This document contains assignments given in LSU EE 4720 over many semesters. It was automatically generated and so some solutions (and even some assignments) are possibly missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at `https://www.ece.lsu.edu/ee4720/prev.html`.

Contents

1 Spring 2025

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Midterm Examination

Friday, 21 March 2025 9:30-10:20 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [30 pts] Appearing on the facing page is the MIPS implementation that includes the `addsc` from Homework 3.

(*a*) The first four code fragments below will execute as shown with the illustrated control logic (from the Homework 3 solution), but the logic won't generate the stall for the last fragment.

☐ Add control logic to the implementation so that *all* of the code fragments execute as shown. That is, add logic to generate the stall signal ☐ for the last fragment ☐ without changing whether the others stall.

```
lw R5, 8(r2)          IF ID EX ME WB        # Correctly stalls with existing logic.
addsc r3, r4, R5, 7      IF ID -> EX ME WB

lw R4, 8(r2)          IF ID EX ME WB        # Correct with existing logic.
addsc r3, R4, r5, 7      IF ID EX ME WB

xori R4, r2, 8        IF ID EX ME WB        # Correct with existing logic.
and r6, R4, r5           IF ID EX ME WB

lw R5, 8(r2)          IF ID EX ME WB        # Correctly stalls with existing logic.
xor r6, r4, R5           IF ID -> EX ME WB

# Cycle               0  1  2  3  4  5  6
lw R4, 8(r2)          IF ID EX ME WB        # Should stall but doesn't with existing logic.
xor r6, R4, r5           IF ID -> EX ME WB
```

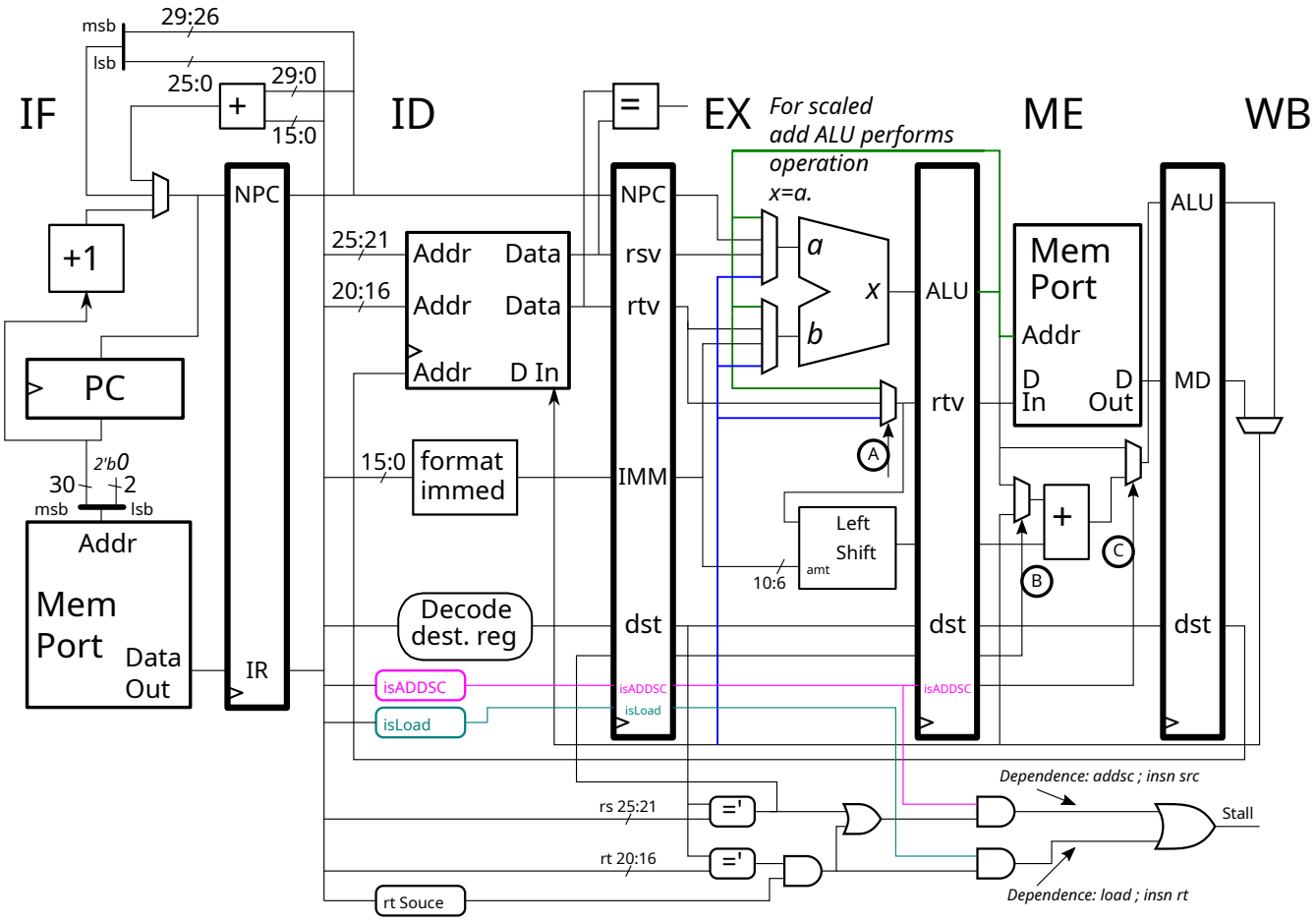
(*b*) Notice that in the first two fragments below the `addsc` shift amount is zero, and so those instructions just add. In the first fragment `addsc` executes in `ME` due to the load dependence, but in the second fragment it executes in `EX` so it can avoid stalling the `or`. *Note: The material about* `doADDSC` *described below was not in the original exam.*

☐ Modify the control logic so that an `addsc` with a zero shift executes as shown below. Do so by ☐ relabeling the `isADDSC` pipeline latches to `doADDSC`. Set this signal to `1` only if there is an `addsc` in `ID` that needs to execute in `ME`. ☐ The logic should not break correct behavior for other cases, such as the ones above.

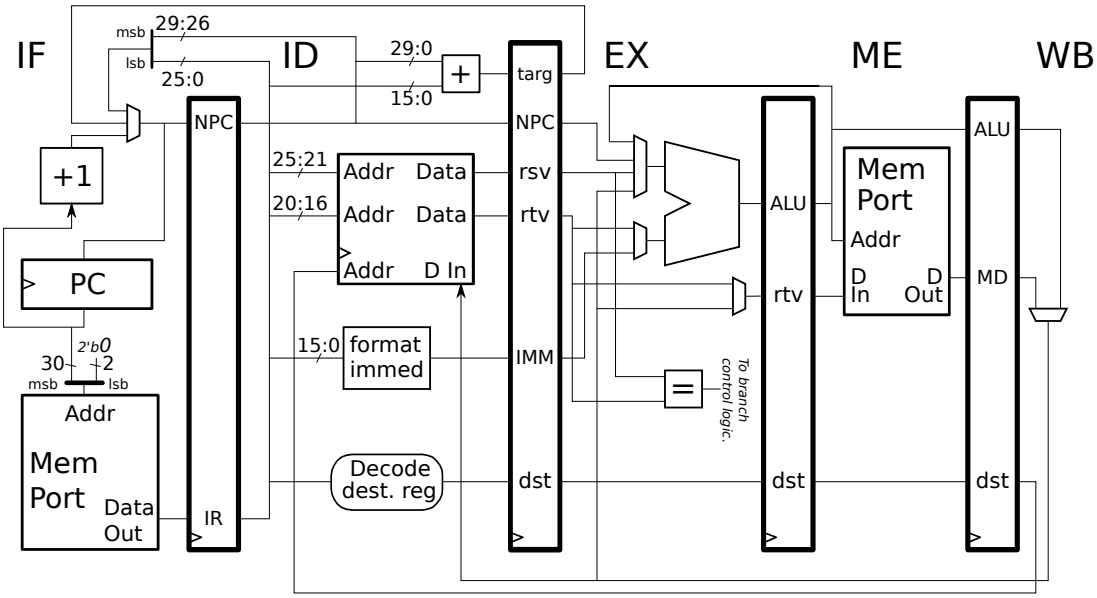
```
# Cycle               0  1  2  3  4  5  6  7  Fragment b1 - addsc adds in ME.
lw R2, 8(r9)          IF ID EX ME WB
addsc r1, R2, r3, 0      IF ID EX ME WB
or    r5, r10, r6           IF ID EX ME WB

# Cycle               0  1  2  3  4  5  6  7  Fragment b2 - addsc adds in EX
andi R2, r9, 8        IF ID EX ME WB
addsc R1, R2, r3, 0      IF ID EX ME WB
or    r5, R1, R6            IF ID EX ME WB

# Cycle               0  1  2  3  4  5  6  7  Fragment b3 - addsc adds in ME.
lw R6, 8(r9)          IF ID EX ME WB
addsc R1, r2, r3, 4      IF ID EX ME WB
or    r5, R1, R6            IF ID -> EX ME WB
```

Staple This Side

Staple This Side



Problem 2: [30 pts] In the MIPS implementation below pay attention to bypass paths and how the branch is resolved.



☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
add r1, r2, r3

sub r4, r1, r5

and r6, r7, r4

sw  r1, 0(r6)
```

☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
addi r1, r1, 1

sw r1, 0(r2)

sw r1, 4(r2)
```


Staple This Side

Staple This Side

☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
lw r1, 0(r2)

lw r4, 4(r2)

sw r1, 0(r3)

sw r4, 4(r3)
```

☐ Show the execution of the code below with ☐ the branch taken on the implementation above. ☐ Don't forget to check for dependencies! ☐ Pay attention to branch behavior.

```
add r1, r2, r3

beq r1, r4, TARG

sw r1, 0(r8)

and r5, r1, r6

ori r5, r5, 0x6

sw r5, 4(r8)


TARG:
 lw r1, 8(r8)
```

☐ Show how the inputs to the [=] box in EX can be changed to eliminate stall(s) ☐ in the example above, and stalls for other kinds of ☐ dependenices. ☐ Do not add hardware, just change the inputs.

Problem 3: [40 pts] Answer each question below.

(*a*) In the routine below `r4` holds an integer, call its value x, and `f1` holds a single-precision float, call its value y. Complete the routine so that register `f9` holds $x \times y$ in single-precision floating point.

☐ Complete the routine so that `f9` is written with the product of the values of `r4` and `f1`. The solution only requires a few instructions, ☐ don't try to fill the entire page.

```
add r4, r5, r5
add.s f1, f2, f3

# At this point r4 holds an integer and f1 holds a single-precision float.
```


Staple This Side

(*b*) The three MIPS code fragments below each do the same thing, and infinite loops are not the problem.

```
loop:   # Fragment A
        sw $t4, 0($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4

loop:   # Fragment B
        sw $t4, 0($t5)
        sw $t4, 4($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 8

loop:   # Fragment C
        sb $t4, 0($t5)
        sb $t4, 1($t5)
        sb $t4, 2($t5)
        sb $t4, 3($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4
```

☐ Which code fragment is the fastest, ◯ *Fragment A*, ◯ *Fragment B*, or ◯ *Fragment C*?

☐ Which code fragment is the slowest, ◯ *Fragment A*, ◯ *Fragment B*, or ◯ *Fragment C*?

☐ Explain choice of ☐ fastest and ☐ slowest fragment, and ☐ include a good definition of fast.

☐ Assume that the contents of `t5` and `t3` refer to a range of valid memory addresses. Which fragment(s) put a restriction on the value of `t5`? ☐ Explain. Assume that `t3` is always chosen to avoid an infinite loop.

(*c*) When designing a RISC ISA what is the most important criterion when considering possible instructions based on the material presented in class?

☐ Most important factor when deciding whether an instruction should be added to a RISC ISA.

☐ Give an example of an instruction unsuitable for RISC and ☐ explain how the criterion makes it unsuitable.

(*d*) CISC ISAs have powerful instructions, such as `add 4(r1), (r2), ((r3))` or a call instruction that automatically saves registers.

☐ What is the benefit of powerful instructions, especially in the days when memory was made by people sewing wires around little metal rings.

(*e*) Intel has updated IA-32 (a.k.a. x86) since the 1980s, and later added a 64-bit variant, Intel-64. Recall that nobody actually likes IA-32.

☐ So why did Intel's customers continue to buy implementations of IA-32 and Intel 64 rather than switching to a better-designed ISA? (Note that Apple is an exception to the rule that computer makers don't switch ISAs.)

(*f*) MIPS uses the `func` field as an opcode extension field.

☐ Why is an opcode extension field needed?

☐ Why didn't they just make the opcode longer when designing MIPS?

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Thursday, 8 May 2025 7:30-9:30 CDT

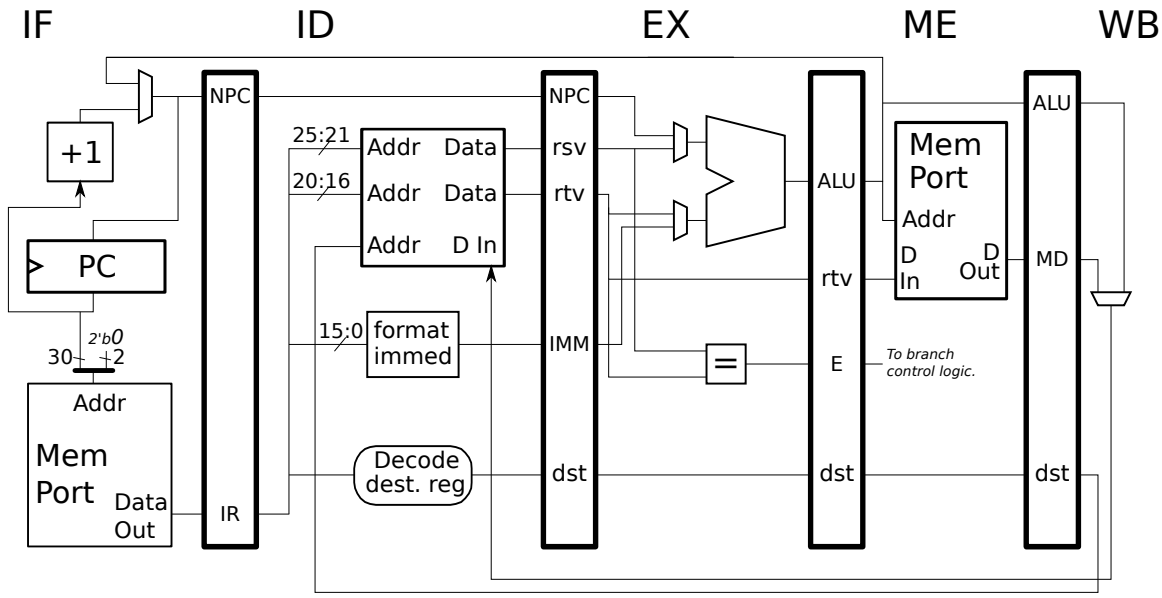
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

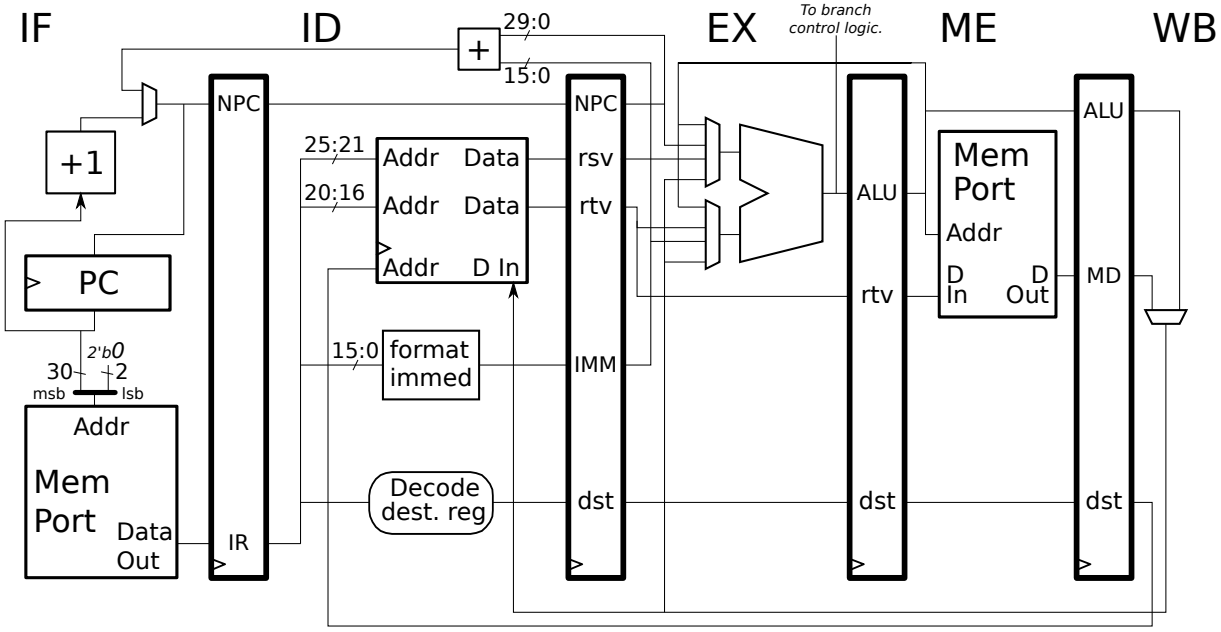
Problem 1: (20 pts) Appearing below are MIPS implementations and code fragments. Show execution (a pipeline execution diagram) of the code on the accompanying implementations.



☐ Show execution of the code below on the implementation above ☐ with the branch taken. ☐ Check for dependencies, including ☐ those for the branch.

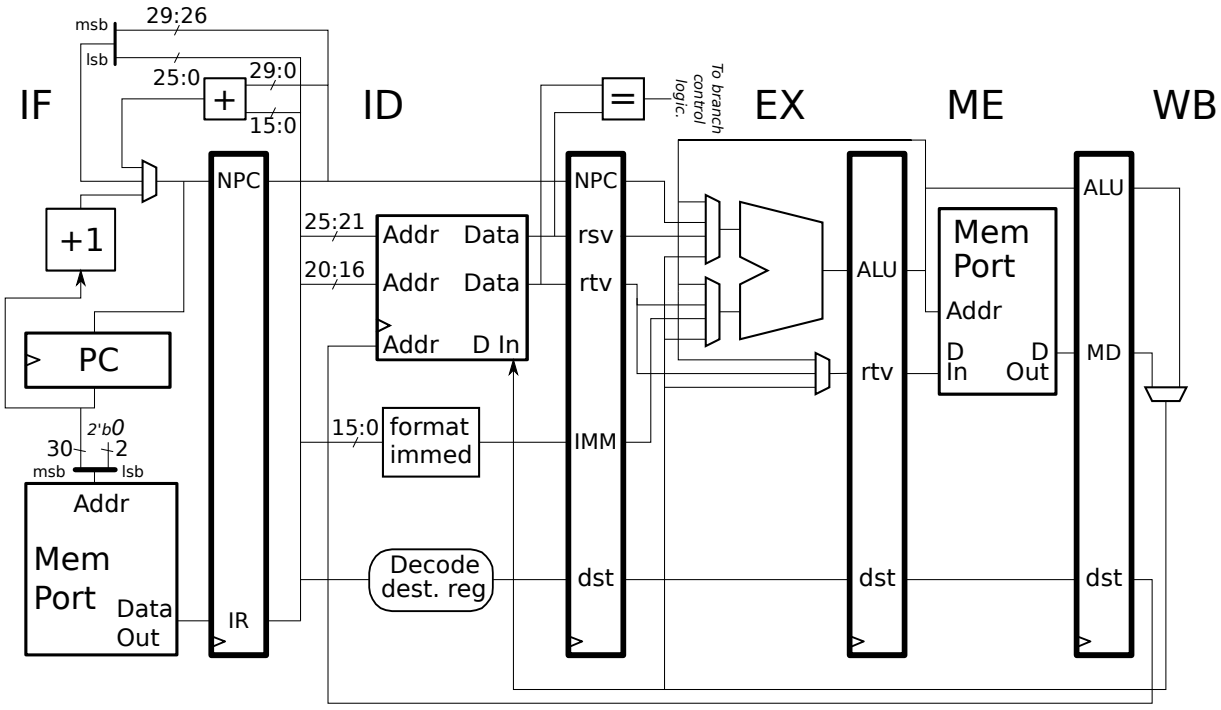
```
# Cycle

lw r2, 4(r5)

slt r1, r2, r3

bne r1, r0  SKIP

lw r4, 0(r5)

ori r7, r4, 0xaa

addi r10, r11, 12

sub r11, r12, r14

SKIP:
andi r6, r4, 0xf0

addi r5, r5, 4
```


Staple This Side



☐ Show execution of the code below on the implementation above (debuting in this exam) ☐ with the branch taken. ☐ Check for dependencies, including ☐ those for the branch.

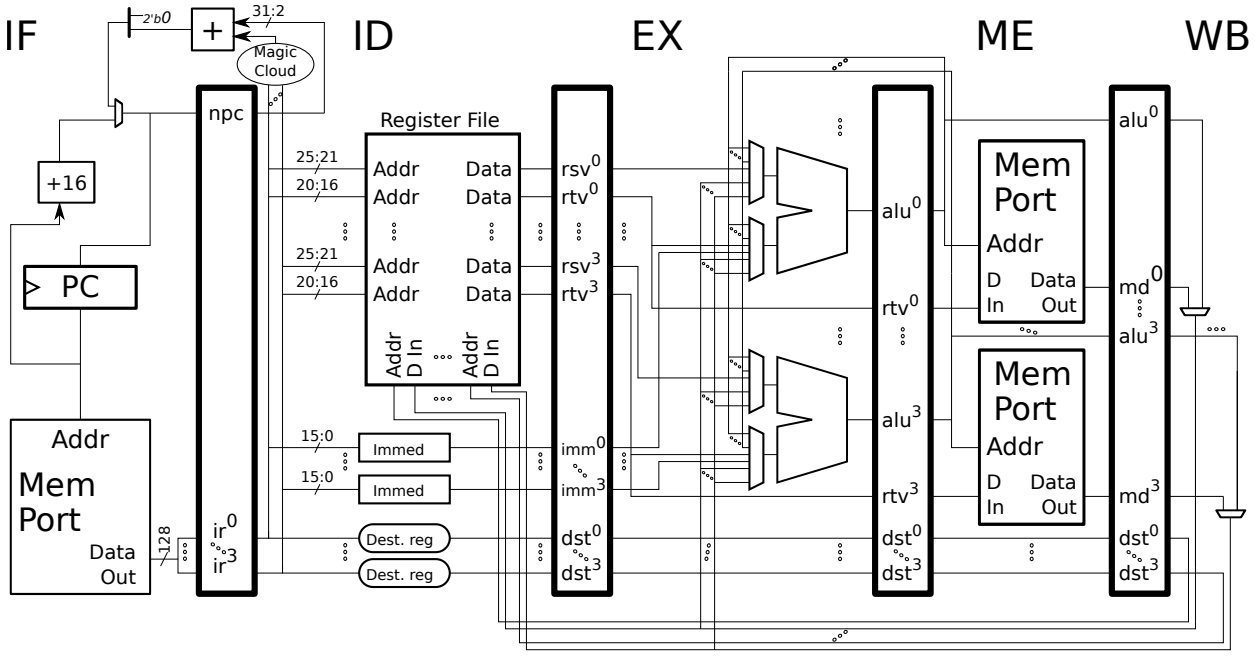
```
# Cycle

lw r2, 4(r5)

slt r1, r2, r3

bne r1, r0  SKIP

lw r4, 0(r5)

ori r7, r4, 0xaa

addi r10, r11, 12

sub r11, r12, r14

SKIP:
andi r6, r4, 0xf0

addi r5, r5, 4
```



☐ Show execution of the code below on the implementation above ☐ with the branch taken. ☐ Check for dependencies, including ☐ those for the branch.

```
# Cycle

lw r2, 4(r5)

slt r1, r2, r3

bne r1, r0  SKIP

lw r4, 0(r5)

ori r7, r4, 0xaa

addi r10, r11, 12

sub r11, r12, r14

SKIP:
andi r6, r4, 0xf0

addi r5, r5, 4
```


Staple This Side



☐ Show execution of the code below with ☐ the branch taken on the ☐ 4-way superscalar implementation above in which ☐ fetch groups are not aligned. ☐ Use course assumptions about instruction ordering and branch handling. ☒ Use an `x` to show where instructions gets squashed, don't omit `IF` of instructions that will be squashed. ☒ Check for dependencies. ☒ The code below is different than the previous parts.

```
# Cycle

lw r2, 4(r5)


sub r1, r2, r3


bne r9, r0  SKIP


lw r4, 0(r5)


ori r7, r8, 0xaa


SKIP:
and r6, r4, r7


addi r5, r5, 4
```

Problem 2: (20 pts) Recall that in Homework 5 Problem 2b the FP add hop paths from the 2024 final exam were to be replaced by bypass paths. Those bypass paths are shown in green on the facing page, which shows a slightly cleaned up solution.

(*a*) The bypass paths shown are correct, however the `WF.wa` pipeline latch output (on the right side) is unconnected and the `Addr` and `WE` inputs to the FP register file are wrong.

☐ Modify the hardware so that the `WE` (write enable) input to FP Reg File is correct. ☐ Don't forget about the unconnected `WF.wa`.

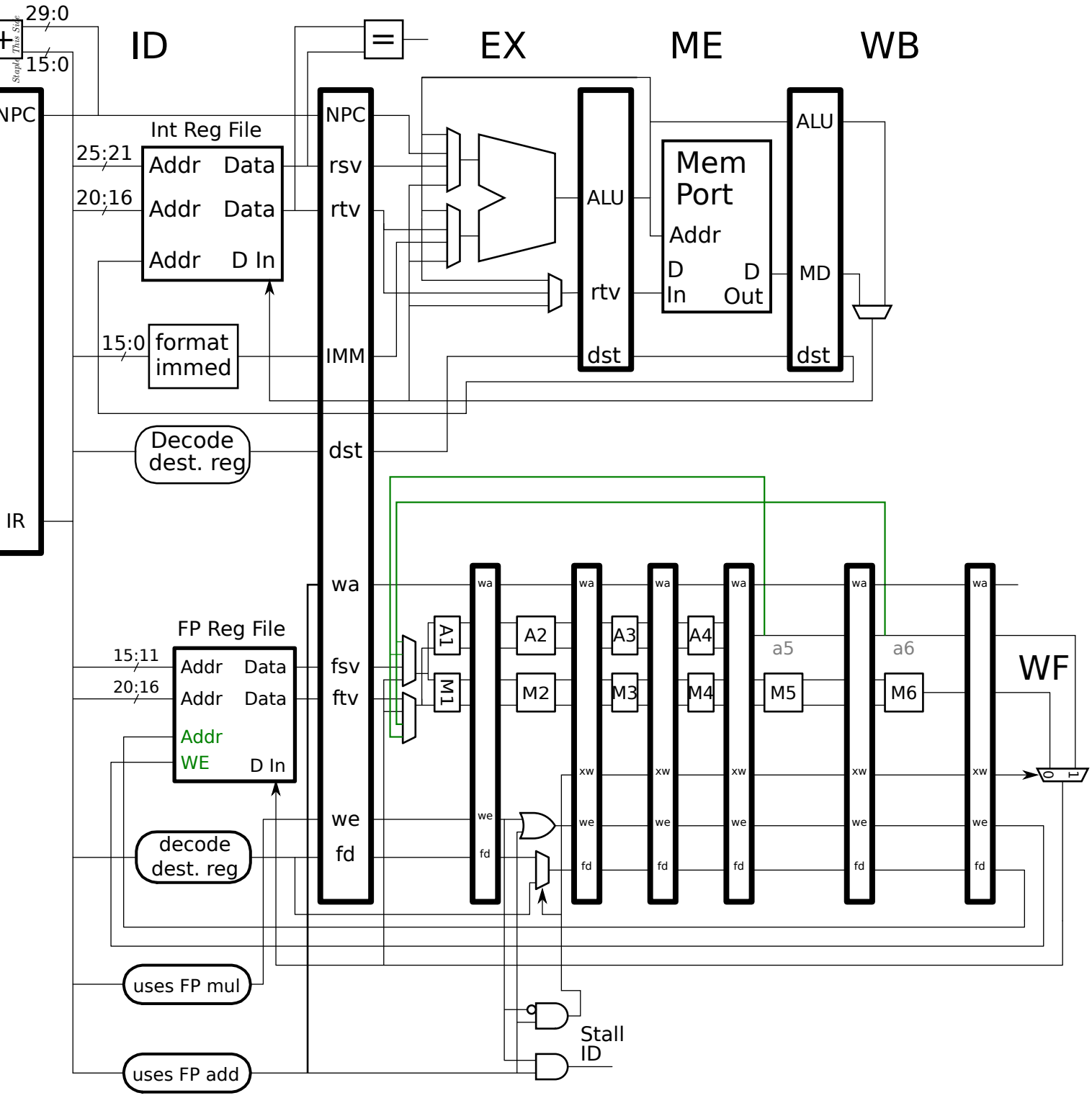
☐ Modify the hardware so that the `Addr` (destination register number) input to FP Reg File is correct. ☐ Think about deleting hardware rather than adding new hardware (other than reconnecting wires).

☐ Modify the hardware so that the select signal for the `WF` mux is correct. ☐ Think about cost, remove unneeded hardware.

☐ *Don't overdo it.* A solution to this part consists mostly of crossing things out and reconnecting things. Only a few gates need to be added.

Staple This Side

The IF stage is not shown to make space.



(*b*) Those expecting a bypass mux control logic problem have not been forgotten! On the facing page is the homework solution but with a part covered, and with parts of the integer pipeline off the page to make space. Assume that the values in the `wa`, `fd`, and `we` pipeline latches in stages `M3` and later are correct. Also assume that the instruction in `ID` does not need to stall. *Stall logic might be asked for in a 2026 homework assignment based on this problem.* Some control logic for the lower FP bypass mux is shown. The mux is blue to stand out.

☐ The wire labeled `SFTV` is the mux select signal. Provide a path for it to the lower bypass mux feeding `M1` and `A1`. ☐ Assume that the critical path passes through `M1` and `A1`.

☐ Complete the hardware to compute `SFTV`. ☐ Note that `wa` and `we` should be used and their values can be based on your part (a) solution.

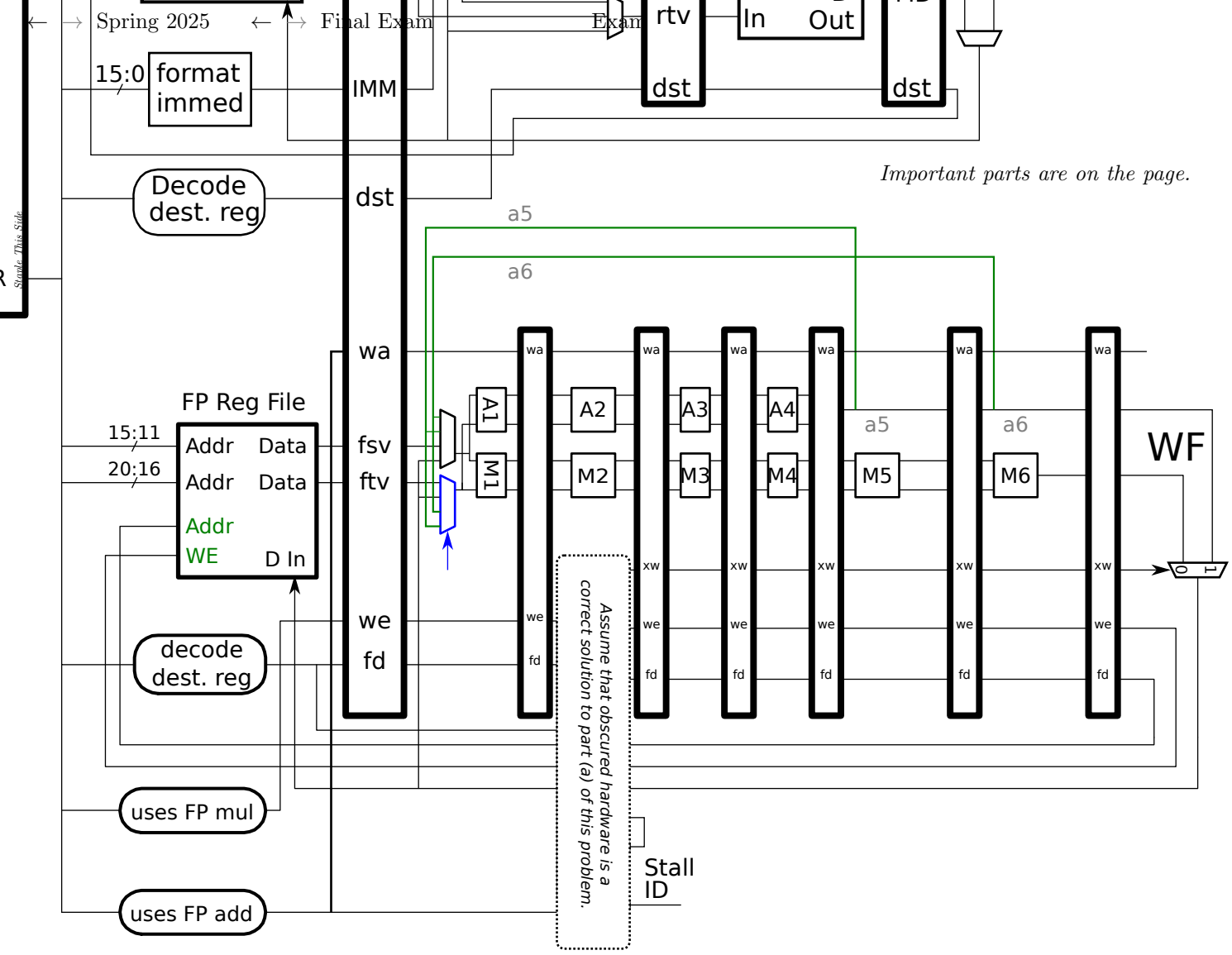
☐ Explain why a $=$ and not a $='$ should be used in the solution.

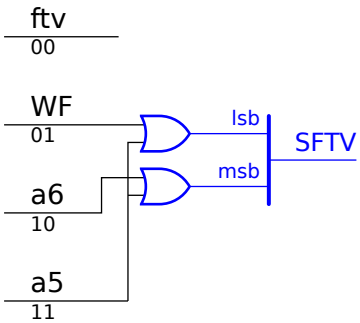
Use the execution diagram below to help keep track of where instructions might be when designing control logic.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
mul.s F1, f2, f3 IF ID M1 M2 M3 M4 M5 M6 WF
add.s F7, f4, f5    IF ID A1 A2 A3 A4 a5 a6 WF
add.s F8, f6, f6       IF ID A1 A2 A3 A4 a5 a6 WF
add r1, r2, r3            IF ID EX ME WB
sub r1, r2, r3               IF ID EX ME WB
add.s f12, f11, F7              IF ID A1 A2 A3 A4 a5 a6 WF
add.s f10, f11, F1                 IF ID A1 A2 A3 A4 a5 a6 WF
add.s f14, f11, F8                    IF ID A1 A2 A3 A4 a5 a6 WF
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Staple This Side

Staple This Side





Important parts are on the page.

Staple This Side

Problem 3: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. *Commas are used* to emphasize the repeating patterns. One system has a bimodal predictor, and the others use local predictors. Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T  N  T  T  N  T  N  T,  T  N  T  T  N  T  N  T,  T  N  T  T  N  T  N  T,

B2:   N  N  T, N  N  T,  N  N  T, N  N  T,  N  N  T, N  N  T, ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`? ☐ Be sure to base the accuracy on a warmed-up 2-bit counter and repeating pattern.

For help in solving the local predictor problems the 8 possible B1 pattern rotations and the three possible B2 rotations are shown below. The table to the right shows them sorted to facilitate computing local history accuracies.

Unsorted		Sorted	
TnTTnTnTT	B1	TTnTTnTnT	B1
nTTnTnTTn	B1	TTnTnTTnT	B1
TTnTnTTnT	B1	TnTTnTTnT	B1
TnTnTTnTT	B1	TnTTnTnTT	B1
nTnTTnTTn	B1	TnTnTTnTT	B1
TnTTnTTnT	B1	TnnTnnTnn	b2
nTTnTTnTn	B1	nTTnTTnTn	B1
TTnTTnTnT	B1	nTTnTnTTn	B1
nnTnnTnnT	b2	nTnTTnTTn	B1
nTnnTnnTn	b2	nTnnTnnTn	b2
TnnTnnTnn	b2	nnTnnTnnT	b2

☐ What is the accuracy of an 8-outcome local history predictor on B1 ignoring B2.

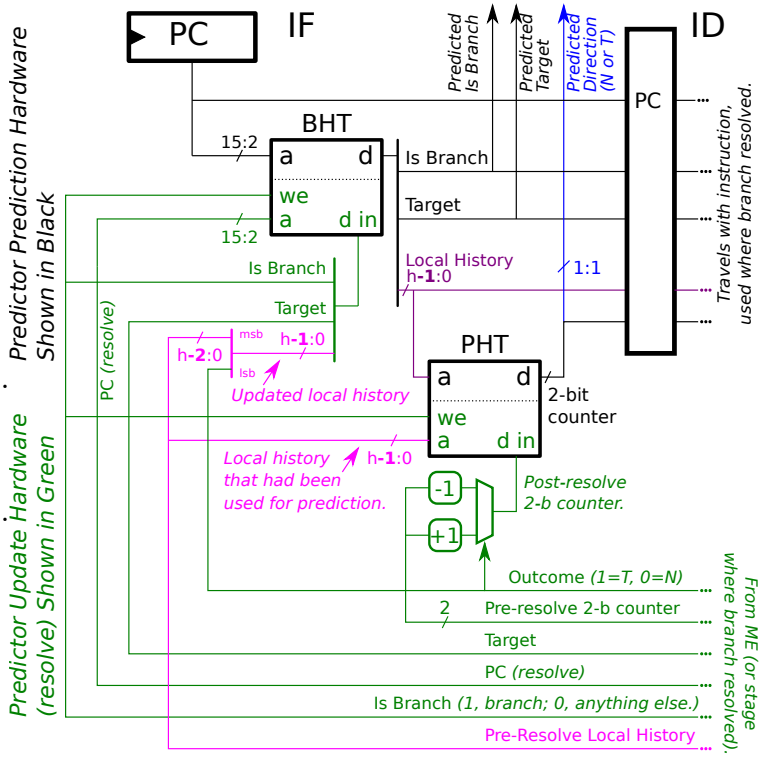
☐ What is the accuracy of a 3-outcome local history predictor on B1 ignoring B2.

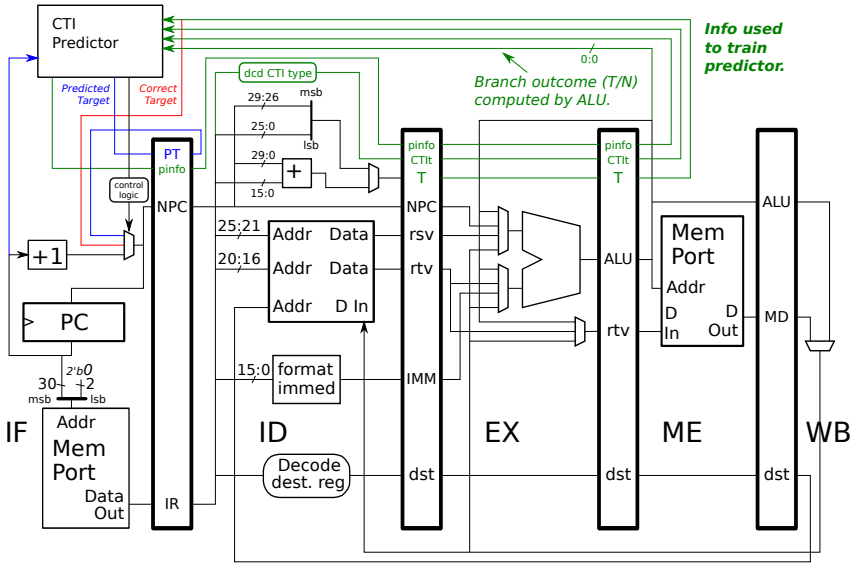
☐ What is the accuracy of a 3-outcome local history predictor on B1 **taking into account** B2. ☐ Note that the B2 pattern repeats faster than the B1 pattern.

Staple This Side

(*b*) The diagrams show a local predictor (to the right) and how a predictor might be integrated into the MIPS pipeline (below). Because only bits 15:2 of an instruction address are used in the BHT lookup it is possible to predict one branch, say at address `0x10024` in the code fragment below, using the BHT entry for a different branch, say at `0x30024`.

```
0x10024 beq r1, r2, TARGa   T T T...

# Far away.

0x30024 beq r3, r4, TARGb   T T T...
```



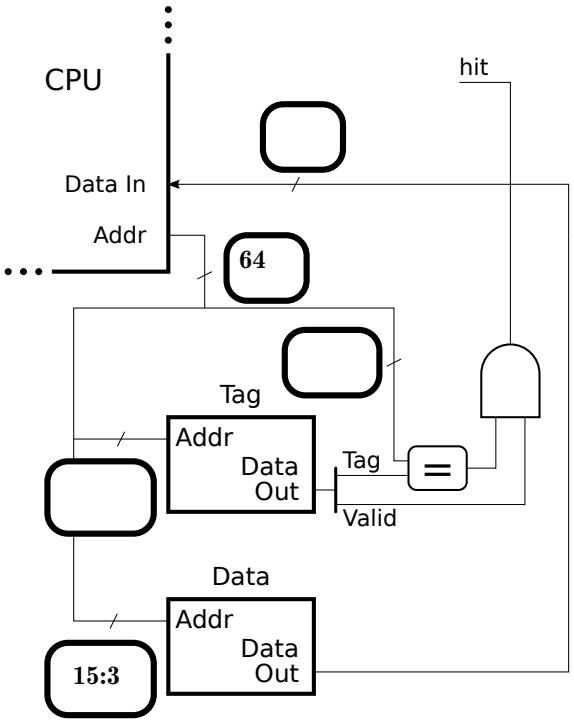


☐ Suppose both branches are always taken and that they are correctly predicted taken. Why would there still be a misprediction when predicting `0x10024` that can result in the wrong target being reached?

☐ Add hardware to detect the problem, the output should be labeled `MISPRED`. ☐ The hardware is very simple.

Problem 4: (15 pts) The diagram below is for a direct-mapped cache with a 64 byte (2^6 B) line size. The character size is a byte.
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the number of bits or bit ranges in the unfilled boxes above.

☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: 

☐ Cache Capacity, in Bytes (how much data can it cache).

Staple This Side

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 256 bytes (characters). The code fragment starts with the cache cold (empty); consider only accesses to the array. Of course, $2^8 = 256$.

(*b*) Find the hit ratio executing the code below.

```
int64_t sum = 0;
int64_t *a = 0x2000000; // sizeof(int64_t) == 8
int ILIMIT = 1 << 14;    // = 2^14

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

(*c*) The program from the previous part should run faster on a system with longer line size.

☐ In general, what is the disadvantage of a longer line size?

☐ Describe the characteristics of a program that runs better with shorter line sizes.

Problem 5: (25 pts) Answer each question below.

(*a*) Show the encoding of the MIPS instructions below.

☐ Show encoding. ☐ Label fields, and show specific values where possible. ☐ Pay attention to the format that this particular instruction uses.

```
sll r9, r8, 7
```

☐ Show encoding of `bne`. ☐ Label fields, and show numeric values where possible, ☐ including the field encoding the branch target.

```
bne r5, r6, SKIP     # Show encoding of this instruction only.
sw r4, 5(r6)
lui r8, 0x5678
addi r8, r8, 0x1234
SKIP:
add.s f1, f2, f3
```

(*b*) The code below uses MIPS pseudo instruction `li` (load immediate) several times.

```
li r1, 0x12345678
li r2, 0x990000
li r3, 0x8
```

☐ Rewrite the code using real MIPS instructions.

Staple This Side

(*c*) The organization SPEC decides the programs that go into the very popular and influential SPECcpu benchmarks and how they should be run. The goal is to measure performance on a set of programs that typical users run.

Suppose there are rumors that company E has successfully bribed SPEC to omit benchmarks from the next SPECcpu suite that make company E's products look bad in comparison to other companies' products.

☐ How might ordinary users gauge whether these rumors are true based upon how SPEC is organized?

(*d*) One reason the VAX CALLS instruction was not suitable for a RISC ISA was because it might need to write several registers to memory.

☐ Why does the need to write several registers to memory make CALLS unsuitable for a RISC ISA?

☐ Why is it not a problem for a CISC ISA?

(*e*) The cost of an n-way superscalar implementation at some point will be proportional to n^2.

☐ What will contribute most to the n^2 cost?

(*f*) Provide two important reasons that an n-way superscalar RISC implementation will not have n times the instruction throughput (IPC) of a scalar RISC implementation on typical code.

☐ Two important reasons it does not have $n\times$ the instruction throughput on typical code.

2 Spring 2024

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Midterm Examination

Wednesday, 3 April 2024 9:30-10:20 CDT

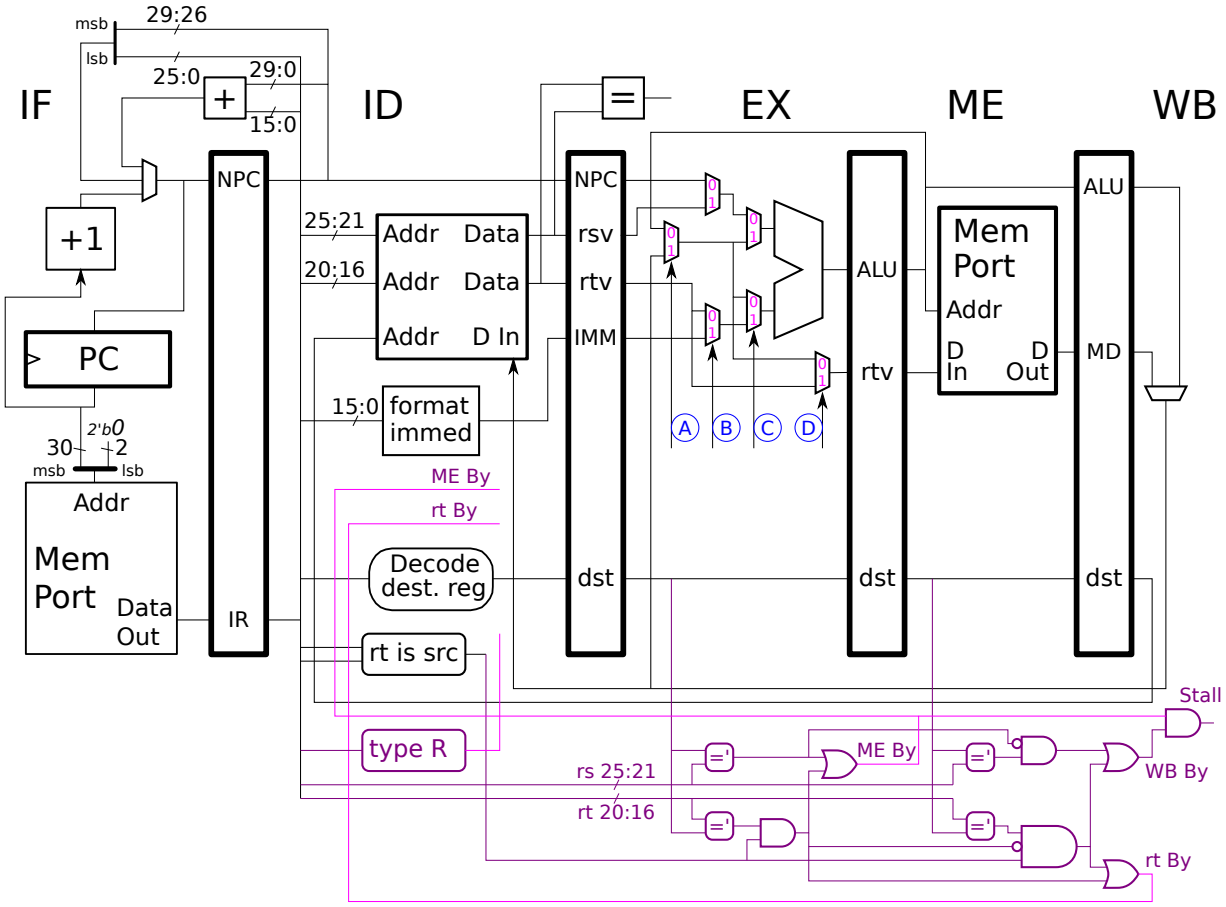
Problem 1	_____	(18 pts)
Problem 2	_____	(17 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(20 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [18 pts] Appearing below is a **changed** version of the MIPS implementation appearing in Homework 3 and the 2020 midterm exam.



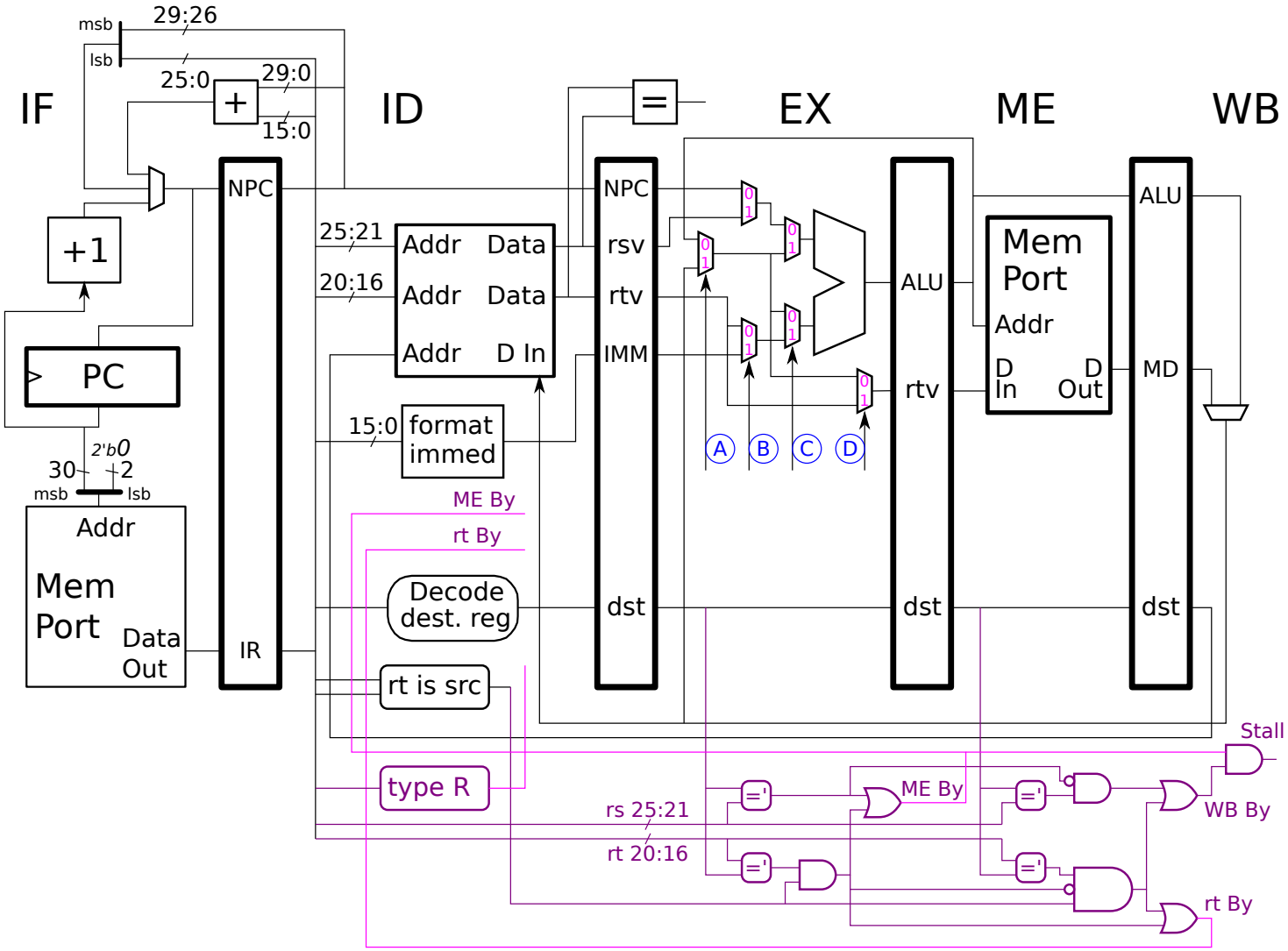
☐ In the table show the select signal values expected for the execution shown below. ☐ Use `X` for select signals that don't matter (that can be either `0` or `1`). ☐ **Don't forget** to check for dependencies

```
# Cycle          0     1     2     3     4     5     6     7
add r1, r2, r3   IF    ID    EX    ME    WB
ori r7, r1, 9          IF    ID    EX    ME    WB
sub r8, r9, r7               IF    ID    EX    ME    WB
sw r7, 5(r6)                       IF    ID    EX    ME    WB
# Cycle          0     1     2     3     4     5     6     7

A

B

# Cycle          0     1     2     3     4     5     6     7

C

D

# Cycle          0     1     2     3     4     5     6     7
```

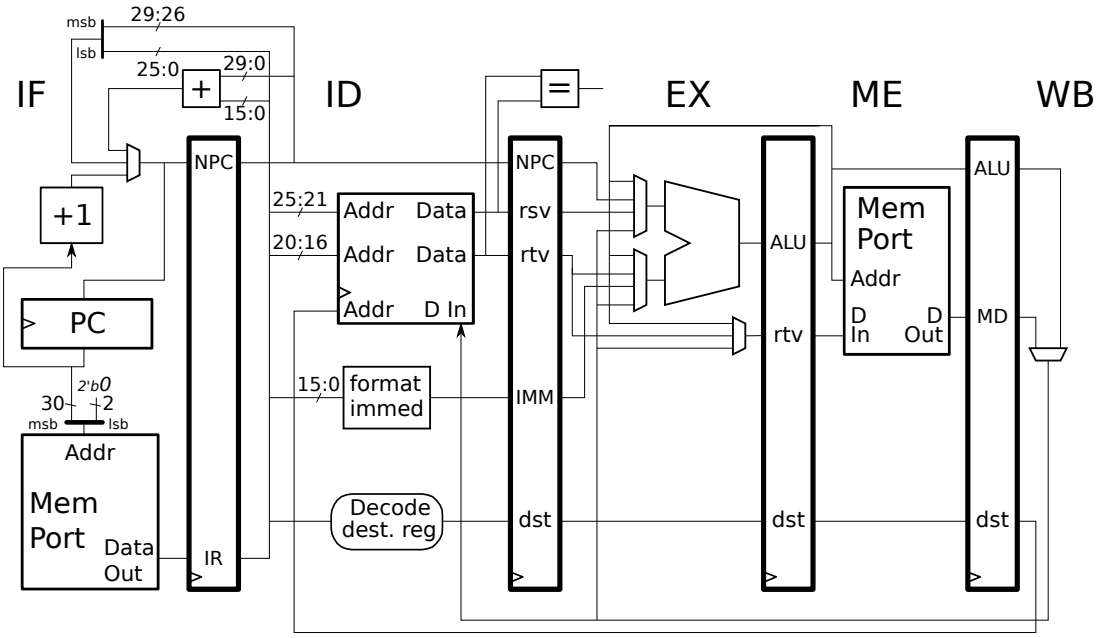
Problem 2: [17 pts] Appearing below is the implementation from the previous problem. It **is not identical** to the Homework 3 implementation. *See the last page of this exam for the Homework 3 Problem 3 solution.*

☐ Design the control logic for the A, B, C, and D select signals.

☐ Take advantage of existing logic, not much more logic is needed. ☐ Make sure that C works for the code fragment in the previous part. ☐ Don't forget that execution is pipelined.



Problem 3: [15 pts] Show the execution of the MIPS code fragments on the implementation.



☐ Show the execution of the fragment below with ☐ the branch taken. ☐ Pay close attention to branch behavior.

```
beq r1, r1, SKIPA

add r2, r3, r4

sub r5, r6, r7

ori r8, r9, 100

xori r10, r11, 101


SKIPA:
lw r12, 0(r14)
```

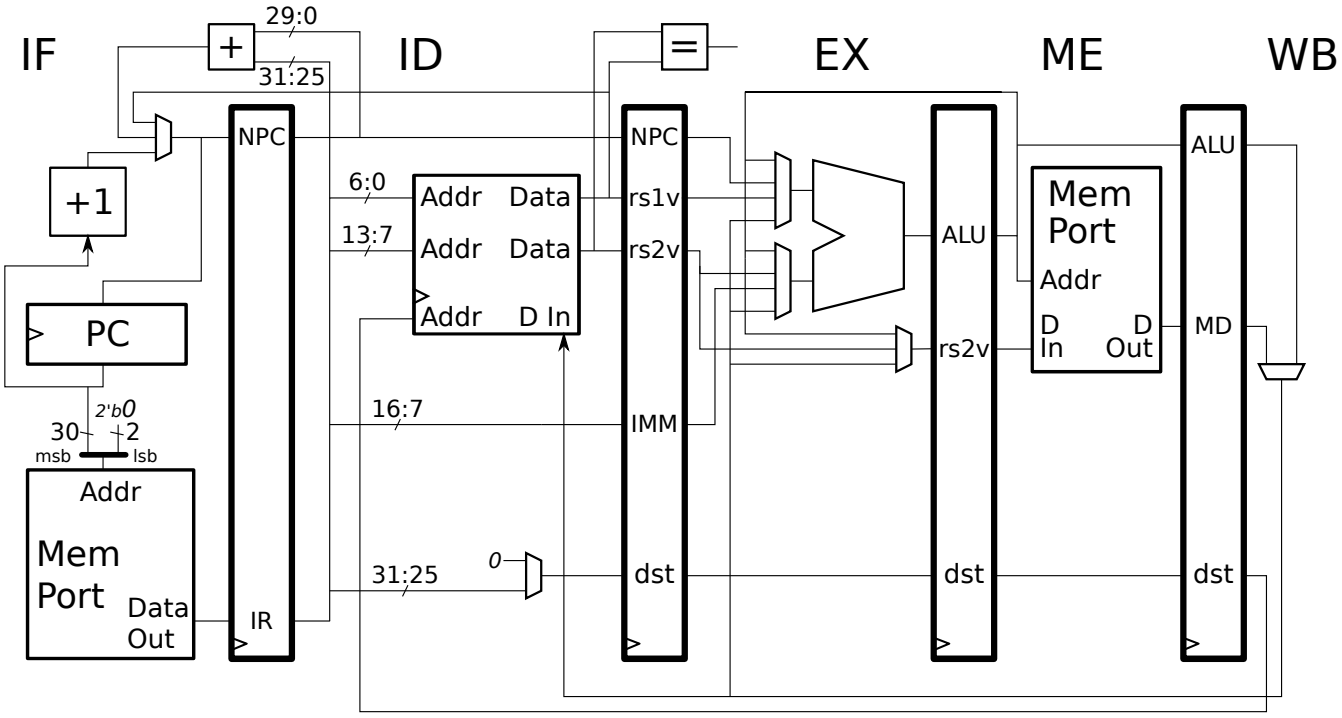
☐ Show the execution of the fragment below. ☐ Be sure to check for dependencies.

```
addi r5, r5, 4

lw r2, 0(r5)

add r1, r2, r3

sub r4, r1, r4
```


Staple This Side

Staple This Side

Problem 4: [15 pts] Appearing below is the implementation of Another RISC ISA (ARI) and incomplete diagrams for the encoding of its MIPS-like R and I formats.



☐ How many registers does ARI have?

☐ What is ARI's immediate size?

☐ Why is it possible to implement an instruction like `lw r1, 4(r2)` but not an instruction like `sw r1, 4(r2)` on the implementation above?

☐ In the spaces below complete ☐ ARI R and ☐ ARI I instruction formats consistent with the implementation. ☐ Be sure to show the opcode field and any opcode extensions that are needed.

ARI R: ☐ `add r1, r2, r3`

31 0

ARI I: ☐ `addi r4, r5, 6`

31 0

Staple This Side

Problem 5: [20 pts] Answer each question below.

(*a*) Show the contents of the destination register after each MIPS I instruction below executes.

```
#           Initially r1 = 0x12345678
sll r2, r1, 16
#                 ☐  r2 =

srl r3, r1, 16
#                 ☐  r3 =

or  r4, r2, r3
#                 ☐  r4 =
```

(*b*) Given the MIPS code below, why might execution never reach the `or` instruction?

```
lw $a0, 0($t0)
jal SOME_CONVENTIONAL_STANDARD_LIBRARY_FUNCTION
addi r31, r31, -8
or $s1, $s1, $v0
```

☐ The `or` instruction won't be reached because:

☐ What will happen instead is:

(*c*) Register `r9` holds the address of the middle of a large memory allocation, and so all the MIPS `lb` instructions below execute with no problem. Not so for the `lw` instructions.

```
lb r11, 0(r9)  # Will execute correctly.
lb r12, 5(r9)  # Will execute correctly.
lb r13, 10(r9) # Will execute correctly.
lb r14, 15(r9) # Will execute correctly.

lw r1, 0(r9)
lw r2, 5(r9)
lw r3, 10(r9)
lw r4, 15(r9)
```

☐ Why won't the rest of the MIPS code execute to completion?

☐ What are the maximum and minimum number of `lw` instructions that will execute before an error occurs, and ☐ briefly explain how the maximum and minimum number are determined by the exact value of `r9`.

(*d*) Simplify MIPS the code fragment below.

```
lbu r1, 0(r10)
lbu r2, 1(r10)
sll r1, r1, 8
or r1, r1, r2
sh r1, 2(r10)
# Note: r1 and r2 not used again.
```

☐ Simplify the code fragment ☐ without changing what it does.

Problem 6: [15 pts] Answer each question below.

(*a*) In class we described three families of ISAs, CISC, VLIW, and RISC.

☐ How do VLIW ISAs differ from both RISC and CISC ISAs?

(*b*) Identify the ISA family of the following ISAs:

MIPS: ◯ *CISC* ◯ *VLIW* ◯ *RISC*

Arm A64: ◯ *CISC* ◯ *VLIW* ◯ *RISC*

Itanium: ◯ *CISC* ◯ *VLIW* ◯ *RISC*

Intel 64 /IA-32: ◯ *CISC* ◯ *VLIW* ◯ *RISC*

VAX: ◯ *CISC* ◯ *VLIW* ◯ *RISC*

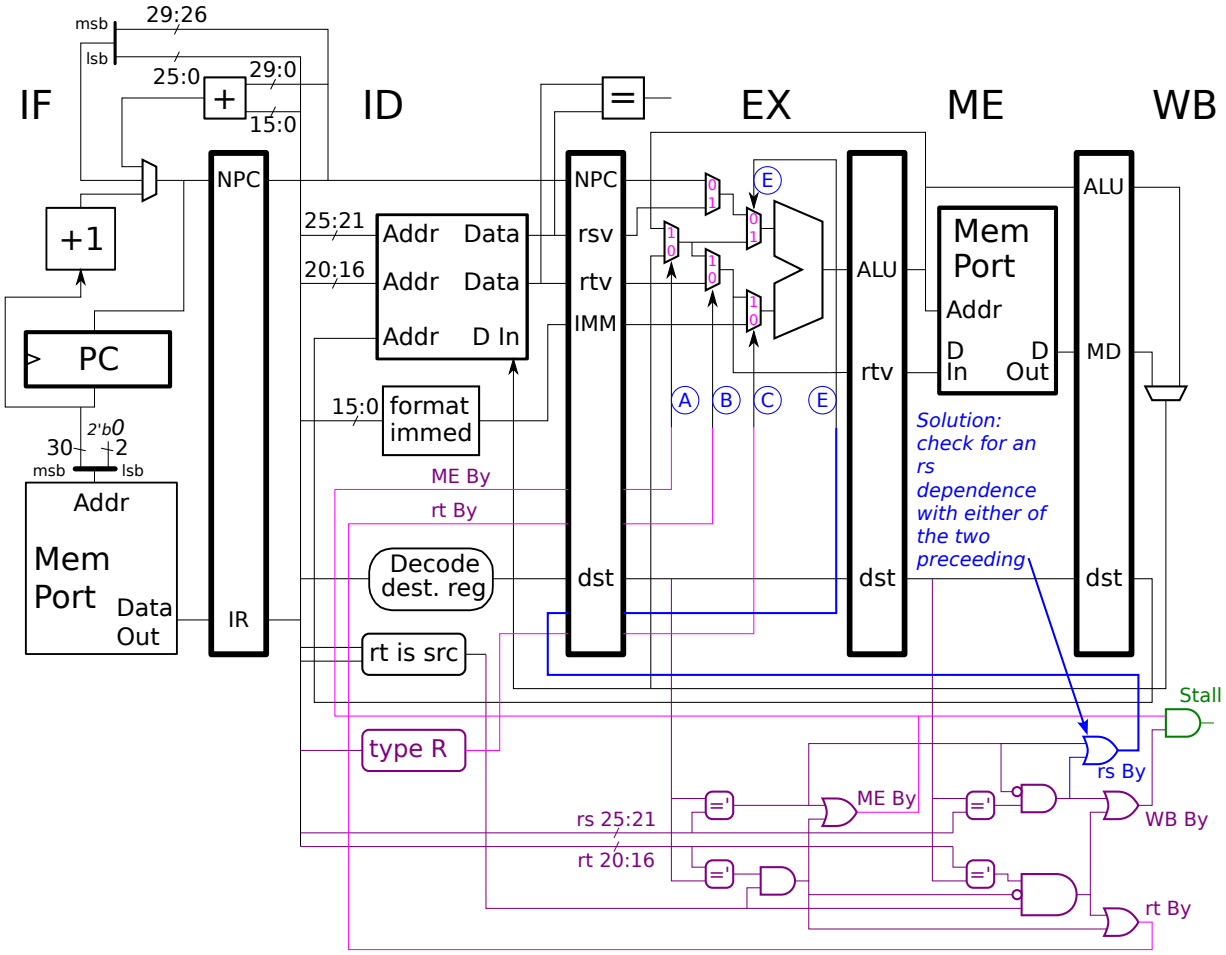
(*c*) The statement below is wrong.

CISC ISAs can have large immediate values, but at the cost of having large instructions. That is why programs in CISC ISAs are large compared to those in RISC ISAs.

☐ What is correct in the statement above?

☐ What is wrong in the statement above?

Appearing below is part of the solution to Homework 3 Problem 3. It may be helpful in solving Problem 2 in this exam.



Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Thursday, 9 May 2024 17:30-19:30 CDT

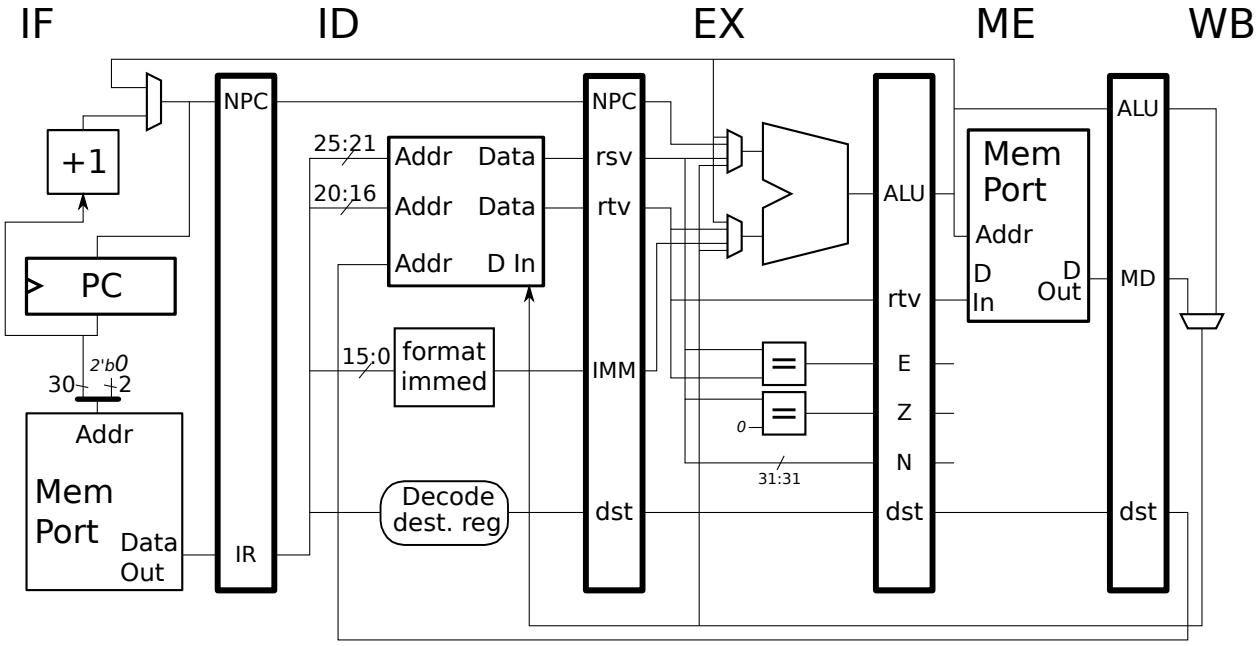
Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(8 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

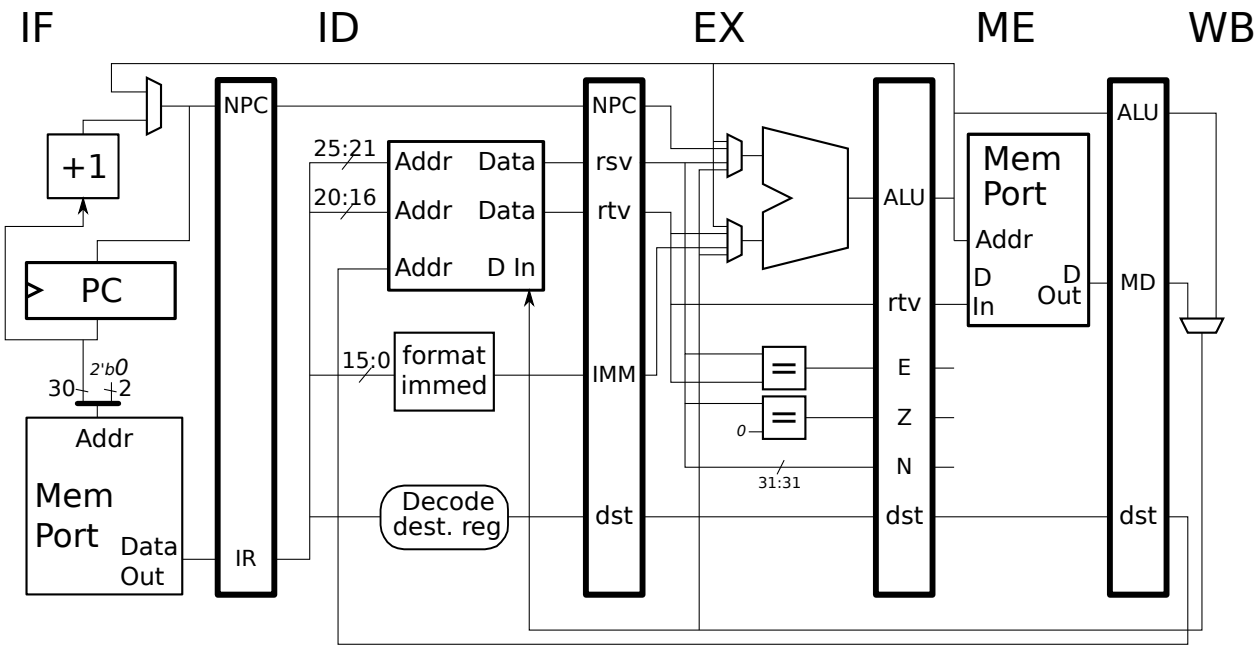
Problem 1: (20 pts) Appearing below are MIPS implementations and code fragments. Show execution (a pipeline execution diagram) of the code on the accompanying implementations.



☐ Show execution of the code below on the implementation above. ☐ Check for dependencies.

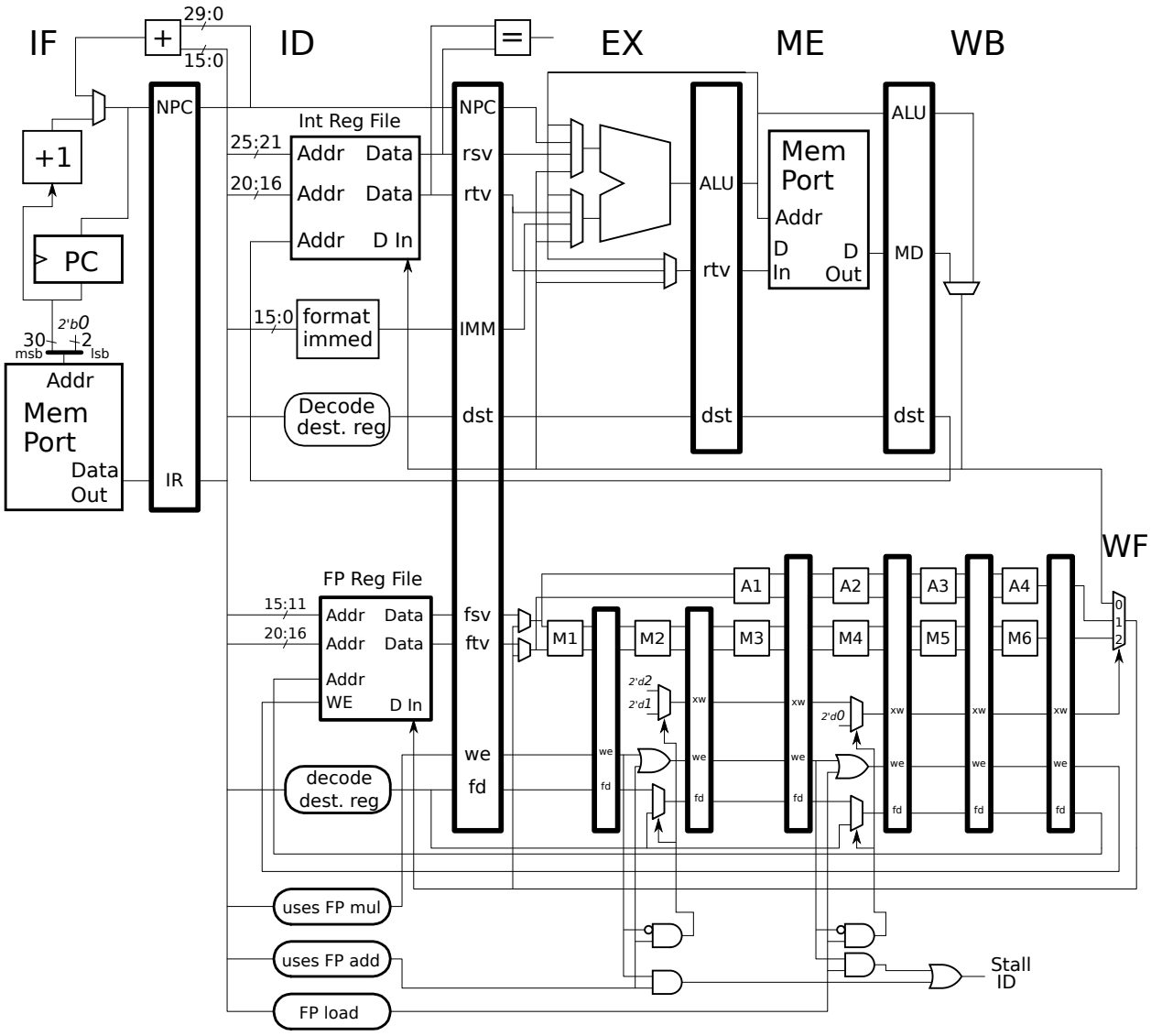
```
add r6, r7, r8

lw r9, 0(r6)

add r1, r2, r9

sub r4, r1, r5

sw r4, 0(r5)
```


Staple This Side



☐ Show execution of the code below on the implementation above ☐ until the second fetch of `lw`. ☐ Show when and where instructions are squashed (with an `x`). ☐ The branch must be taken. ☐ Pay close attention to branch behavior. ☐ Check for dependencies.

```
LOOP:
 lw r3, 0(r4)

 addi r4, r4, 4

 bne r1, r2, LOOP

 addi r1, r1, r3

 sw r3, 0(r9)

 sw r4, 4(r9)

 sw r2, 8(r9)
```



☐ Complete the execution of the code below for the implementation above. ☐ Check for dependencies, ☐ don't overlook the first two `mul.s` instructions.

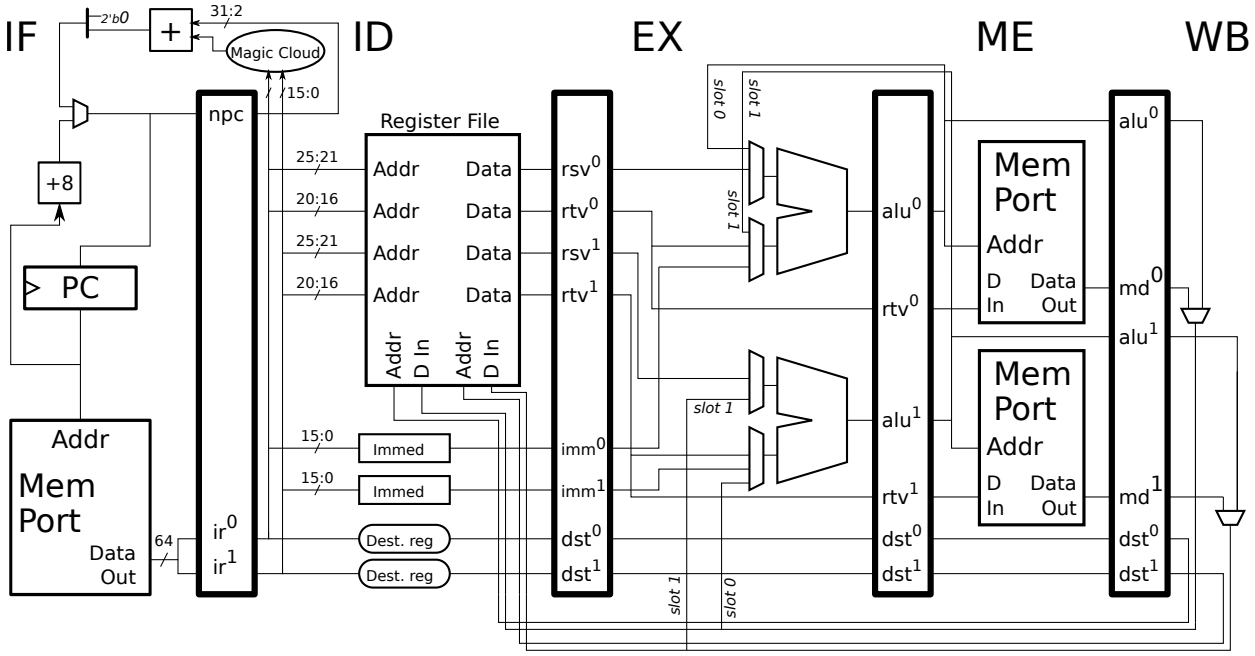
```
# Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12 ...
mul.s f1, f2, f3   IF   ID   M1   M2   M3   M4   M5   M6   WF
mul.s f4, f5, f6        IF   ID   M1   M2   M3   M4   M5   M6   WF


add.s f8, f15, f16


add.s f7, f4, f9


lwc1 f11, 0(r1)


add.s f10, f11, f1

# Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12...
```


Staple This Side

This page left mostly blank to provide room for the pipeline execution diagram.

Problem 2: (15 pts) Answer the following questions on superscalar MIPS implementations.

(*a*) The superscalar MIPS implementation below has only four bypass paths, one per ALU multiplexor. The paths have labels, slot 0 and slot 1 (showing where they originate).

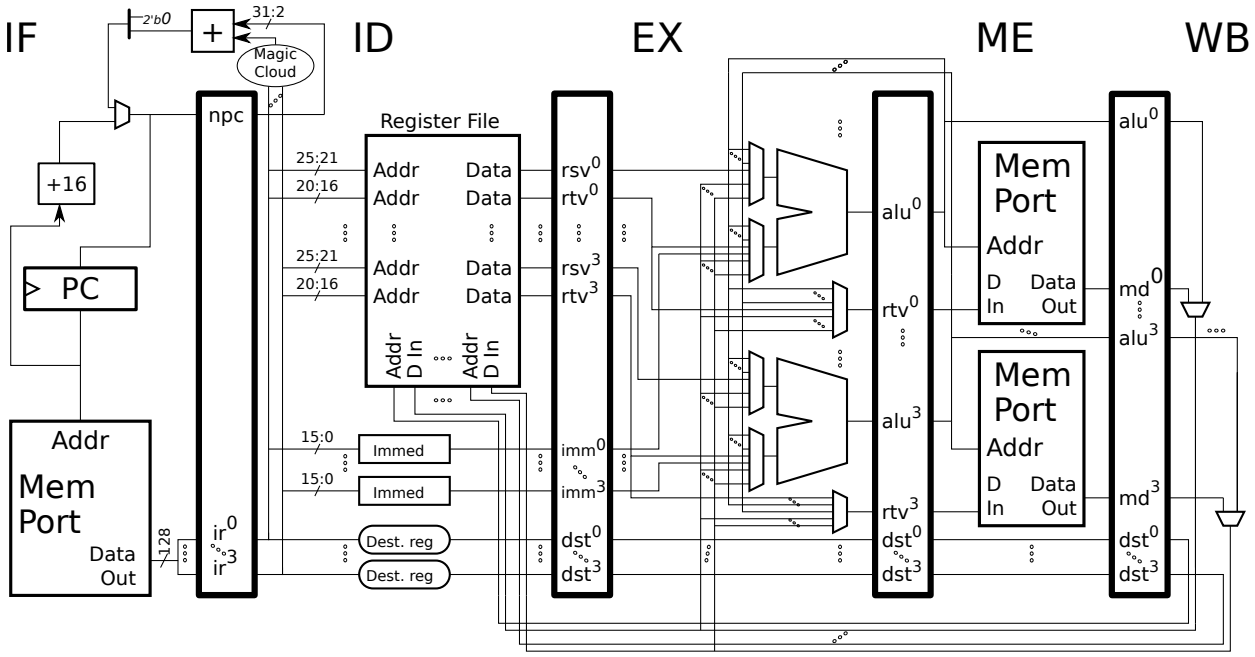


☐ Complete the code fragment below (by adding registers) so that it ☐ uses all four bypass paths **in cycle 4**.

```
# Cycle          0  1  2  3  4  5  6
add              IF ID EX ME WB
sub              IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
or                  IF ID EX ME WB
xor                 IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
and                    IF ID EX ME WB
slt                    IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
```


Staple This Side

(*b*) Show the execution of the code below on the following 4-way superscalar MIPS implementation. As we have been doing this semester, instruction fetches are not aligned (the address can be any multiple of 4).



☐ Show execution on this ☐ **4-way** superscalar MIPS implementation, with ☐ the **branch taken**. ☐ Show all squashes with an x. ☐ Check for dependencies and ☐ pay attention to branch behavior.

```
 beq r2, r3 SKIP

 addi r1, r1, 8

 add r2, r2, r5

SKIP:
 sw r2, -4(r1)

 add r1, r1, r5

 lw r2, 0(r1)

 xori r2, r2, 0xaa

 slt r8, r1, r9
```

Problem 3: (25 pts) When the modifications to the MIPS implementation on the facing page are complete an `add.s` instruction will not have to stall to avoid a structural hazard with preceding `mul.s` instructions. Here an `add.s` instruction can pass through the same number of stages as a `mul.s`, so there is no possibility of a stall due to a structural hazard at `WF`. In the execution below `add.s f14` avoids such structural hazard stalls by passing through the two extra stages, `a5` and `a6`. But to avoid the necessity of *always* having to pass through those two extra stages an `add.s` can use *hop* multiplexors to skip ahead to `WF` early. The `add.s f10` skips over two stages, and `add.s f7` skips over one stage.

To keep the problem description from getting too long the following interesting material was not included in the original final exam. Hopping ahead this way is probably not the best way to deal with structural hazards, even if the avoided structural hazard stalls justified the cost of the pipeline latches. The reason is that those two new hop multiplexors could instead be used to implement bypass paths from `a5` *and* `a6` *into the* `M1` *and* `A1` *functional units. (See the Fall 2006 final exam.) With such bypasses possible there would no longer be a need to write back early. Hmmm, this may turn into a Fall 2025 question.*

New and important hardware is shown in blue. The hardware generating signals `xw`, `we`, and `fd` is from the old design and needs to be modified. Hardware for `lwc1` has been removed, it is not part of this problem. When solved correctly code should execute as shown below:

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
mul.s f1, f2, f3   IF ID M1 M2 M3 M4 M5 M6 WF
mul.s f4, f5, f6      IF ID M1 M2 M3 M4 M5 M6 WF
add.s f14, f15, f16      IF ID A1 A2 A3 A4 a5 a6 WF
sub r1, r2, r3              IF ID EX ME WB
add.s f7, f8, f9               IF ID A1 A2 A3 A4 a5 WF
or r4, r5, r6                     IF ID EX ME WB
add.s f10, f11, f12                  IF ID A1 A2 A3 A4 WF
a4/a5.wa                                  1     1     1
a5/a6.wa                                     1     1     0
a6/WF.wa                                        1     0     0
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

In cycle 11 the `add.s f7` uses the lower input of Hop6 to hop from `a6` to `WF` (which is why `a6` is not shown). In cycle 12 `add.s f10` uses the upper input of Hop6 to hop from `a5` to `WF`.

☐ Connect logic to the inputs of AND gates W5 and W6 so that the `wa` (write add) signal will be `0` for an instruction that hopped out of a stage, preventing a hopping `add.s` from writing back twice. See sample `wa` values in the execution above.

☐ Add select signals to the two hop multiplexors. ☐ *Little or no logic is required. For partial credit assume* `wa` *is correct.*

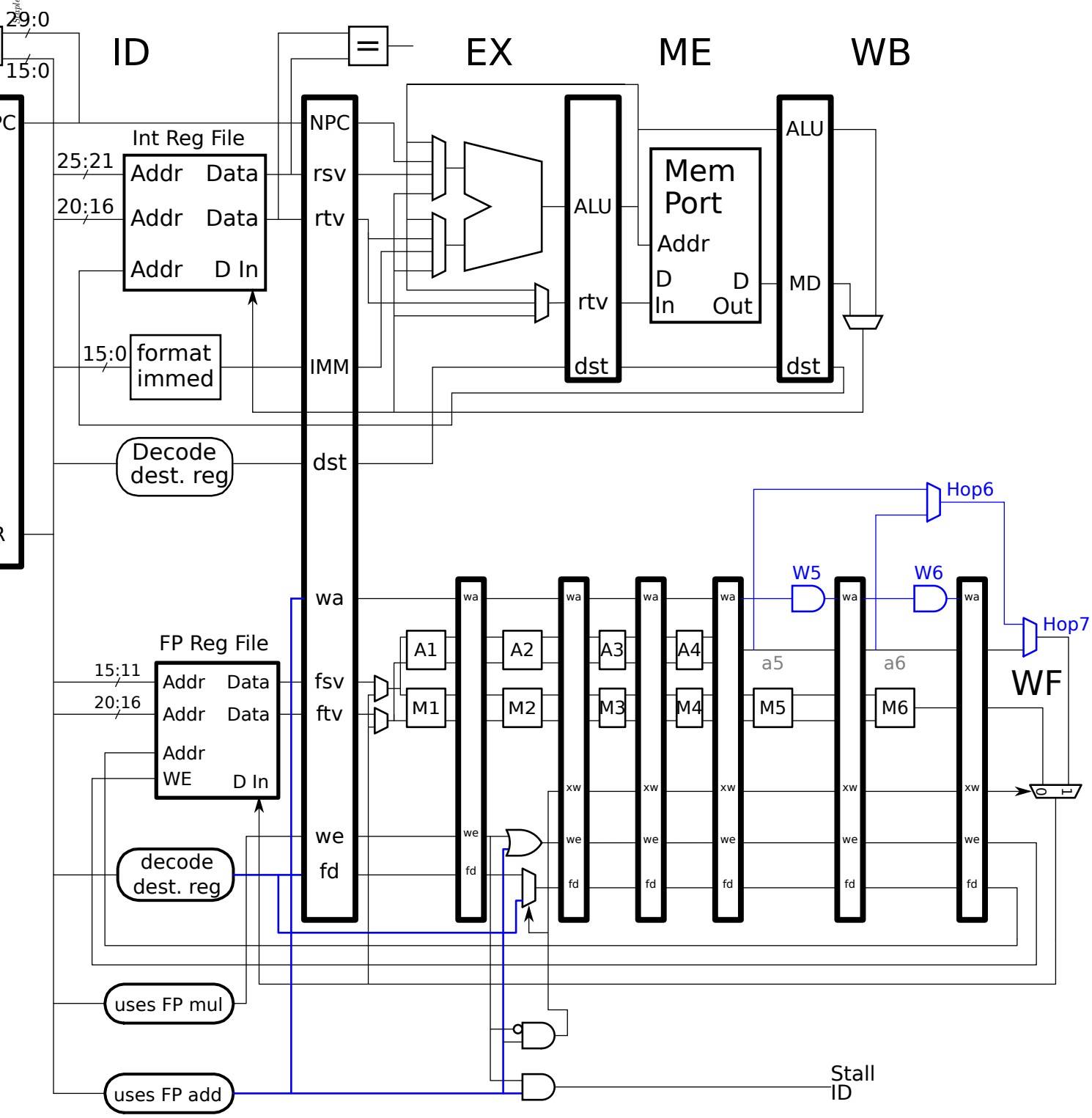
☐ Modify the design so that `fd` is correct for ☐ `mul.s` and ☐ the hopping `add.s` instructions.

☐ Modify the `xw` logic so that it works for hopping `add.s` instructions and the `mul.s`.

☐ Modify `we` so that it is correct for hopping `add.s` and `mul.s` instructions.

☐ Remove logic that is no longer needed. That can include logic for `xw` or `we`, depending on the solution.

The IF stage is not shown to make space.



Problem 4: (12 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor, and the other uses a 3-outcome local history predictor.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  N  N  T  T  T  N  T    N  N  T  T  T  N  T     N  N  T  T  T  N  T   <- Outcome

B2:   N  T  N  T  N  T  N    T  N  T  N  T  N  T     N  T  N  T  N  T  N  <- Outcome
```

☐ What is the accuracy of the bimodal predictor on branch `B1`? ☐ Be sure to base the accuracy on a repeating pattern.

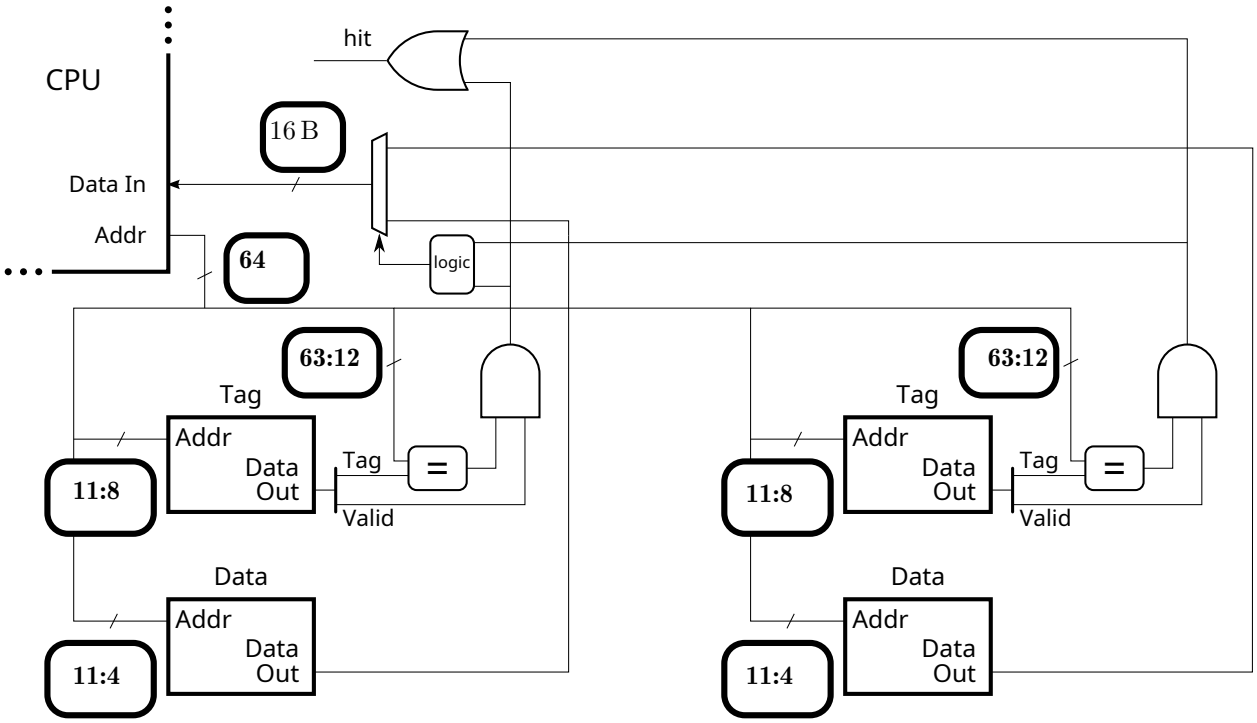
☐ What is the accuracy of 3-outcome local history predictor on B1 ignoring B2.

☐ What is the accuracy of 3-outcome local history predictor on B1 **taking into account** B2. ☐ Note that the B2 pattern repeats faster than the B1 pattern.

Staple This Side

Problem 5: (8 pts) The diagram below is for a two-way set associative cache.
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: 

0

☐ Cache Capacity, in Bytes (how much data can it cache).

☐ Line Size ☐ Indicate Unit!!:

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 bytes (characters). The code fragment starts with the cache cold (empty); consider only accesses to the array. Of course, $2^6 = 64$.

(*b*) Find the hit ratio executing the code below.

```
int64_t sum = 0;
int64_t *a = 0x2000000; // sizeof(int64_t) == 8
int ILIMIT = 1 << 14;    // = 2^14

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Staple This Side

Problem 6: (20 pts) Answer each question below.

(*a*) For each item below provide a code fragment exhibiting the indicated dependency. For your solving convenience one instruction is already shown.

☐ Complete the code fragment so that it exhibits a true dependence. ☐ Circle the register carrying the dependence.

```
add  r1,  r2,  r3
```

☐ Complete the code fragment so that it exhibits an output dependence. ☐ Circle the register carrying the dependence.

```
add  r1,  r2,  r3
```

☐ Complete the code fragment so that it exhibits an anti-dependence. ☐ Circle the register carrying the dependence.

```
add  r1,  r2,  r3
```

(*b*) Show the encoding of the instructions below. For the FP instruction infer the encoding from the implementation diagrams in other problems.

☐ Show encoding. ☐ Label fields, and show specific values where possible.

```
add r1, r2, r3
```

☐ Show encoding. ☐ Label fields, and show specific values where possible.

```
sw r4, 5(r6)
```

☐ Show encoding. ☐ Label fields, and show specific values where possible.

```
add.s f1, f2, f3
```

(*c*) Based on the execution below, why can't the exception raised by the `mul.s` instruction be precise? What can't the handler do after it returns that could be done if the exception were precise?

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10
mul.s f4, f5, f6  IF ID M1 M2 M3 M4 M5*M6*WF
addi r6, r6, 1       IF ID EX ME WB

Handler:
sw r1, 4(fp)                              IF ID ..
```

☐ `mul.s` exception can't be precise because:

☐ If the exception were precise the handler could:

(*d*) Many early RISC ISAs avoided branch instructions that compared registers, such as `blt r1, r2, TARG`, (branch less than) because that would lower the clock frequency. Later ISAs have included them. Assume that the time needed to compute `r1 < r2` has not changed.

☐ Explain why newer ISAs can have instructions like `blt r1, r2, TARG` without slowing the clock frequency, given that comparison is no faster? ☐ In your answer indicate where branches might resolve and how penalty is avoided.

(*e*) A design team is trying to decide between including bypass path A or bypass path B. With the target workload compiled without optimization the implementation with path A is faster. With the workload compiled with optimization the implementation with path B is faster.

☐ Which should be used ◯ *bypass path A* or ◯ *bypass path B.* ☐ Explain.

Staple This Side

This page intentionally left blank.

Staple This Side

3 Spring 2023

Name _____

Staple This Side

Computer Architecture
LSU EE 4720
Midterm Examination

Wednesday, 29 March 2023 9:30-10:20 CDT

Problem 1	_____	(17 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(16 pts)
Problem 4	_____	(16 pts)
Problem 5	_____	(16 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

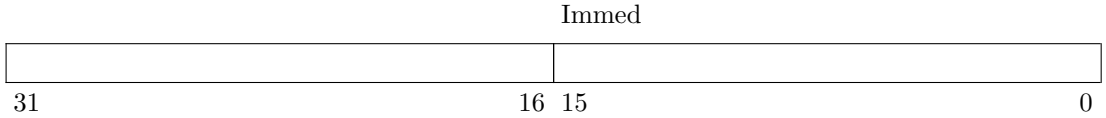
Problem 1: [17 pts] Candidate MIPS instruction `subir r1, 22, r3` is to compute `r1=22-r3`, which can't be done with a single existing MIPS instruction. The `22` is taken from instruction bits 15:0, which is the immediate field of Type-I instructions.

The `subir` instruction is to be encoded so that it can be executed by the implementation to the right **with the ALU computing** $X = A - B$, the same operation used by existing subtract instructions. Notice that in the implementation the **immediate connects to both** ALU inputs.

(*a*) Show how `subir r1, 22, r3` instruction would be encoded for this hardware.

☐ Show encoding of `subir r1, 22, r3`. Be sure to show ☐ the position of the fields and ☐ the field values for the sample instruction.

☐ Be sure that the encoding fits with the illustrated hardware and other MIPS instructions.



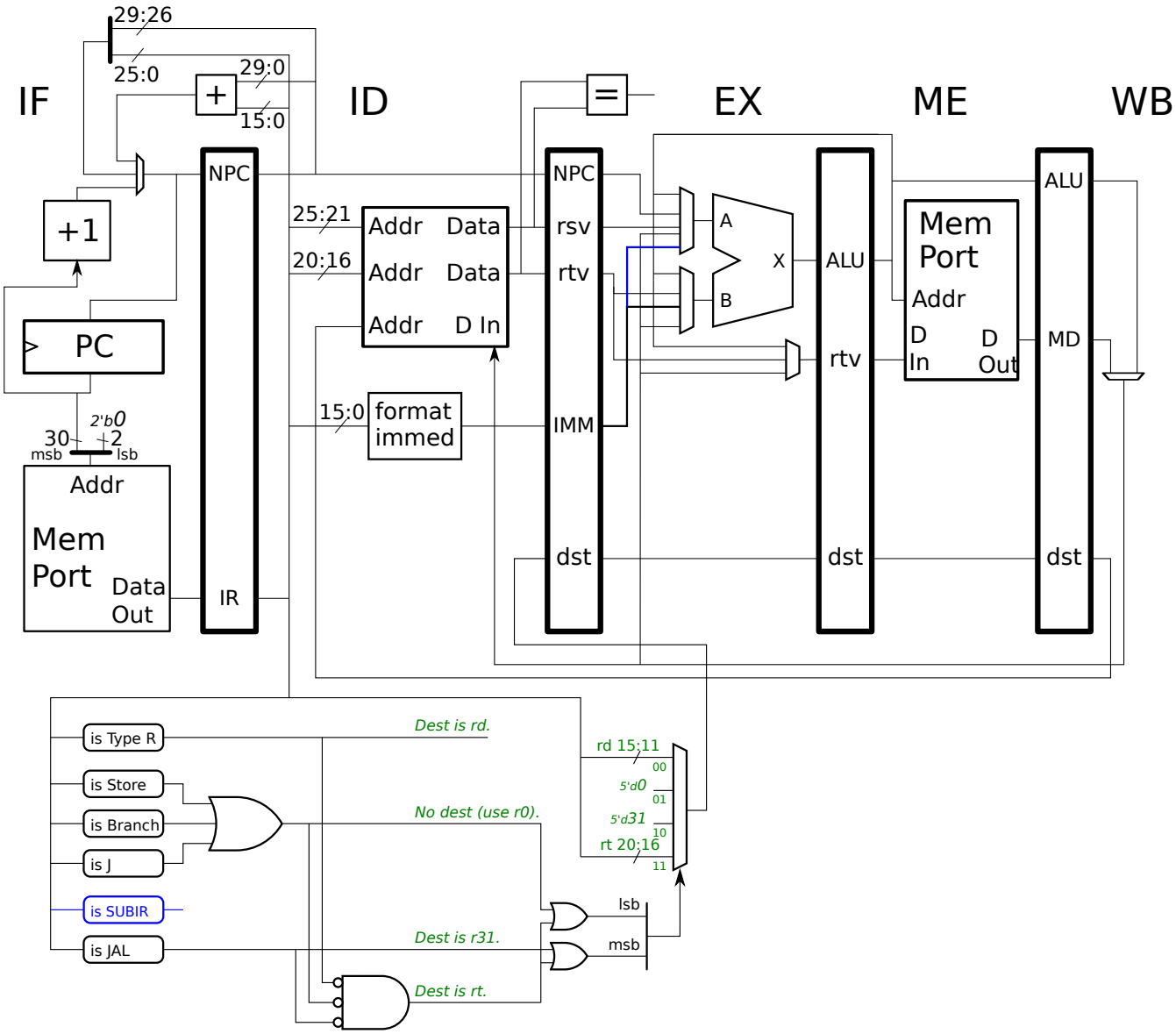
(*b*) Some control logic is shown for the implementation.

☐ Modify the control logic that computes `dst` so that `subir` executes correctly. ☐ **Do not** design control logic for the ALU multiplexors.

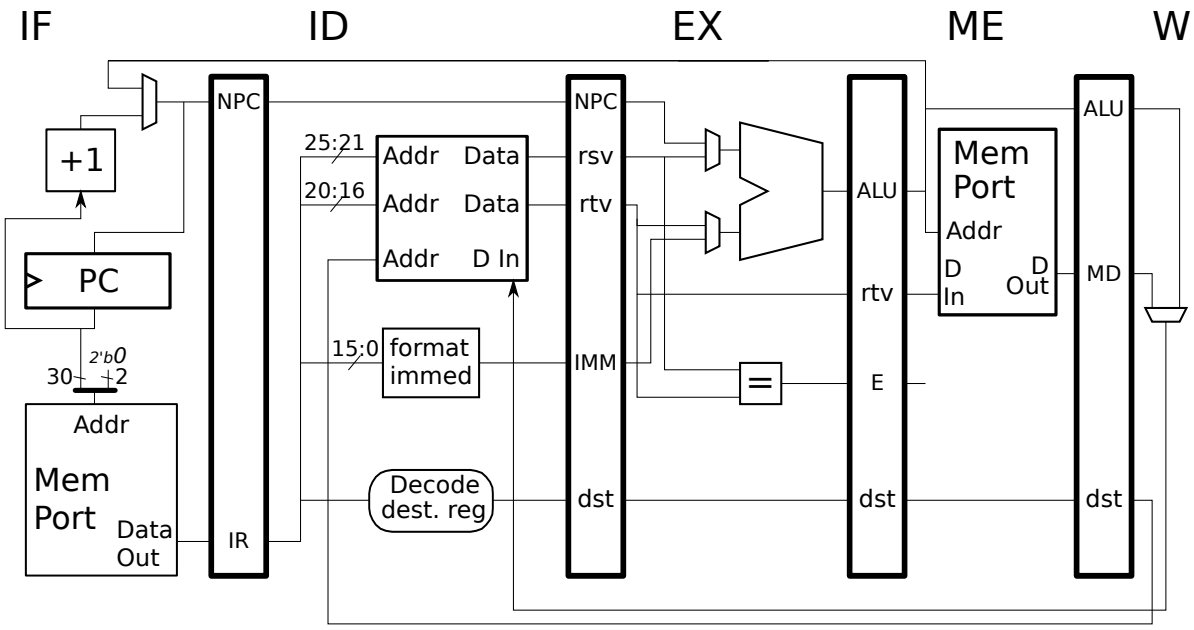
☐ The control logic should not break existing instructions.

☐ The control logic changes should be consistent with your answer to the previous part.

Staple This Side



Problem 2: [20 pts] Show the execution of the code fragments below on their accompanying MIPS implementations.



☐ Show the execution of the code fragment below on ☐ the implementation above. ☐ Be sure to check for dependencies.

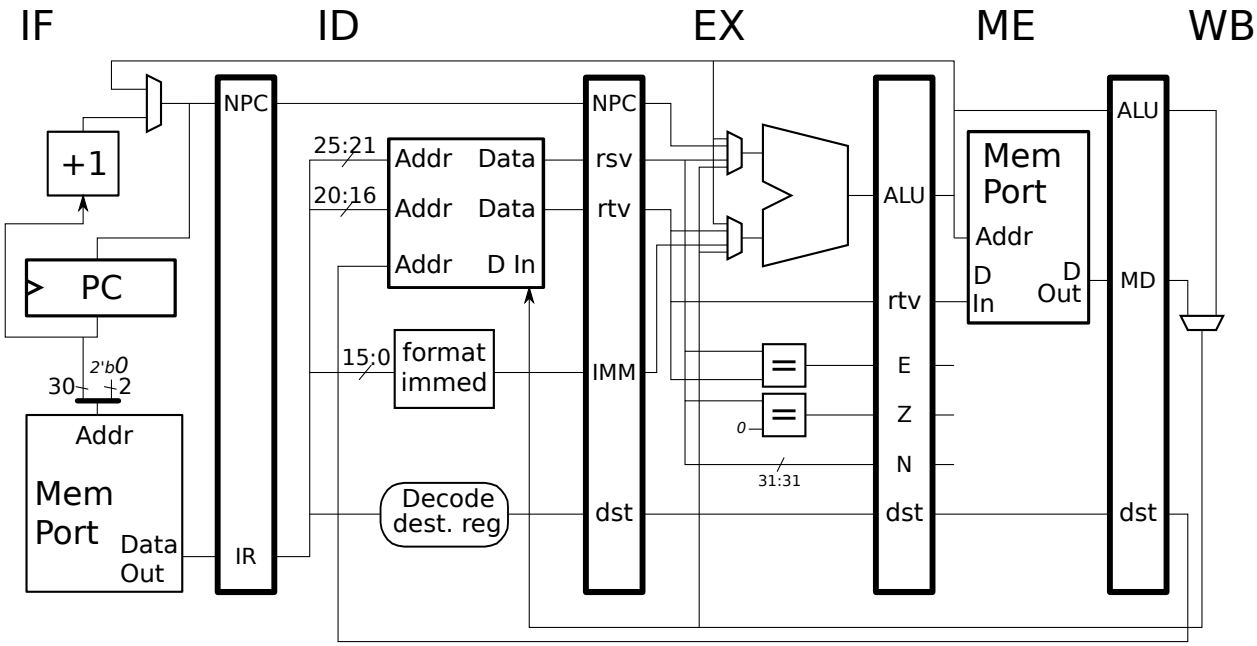
```
addi r1, r2, 4
lw   r3, 0(r1)
sw   r1, 4(r3)
```

☐ Show the execution of the code fragment below on ☐ the implementation above. ☐ Be sure to check for dependencies.

```
addi r1, r2, 4
sw   r1, 4(r3)
lw   r3, 0(r1)
```


Staple This Side

Problem 2, continued:



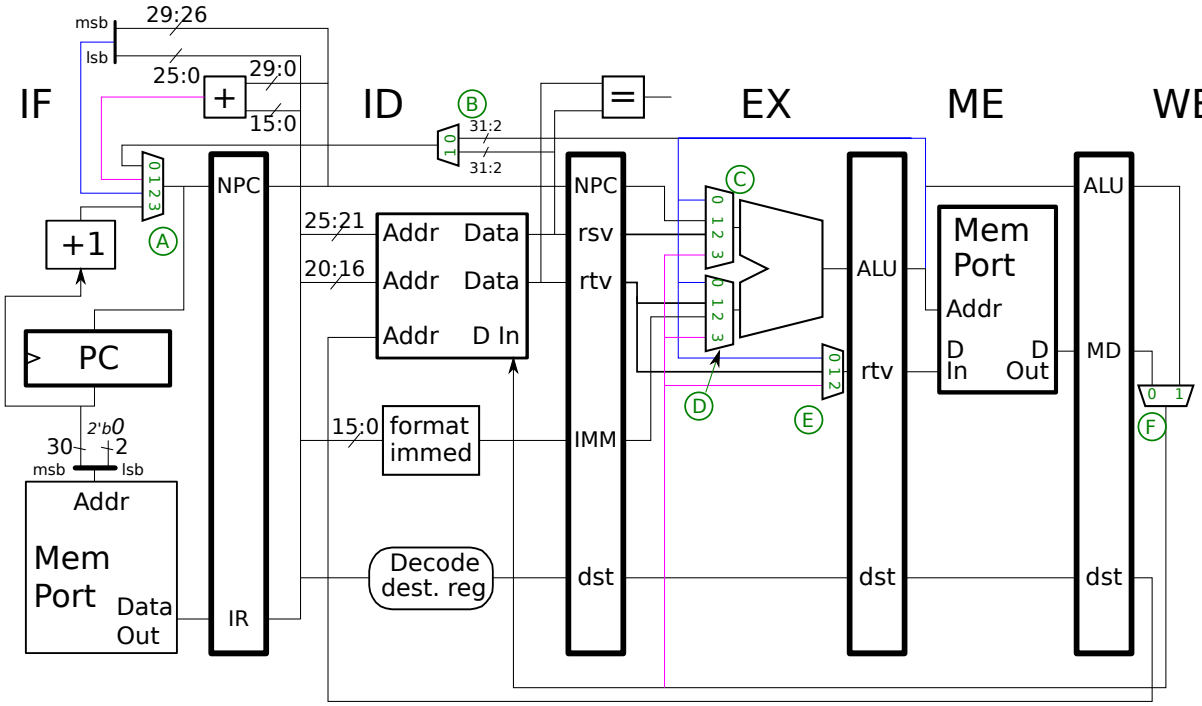
☐ Show the execution of the code fragment below on ☐ **the implementation above**. ☐ Be sure to check for dependencies.

```
addi r1, r2, 4
lw   r3, 0(r1)
sw   r1, 4(r3)
```

☐ Show the execution of the code fragment below on ☐ the implementation above. ☐ Be sure to check for dependencies.

```
addi r1, r2, 4
sw   r1, 4(r3)
lw   r3, 0(r1)
```

Problem 3: [16 pts] Appearing below is the MIPS implementation with labeled multiplexor select signals from Homework 3. Following that is an execution diagram along with a row showing select signal values for the D multiplexor. The first instruction, `add`, is shown.



☐ Complete the code fragment so that it produces the values shown for D.

# Cycle	0	1	2	3	4	5	6	7	8
add r1, r2, r3	IF	ID	EX	ME	WB				
		IF	ID	EX	ME	WB			
			IF	ID	EX	ME	WB		
				IF	ID	EX	ME	WB	
					IF	ID	EX	ME	WB

# Cycle	0	1	2	3	4	5	6	7	8
D:			1	0	1	2	3		

Staple This Side

Problem 4: [16 pts] Rewrite each code fragment below so that it uses fewer instructions.

☐ Simplify code fragment.

```
addi r1, r0, 123
add  r1, r1, r2
```

☐ Simplify code fragment.

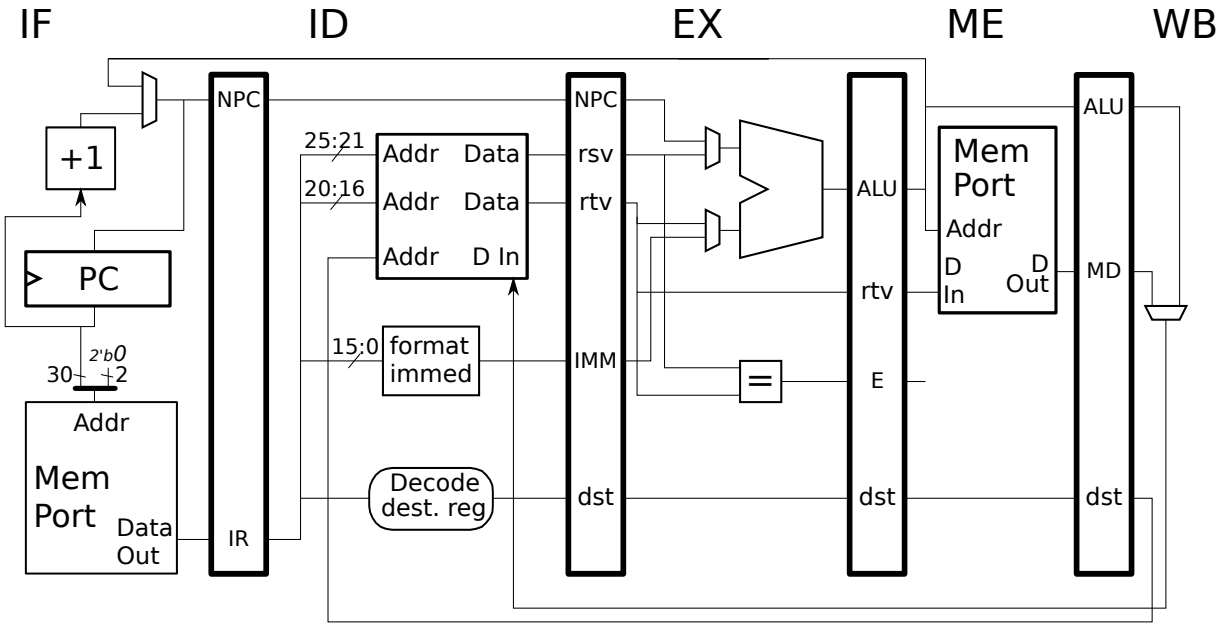
```
lw r1, 0(r2)
addi r2, r2, 4
lw r2, 0(r2)
```

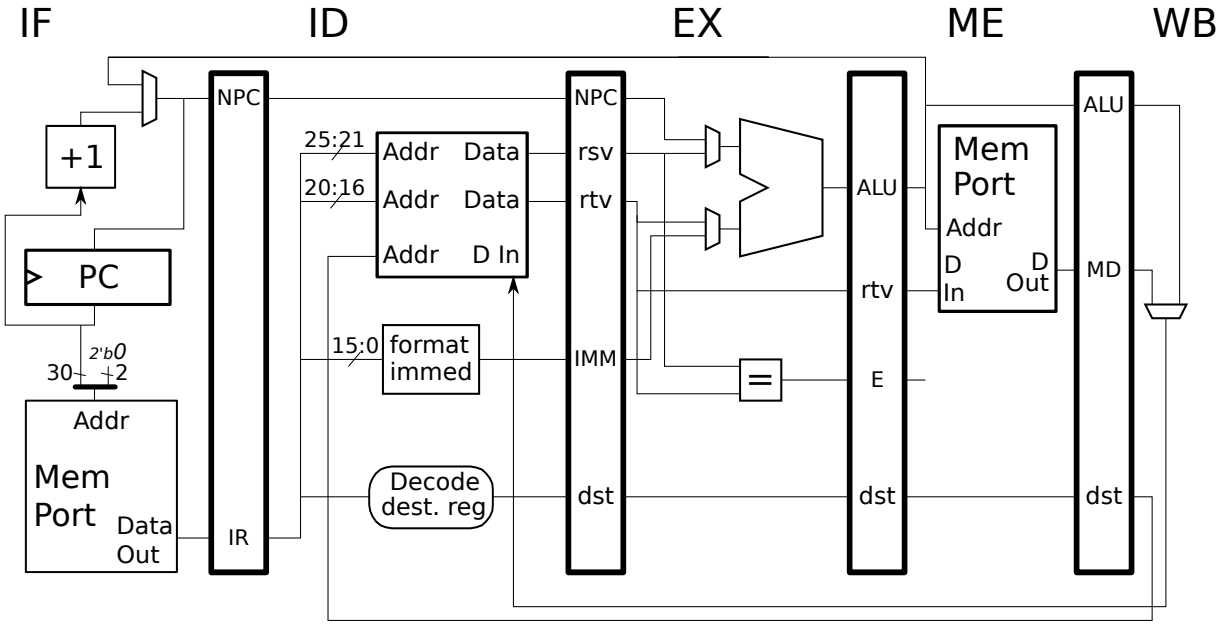
☐ Simplify code fragment.

```
sub r1, r2, r3
beq r1, r0,  TARG
lw r1, 0(r4)
```

Problem 5: [16 pts] Appearing below are two identical illustrations of one of our MIPS implementations. To the right are three executions of a code fragment, only one of which is possible on the implementation.

Identify the execution that is possible. For each of the executions that is not possible modify one of the illustrations below so that it is. The modification is very simple, just consider the target address. A few well chosen lines will suffice. No logic gates.





☐ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ◯ *No*.

☐ If not, modify the implementation so that it is and ☐ label your modifications *A*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IFx
 lui r5, 0x1234
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10              IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
```

☐ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ◯ *No*.

☐ If not, modify the implementation so that it is and ☐ label your modifications *B*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF IDx
 lui r5, 0x1234             IFx
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10                 IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
```

☐ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ◯ *No*.

☐ If not, modify the implementation so that it is and ☐ label your modifications *C*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF ID EXx
 lui r5, 0x1234             IF IDx
 ori r5, r5, 0x6789            IFx
TARG:
 xor r8,r9,r10                    IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
```

Problem 6: [15 pts] Answer each question below.

(*a*) Company *A* and *B* both come out with a new computer each year. Company *A* changes both the ISA and implementation each year. Company *B* changes only the implementation each year but uses the same ISA.

☐ Which company is following accepted practice?

☐ Which company's customers are more likely to stay with the company when it is time to upgrade to a new model? ☐ Explain.

(*b*) In MIPS `nop` is a pseudo instruction.

☐ What is a pseudo instruction?

☐ Does having too many pseudo instructions make implementations too expensive? ☐ Explain.

(*c*) The first code fragment below, from code presented in the course, loads element `i` of an array of integers. (Here integers are four bytes.) Complete the second code fragment so that it loads element `i` from an array of shorts (A short is two bytes.).

```
# C CODE                        #  ASM REGISTER = C VARIABLE NAME
# int *a; ...          # $s1 = a;  $t0 = i    sizeof(int) = 4 chars.
# x = a[i];

sll $t5, $t0, 2       # $t5 -> i * 4;  Each element is four characters.
add $t5, $s1, $t5     # $t5 -> &a[i]  (Address of a[i].)
lw $t1, 0($t5)        # x = a[i];   $t1 -> a[i]
```

☐ Complete code below so that it loads a short.

```
# C CODE                        #  ASM REGISTER = C VARIABLE NAME
# short *a; ...          # $s1 = a;  $t0 = i    sizeof(short) = 2 chars.
# x = a[i];
```

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Monday, 8 May 2023 10:00-12:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

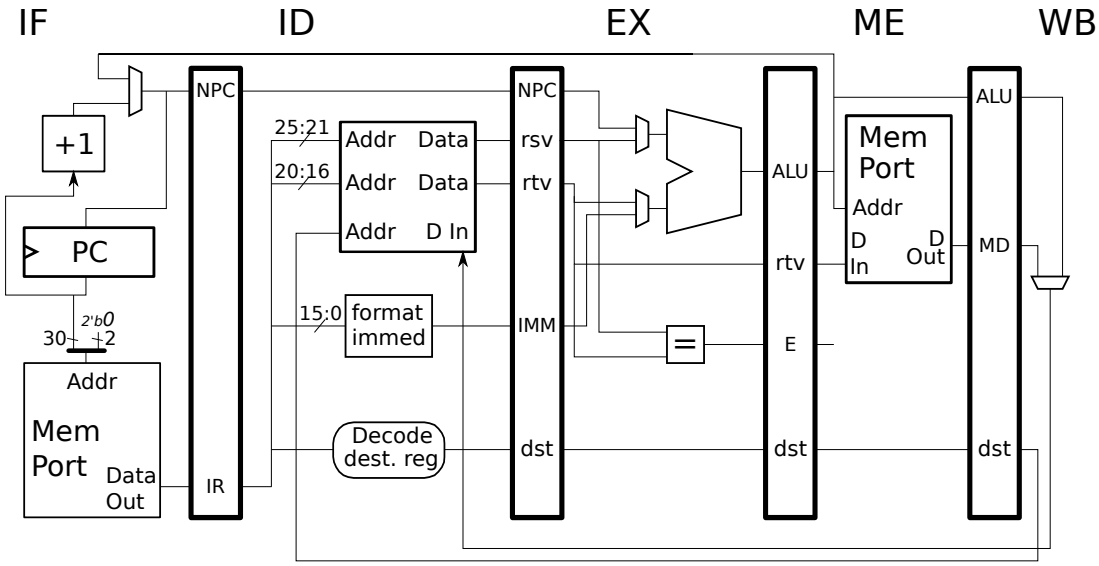
Alias _____

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

Problem 1: (25 pts) Show the execution of the code fragments on the following implementations. In each case the branch is taken.

(*a*) Show the execution on this basic MIPS implementation.



☐ Show execution for the case where the branch is taken. ☐ Check for dependencies. ☐ Base execution on hardware shown. ☐ Pay close attention to branch behavior.

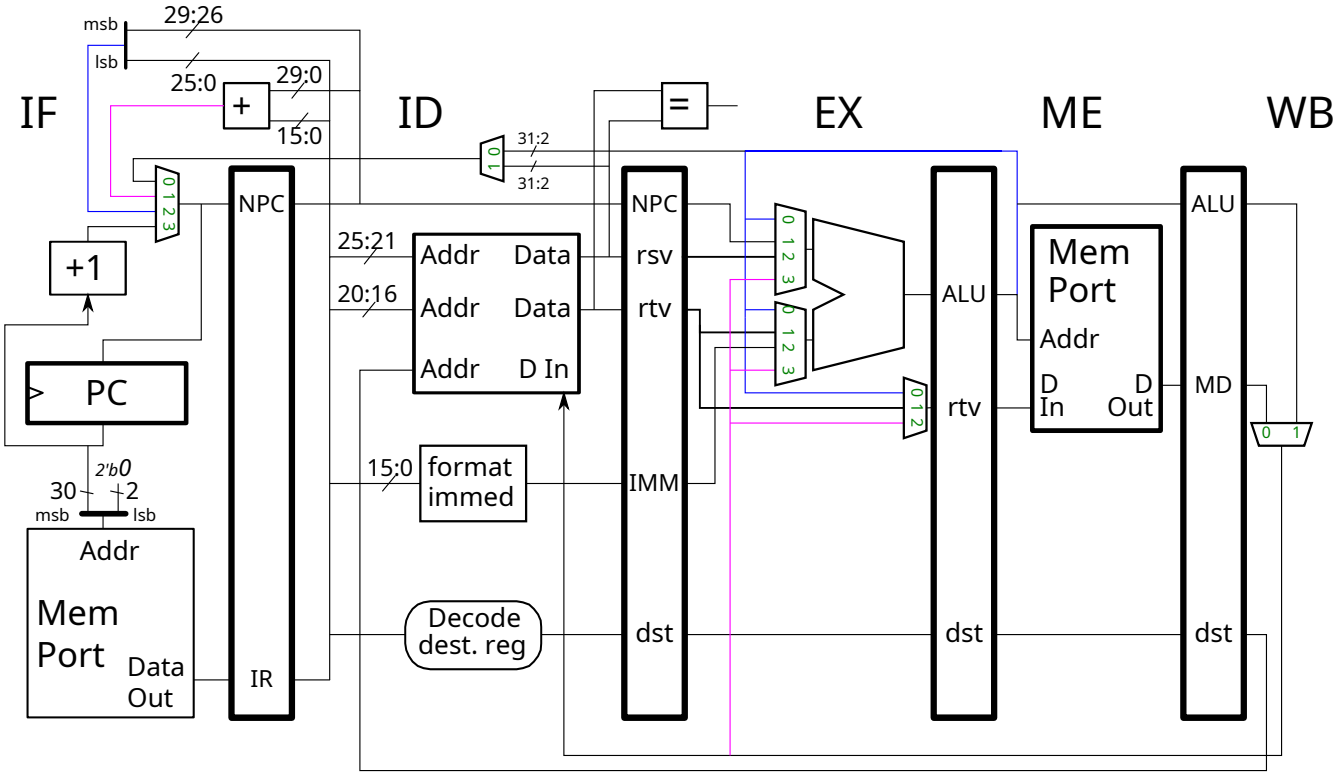
```
 addi r6, r6, 1

 lui r5, 0xf00d

 lw r3, 0x81b4(r5)

 bne r1, r2, TARG

 or r8, r3, r6

 sw r8, 0x8120(r5)

 lw r3, 0x8200(r5)

 addi r5, r5, 16

TARG:
 sll r10, r3, 8

 add r11, r10, r12
```


Staple This Side

This page left blank to provide extra space for the solution.

Staple This Side

(*b*) Appearing below is a MIPS implementation and an **incorrect** execution of a code fragment on that implementation. The code executes more slowly than it would on the implementation. Modify **the implementation** so that the execution is correct. Your modifications will reduce the cost of the implementation.

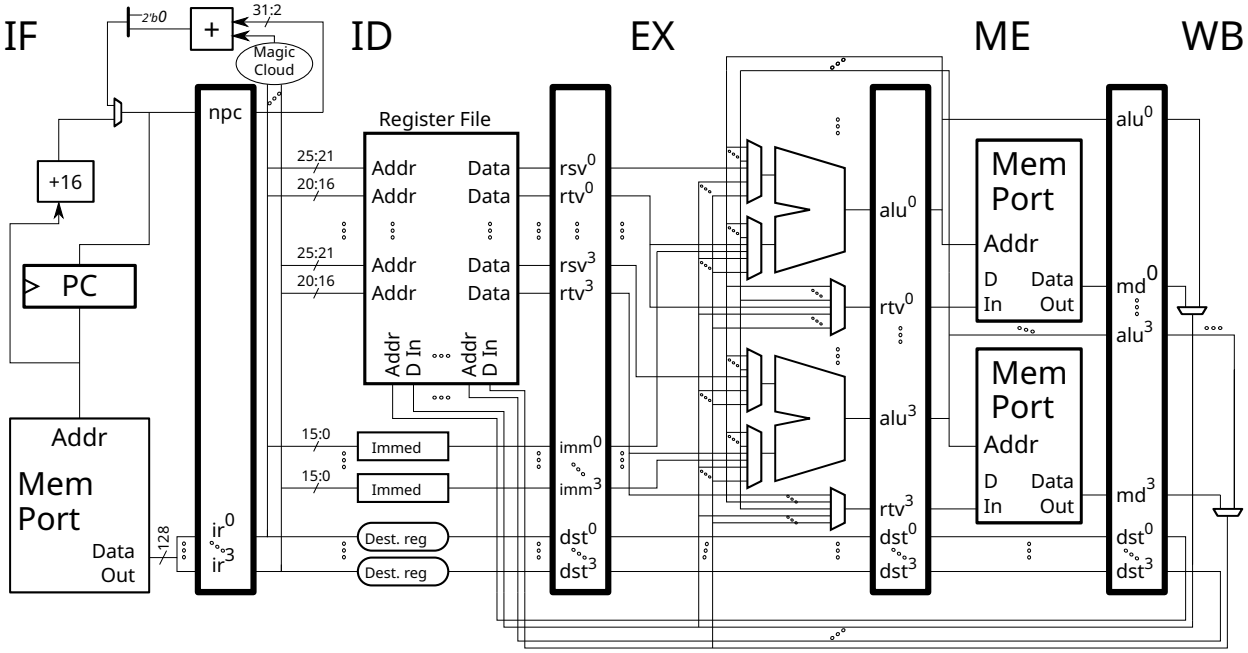


☐ Modify the implementation above so that the code below executes as shown.

☐ **Make as few** changes as possible. For example, remove a bypass path from a mux input, rather than the entire bypass path.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10
addi r6, r6, 1     IF ID EX ME WB
lui r5, 0xf00d        IF ID EX ME WB
lw r3, 0x81b4(r5)        IF ID -> EX ME WB
bne r1, r2, TARG            IF -> ID EX ME WB
or r8, r6, r3                     IF ID -> EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10
```

(*c*) Appearing below is a **4-way** superscalar MIPS implementation.

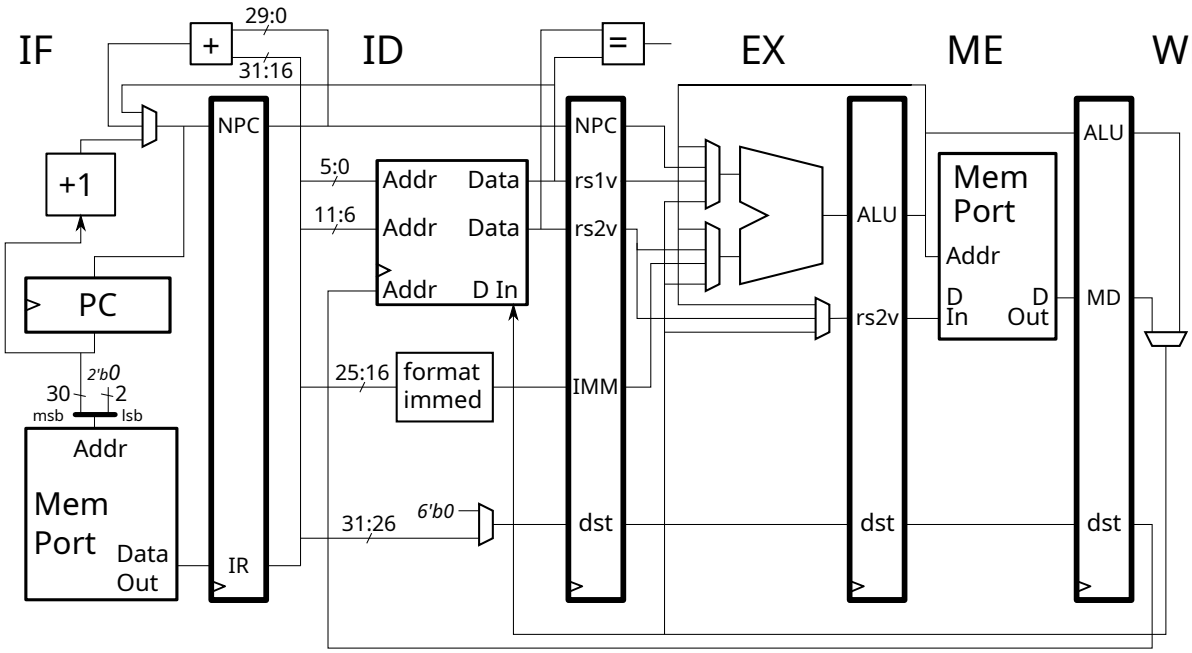


☐ Show execution on the 4-way superscalar implementation ☐ with branch taken.

☐ Pay attention to ☐ branch behavior and ☐ the order of instructions within a stage.

```
addi r6, r6, 1

lui r5, 0xf00d

lw r3, 0x81b4(r5)

bne r1, r2, TARG

or r8, r3, r6

sw r8, 0x8120(r5)
...

TARG:
sll r10, r3, 8

add r11, r10, r12
```

Problem 2: (25 pts) The implementation of *ANRI*, a new RISC ISA, appears below. Many instructions are similar to those of MIPS, though they differ in format and other features. Like MIPS, ANRI registers are named `r0`, `r1`, ...



(*a*) First, an easy question:

☐ How does ANRI differ from MIPS in the ☐ number of registers and ☐ the immediate size?

(*b*) Like MIPS, ANRI's Format R is used for three-register instructions such as `add r1, r2, r3`. Show a possible ANRI Format R consistent with the hardware.

☐ Show the bit positions and the name of each field in ANRI Format R based on reasonable guesses.

☐ Show possible field values for `add r1, r2, r3` ☐ and for `or r1, r2, r3`. (Two instructions for a reason.)

Staple This Side

(*c*) Like MIPS, ANRI's Format I is used for immediate instructions such as `addi r4, r5, 6`. Assume that like MIPS, **each of the dozens of ANRI arithmetic and logical instructions has an immediate variant**.

☐ Show the bit position and name of each instruction field in ANRI Format I based on reasonable guesses and ☐ heeding the bold text above. ☐ Show possible field values for `addi r4, r5, 6` ☐ and for `ori r4, r5, 6`.

(*d*) Consider load and store instructions.

☐ Show encoding of `lw r7, 8(r9)` and ☐ `sw r10, 12(r11)` in ANRI Format I, or something similar. Don't forget to ☐ base the encoding on the hardware.

(*e*) Consider procedure call instructions.

☐ Based on the implementation, why does it appear that ANRI would lack the equivalent of MIPS `jal SomeProcedure` though it could still encode the equivalent of `jalr r1, r2`.

(*f*) Based on the hardware above, ANRI would lack a means of loading an arbitrary 32-bit constant into a register using two instructions. Modify the hardware so that ANRI could encode an instruction like MIPS `lui`, one that could be used to load an arbitrary 32-bit constant into a register using two instructions.

☐ Modify hardware to implement an instruction to help loading a 32-bit constant.

☐ Show the format and encoding of this new instruction. ☐ The format must fit in as much as possible with existing formats.

Problem 3: (20 pts) In the incomplete MIPS implementation to the right the FP multiply unit has its own write port to the FP register file, shown in blue and labeled `WM` in several places. Because of this new `MW` port the `sub.s` instruction in the execution below does not stall, both the `sub.s` and `mul.s` can write back in cycle 8. The control logic has not yet been updated for `MW`.

```
# Cycle           0  1  2  3  4  5  6  7  8  9
mul.s f1, f2, f3  IF ID M1 M2 M3 M4 M5 M6 WM       # Uses MW, the mult-only write port.
add.s f4, f5, f6     IF ID A1 A2 A3 A4 WF
sub.s f7, f8, f9        IF ID A1 A2 A3 A4 WF       # No stall!
lwc1 f10, 0(r11)           IF ID ----> EX ME WF    # Stall due to WF str hazard.
add.s f12, f1, f14            IF ----> ID -> A1... # Stall due to dep with mul.s
```

(*a*) With the illustrated hardware a result cannot be bypassed from a `mul.s` to another instruction. The last `add.s` suffers a stall because of that. Add bypass hardware for such cases.

☐ Add the bypass hardware. ☐ Try to keep cost down by using one mux.

(*b*) Modify the control logic so that it no longer stalls instructions that would write back through `WF` at the same time as a preceding `mul.s`. *Hint: This is just a matter of crossing things out.*

☐ Modify logic to eliminate stalls due to `mul.s` ☐ **but retain stalls** for instructions contending for `WF`, such as `lwc1` in execution above.

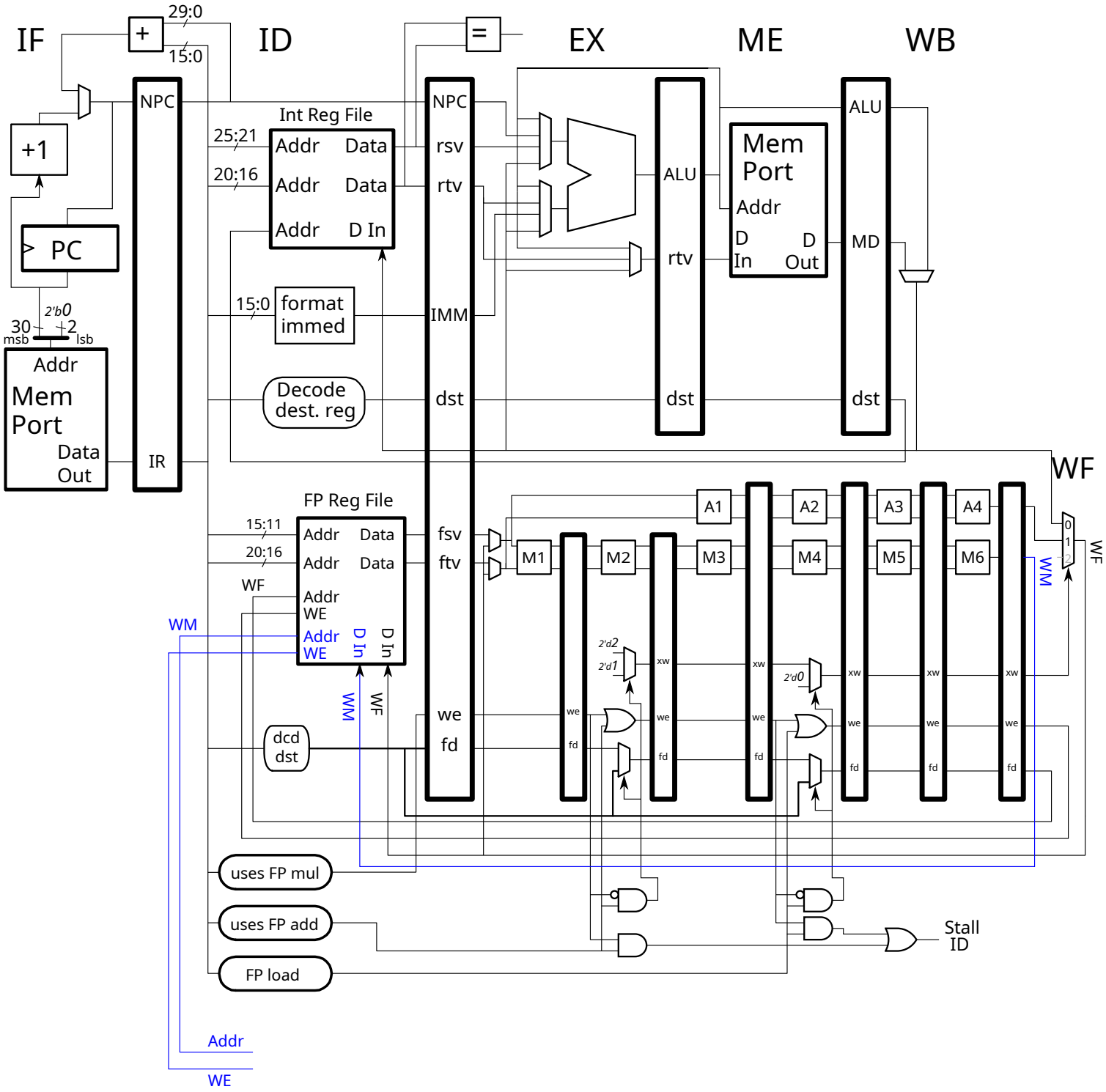
(*c*) Provide the correct `Addr` and `WE` signals to the `WM` and `WF` ports of the FP register file. Note that the `WF` ports are connected, but based on the original version. The `WM` port wires are shown unconnected on the lower-left of the diagram.

☐ Add hardware for the `MW Addr` and `WE` signals, ☐ and make changes to the `WF Addr` and `WE` signals.

☐ Cross out unneeded hardware and ☐ simplify remaining hardware where possible.

Attention perfectionists: Get the SVG source for the hardware at
`https://www.ece.lsu.edu/ee4720/2023/fe-ill-fp-2wr.svg` *and edit it yourself!*

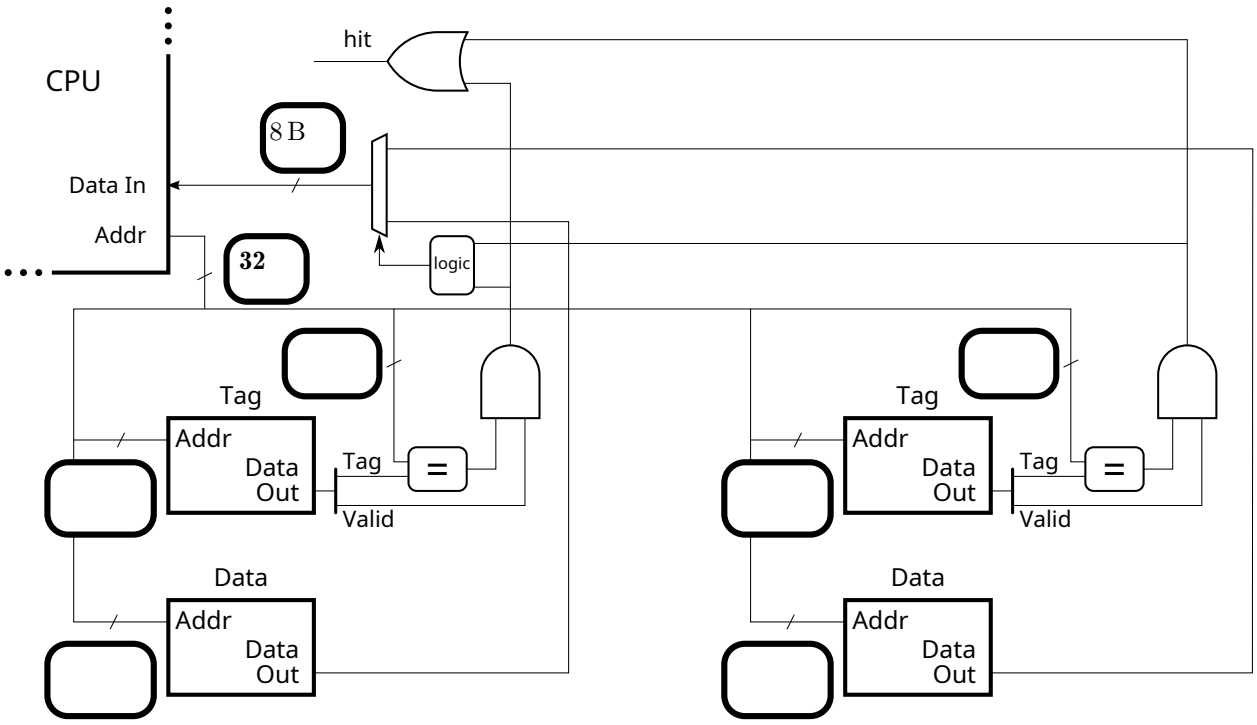
Staple This Side



Problem 4: (10 pts) The diagram below is for a 4 MiB two-way set-associative cache with a line size of 128 B. The character size is the usual 8 bits. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $128 = 2^7$.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: | | | |

0

The code in the problem belows run on a cache with a line size of 128 B (which is 2^7 B). The code fragment starts with the cache cold (empty); consider only accesses to the array.

(*b*) Find the hit ratio executing the code below.

```
float sum = 0;
bfloat16_t *a = 0x2000000; // sizeof(bfloat16_t) == 2
int ILIMIT = 1 << 11;    // = 2^11

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 5: (20 pts) Answer each question below.

(*a*) How does the ARM A64 `fcvtzs` instruction differ from MIPS `trunc.w.s` instruction? These were the instructions used in `sum_thing_unusual` from Homework 5.

☐ The difference between `fcvtzs` and `trunc.w.s` is:

(*b*) With the SPEC CPU benchmarks it is the testers responsibility to compile and run the benchmarks.

☐ A brand-new implementation has many more bypass paths than the old implementation. Why might the results of a tester-compiles test (like SPEC CPU) show better performance on the new implementation than on the old implementation, while with a pre-compiled test the old and new implementations would show the same performance?

(*c*) Appearing below are some hypothetical CISC instructions.

```
 # Some Hypothetical CISC Instructions
I1: add r1, r2, 0x12345678          # r1 = r2 + 0x12345678
I2: add (r1), r2, 0x1234            # Mem[r1] = r2 + 0x1234
I3: add (r1), 0xff04(r2), 0x1234    # Mem[r1] = Mem[r2+0xff04] + 0x1234
I4: add r1, (r2), 8(r3)             # r1 = Mem[r2] + Mem[r3+8]
I5: add (r1), r2, ((r3))            # Mem[r1] = r2 + Mem[ Mem[r3] ]
```

☐ Which of these instructions could not easily be included in an ISA with 32-bit fixed-length instructions? ☐ Explain.

☐ Which of these instructions would be difficult to implement in a pipelined implementation, even if `IF` and `ID` could easily handle variable-length instructions? ☐ Explain.

(*d*) Consider a 4-way superscalar implementation of a conventional ISA, like MIPS, that has **just one memory port** in the `ME` stage. Also consider *Hy4VI*, a hypothetical 4-slot VLIW ISA in which only slot 1 can contain a load or store instruction.

☐ Why might the memory port and related hardware in the `ME` stage of a Hy4VI implementation cost less than that hardware in the `ME` stage in the 4-way RISC implementation?

☐ Why might code written in the conventional ISA enjoy an advantage over code in Hy4VI when running on respective **future** implementations even when the performance of that code is the same on current implementations? The reason given ☐ must have something to do with load and store instructions.

4 Spring 2022

Name _____

Computer Architecture
LSU EE 4720
Midterm Examination

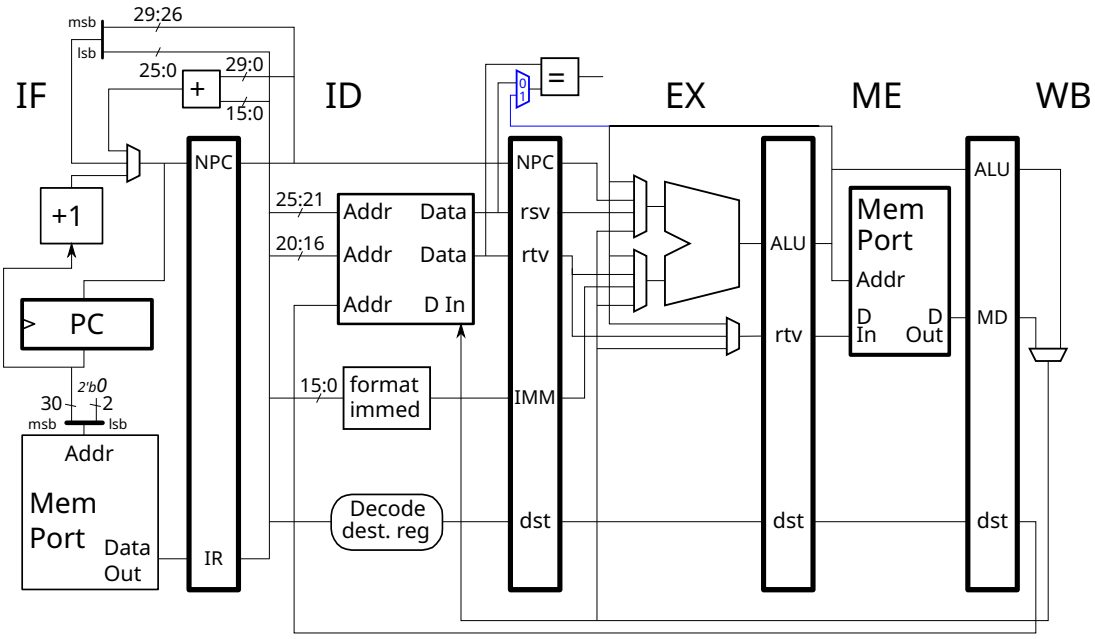
Wednesday, 30 March 2022 9:30-10:20 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The code fragment below is to execute on the illustrated implementation. Show its execution and compute the instruction throughput (IPC) for a large number of iterations. (Note: `sh` is store half.)



☐ Show execution of code below.

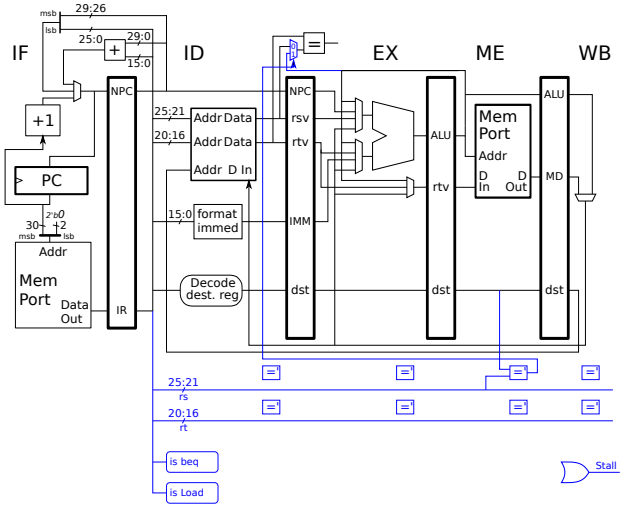
☐ Mark each input to the `rtv` mux (in `EX`) ☐ and by the branch comparison (blue) mux used by the code below.

☐ Compute instruction throughput (IPC) for a large number of iterations.

```
 lw r1, 0(r2)

LOOP:
 addi r2, r2, 4

 sh r1, -2(r2)

 lw r3, -4(r2)

 bne r3, r1, LOOP

 lw r1, 0(r2)
```

Problem 2: [30 pts] Appearing below (and larger on the next page) is a MIPS implementation based on the solution to Homework 4 Problem 2, in which control logic for a branch bypass was designed. The diagram includes a `Stall` signal in the lower right. Add control logic to set the stall signal to `1` when a `beq` needs to stall due to a dependence that can't be bypassed.

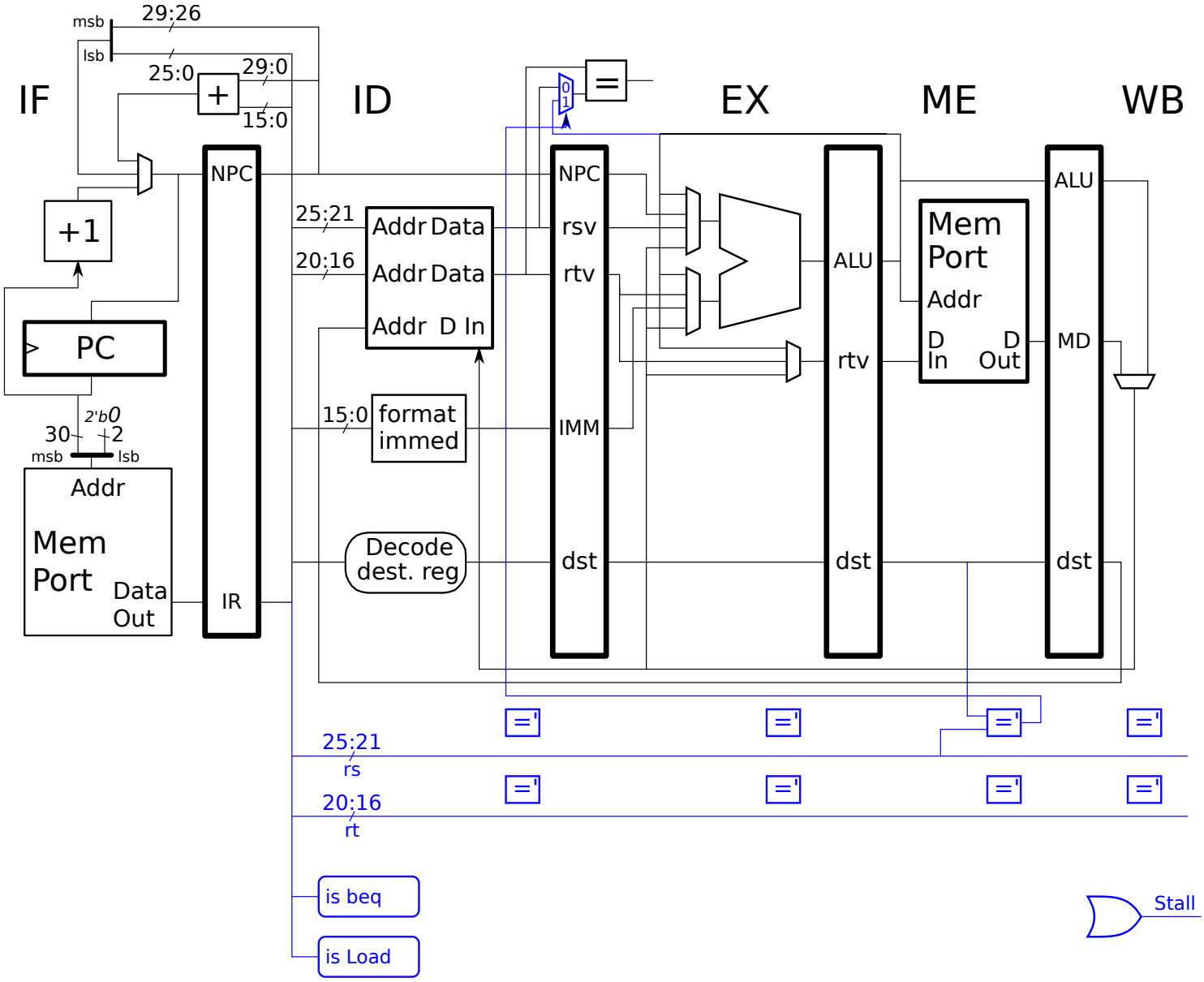
Appearing below are some code fragments. Complete executions are shown for the first two, in the others the executions are incomplete. The control logic should work with these code fragments. It may be helpful to complete the executions.



Use next page for solution.

```
# Cycle              0   1   2   3   4   5   6   7
  addi r1, r2, 3     IF  ID  EX  ME  WB
  beq r1, r4, TARG       IF  ID  ->  EX  ME  WB
  nop                    IF  ->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8
  addi r1, r2, 3     IF  ID  EX  ME  WB
  beq r4, r1, TARG       IF  ID  ----->  EX  ME  WB
  nop                    IF  ----->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8
  lw r1, 0(r2)       IF  ID  EX  ME  WB  # Note: Intentionally incomplete.
  beq r1, r4, TARG       IF  ID
  nop                    IF

# Cycle              0   1   2   3   4   5   6   7   8
  lw r1, 0(r2)       IF  ID  EX  ME  WB  # Note: Intentionally incomplete.
  beq r4, r1, TARG       IF  ID
  nop                    IF

# Cycle              0   1   2   3   4   5   6   7   8
  lw r9, 0(r2)       IF  ID  EX  ME  WB  # Note: Intentionally incomplete.
  beq r1, r4, TARG       IF  ID
  nop                    IF
```

☐ Design control logic to generate the stalls for a `beq`. Show connections to the input of the OR gate on the lower right. ☐ Make sure that the logic handles the cases above and for similar situations. ☐ Use as many or as few comparison units, [=’], as you need.



Problem 3: [40 pts] Answer each question below.

(*a*) The MIPS code below loads, stores, and loads again. The two sets of tables further below show the contents of memory before and after the code executes. Numbers in the table are hexadecimal. The code runs on a big-endian system.

```
 # Initially r2 = 0x1200
LOOP:
 lw r1, 0(r2)
 sb r1, 1(r2)
 lw r3, 0(r2)
 bne r1, r3, LOOP
 addi r2, r2, 4
```

☐ Modify the *After* column so that it shows the contents of memory after the code executes.

Memory Address	Before Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

Memory Address	After Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

☐ Modify **one row** in the *Before* column below so that the code above executes just one iteration.

Memory Address	Before Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

Memory Address	After Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

(*b*) Show the encoding of each MIPS instruction below. (That is, show the layout of the 32 bits in the instruction.) Fill fields with numeric values whenever possible, such as for register numbers and immediate values. For unknown opcodes and func field values show some kind of name.

☐ Show encoding of: `lw r1, 2(r3)`.

☐ Show encoding of: `xor r4, r5, r6`.

☐ Show encoding of: `addi r7, r8, 9`.

(*c*) Arm A32 is a 32-bit ISA, Arm A64 (Aarch64) is a 64-bit ISA.

☐ What does the n in n-bit ISA refer to?

☐ Name an application or kind of device for which a 32-bit ISA has an advantage, and ☐ describe the advantage.

☐ Name an application or kind of device for which a 64-bit ISA is a requirement or a big advantage, and ☐ describe the requirement/advantage.

(*d*) In the statement below the description of how ISAs and implementations are developed is different than how they are typically developed in accepted practice.

> *By finalizing an ISA after its implementation is complete it is assured that the ISA exactly describes the implementation and that the implementation makes the best use of the technology at hand.*

☐ How is this statement of ISA and implementation development different than accepted practice? ☐ What is the disadvantage of the approach described in the statement (ignoring the "technology at hand" part)?

☐ The phrase "makes the best use of the technology at hand" is correct. Explain why accepted practice of ISA and implementation development may not make the best use of technology. *Hint: think about the number of bits in a register.*

(*e*) Answer the following about CISC ISAs.

☐ What feature of CISC ISAs allow them to have large, say 32-bit, immediate values?

☐ Why can't a RISC ISA like MIPS practically have 32-bit immediates?

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Monday, 9 May 2022 10:00-12:00 CDT

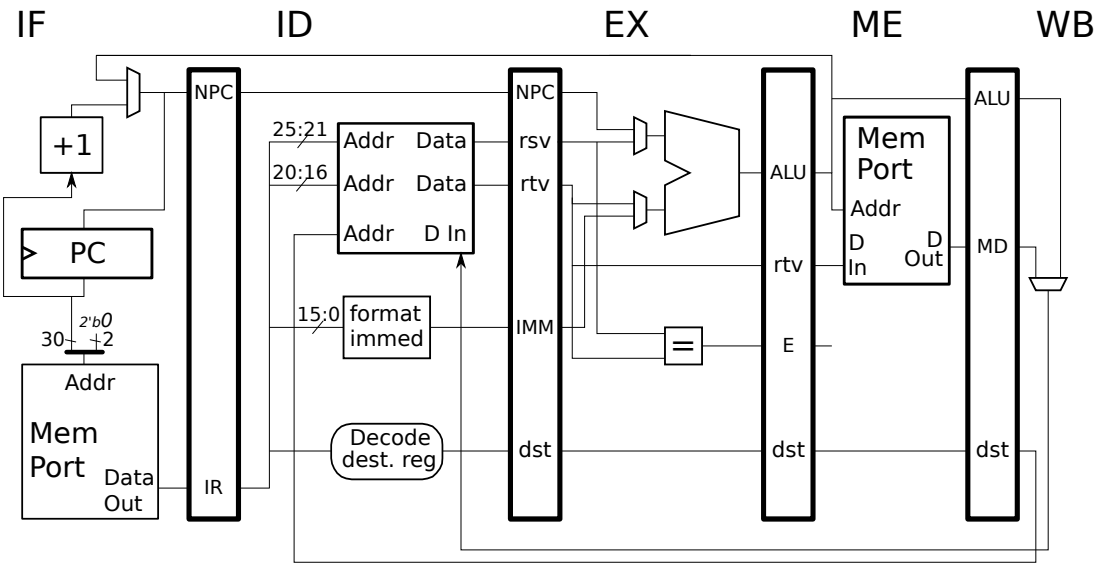
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

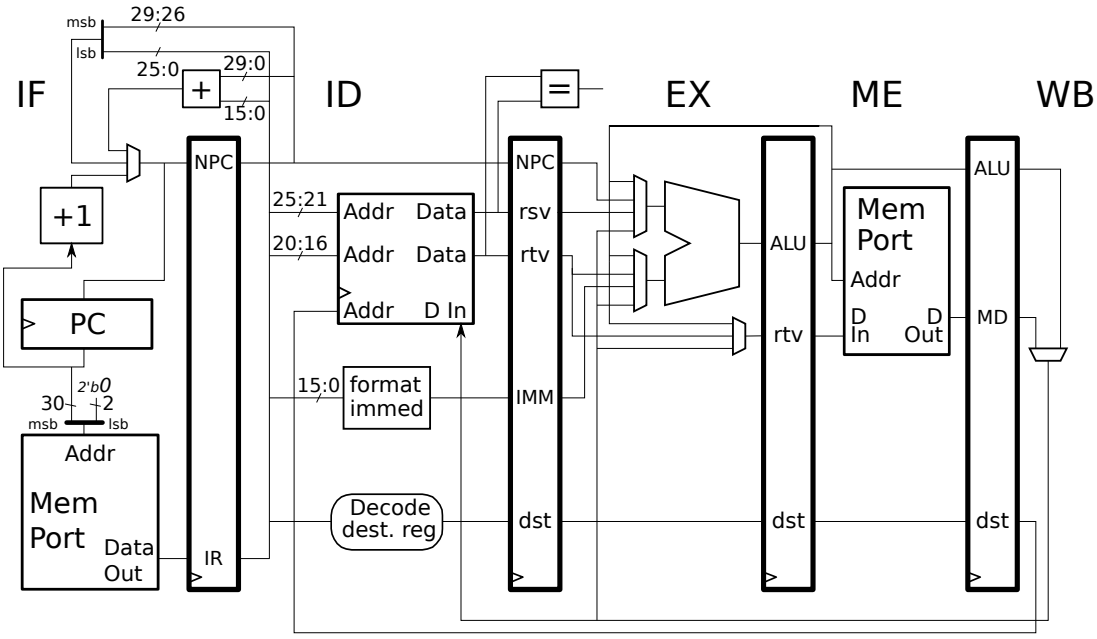
Problem 1: (20 pts) Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). **As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot.** Sorry for yelling, but I hate it when students miss things.



☐ Show execution and ☐ determine instruction throughput (IPC) based on a large number of iterations.

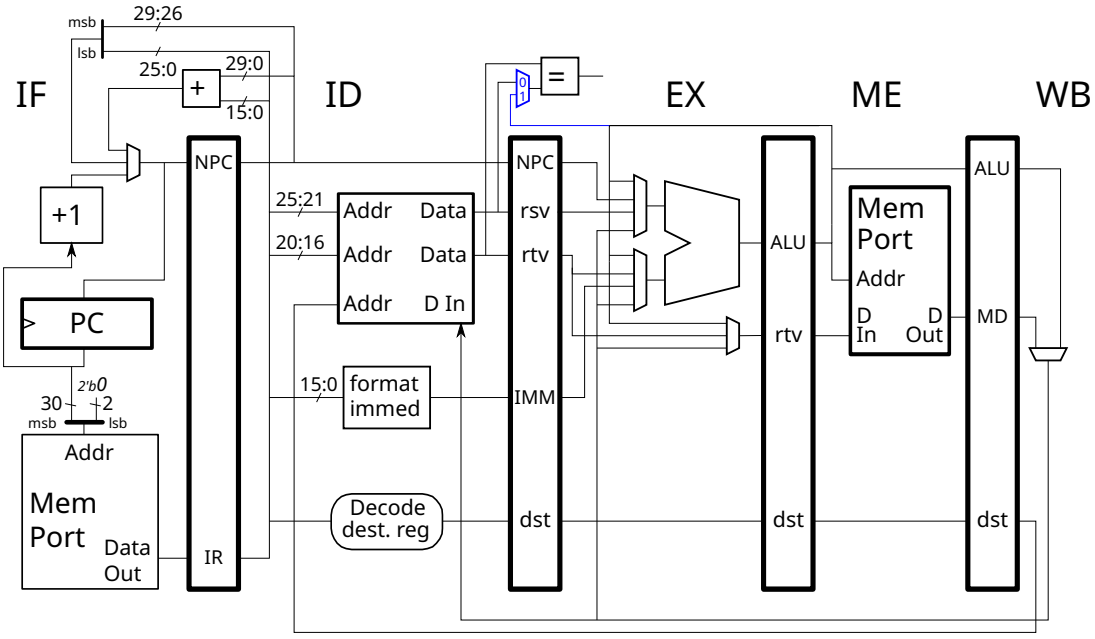
```
LOOP:
 bne r1, r2, LOOP

 addi r1, r1, 4

 xor r5, r6, r7

 sub r8, r9, r10
```


Staple This Side



☐ Show execution and ☐ determine instruction throughput (IPC) based on a large number of iterations.

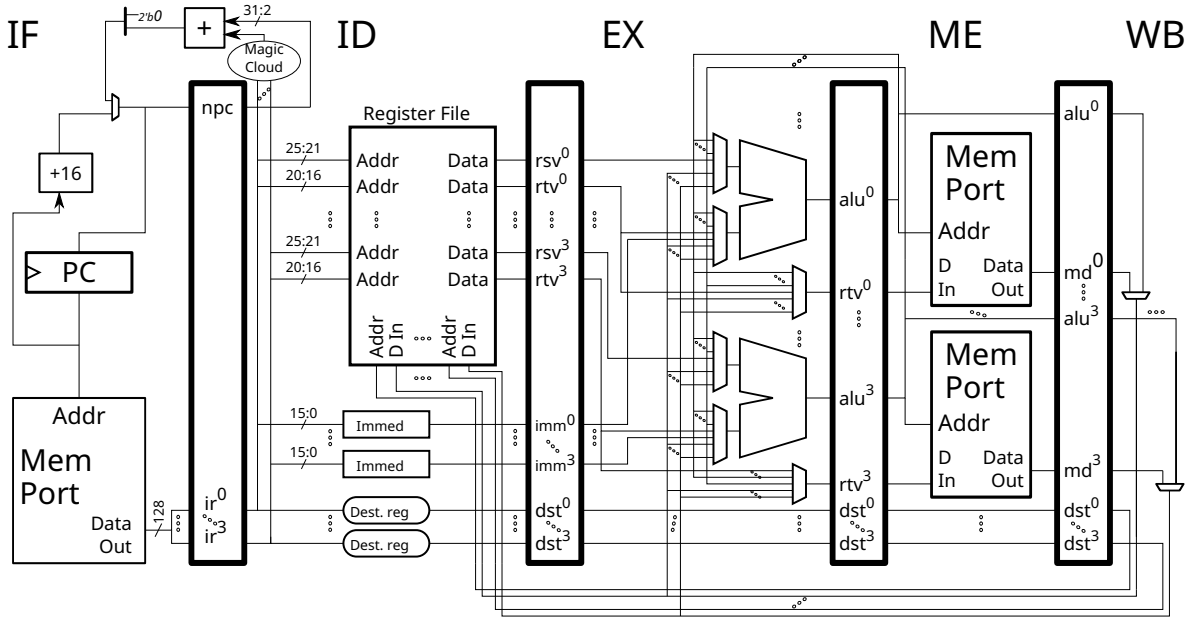
```
LOOP:
 bne r1, r2, LOOP

 addi r1, r1, 4

 xor r5, r6, r7

 sub r8, r9, r10
```



☐ Show execution and ☐ determine instruction throughput (IPC) based on a large number of iterations.

```
LOOP:
 bne r1, r2, LOOP

 addi r1, r1, 4

 xor r5, r6, r7

 sub r8, r9, r10
```

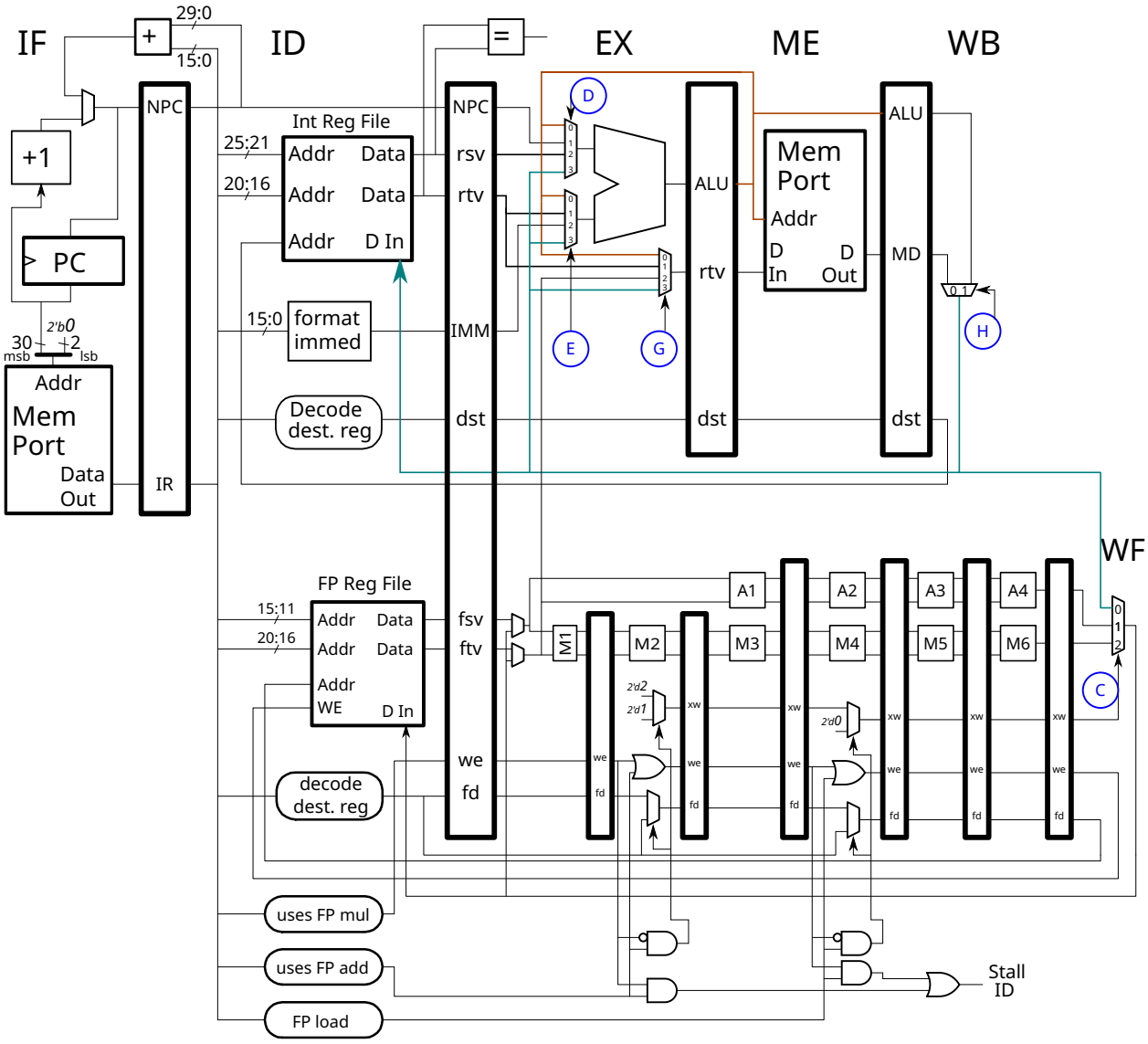

Staple This Side



☐ For the **4-way** superscalar MIPS above show execution until the fetch of the `lw r1` in the second iteration.
☐ Show instruction throughput (IPC) assuming a large number of iterations.

```
LOOP:
 lw r1, 0(r2)

 lw r3, 4(r2)

 add r4, r1, r3

 sw r4, 0(r6)

 sub r5, r1, r3

 sw r5, 4(r6)

 addi r6, r6, 4

 bne r2, r9, LOOP

 addi r2, r2, 8

 xor r10, r11, r12
```

Problem 2: (20 pts) Appearing below is our MIPS implementation with a floating-point pipeline. The select inputs of some multiplexors are labeled with a letter. Also, the inputs to some multiplexors have been colored to make them easier to follow.



☐ Show the values on the select inputs (D, E, and H) expected from the execution shown below. ☐ Leave a signal **blank** if it does not affect execution.

```
# Cycle          0    1    2    3    4    5    6

D:

E:

H:
```

```
# Cycle          0    1    2    3    4    5    6
add R3, r5, r6   IF   ID   EX   ME   WB
addi r2, R3, 4        IF   ID   EX   ME   WB
lw r1, 0(R3)               IF   ID   EX   ME   WB
# Cycle          0    1    2    3    4    5    6
```

☐ Show instructions that could have produced the select input (D,E,G, and H) values shown below. ☐ Take dependencies into account when choosing register numbers.

```
# Cycle                  0    1    2    3    4    5    6
D:                                2    2    3
E:                                2    1    2
G:                                          0
H:                                          0    1
# Cycle                  0    1    2    3    4    5    6
                         IF   ID   EX   ME   WB

                              IF   ID   EX   ME   WB

                                   IF   ID   EX   ME   WB

# Cycle                  0    1    2    3    4    5    6
```

☐ Show the values on the select inputs expected from the execution shown below. ☐ Leave an input **blank** if it does not affect execution.

```
# Cycle          0    1    2    3    4    5    6    7    8    9

G:

H:

C:
```

```
# Cycle          0    1    2    3    4    5    6    7    8    9
lwc1 f1, 0(r5)   IF   ID   EX   ME   WF
swc1 f2, 0(r7)        IF   ID   EX   ME   WB
mtc1 f3, r8                IF   ID   EX   ME   WF
add.s f4, f5, f6           IF   ID   A1   A2   A3   A4   WF
# Cycle          0    1    2    3    4    5    6    7    8    9
```

Problem 3: (20 pts) Appearing to the right is the *early writeback* 2-way superscalar implementation from the 2021 Final Exam and 2022 Homework 6. Recall that in this implementation if slot 1 contains a load instruction then slot 1 writes back when it reaches `WB` but slot 0 writes back early, in `ME/MW`. If slot 1 does not contain a load instruction slot 0 writes back when it reaches `WB` and slot 1 writes back in `ME/MW`. This is illustrated in the execution below.

```
# Cycle          0  1  2  3  4  5  6
add r2, r3, r4   IF ID EX ME WB    # Slot 0 uses WB since slot 1 isn't a load.
and r1, r5, r6   IF ID EX MW
or r10, r9, r7      IF ID EX MW    # Slot 0 uses MW since slot 1 is a load.
lw r6, 8(r11)       IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
```

(*a*) Consider the execution of the code below:

```
# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX ME WB
sub r1, r2, r5   IF ID -> EX MW
```

The `sub` stalls because it needs to wait for `r2` from the `add`. Add control logic to generate a stall when slot 1 depends on slot 0. The output of rs src and rt src are `1` when the slot-1 instruction uses the `rs` and `rt` registers as sources. Use these in your solution.

☐ Add hardware to generate a stall (see the big OR gate) when slot 1 depends on slot 0.

(*b*) In the first code fragment below the `lw r5` stalls, but because the `lw` has a zero immediate it could have just used the value computed by the `add` instruction (since there is no need to add anything to it). In the second execution Mux A is used to perform a *lateral bypass* from the `add` to the `lw` during cycle 2, avoiding the stall.

Modify the control logic so that a lateral bypass will be performed when there is a zero-immediate load in slot 1 that depends on the slot-0 instruction. Other code, such as the examples at the top of this problem, should continue to work correctly.

```
# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX ME WB
lw r5, 0(r2)     IF ID -> EX ME WB

# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX MW      # add executes normally.
lw r5, 0(r2)     IF ID EX ME WB   # Lateral Bypass: lw uses slot-0 alu value.
```

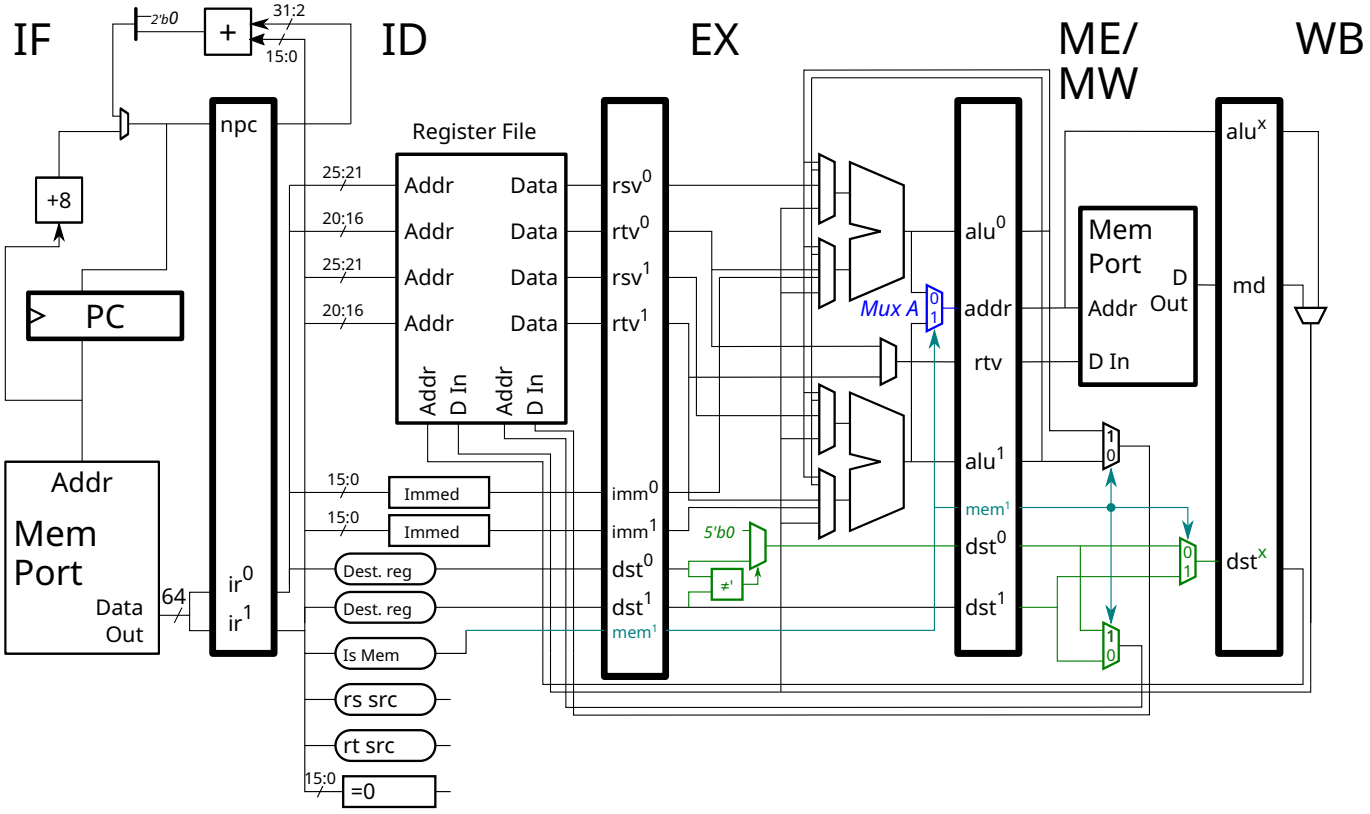
☐ Add logic to detect whether a lateral bypass is possible, and if so, suppress the stall from part a.

☐ Modify the control logic to implement a lateral bypass, in part using Mux A.

☐ Make sure that ☐ early writeback continues to work correctly in other cases and ☐ that the destination of the slot 0 and slot 1 instructions are written to the correct registers.

☐ As engineers always do, pay attention to cost and performance.

Staple This Side



Problem 4: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with an 6-outcome local history.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  N  N  T  T  N  T    T  N  N  T  T  N  T    T  N  N  T  T  N  T      <- Outcome

          All N's          All T's             All N's          All T's
      ---------------  --------------      ---------------  --------------  ...
B2:    N  N  ...  N  N  T  T  ... T  T      N  N  ...  N  N  T  T  ... T  T  <- Outcome
       1  2       7  8  9 10        16      1  2       7  8  9 10        16  <- Position
```

☐ What is the accuracy of the bimodal predictor on branch `B1`? ☐ Be sure to base the accuracy on a repeating pattern.

☐ What is the accuracy of the local predictor on B1 ignoring B2.

☐ What is the accuracy of the local predictor on B2 ignoring B1.

☐ What is the accuracy of the bimodal predictor on branch B2?

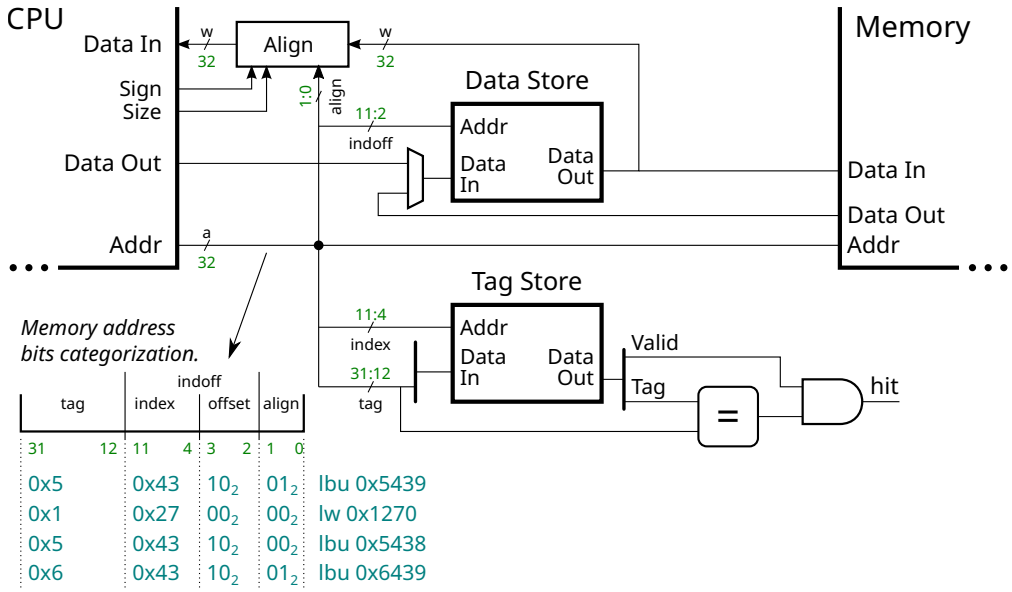
☐ What is the shortest history size for which the local history predictor is better than the bimodal predictor on branch B2?

Staple This Side

This page intentionally left blank, except for this notice.

Problem 5: (20 pts) Answer each question below.

(*a*) The diagram below shows a simple direct-mapped cache and the address bit categorization of four lookup addresses (`0x5439, 0x1270, ..`).



Two kinds of memory are used in the diagram above, fast/expensive and slow/cheap.

☐ On the diagram above show which blocks are fast and which blocks are slow.

Suppose that the cache is initially cold (there is nothing in the cache). Show the outcome, hit or miss, of each of the four lookups.

☐ Show outcome, hit or miss, on diagram above.

Find the addresses requested below.

☐ After the four lookups, what is the smallest address that will hit the cache.

☐ After these four lookups, what is the largest address that will hit the cache.

(*b*) Show the encoding for the `beq` and `lw` as used in the code below. Be sure to include the immediate value.

```
 addi r6, r0, 10
 beq r2, r6, SKIP
 lw r1, 4(r3)
 add r1, r1, r5
SKIP:
 and r9, r9, r1
```

☐ Encoding of `beq`. ☐ Be sure to show a value for the immediate field.

☐ Encoding of `lw`. ☐ Be sure to show a value for the immediate field.

(*c*) Answer the following about ISA families.

☐ Which style of implementation are RISC ISAs designed for?

☐ Which style of implementation are VLIW ISAs designed for?

(*d*) Some early RISC ISAs omitted useful magnitude-comparison branch instructions such as `bgt r1, r2, TARG` in which the branch is taken if `r1 > r2`. As branch prediction became more common magnitude-comparison branch instructions were added to RISC ISAs. One might argue that with branch prediction the cost and performance impact of magnitude-comparison instructions was lower.

☐ Explain how the cost of implementing `bgt` is lower with branch prediction than without.

☐ Explain how the performance impact of implementing `bgt` is lower with branch prediction than without.

5 Spring 2021

Name _____

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Friday, 26 March 2021 to Monday, 29 March 2021 16:00 CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \;\wedge\; r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

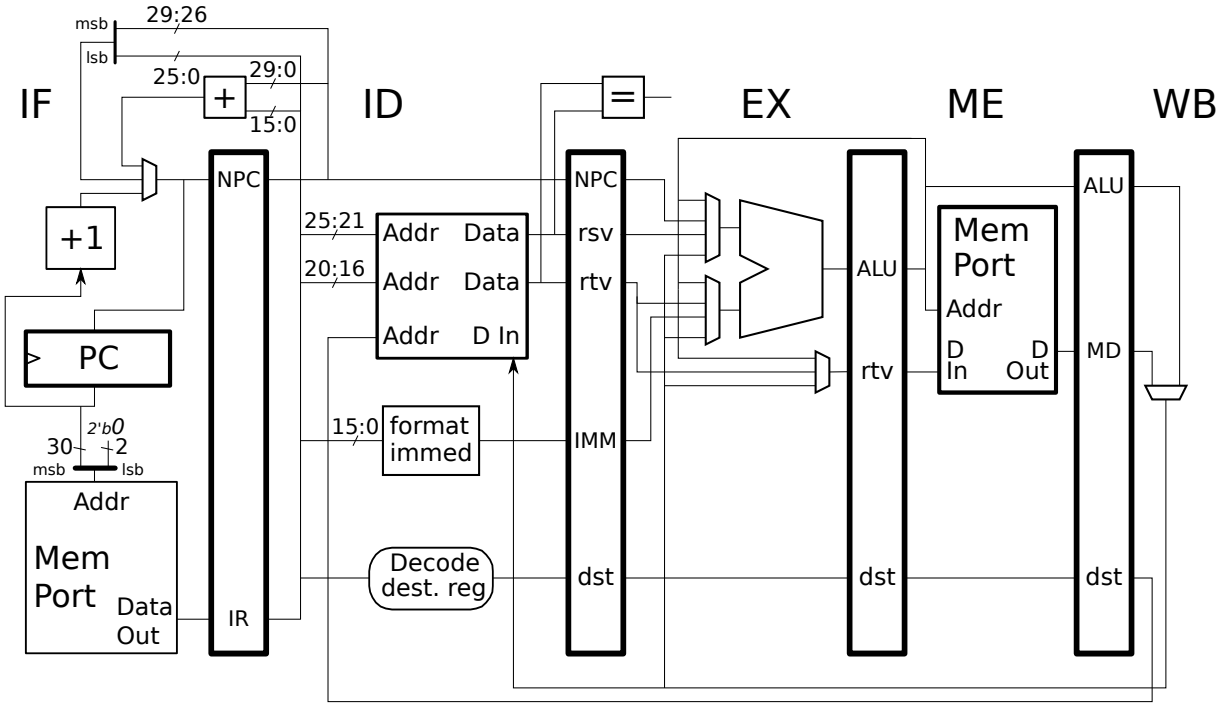
Problem 1: [30 pts] One instruction that might have come in handy for Homework 2 is the proposed `lbit`, load bit, instruction. Consider `lbit r1, (r2..r3)`. This instruction will load a single bit from memory into `r1`. Register `r2` holds a base address and `r3` holds a bit offset. The bit offset is relative to the most-significant bit of the byte at address `r2`. So if `r3` is zero the MSB is loaded into `r1`. If `r3` is 7 the LSB of the byte at `r2` is loaded into `r1`, if `r3` is 8 the MSB of the byte at `r2+1` is loaded into `r1`, etc. (As with Homework 1 and 2, bit ordering is big-endian.) To help understanding `lbit` there are two code fragments below. They do the same thing, the first uses `lbit`, the second uses existing MIPS instructions.

```
# Proposed Instruction
lbit r1, (r2..r3)

# Equivalent MIPS Code
sra r9, r3, 3
add r9, r2, r9
lbu r1, 0(r9)
sll r1, r1, 24
andi r9, r3, 0x7
sllv r1, r1, r9
srl r1, r1, 31
```

(*a*) Modify the illustrated MIPS implementation so that it implements `lbit`, omitting control logic. Assume that the memory port will be set to perform a read byte unsigned operation (the same operation as would be performed for the `lbu` instruction) and the ALU will be set to perform an add operation. (That is, don't assume or try to add new operations for the memory port nor for the ALU.) The modifications should provide the appropriate address to the memory port and should place the appropriate bit in the destination register.

As always, assume that the critical path is through the memory port. For this problem it is okay to put additional non-control logic in the `WB` stage.

- ☐ Add logic to compute the correct load address.
- ☐ Add logic to extract the needed bit.
- ☐ There is no need to show control logic.
- ☐ Don't assume or implement new Mem Port or ALU operations.
- ☐ It's okay to add logic to the `WB` stage.
- ☐ Pay attention to performance.
- ☐ Pay attention to cost. ☐ Do not show functional units that are more complicated than necessary. ☐ Use existing pipeline latches and other data carrying paths when possible.
- ☐ As always, do not break other instructions.


IF
ID
EX
ME
WB
msb
lsb
29:26
25:0
29:0
15:0
+
=
+1
PC
NPC
IR
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format immed
Decode dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem Port
Addr
D In
D Out
ALU
MD
dst
2'b0
30
2
msb
lsb
Addr
Mem Port
Data Out

(*b*) Show the execution of the code fragments below on your implementation. Add reasonable bypass paths to eliminate stalls.

☐ Add reasonable bypass paths to avoid stalls that would be suffered by the code below.

☐ Show execution of each code fragment (with reasonable bypass paths).

```
# Fragment A
addi r3, r3, 1

lbit r1, (r2..r3)

add r4, r4, r1
```

```
# Fragment B
lbit r1, (r2..r3)

addi r3, r3, 1

add r4, r4, r1
```

```
# Fragment C
lbit r1, (r2..r3)

bne r1, r0  TARG

addi r3, r3, 1


TARG:
xor r8, r9, r10
```

(*c*) Consider another instruction `ebit`, extract bit. Consider `ebit r1, r2, r3`. This instruction extracts the bit at position `r3` from `r2` and writes it to `r1`. The MSB is at position 0. The bit position is in the least significant five bits of `r3`, other bits of `r3` are ignored.

Both `lbit` and `ebit` extract a bit from a value, so it is possible to use some of the hardware for `lbit` to implement `ebit`. One difference is that `lbit` extracts an 8-bit quantity while `ebit` extracts a bit from a 32-bit quantity. If the hardware were shared, the `lbit` hardware would have to be upgraded to handle 32-bit values.

Ignoring whether such sharing really is a good idea, modify the implementation of `lbit` so that it could implement `ebit` using hardware shared with `lbit`.

☐ Modify MIPS hardware to implement `ebit` using hardware shared with `lbit`.

☐ No need to show control logic.

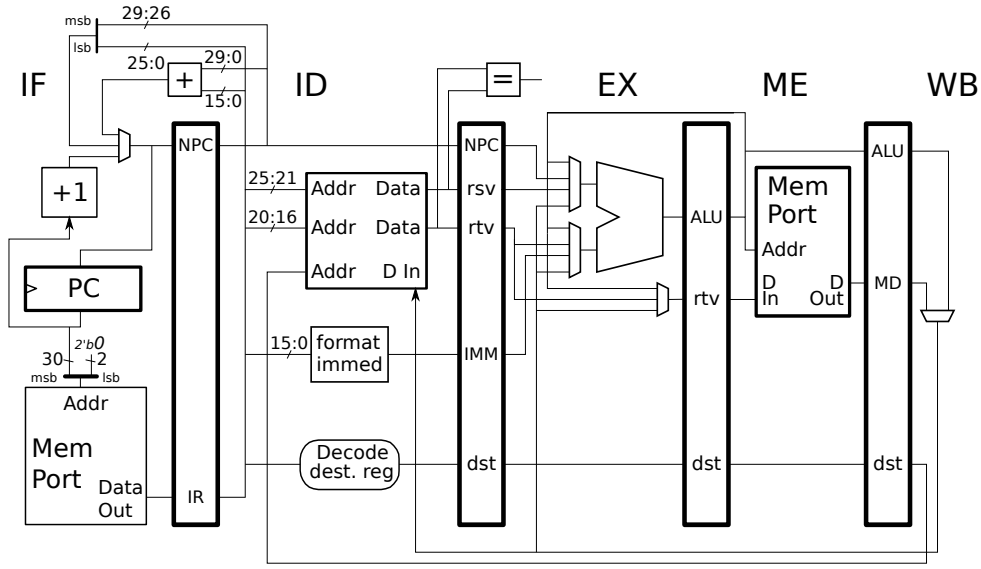
(*d*) Explain why an implementation sharing `ebit` and `lbit` hardware would execute the code fragment below slowly and describe a faster alternative.

```
ebit r1, r2, r3
add r4, r4, r1
```

☐ Why does the shared hardware implementation slow code below?

☐ Why is an implementation of `ebit` that is similar to other computation instructions faster?

Problem 2: [15 pts] Consider the pointer-chasing loop below. Assume that the loop executes many iterations on the illustrated hardware.



(*a*) Show an execution of the loop below for enough iterations—at least two—to compute the IPC (inverse of CPI). The IPC is the number of executed instructions divided by the number of cycles. Compute it for a very large number of iterations.

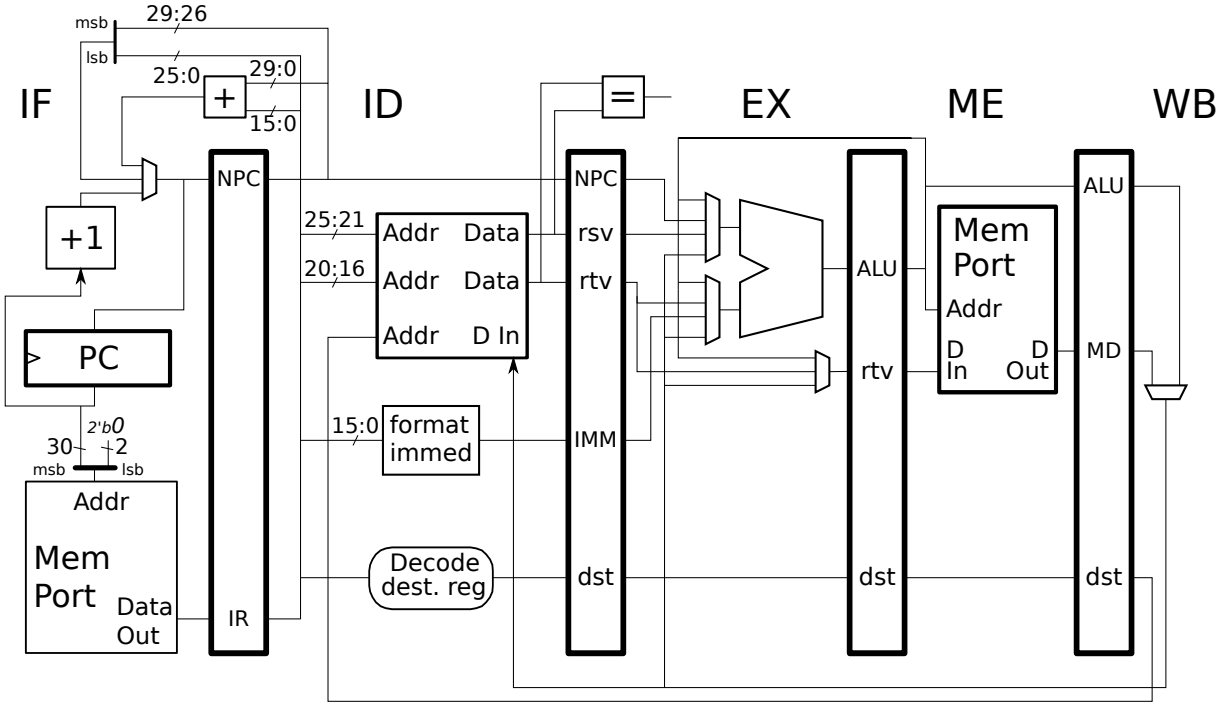
☐ Show execution.

☐ Compute IPC for a large number of iterations.

☐ Check for dependencies and available bypass paths.

```
 lw r3, 8(r3)
LOOP:
 lw r1, 4(r3)

 sw r1, 0(r3)

 bne r1, r5 LOOP

 lw r3, 8(r3)

 add r5, r3, r9
```

(*b*) If the previous part were solved correctly, then there should be two stalls per iteration. One stall could be eliminated by a bypass path, but the other could not (without increasing critical path). For each stall in your execution (even if there are more or less than two) show a reasonable bypass path that would avoid the stall or else explain why such a bypass is not reasonable.



☐ Show reasonable bypass paths needed to avoid stalls on your code.

☐ For each stall that could not be eliminated with a bypass path, explain why:

Problem 3: [15 pts] Appearing below are two candidate MIPS instructions, `jca`, *jump case add*, and `jcc`, *jump case concatenate*, that can be used to implement C-style `switch` statements. The instructions are designed for case statements that each consist of up to eight instructions. In both instructions the `rs` register (register `r1` in the examples) holds the address of case statement zero. Case statement 1 starts at address `r1+32`, case statement 2 starts at address `r1+32*2`, etc. The `rt` register (`r2` in the examples) holds the number of the case statement to jump to, so the address to jump to is `r1+32*r2`. The only difference between the two instructions is that in `jca` the value of `r1` must be a multiple of 4 (since instruction addresses are aligned) while in `jcc` the value of `r1` must be a multiple of 4096 (the 12 least-significant bits must be zero) and `r2` must be less than 128. Like other MIPS control transfers, both instructions have a 1-instruction delay slot. Note that `jca r1, r0` is equivalent to `jr r1`.

The code below shows the use of `jca` and an equivalent code fragment that uses only existing MIPS instructions.

```
# Candidate Instruction
jca r1, r2   # Jump to r1 + r2 * 32
nop

# Another Candidate Instruction
jcc r1, r2   # Jump to { r1[31:12], r2[6:0], 5'b0 }
nop

# Equivalent code to jca (and partly jcc) using existing MIPS instructions.
sll r9, r2, 5
add r9, r9, r1
jr r9
nop
```

A resolve-in-`ID` implementation of `jcc` can be designed at low cost and with no risk of lengthening the critical path. In contrast, a resolve-in-`ID` implementation of `jca` would add to cost and risk critical path impact.

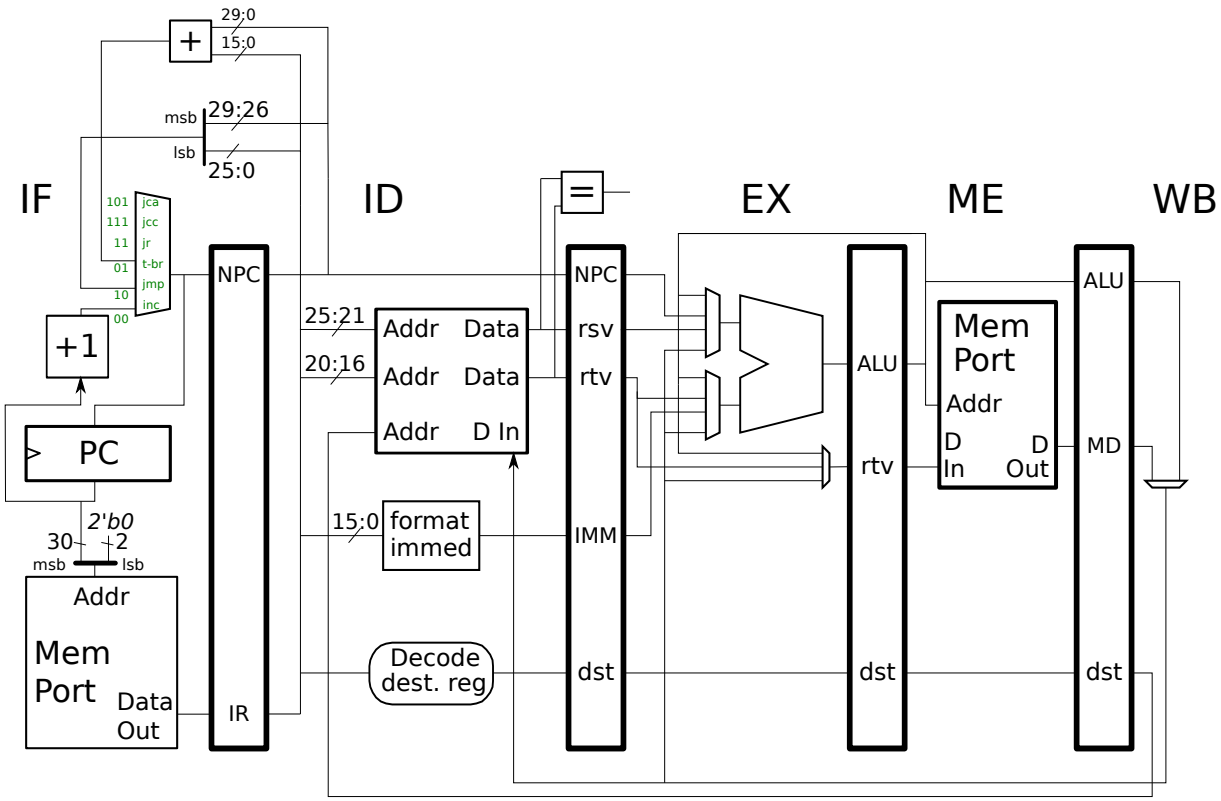
(*a*) Show the datapath changes to the MIPS pipeline on the next page needed for resolve-in-`ID` implementations of the two instructions.

☐ Show datapath changes (not control logic) for resolve-in-`ID` implementation of ☐ `jca` and ☐ `jcc`.

☐ As always, pay attention to cost and performance.

(*b*) Explain why computing a branch target, which is done using an adder, has no critical path impact while there is critical path impact for `jca`.

☐ Why can a branch safely use an adder in `ID`, but not `jca`?


IF
ID
EX
ME
WB
+
29:0
15:0
msb
29:26
lsb
25:0
101 jca
111 jcc
11 jr
01 t-br
10 jmp
00 inc
+1
PC
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
NPC
IR
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format
immed
Decode
dest. reg
=
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst

Problem 4: [40 pts] Answer each question below.

(*a*) MIPS branches have one delay slot. That enables five-stage scalar MIPS implementations to fetch the delay-slot instruction while resolving the branch. So, is the delay slot a feature of the ISA or a feature of the implementation?

☐ Is a delay slot an ISA feature or an implementation feature? ☐ Explain.

(*b*) There are 32 MIPS integer (general-purpose) registers, usually called `r0` to `r31`. But these registers are also given names, which are shown in the table below. Suppose we wanted to rearrange the names. For example, suppose we wanted to name register `r16` `t8` (instead of name `r24` `t8`) and make `r24` the new `k0`. Which registers could we rearrange without changing the ISA? It must be possible to use the registers for the purpose suggested by their names after rearranging.

```
Names     Numbers Suggested Usage
$zero:    0       The constant zero.
$at:      1       Reserved for assembler.
$v0-$v1:  2-3     Return value
$a0-$a3:  4-7     Argument
$t0-$t7:  8-15    Temporary (Not preserved by callee.)
$s0-$s7: 16-23    Saved by callee.
$t8-$t9: 24-25    Temporary (Not preserved by callee.)
$k0-$k1: 26-27    Reserved for kernel (operating system).
$gp      28       Global Pointer
$sp      29       Stack Pointer
$fp      30       Frame Pointer
$ra:     31       Return address.
```

☐ Which register numbers can get new names without having to change the ISA? ☐ Explain.

(*c*) The code fragment below adds 1 to the floating-point value in register `f2` and puts the sum in `f3`.

```
addi $t1, $0, 1    # Integer 1
mtc1 $t1, $f1
cvt.s.w $f1, $f1
add.s $f3, $f1, $f2
```

☐ Explain what the `cvt.s.w` instruction does in the code above.

☐ Re-write the code so that it adds a 1, but does so without the `cvt.s.w` instruction.

(*d*) In early RISC ISAs, including MIPS I, floating-point registers were 32 bits and yet many of these ISAs had double-precision (64-bit) floating-point instructions. Where do these instructions find their 64-bit operands?

☐ MIPS-I gets 64-bit floating-point operands . . .

(*e*) ARM A64 and RISC-V RV64 are both late RISC ISAs. But ARM A64 has many more instructions than RISC-V. How does having more instructions help A64 and fewer instructions help RISC-V?

☐ Lots of instructions help A64 because . . .

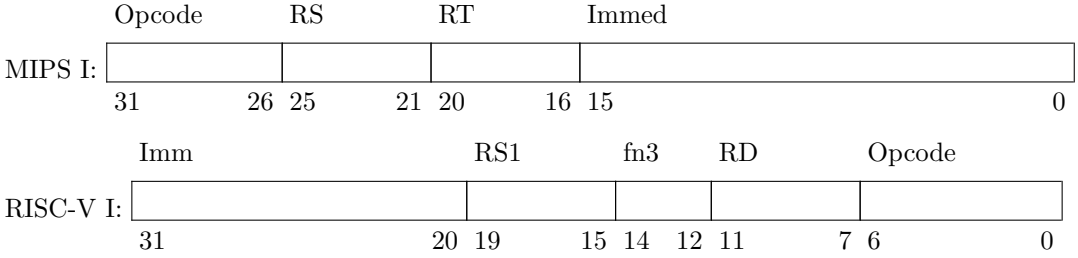
☐ Fewer of instructions help RISC-V because . . .

(*f*) Explain the problem with this statement:

Implementations of CISC ISAs were slow because of complex instructions. Only later did computer engineers discover that with simpler RISC ISAs implementations could be made faster.

☐ The statement is misleading or incorrect because . . .

(*g*) Appearing below are MIPS and RISC ISAs' immediate formats.



☐ What advantage does the MIPS format have?

☐ Show an example of a MIPS instruction that could not be encoded in the RISC-V format.

☐ What advantage does the RISC-V format have? (Another question on this exam implies this advantage is wasted.)

(*h*) Compilers optimize by scheduling (rearranging) instructions to avoid stalls due to true dependencies. In that case, why do we need to have bypass paths?

☐ Bypass paths are needed despite optimizations because:

Name _____

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Tuesday, 27 April 2021 to Friday, 30 April 2021 16:00 (4 PM) CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. For example, don't Web-search the text of a problem unless the problem specifically allows it. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman. **Suspected violation of these rules will be reported to the Dean of Students as a violation of the Student Code of Conduct.**

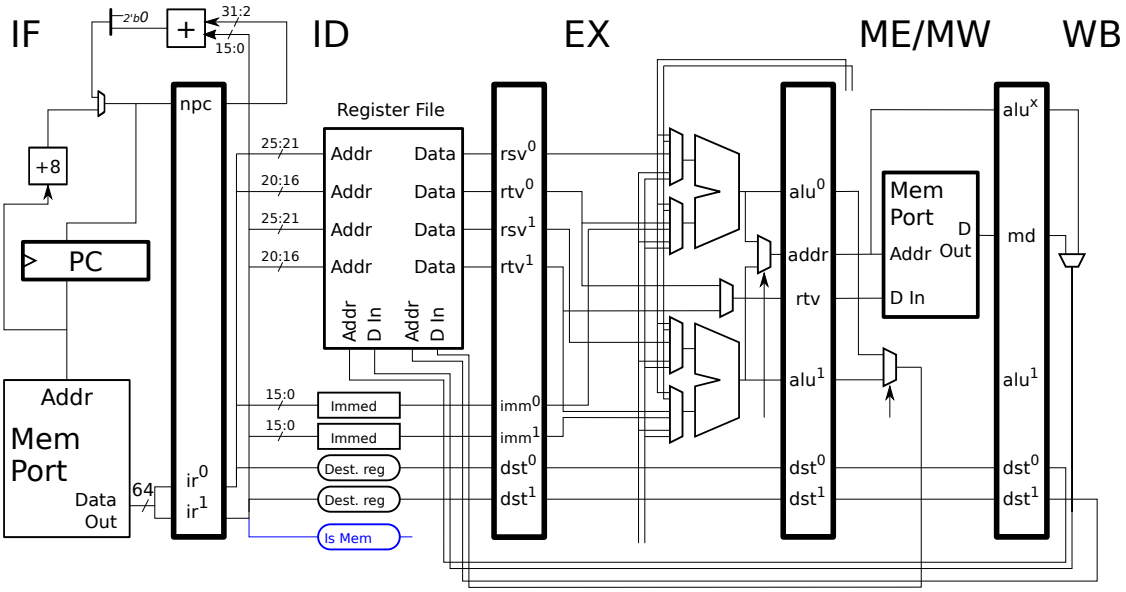
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \;\wedge\; r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Thank you for your effort in EE 4720!

Problem 1: (25 pts) Appearing below (and larger a few pages ahead) is an idea for a 2-way superscalar MIPS implementation with an unorthodox feature: The register file is written both by an instruction in the `ME/MW` stage (the former `ME` stage) and by an instruction in the `WB` stage. A memory instruction would have to write back in the `WB` stage, but a non-memory instruction could write back in either stage. Consider the execution below. For the first pair, `lw` and `addi`, the `lw` must use `WB` and the `addi` must use `MW`. The situation is similar for the second pair except that the memory instruction, `lb`, is in slot 1 rather than slot 0. When neither slot holds a memory instruction (the pair fetched in cycle 2) either one (but not both!) could use `MW`. If both use the memory port, the later one should stall. In execution diagrams label `MW` is used by an instruction that writes back in that stage, and `ME` is used by an instruction that will write back in the `WB` stage.

Put solution on diagram several pages ahead!



Put solution on diagram several pages ahead!

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID EX ME WB
addi r2, r2, 4   IF ID EX MW
sub r3, r4, r5      IF ID EX MW
lb r7, 5(r2)        IF ID EX ME WB
and r9, r10, r11       IF ID EX ME WB
or r12, r13, r14       IF ID EX MW
lb r16, 0(r7)             IF ID EX ME WB
lh r17, 2(r7)             IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

This idea has a potential cost benefit, but it must be thought through because the order in which registers are written can vary. (Which is a scary thing to those worried about correctness.) One cost benefit can be seen in the diagram. The `WB.alu1` pipeline latch is no longer needed. The label is still there but it is not connected (and won't be). Other cost-saving changes are part of the subproblems below.

☐ For all parts of this problem remember to pay attention to cost and performance. ☐ For example, don't connect a bypass path that will never be needed.

(*a*) The `D In` connections to the write ports of the register file have been changed, but the register number inputs, `Addr`, are the same. Modify the hardware so that the `Addr` inputs to the write ports get the correct register number. For this part don't worry about two instructions writing the same register number.

☐ Modify hardware (next page, not above) so that `Addr` inputs of the register file write ports get the correct register number.

(*b*) The diagram shows one cost savings: the `WB.alu1` pipeline latch is no longer needed. Notice that the four bypass connections to the `EX` stage are unconnected. Reconnect them as needed so that any dependency that could be bypassed in the unmodified superscalar can be bypassed here. Leave a bypass unconnected if not needed.

☐ Re-connect bypass paths (on next page, not above), possibly adding or modifying other hardware to bypass values.

(*c*) Notice that there is an unconnected select signal in `EX` and `MW`. Design control logic for these and for any multiplexors used to provide the correct register numbers (the first subpart above). The Is Mem logic in `ID` should be helpful. *Hint: There is not that much to do for this part. The Is Mem block should come in handy.*

☐ Connect select signals in `EX` and `MW`.

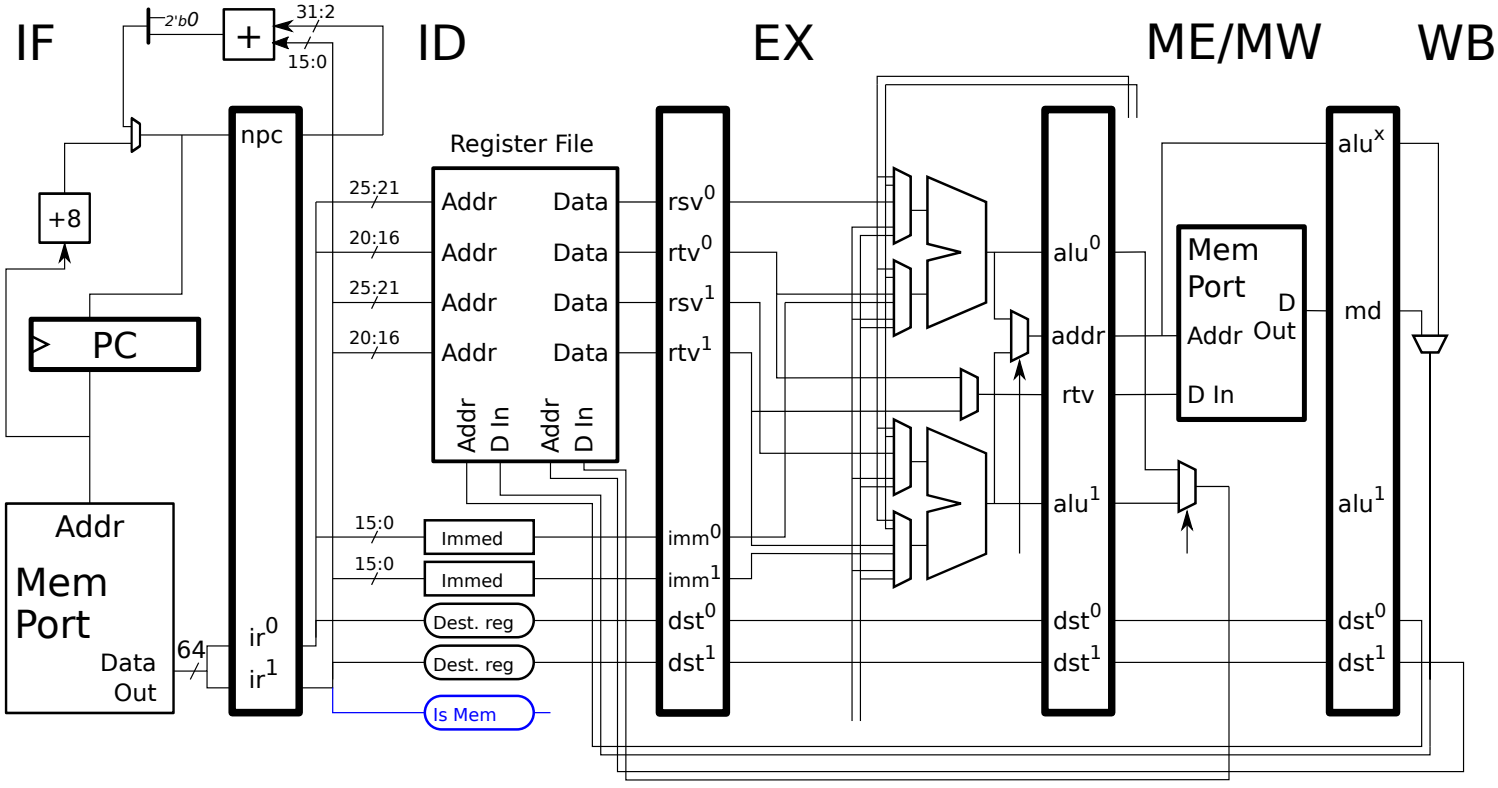
☐ Connect select signals for any multiplexors used for register numbers.

(*d*) Notice that in the execution below the `and` writes `r9` after the `or`. It looks like the `add` instruction will get the value written by the `and` rather than the `or`. That's not right!

```
# Cycle            0  1  2  3  4  5 ... 9  10 11 12 13
and r9, r10, r11   IF ID EX ME WB
or r9, r7, r14     IF ID EX MW
# .. later..
add r1, r1, r9                          IF ID EX MW WB
```

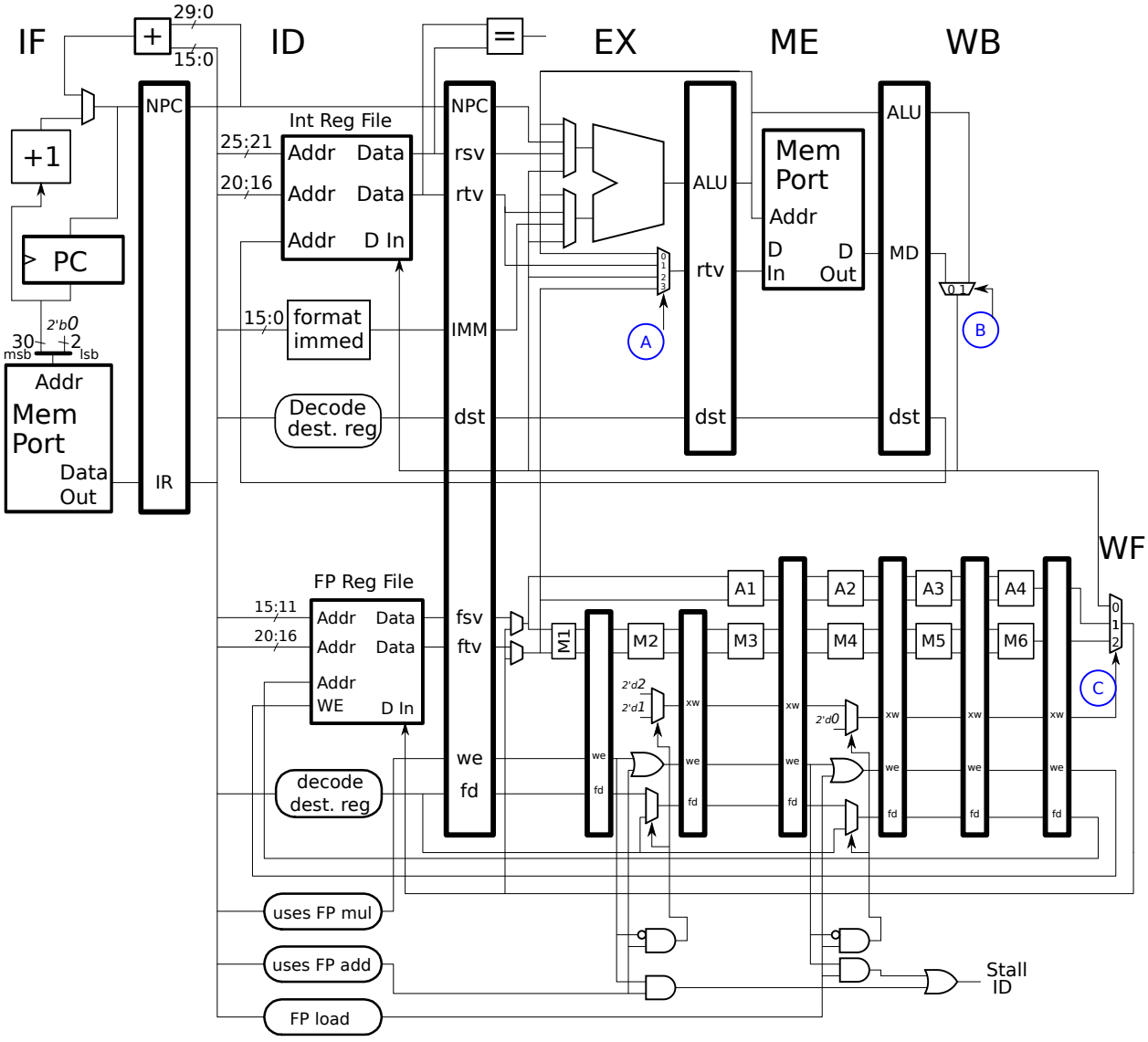
☐ Add control logic to detect such WAW hazards and which will substitute `r0` for the destination of the earlier instruction.

Use your favorite SVG or plain text editor on the SVG source for the implementation `https://www.ece.lsu.edu/ee4720/2021/fe-ss-px.svg` and logic gates `https://www.ece.lsu.edu/ee4720/2021/g.svg`.



Problem 2: (25 pts) Show the execution of the code fragments as requested below.

(*a*) The MIPS implementation below is similar to the one used in class, but with some added bypass paths for FP instructions and some labeled multiplexor select signals.



Appearing on the next page is a code fragment, and above the code fragment are the labels `A`, `B`, and `C`. These labels correspond to those used in the implementation.

Show the execution of the code below on this pipeline long enough to determine the IPC for a long number of iterations. Of course, that means the branch is taken. Show the value of each labeled select signal in those cycles it is being used.

☐ Show the execution on the illustrated implementation.

☐ Show the values of the labeled select signals, A, B, and C when they are in use.

☐ Find the IPC for a large number of iterations.

A

B

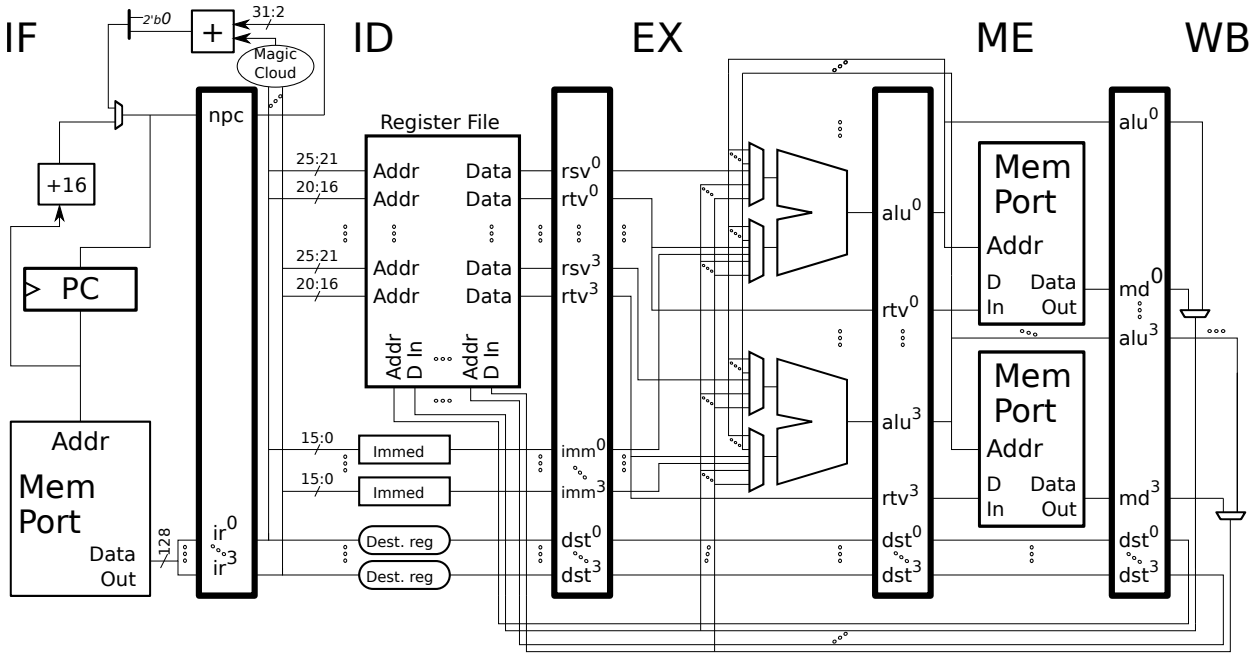
C

```
LOOP:
 lwc1 f1, 0(r1)

 lwc1 f2, 4(r1)

 add.s f3, f1, f2

 swc1 f3, 8(r1)

 bne r1, r2  LOOP

 addi r1, r1, 12

 xor r5, r1, r6
```

(*b*) There should have been stalls in the execution of the code above. Re-write the code so that it executes with as few stalls as possible and still computes the same result. It is okay to add extra instructions before and after the loop. For convenience assume that the code executes for at least two iterations. But don't unroll the loop.

☐ Re-write code to avoid stalls. ☐ Code must compute the same values. ☐ Can re-arrange instructions, change registers, and put instructions before the start of the loop.

(*c*) Appearing below is a 4-way superscalar MIPS implementation. In this implementation fetch is not aligned (which makes things easier). Also, there is no branch prediction, which is how we have been doing things in class.



☐ Show the execution of the code below for enough iterations to determine IPC. (Note: There is no need to put slot numbers on the stage labels.) ☐ Don't forget that it is 4-way superscalar.

LOOP:

```
lw r10, 0(r1)

add r3, r10, r3

sw r3, 0(r5)

addi r5, r5, 4

bne r1, r9, LOOP

addi r1, r1, 4

lb r8, 0(r9)

xor r11, r8, r10
```

(*d*) The code fragment below is to execute on the same 4-way superscalar MIPS implementation. Notice that loop body in the code fragment below contains two copies of the loop body from the loop in the previous subproblem. So one iteration of the loop below does the work of two iterations of the loop from the previous problem. This is the first step in the application of a technique called *loop unrolling*. The loop from the previous problem has been unrolled by *degree 2*. The next step is to re-arrange the instructions, and possibly eliminate those that are no longer needed. Complete this step, of course without changing what the code does. Finally, to eliminate all stalls *software pipelining* will need to be used: values computed used in one iteration will have to come from instructions executed in the prior iteration.

```
LOOP:
 lw r10, 0(r1)
 add r3, r10, r3
 sw r3, 0(r5)
 addi r5, r5, 4
 addi r1, r1, 4
 lw r10, 0(r1)
 add r3, r10, r3
 sw r3, 0(r5)
 addi r5, r5, 4
 bne r1, r9, LOOP
 addi r1, r1, 4
 lb r8, 0(r9)
 xor r11, r8, r10
```

☐ Re-write the loop above so that it runs more efficiently on the 4-way MIPS implementation. ☐ Use fewer instructions, even if doing so does not help with the degree 2 unroll, in the expectation that it might be beneficial at higher unroll degrees.

Problem 3: (25 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with a 10-outcome local history.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  T  T  N  T  N    T  T  T  N  T  N    T  T  T  N  T  N   <- Outcome
     1  2  3  4  5  6    1  2  3  4  5  6    1  2  3  4  5  6   <- Outcome Pos.

BA:

B2:   N  N  N ...  N  N  T  T  T    N  N  N ...  N  N  T  T  T
      1  2  3     11 12 13 14 15    1  2  3     11 12 13 14 15  <- Outcome Pos.
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the local predictor on B1 ignoring B2.

☐ What is the accuracy of the local predictor on B2 ignoring B1.

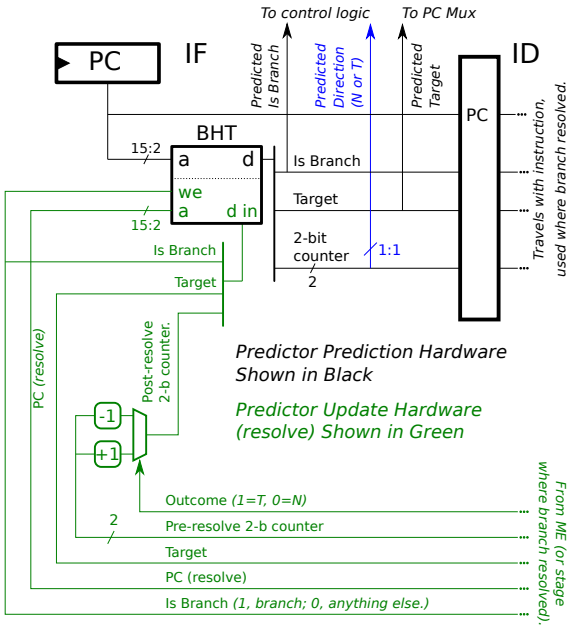
☐ What is the longest local history size for which branch B1 and branch B2 will interfere with each other on the local predictor? (The question is for a local predictor, not a global predictor.)

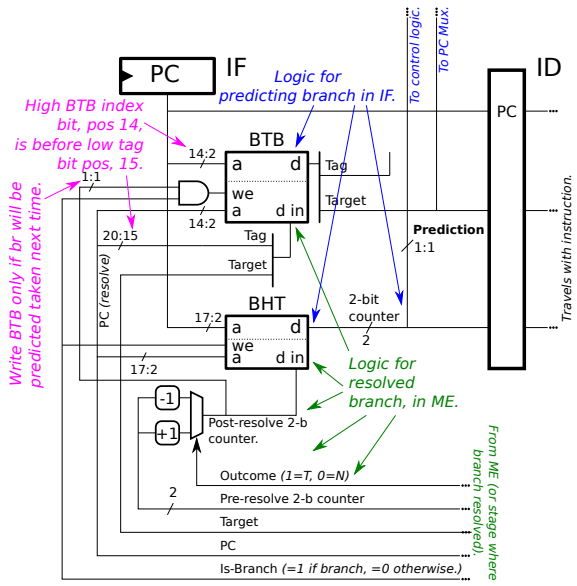
(*b*) If branch B1 at position 6 is mispredicted `T` (taken) then data at address $a + x$ will be brought into the cache. An adversary would like to learn the value of x. To do so the adversary, running in another process on the same core as the process using B1, will force a misprediction of 6. Then the adversary will make careful timing measurements of loads to addresses starting at a to determine what address was loaded, and so learning x.

Branch BA is part of the adversary's code. Find a pattern for BA that will force B1 to be mispredicted by the 10-outcome local predictor at position 6 (but not at other positions). The misprediction does not need to occur every time position 6 in B1 is executed.

☐ Pattern for BA that forces misprediction of B1 at position 6 on 10-outcome local predictor.

(*c*) Below on the left is a plain bimodal predictor as first described in class. Below on the right is a refined version that makes better use of storage by splitting the BHT into two tables, a *branch target buffer*, holding the target and a *tag*, and a BHT holding only the two-bit counters. The tag field replaces the `IsBranch` field and is used like a cache tag. (Though a cache tag would include bits `31:15`.) The control logic will compare the tag retrieved from the BTB with bits `20:15` of the PC, if they match a prediction will be made using the 2-bit counter from the BHT, if the tag doesn't match it is assumed that the instruction is not a branch. See 2017 Homework 8 Problem 3 for additional explanation.





The memory address and behavior of two branches from two programs, A and B, are shown below. One program runs better on the plain predictor than the BTB predictor, the other program runs better on the BTB predictor. Identify which is better, and why, as requested below.

```
# Program A
B1: 0x9000:  T T T ...
B2: 0x1000:   T T T ...
```

☐ Prediction of branches in A better on ◯ *Plain* or ◯ *BTB* predictor. ☐ Explain why other predictor does worse on this program. ☐ Pay attention to address of branches.

```
# Program B
B3: 0x11000:  N N N T  N N N T  N N N T    ...
B4: 0x21000:   T T T N  T T T N  T T T N    ...
```

☐ Prediction of branches in B better on ◯ *Plain* or ◯ *BTB* predictor. ☐ Explain why other predictor does worse on this program. ☐ Pay attention to address of branches.

Problem 4: (25 pts) Answer each question below.

(*a*) A program runs with fewer stalls on an 2-way superscalar than on an 8-way superscalar statically scheduled processor. Both have the same clock frequency and are otherwise comparable.

☐ Does that mean the program runs faster on the 2-way processor? ☐ Explain.

(*b*) Shorten the code fragments below.

☐ Shorten code fragment.

```
addi r1, r0, 0x4755
sll r1, r1, 16
addi r1, r1, 0x4720
lw r1, 0(r1)
```

☐ Shorten code fragment.

```
lw r3, 0(r1)
addi r2, r0, 4
add r1, r1, r2
lbu r1, 0(r1)
```

(*c*) Appearing below are three code fragments. These are to run on a machine with vector instructions and four-lane vector units. Indicate whether each fragment can easily be replaced by a vector instruction, and the reason why or why not.

☐ Fragment below ◯ *can* or ◯ *cannot* be replaced by a vector instruction. ☐ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
mul.s f7,  f8,  f9
sub.s f10, f11, f12
```

☐ Fragment below ◯ *can* or ◯ *cannot* be replaced by a vector instruction. ☐ Explain.

```
add.s F1,  f2,  f3
add.s f4,  F1,  f6
add.s F7,  f8,  f9
add.s f10, F7, f12
```

☐ Fragment below ◯ *can* or ◯ *cannot* be replaced by a vector instruction. ☐ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
add.s f7,  f8,  f9
add.s f10, f11, f12
```

(*d*) Let's break something! As we know, an implementation of an ISA must execute a program as defined in the ISA. The Broeken Company is going to sell a 4-way superscalar statically scheduled five-stage "implementation" of ARM A64, call that the Broeken4 implementation. (There is no space between the Broeken and the 4. That's so you don't notice the 4.) We know how smoothly our five-stage scalar MIPS implementation handles branches: zero penalty because of the delay slot. But A64 branches don't have a delay slot. That didn't stop the Broeken Company from changing things a bit. In the Broeken4 implementation the number of delay slots for a branch can vary from 4 to 7. If a branch is in slot 0 of its fetch group there are 7 delay slots. Branches in slots 1, 2, and 3 have 6, 5, and 4 slots, respectively. If the branch is not stalled by dependencies there will be zero branch penalty. This chip performs about 20% better than the (rule-abiding) competition on performance and energy efficiency benchmarks.

Computer engineering professors obviously will hate the Broeken company for ignoring what an implementation of an ISA should do. But what about others? Gauge the reaction of each group below.

Note: If this were an in-class exam questions would be shorter.

☐ Compiler writers will be ◯ *positive, happy, and supportive* or ◯ *negative, irritated, and obstreperous* . ☐ Explain.

☐ Buyers of laptops and desktops using Broeken4 will be ◯ *positive, satisfied* or ◯ *negative, seeking a refund* . ☐ Explain.

☐ Engineers using Broeken4 in embedded devices, such as the controller chip in a microwave oven, will be ◯ *positive, satisfied* or ◯ *negative, seeking a new supplier* . ☐ Explain.

(*e*) The execution below is taken from an illustration of how exceptions work from a class lecture. The `lw` raises an exception in cycle 4 and in response the handler starts in cycle 5. The first instruction that the handler executes is `sw`. It is likely that the handler is saving registers to the stack before attending to the event causing the exception. Presumably the handler will save every register that it plans to modify (or to be safe, all registers). To reduce the number of registers saved, why not split the work between the interrupted code and the handler. As with the ABI rules for procedure calls, why not have the interrupted code save the caller-save registers it wants preserved (`t0-t9`, etc.) so that the handler would only need to save the callee-save registers it plans to overwrite (`s0-s7`, etc.)?

```
# Cycle:                0  1  2  3  4  5 ...          99 100 ...
        add  r1, r2, r3  IF ID EX ME WB
        lw   r6, 0(r1)      IF ID EX ME*x             IF ID EX ME WB
        or   r5, r6, r7        IF ID EXx                 IF ID EX MEx
        xor  r10, r11, r12        IF IDx                    IF ID EXx
        and  r20, r21, r22           IFx                       IF IDx
...
Handler:
        sw ...                          IF ...                      IF ...
        ...
        eret (exception return)                 IF ID EX ME WB
```

☐ Why can't interrupted program save the caller-save registers before the handler starts, potentially reducing work?

6 Spring 2020

Name _____

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Tuesday, 14 April 2020 to Friday, 17 April 2020 23:59 CDT

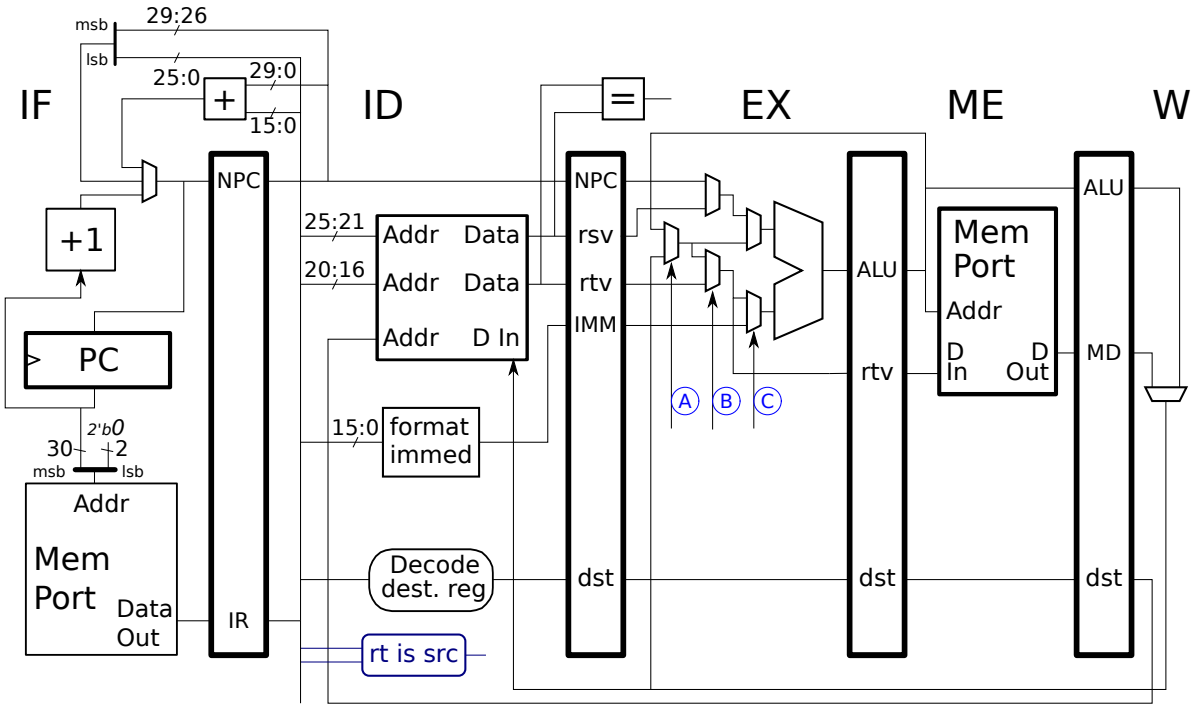
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(15 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Don't be Foolish!

Problem 1: [15 pts] The pipeline below is a slightly lower cost version of the bypassed MIPS implementation that we've been using. The cost saving is achieved by not allowing an instruction to use a bypassed value from both the ME and WB stage, the value must come from one stage or the other. Select inputs are shown for three of the re-done EX stage multiplexors, they are labeled A, B, and C. For this problem assume that they are connected to properly designed control logic.



(*a*) Show the values on the labeled select signals for an execution of the code below for those cycles in which an instruction below is in the EX stage. If the value on a select signal does not matter, show an X.

☐ Show values of A, B, and C for when EX occupied by code below. ☐ Use X if value does not matter, blank when no insn in EX.

```
#    Cycle        0     1     2     3     4     5     6
 add r1, r2, r3   IF    ID    EX    ME    WB
 sub r4, r5, r1         IF    ID    EX    ME    WB
 sw r6, 8(r1)           IF    ID    EX    ME    WB

#    Cycle        0     1     2     3     4     5     6
 A

 B

 C
#    Cycle        0     1     2     3     4     5     6
```

(*b*) Show a code fragment that would stall on the implementation above but would not stall on our usual bypassed MIPS (which appears in Problem 3).

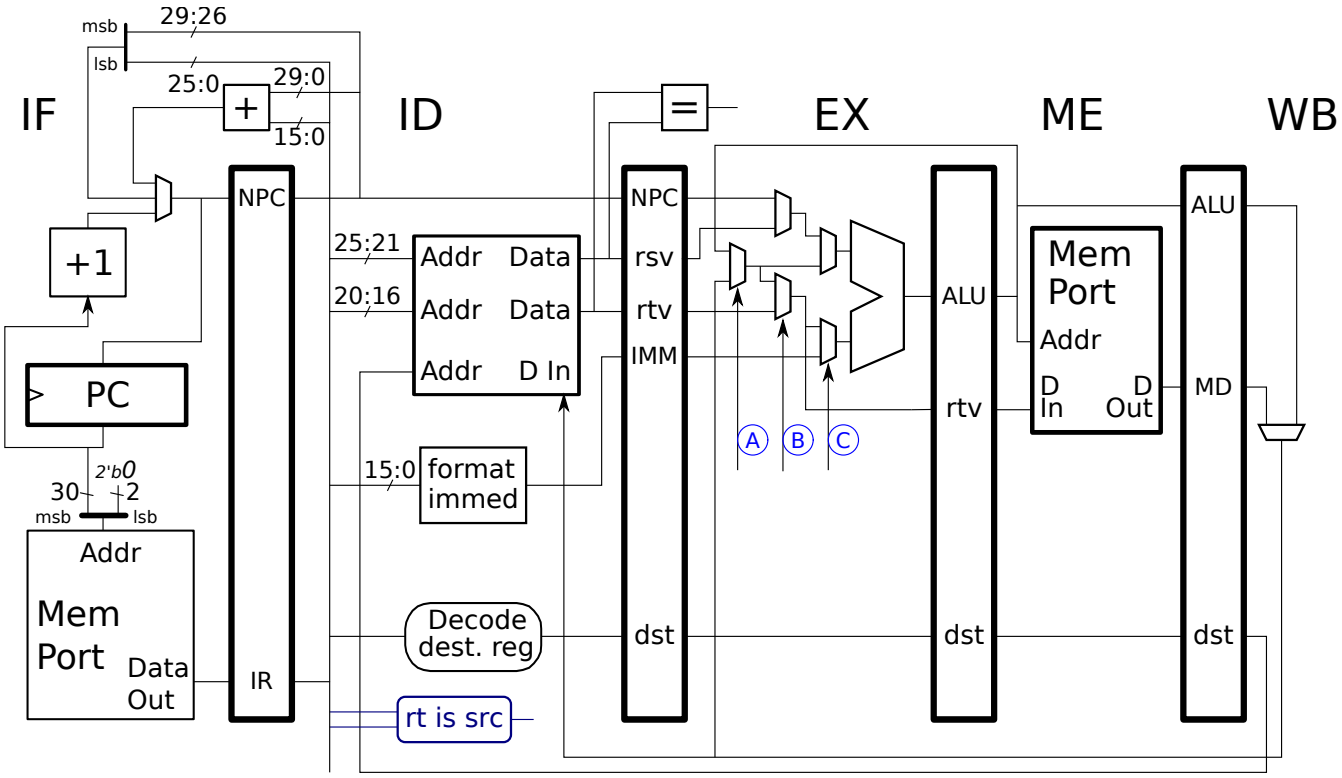
☐ Code fragment that stalls on this implementation, but not our usual 5-stage MIPS.

Problem 2: [25 pts] Appearing below is the lower-cost MIPS implementation from the previous problem. Design the control logic specified below. The output of rt is src is 1 if the `rt` field of the instruction specifies a source value, as it does in most type R but only a few type I, such as `sw`. The Inkscape SVG source for the image below can be found at `https://www.ece.lsu.edu/ee4720/2020/mt-p1.svg`.

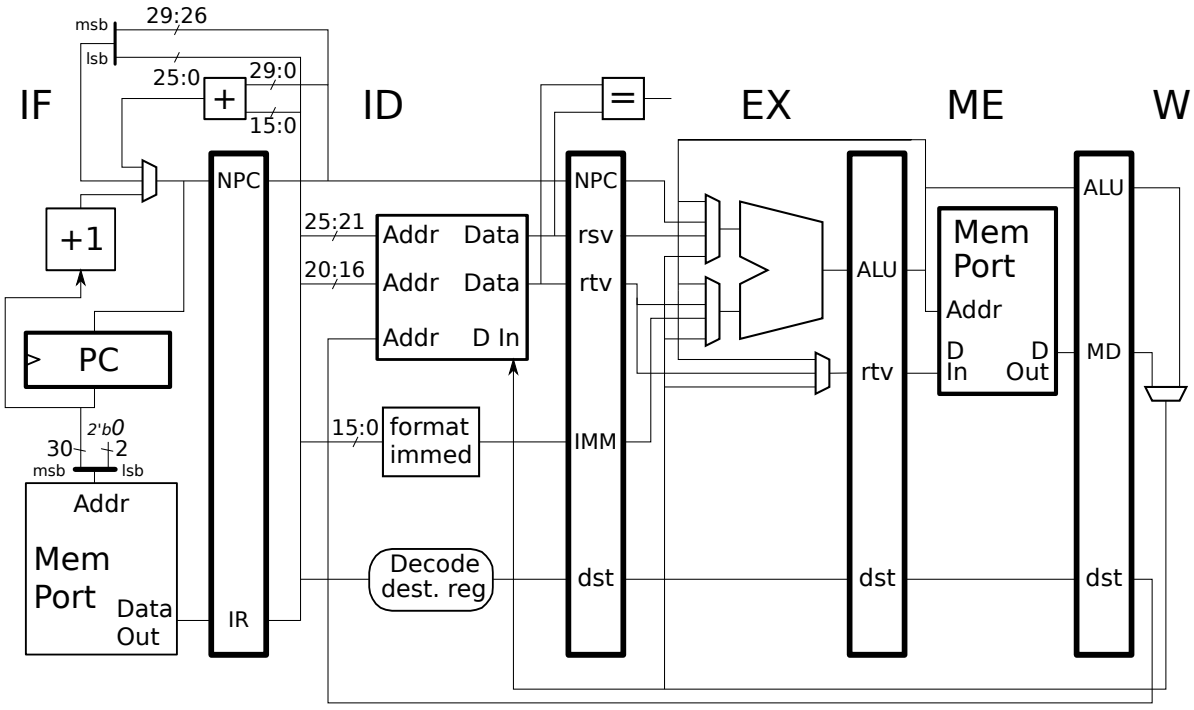
☐ Design control logic for the labeled multiplexor select signals, A, B, and C.

☐ Design control logic to generate a stall signal when a bypass would have been from both `ME` and `WB`.

☐ Pay attention to the usual stuff: ☐ Cost and critical path. ☐ The stage that instructions are in when the select signals are computed and the stage in which they are used.



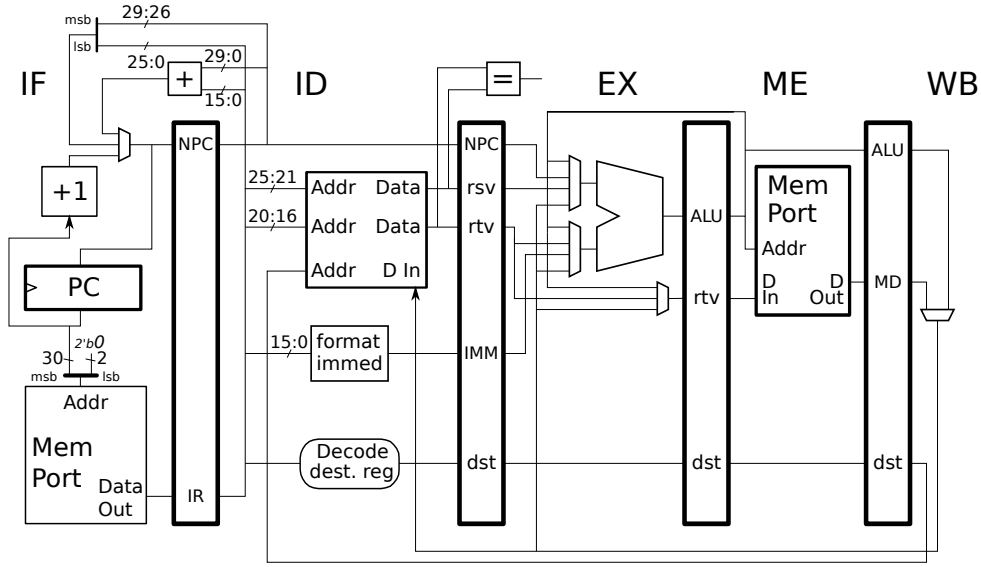
Problem 3: [15 pts] Show the execution of the code fragment below on the illustrated implementation.



☐ Show execution. ☐ Note that the branch is taken. ☐ Pay attention to the timing of the branch.
☐ Check for dependencies, ☐ including for the branch.

```
lw r1, 0(r2)

slt r3, r1, r4

# Branch is taken.
beq r3, r0  SKIP

addi r2, r2, 4

xor r5, r5, r9

or  r6, r6, r9

SKIP:
addi r7, r7, 4

sw r1, 0(r7)
```

Problem 4: [15 pts] The code fragment below runs inefficiently. Modify the code so that it runs faster on the implementation below. Instructions can be re-arranged, changed, or removed, and registers can be changed. Don't forget that the modified needs to do the same thing as the original code.

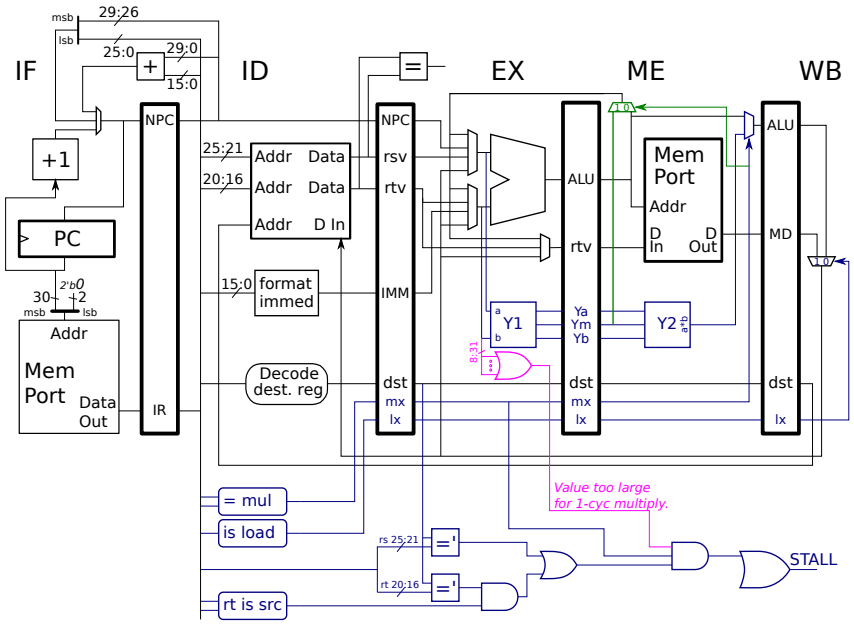


☐ Re-write code so that it is faster but, of course, does the same thing as the original.

```
LOOP:
 lw r1, 0(r2)
 andi r1, r1, 0xff
 addi r2, r2, 4
 lw r3, 0(r2)
 srl r3, r3, 24
 add r9, r9, r1
 add r9, r9, r3
 addi r2, r2, 4
 sub r8, r2, r11
 bne r8, r0  LOOP
 nop
```

Problem 5: [30 pts] Answer each question below.

(*a*) The code fragments below are to run on the implementation with the small-multiply bypass from Homework 3 and shown to the right. For each code fragment, indicate whether our small-value bypass feature always eliminates a stall, sometimes, or never? Explain.



☐ Eliminates stall on code below: ◯ *Always* ◯ *Sometimes* ◯ *Never*

```
andi r3, r5, 0x3f
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☐ Eliminates stall on code below: ◯ *Always* ◯ *Sometimes* ◯ *Never*

```
ori r3, r5, 0x63f
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☐ Eliminates stall on code below: ◯ *Always* ◯ *Sometimes* ◯ *Never*

```
lbu r3, 0(r4)
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☐ Eliminates stall on code below: ◯ *Always* ◯ *Sometimes* ◯ *Never*

```
lw r3, 0(r4)
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

(*b*) In typical practice a company decides upon an ISA, and then makes multiple implementations of that ISA. Let H_I and L_I be two implementations of ISA I, H_I is a high-end system and L_I is low-cost. Let ISA E (for expensive) be an ISA designed for high-end systems, and ISA C (for cheap) be an ISA designed for low-cost systems, and let H_E and L_C be their implementations. All three ISAs and all four implementations were designed by skilled engineers with lots of resources.

☐ Why might H_E be better than H_I and why might L_C be better than L_I? The same reason should apply to both. The answer is related to the ISAs used for the implementations.

☐ Even if L_C is better than L_I, why might a user still choose L_I?

(*c*) Consider the preparation of a set of SPECcpu results. For each item below indicate who is responsible, SPEC (the organization) or the tester. Also indicate what would be the problem if it were the other way around. For example, if you answered that SPEC chooses the benchmarks, then explain the disadvantage of having the tester choose the benchmarks.

☐ Choose the benchmarks: ◯ *SPEC* or ◯ *The Tester*

☐ Problem if it were the other way around:

☐ Choose the benchmark input data: ◯ *SPEC* or ◯ *The Tester*

☐ Problem if it were the other way around:

☐ Choose the benchmark training data: ◯ *SPEC* or ◯ *The Tester*

☐ Problem if it were the other way around:

☐ Choose the compiler: ◯ *SPEC* or ◯ *The Tester*

☐ Problem if it were the other way around:

☐ Choose the compiler optimization flags: ◯ *SPEC* or ◯ *The Tester*

☐ Problem if it were the other way around:

(*d*) The IA-32 ISA has been described as Intel's golden handcuffs. Who slapped on those handcuffs? What does the gold refer to? What do the handcuffs refer to? *This was discussed in class, but it is okay to use Web searches to answer this question.*

☐ The reason for these handcuffs is:

☐ They are golden because:

☐ They are handcuffs (a restriction) because:

(*e*) Appearing below are some hypothetical instructions. Indicate whether each instruction is a better candidate for a RISC ISA or a CISC ISA. Explain why.

☐ Is the instruction below more ◯ *RISC* or ◯ *CISC* like? ☐ Explain.

```
addi r1, r2, 0x12345678
```

☐ Is the instruction below more ◯ *RISC* or ◯ *CISC* like? ☐ Explain.

```
lw r1, (r2+r3)  # Load r1 = Mem[ r2 + r3 ]
```

☐ Is the instruction below more ◯ *RISC* or ◯ *CISC* like? ☐ Explain.

```
bgt r1, r2, TARG  # Branch if r1 < r2
```

☐ Is the instruction below more ◯ *RISC* or ◯ *CISC* like? ☐ Explain.

```
add (r1), r2, (r3)   # Mem[r1] = r2 + Mem[r3]
```

Name _____

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Wednesday, 6 May 2020 to Saturday, 9 May 2020 5:00 (5 AM) CDT

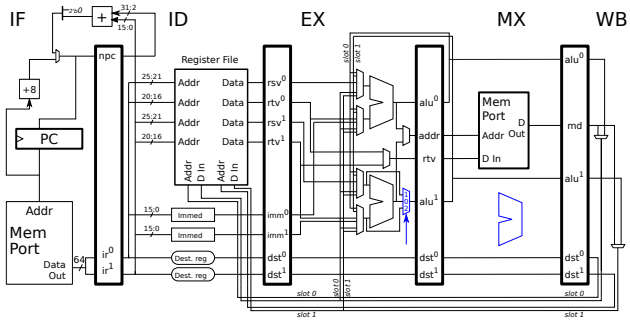
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Help Keep Everybody Safe!

Problem 1: (30 pts) The two-way superscalar MIPS implementation below has an ALU in the `MX` (née `ME`) stage, call it the *second-chance ALU*. For this problem the second-chance ALU will be connected so that two stall situations are avoided. *A slightly similar problem appeared on the Spring 2006 Midterm Exam in Problem 2. It's okay to look at the problem and solution.*



Yes, it's small! Use the next page for the solution.

(*a*) The `sub` in the code below suffers a stall due to a dependence with the other instruction in the same fetch group. Connect the second-chance ALU so that the stall is avoided. The changes must not break existing functionality and must not result in stalls for unrelated code. In particular note that the `add` does not stall in either version.

```
# Cycle           0  1  2  3  4  5  6  7   # Unmodified Implementation
ori R3, r1, 0xff  IF ID EX ME WB
xor R2, r8, r9    IF ID EX ME WB
add R1, R2, R3       IF ID EX ME WB
sub R4, R1, r5       IF ID -> EX ME WB
and r7, r8, R4          IF -> ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7

# Cycle           0  1  2  3  4  5  6  7   # With second-chance ALU.
ori R3, r1, 0xff  IF ID EX MX WB
xor R2, r8, r9    IF ID EX MX WB
add R1, R2, R3       IF ID EX MX WB        # No stall in either implementation.
sub R4, R1, r5       IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4          IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle           0  1  2  3  4  5  6  7
```

☐ Connect second-chance ALU to avoid the stall by the `sub` and allowing code to execute as in the sample above. ☐ The connections should work for any pair of dependent, ALU-using, non-memory instructions.

☐ Pay attention to cost. Assume that a pipeline latch bit costs twice as much as a multiplexor bit.

☐ Do not add unneeded bypass paths. ☐ Don't break existing functionality.

(*b*) The code below suffers a load/use stall. Add the minimum number of connections to the second-chance ALU so that such load/use stalls (in which the using instruction is in slot 1) can be avoided.

```
add r3, r2, r3    IF ID EX ME WB
lw r4, 5(r1)      IF ID EX ME WB
ori r6, r1, 0xff     IF ID EX ME WB
sub r5, r3, r4       IF ID -> EX ME WB
```

☐ Add connections to the second-chance ALU to avoid load/use stalls when the using instruction (such as the `sub` in the example) is in slot 1.

☐ Pay attention to cost, use the same cost assumption as given in the previous part.

(*c*) In the code below the `sub` does not stall due to the second-chance ALU but the `and` does stall. Add control logic to generate a stall signal for cases such as this.

```
# Cycle           0  1  2  3  4  5  6  7
ori R3, r1, 0xff  IF ID EX MX WB
xor r2, r8, r9    IF ID EX MX WB
add R1, r2, R3       IF ID EX MX WB
sub R4, R1, r5       IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4          IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle           0  1  2  3  4  5  6  7
```

☐ Add logic to generate a stall signal for the situation described above. ☐ The logic should work for any instruction dependent on an instruction using the second-chance ALU.

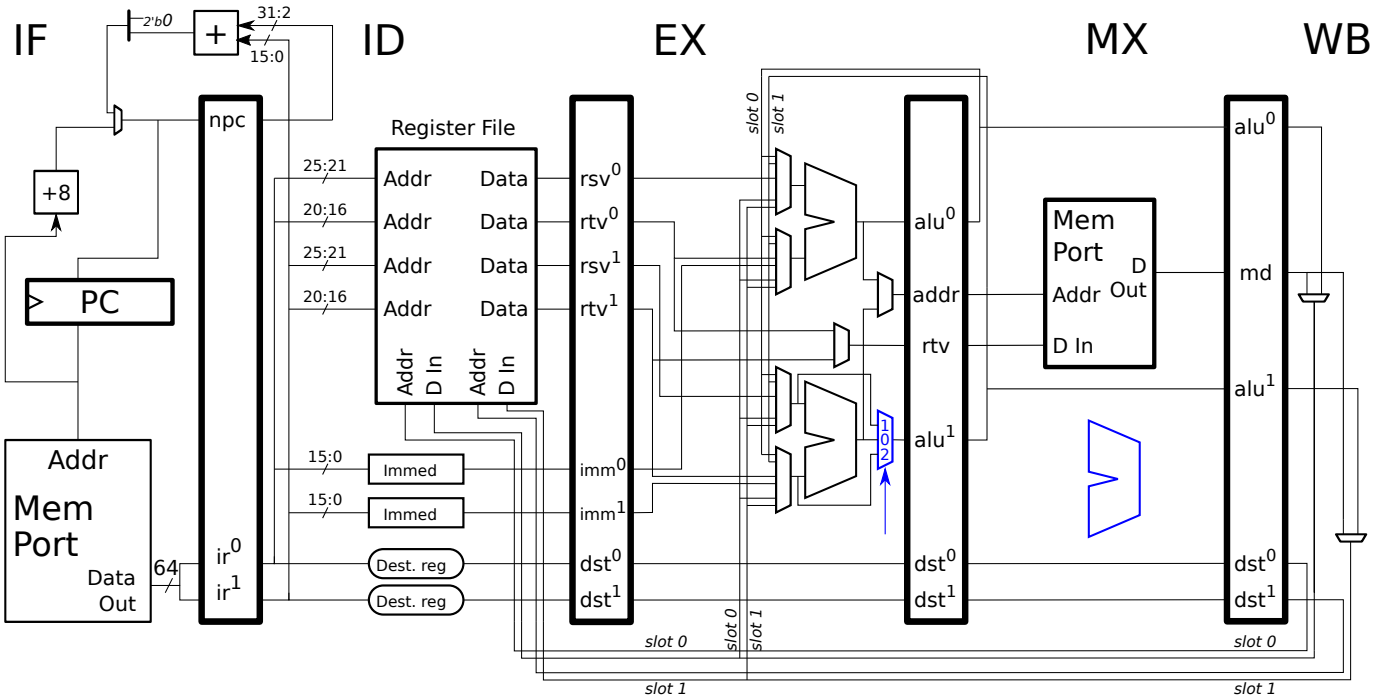
☐ Pay attention to cost, use the same cost assumption as given in the previous part.

(*d*) Generate the select signal for the `EX` stage multiplexor shown in blue. The control logic should work for the intra-group dependence case. (The control logic does not need to work for the load/use case.)

☐ Add logic for the select signal for the intra-group dependence. ☐ The logic should work for any pair of dependent, ALU-using, non-memory instructions.

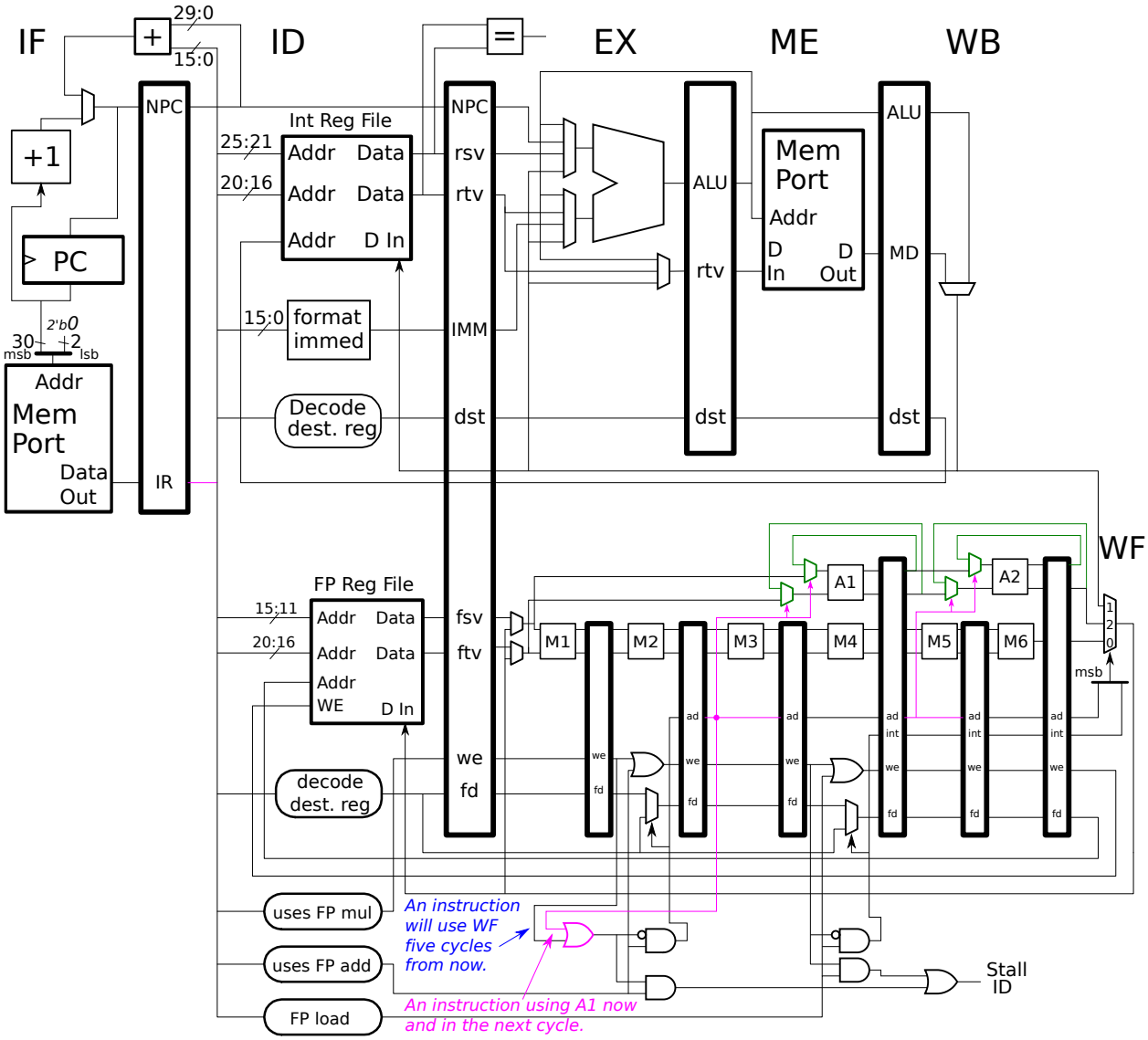
☐ Pay attention to cost, use the same cost assumption as given in the previous part.

The Inkscape SVG source is at `https://www.ece.lsu.edu/ee4720/2020/fe-p1-v2-ss.svg`.



Problem 2: (25 pts) Show the execution of the code fragments as requested below.

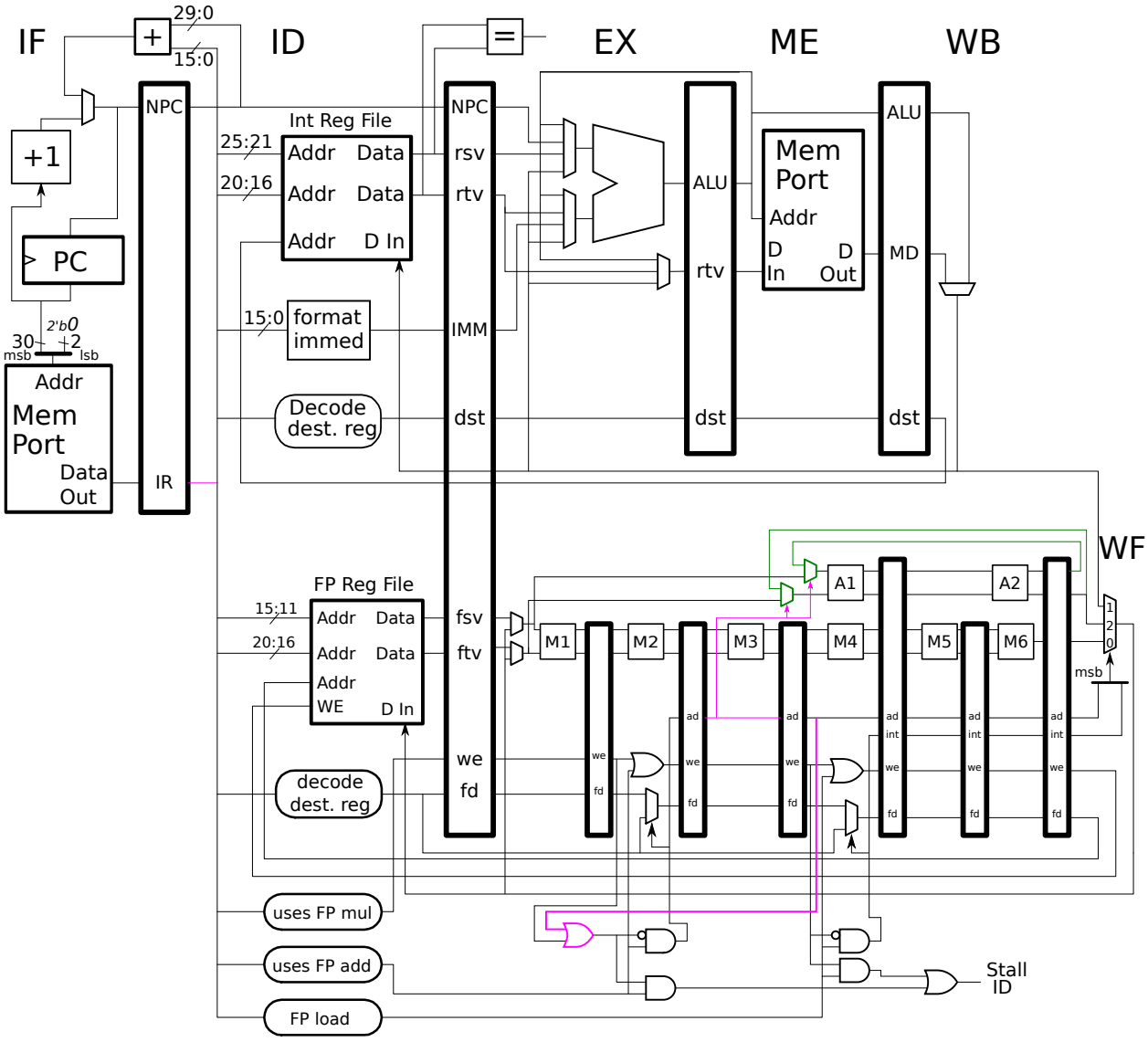
(*a*) Show the execution on the FP pipeline below, note that the adder unit has an initiation interval of 2.



☐ Show execution. ☐ Pay attention to how the FP add unit should operate. ☐ Don't forget to check for dependencies.

```
lwc1 f2, 0(r1)

add.s f0, f2, f4

add.s f1, f2, f5

add.s f3, f1, f6
```

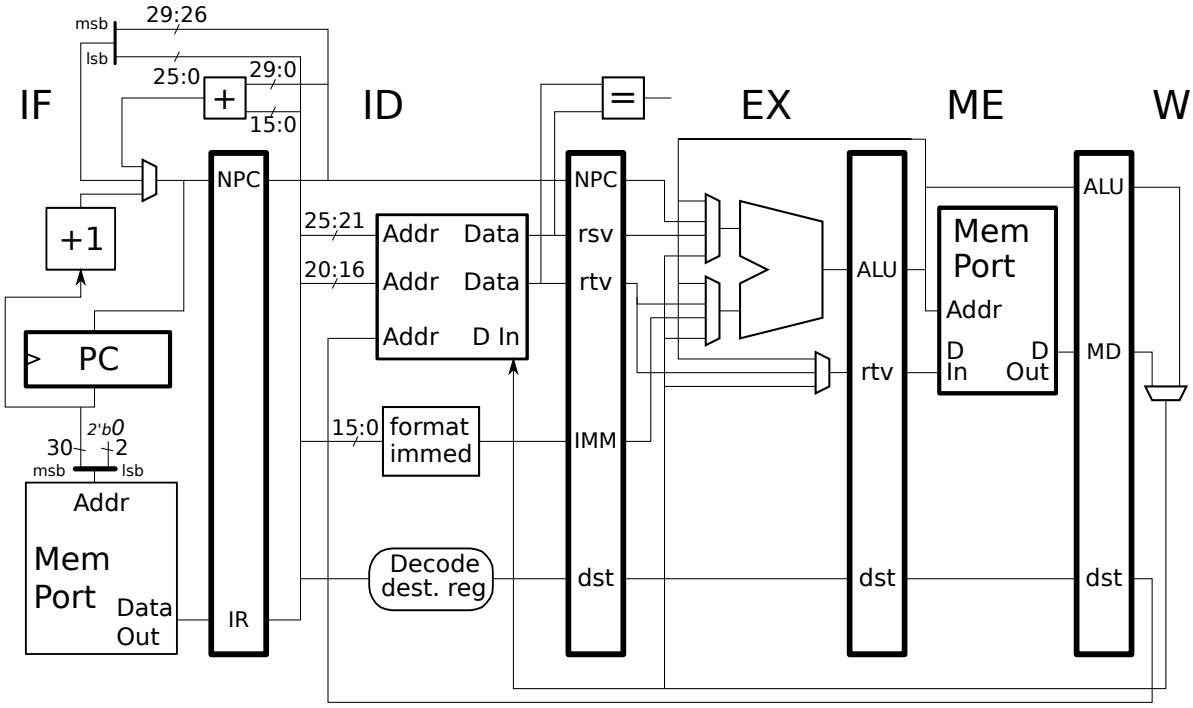
(*b*) Show the execution on the FP pipeline below, note that the adder unit is different than the previous problem and from other examples covered in class.



☐ Show execution. ☐ Pay attention to how the FP add unit should operate. ☐ Don't forget to check for dependencies.

```
lwc1 f2, 0(r1)

add.s f0, f2, f4

add.s f1, f2, f5

add.s f3, f1, f6
```

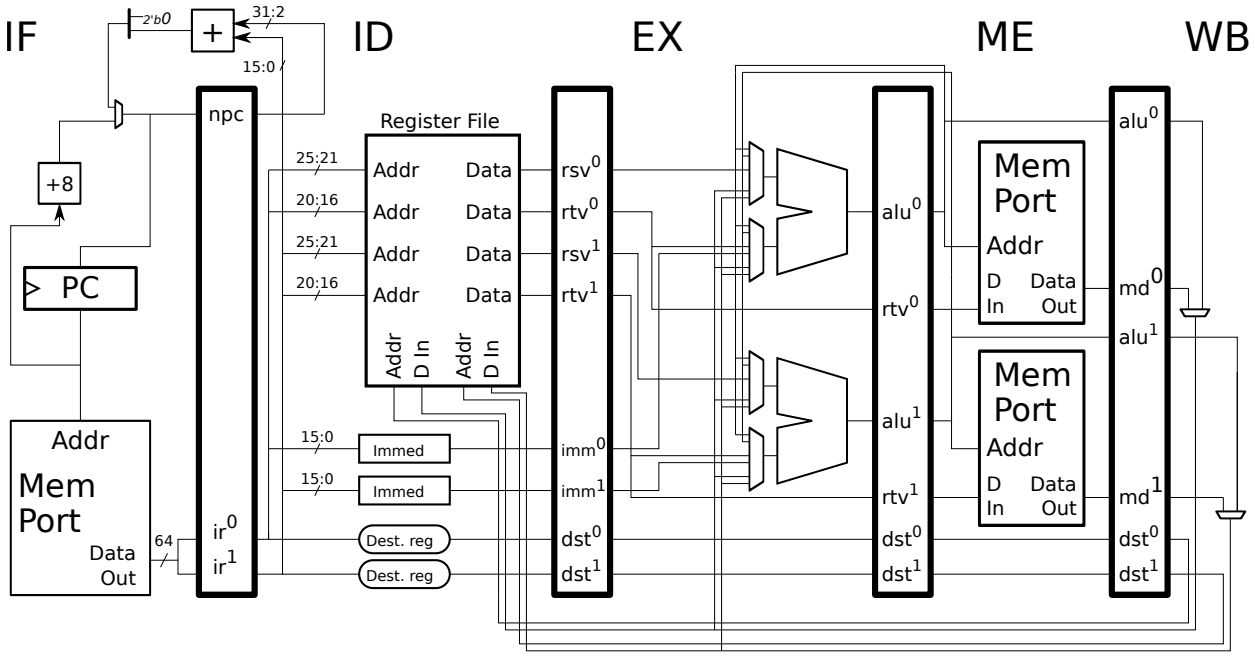
(*c*) Show the execution of the code on the implementation below. Find the CPI for a large number of iterations.



☐ Show execution on the illustrated implementation with the branch taken. ☐ Find the CPI for a large number of iterations.

```
LOOP:
 addi r2, r2, 16

 lw r1, 8(r2)

 sw r1, 12(r3)

 bne r3, r4, LOOP

 addi r3, r3, 32

 sub r10, r3, r2
```

(*d*) Show the execution of the code on the 2-way superscalar MIPS implementation illustrated below, and find the CPI for a large number of iterations. This is not the same as the implementation from Problem 1. Instruction fetch is of aligned groups.



☐ Show execution on the illustrated implementation. ☐ Find the CPI for a large number of iterations.

☐ Take aligned fetch into account, the address of `LOOP` is `0x1000`. ☐ Pay attention to available bypass paths.

```
LOOP:
 addi r2, r2, 16

 lw r1, 8(r2)

 sw r1, 12(r3)

 bne r3, r4, LOOP

 addi r3, r3, 32

 sub r10, r3, r2
```

(*e*) The code fragment below is the same as the one from the previous problem and is to run on the same superscalar system. Re-write the code so that it runs with fewer stalls (and of course does the same thing), and compute the CPI for a large number of iterations. Extra instructions can be added before or after the loop. Do not unroll the loop.

☐ Re-write code to minimize stalls on the superscalar implementation.

☐ Compute the CPI of the re-written code for a large number of iterations.

```
LOOP:
 addi r2, r2, 16
 lw r1, 8(r2)
 sw r1, 12(r3)
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2
```

Problem 3: (15 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and the other systems have a local predictor, the length of the local history is given in the questions below.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   N  N  N  N  T  T    N  N  N  N  T  T   ...

B2:      T  T  T  T  T  T    T  T  T  T  T  T   ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

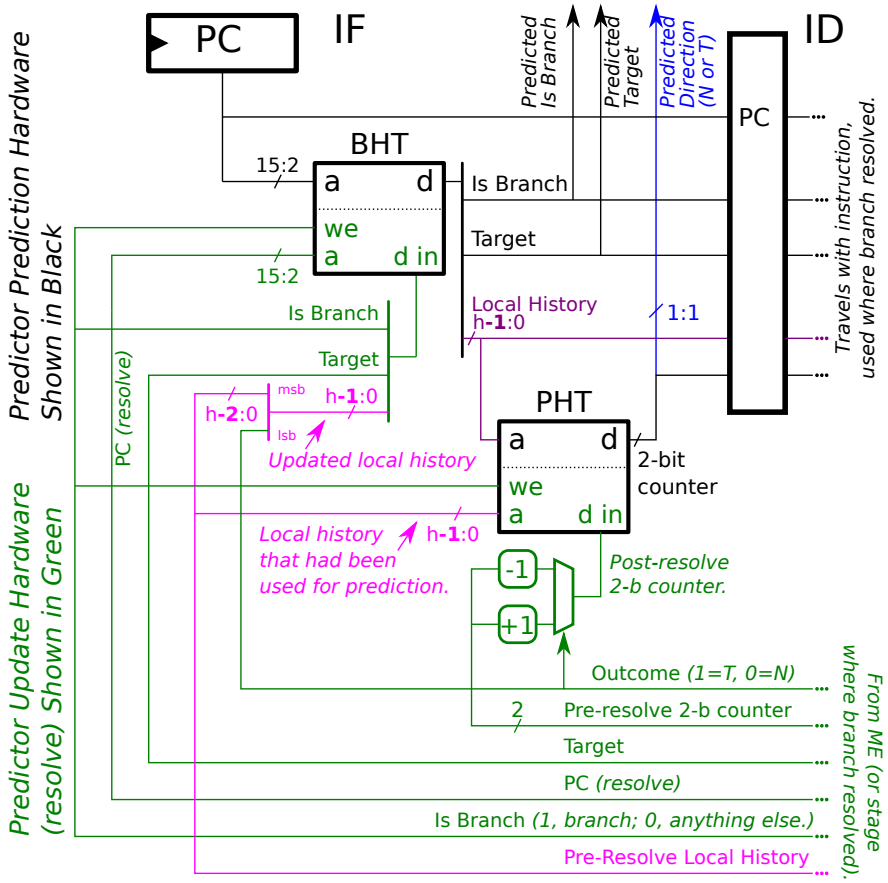
☐ What is the accuracy of a local predictor with a 12-outcome local history on branch `B1` and ignoring `B2`.

☐ What is the accuracy of a local predictor with a 2-outcome local history on branch `B1` and ignoring `B2`.

☐ What is the accuracy of a local predictor with a 2-outcome local history on branch `B1` and taking into account `B2`.

☐ What is the minimum local history size so that branch `B1` is predicted with 100% accuracy, taking into account `B2`.

(*b*) Appearing below is a diagram of a local predictor. The local predictor illustrated has a specific BHT size and an h-outcome local history size. The BHT size does not necessarily match the BHT in the prior part. Determine the amount of storage, in bits, used by the BHT and PHT for a 16-bit local history. Assume that Target is stored efficiently.



☐ Amount of storage for PHT is:

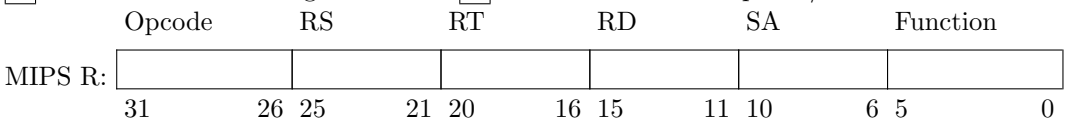
☐ Assumption about Target:

☐ Amount of storage for BHT is:

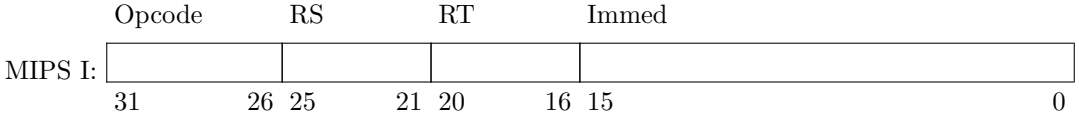
Problem 4: (30 pts) Answer each question below.

(*a*) Appearing below are the three integer MIPS I instruction formats. Consider a modified form of MIPS in which there are 64 rather than 32 integer registers. A goal is compatibility with MIPS-I code and to use as few new opcodes and function field values as possible. Modify each format so that it can use 64 registers and explain what new opcodes (if any) are needed and any assumptions about existing MIPS-I instructions. *Hint: For one case there's nothing to do, for one case many opcodes will be needed, for one case only a few.*

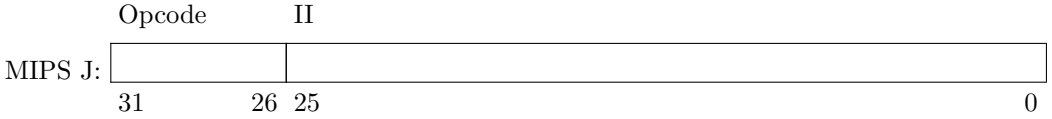
☐ Modification for 64-register MIPS. ☐ Describe what new opcode/func values are needed for, if any.



☐ Modification for 64-register MIPS. ☐ Describe what new opcode/func values are needed for, if any.



☐ Modification for 64-register MIPS. ☐ Describe what new opcode/func values are needed for, if any.



(*b*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock. *Yes, up until this comment the question is identical to one asked on the Spring 2019 final exam.* The SPECcpu benchmarks are run on each chip. Recall that SPECcpu can be run to compute a speed score and a rate score. (Don't confuse speed/rate with base/peak or int/FP.) Feel free to visit the SPEC site to help answering this question.

☐ Which chip would likely score higher (better) on the SPECspeed2017int benchmarks,
◯ *Chip A* or ◯ *Chip B* . ☐ Explain.

☐ Which chip would likely score higher (better) on the SPECrate2017int benchmarks,
◯ *Chip A* or ◯ *Chip B* . ☐ Explain.

(*c*) Our goal is to build a machine that can execute eight floating-point operations per cycle. Two machines are under consideration, an 8-way superscalar system implementing ISA I, and a 2-way superscalar system with an 8-lane vector unit implementing ISA IV, which is like I but with vector instructions. Both machines run at 1 GHz, and both can sustain eight billion floating-point operations per second.

☐ Which machine is likely to be more expensive? ☐ Explain.

☐ Which machine is likely to be faster on typical code? ☐ Explain with ☐ a code example.

(*d*) In MIPS and many other RISC ISAs memory accesses must be aligned. For example, a `lw` instruction, which loads a four-byte value, must load from an address that is a multiple of 4. The execution of a `lw` loading from an address that is not a multiple of 4 will result in an exception (and on Linux system resulting in the Bus Error signal handler being called). As we pointed out in class, integer instructions, and especially load and store instructions, in any reasonable ISA would be required to raise precise exceptions. MIPS is certainly reasonable in this respect.

Suppose that a program uses non-aligned addresses in memory accesses, but is otherwise correct. That is, the program would run correctly if the load could handle a non-aligned address. (After all, CISC ISA load instructions can do it.) But on MIPS it raises an exception as soon as a non-aligned load or store is attempted. Suppose further that re-writing the program is not feasible.

☐ Explain how we can take advantage of precise exceptions so that this program would run correctly. A code example would be nice but not necessary.

☐ Explain why it would be impossible if loads only raised deferred exceptions. (Assume that aligned accesses work fine with such loads.)

(*e*) MIPS has a `slt` (set less than) instruction, but doesn't have a `sge` (set greater than or equal to) instruction. Why not?

☐ Why doesn't MIPS have an `sge` instruction?

(*f*) *Note: The following question was asked about two months after in-person classes were ended for the CoViD-19 pandemic.*

Perhaps many of us are wishing we could go back in time. (Not to warn people, that's obviously futile.) Wish granted. You are in a meeting (in person, not Zoom) with future Turing Prize winners discussing which features to put into their new [airquotes] "RISC" ISA, MIPS.

One attendee is advocating for the inclusion of magnitude comparison branch instructions such as `bgt r1,r2 TARG` (branch if `r1` greater than `r2`). But many others oppose the idea because it would have too much critical path impact. "We can include `bgt` with zero critical-path impact if we use a surprisingly simple but effective technique called branch prediction," you say.

☐ Explain how branch prediction can remove the critical path impact that was a concern at the meeting.

☐ Was the phrase *branch prediction* an anachronism at that fictional meeting? Web-search freely to answer this question.

7 Spring 2019

Name _____

Computer Architecture
LSU EE 4720
Midterm Examination

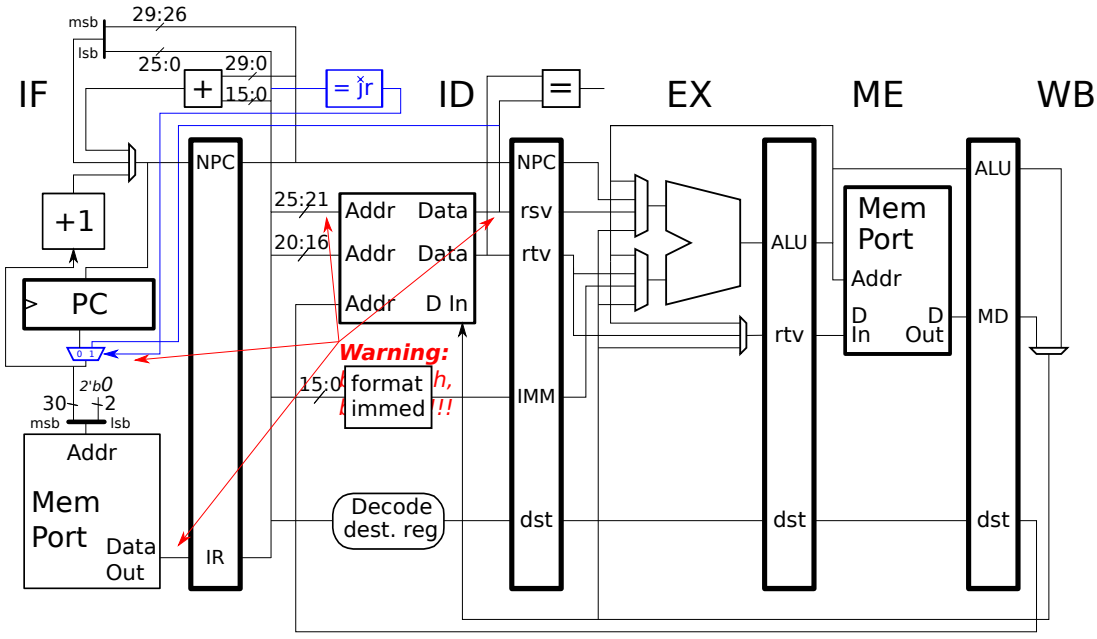
Wednesday, 27 March 2019, 9:30–10:20 CDT

Problem 1	_____	(7 pts)
Problem 2	_____	(17 pts)
Problem 3	_____	(27 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(12 pts)
Problem 6	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

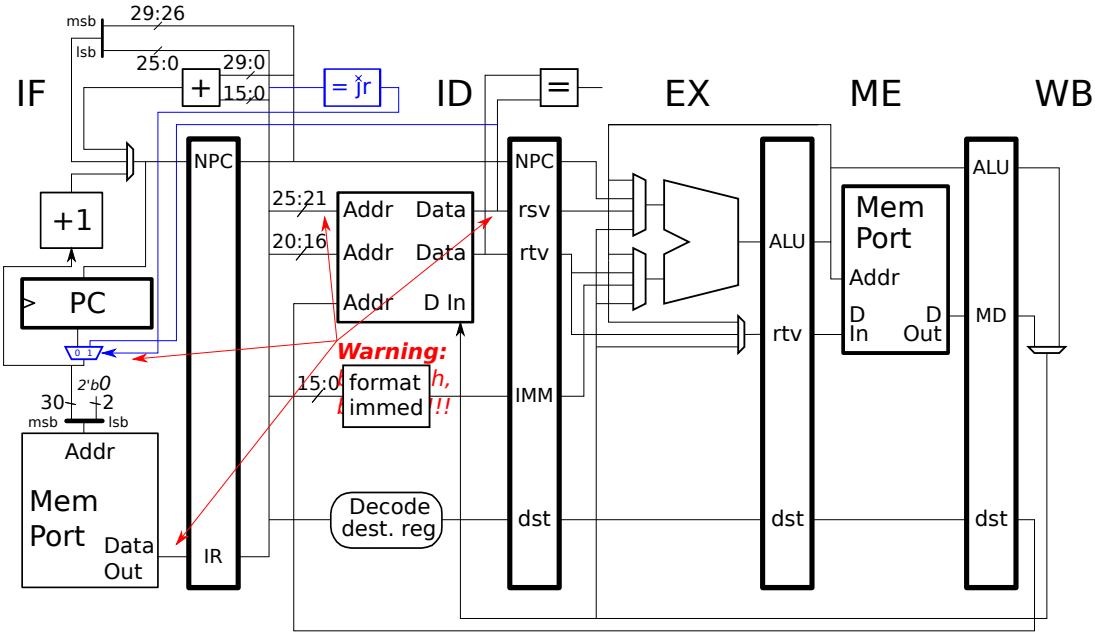
Problem 1: [7 pts] The MIPS pipeline below implements a hypothetical MIPS `jr` instruction, the hardware for `jr` is shown in blue. Don't confuse `jr` with the existing MIPS `jr` instruction.



(*a*) The diagram shows a warning in red with lots of arrows and an explanation. Alas, the explanation is covered by the `format immed` box. **Assume that the hardware for `jr` is correct.** Then what can the warning be about?

☐ Reason for warning:

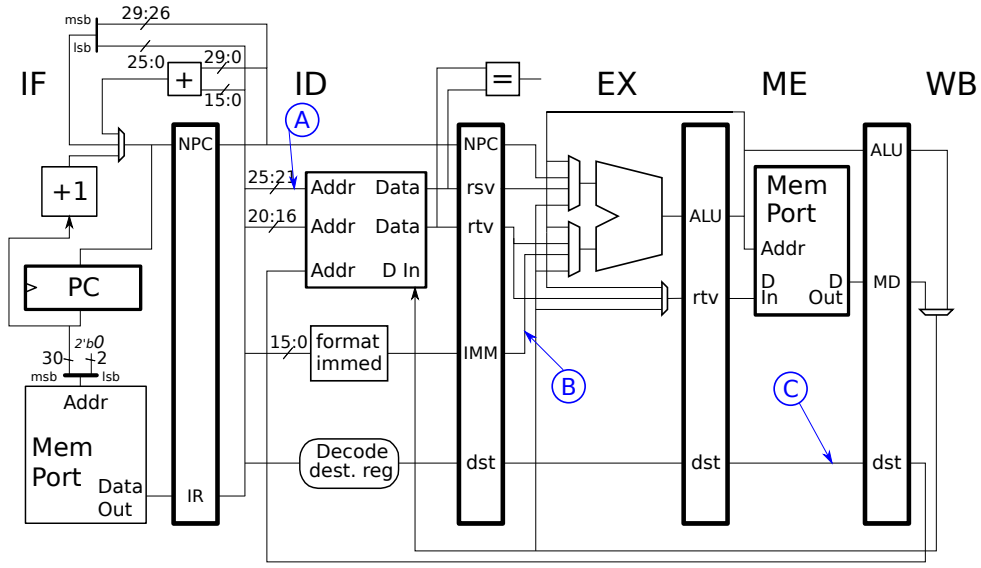
Problem 1, continued: (*b*) *Note: This part did not appear on the exam because the exam was already long enough.* There are two differences between ~~`jr`~~ and `jr`. Fragment A, below, uses `jr`. Complete Fragment B so that it uses ~~`jr`~~, making changes to account for these two differences. Fragment B must jump to the same location and perform the same computation as Fragment A. Register `r9` can be used for intermediate values. *Hint: The differences are when and where.*



☐ Complete code, or for partial credit explain two differences.

```
# Fragment A -- Uses jr. Don't modify it.
lw r1, 0(r6)
jr r1
addi r2, r2, 4
xor r3, r4, r5


# Fragment B --  ☐  Finish code below, use ~~jr~~

lw r1, 0(r6)

addi r2, r2, 4

xor r3, r4, r5
```

Problem 2: [17 pts] The code below executes on the illustrated implementation. The implementation has hardware that enables the `bne` to avoid the stall, but that hardware is not shown.



(*a*) The illustration has several circled letters pointing to wires. In the diagram below show the values on those wires on each cycle the value is used.

☐ Show values for ☐ A, ☐ B, and ☐ C, show these ☐ for each cycle used.

```
LOOP: # Cycle    0    1    2    3    4    5    6    7    8    9    10   11
 lhu r1, 8(r2)   IF   ID   EX   ME   WB
 addi r2, r2, 2       IF   ID   EX   ME   WB
 add r3, r1, r3            IF   ID   EX   ME   WB
 sw r3, 12(r6)             IF   ID   EX   ME   WB
 slt r5, r3, r4                 IF   ID   EX   ME   WB
 bne r5, r0  LOOP               IF   ID   EX   ME   WB  # No stall? Next prob.
 addi r6, r6, 4                      IF   ID   EX   ME   WB

#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11

A:

B:

C:

#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11
```

(*b*) For each dependency below highlight, on the illustration, the multiplexor input that provides the bypassed value. Also indicate the cycle in which the bypass occurs.

☐ Dependence from `lhu` to `add r3`. ☐ Cycle when bypass used.

☐ Dependence from `add r3` to `sw`. ☐ Cycle when bypass used.

☐ Dependence from `add r3` to `slt`. ☐ Cycle when bypass used.

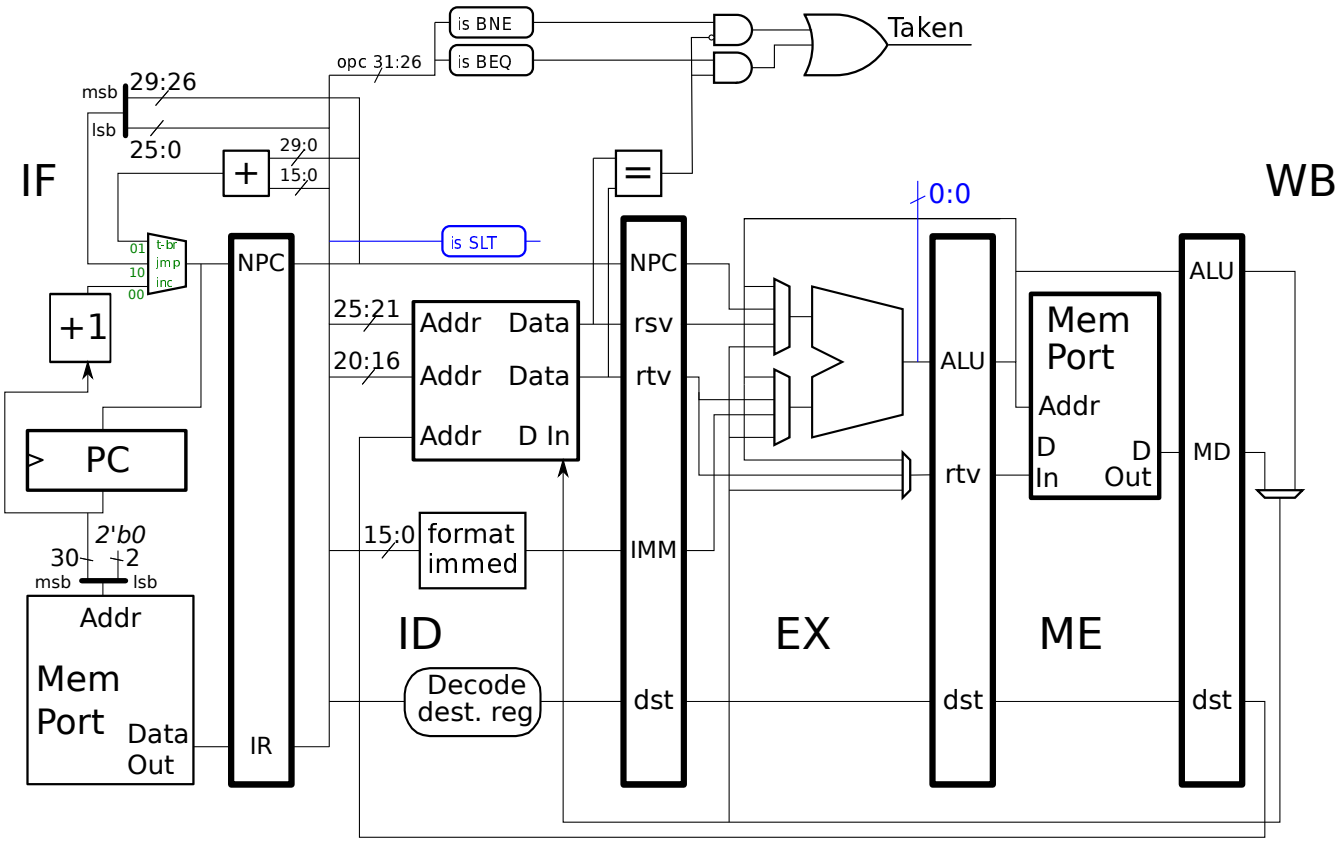
Problem 3: [27 pts] In the previous problem the `bne` did not stall despite a dependence with `slt`. Code with a similar dependence appears below. Add hardware to the implementation below that correctly sets the Taken signal for such `slt rX, rY, rZ` to `bne rX, r0` dependencies. Branches without the dependence should not be effected. Note that the result of `slt` is 0 or 1. Some useful hardware is shown in blue.

```
LOOP: # Cycle      0  1  2  3  4  5  6
 slt r5, r3, r4    IF ID EX ME WB       # Note: r5 is 0 or 1.
 bne r5, r0  LOOP     IF ID EX ME WB
 addi r6, r6, 4          IF ID EX ME WB
```

☐ Add logic to set `Taken` correctly for the dependence described above.

☐ Avoid costly solutions. No part of the solution should operate on 32 bits.

☐ Check that the second branch register is `r0`. ☐ Branches without the dependence should not be effected.



Problem 4: [12 pts] The loop below writes zeros to a range of memory.

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3  # Memory location at which to stop.
 addi r2, r2, -1
LOOP:
 sb r0, 0(r1)
 bne r1, r2, LOOP
 addi r1, r1, 1
```

(*a*) Compute the rate that it zeros memory when it runs on our bypassed 5-stage pipeline. Use an appropriate unit.

☐ Rate at which loop above copies data.

(*b*) Apply loop unrolling and make other changes so that the code writes at the rate of two bytes per clock cycle, assuming favorable values of `r1` and `r3`. State those assumptions.

☐ Show unrolled loop and make other changes for 2 byte per cycle copy.

☐ Assumption about `r1`:

☐ Assumption about `r3`:

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3

 addi r2, r2, -1
LOOP:
```

Problem 5: [12 pts] The MIPS code below adds an integer value loaded from memory to the constant π.

(*a*) Suppose no other instructions needed the value of `a0` that was loaded. Modify the code to use fewer instructions.

☐ Use fewer instructions.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        lw $a0, 0($t2)

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        mtc1 $f1, $a0
        cvt.s.w $f2, $f1
        add.s $f3, $f0, $f2
```

(*b*) Modify the code below so that it would run correctly if `a0` were a floating-point value. Also use fewer instructions.

☐ Fix code so it works correctly if `a0` is FP.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        lw $a0, 0($t2)  # a0 is FP!  ☐  Fix code below for FP a0.

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        mtc1 $f1, $a0
        cvt.s.w $f2, $f1
        add.s $f3, $f0, $f2
```

Problem 6: [25 pts] Answer each question below.

(*a*) SPARC divides the 32 integer registers an instruction can access into four groups, `%l0 to %l7`, `%g0 to %g7`, `%o0 to %o7`, and `%i0 to %i7`. The names reflect how they might be used in programs, and how SPARC's register windowing feature affects them when `save` and `restore` instructions are executed. Explain what the first letter of each group stands for. Explain what happens to the values in those registers when `save` or `restore` instruction is executed.

☐ Word that `l`, `g`, `o`, and `i` each stand for.

☐ What happens to values on a `save` or `restore`.

(*b*) Instructions like `addi r1, r1, 1` occur frequently in programs. For this add-one-to-a-register operation CISC ISAs have a specialized instruction, for example `inc r1`. MIPS lacks such an increment instruction.

☐ Why do MIPS and other RISC ISAs lack such an instruction?

☐ Why do CISC ISAs have such an instruction?

☐ What is the benefit that RISC ISAs don't realize?

(*c*) A CPU design team has an area budget that they can use for bypass paths. There's not enough area for all the bypass paths they'd like. They are simulating these design alternatives to determine which is best.

Explain the role that the compiler people have in this process.

☐ Role for compiler writers in deciding on bypass paths.

(*d*) Provide examples for the following common optimizations. Show code to which this optimization can apply and how it is optimized.

☐ Dead-code elimination. ☐ Example code and code after optimization.

☐ Constant propagation and folding. ☐ Example code and code after optimization.

(*e*) Described below are three tuning levels for C++ programs, two of which are SPECcpu tuning levels the other was just made up. Identify the one which is not a SPEC tuning level, and explain why it isn't.

Level A: *Each C++ program is compiled without optimization.*

☐ Is it a SPEC level? _____ ☐ If so, name it: _____

☐ Indicate the users that will benefit from this level, or why no one would benefit.

Level B: *Each C++ program can have its own set of optimization flags.*

☐ Is it a SPEC level? _____ ☐ If so, name it: _____

☐ Indicate the users that will benefit from this level, or why no one would benefit.

Level C: *All C++ programs must be compiled with the same set of optimization flags.*

☐ Is it a SPEC level? _____ ☐ If so, name it: _____

☐ Indicate the users that will benefit from this level, or why no one would benefit.

Name _____ *Formatted For 2-Sided Printing*

Computer Architecture
LSU EE 4720
Final Examination

1 May 2019, 12:30–14:30 CDT

Problem 1	_____	(22 pts)
Problem 2	_____	(22 pts)
Problem 3	_____	(21 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (22 pts) Notice that the execution of the code fragment below suffers two stalls when executing on our 2-way superscalar MIPS implementation. The `add` stalls due to a dependence with `or` and the `sw` stalls due to a dependence with `add`. The MIPS implementation has three unconnected logic blocks that may be useful. Each must be connected to the opcode and func of the instruction in the appropriate slot. The output of the [=or] is 1 if the instruction is an `or`. The output of [uses rs] is 1 if the instruction uses the `rs` register as a source, likewise for [uses rt].

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID ----> EX ME WB
 addi r6, r6, 8      IF -> ID ----> EX ME WB
```

(*a*) In the execution below the `sw` no longer stalls for `r3`. Add a bypass path that can be used by `sw` to get the `r3` value **in the execution below** but not for other cases.

☐ Add bypass path for `r3` so `sw` executes as shown. ☐ Label the path "Part a", and **do not add** unneeded bypass paths.

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID EX ME WB
 addi r6, r6, 8      IF -> ID EX ME WB
```

(*b*) The `add` stalls due to the dependence with `or` carried by `r1`. Add control logic that detects such a dependence and connect it to the Stall ID OR gate at the lower right. The output of the logic should be `1` for any true dependence between two instructions in a group.

☐ Provide a stall signal when there is a dependence between the two instructions in `ID`.

(*c*) Notice that because the second operand is `r0`, the `or` just copies the value in `r2` to `r1`. Therefore the `add` could have used `r2` instead of `r1` and avoided the stall. Design hardware to perform such *substitutions*. The hardware, including control logic, should detect when an `or` is used as a copy (as above) and if so avoid the stall and deliver the correct source operand to the slot-1 instruction.

```
# Cycle           0  1  2  3  4  5
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID EX ME WB
 sw r3, 4(r6)        IF ID EX ME WB
 addi r6, r6, 8      IF ID EX ME WB
```

☐ Detect the substitution opportunity and ☐ suppress the Stall ID signal (from the previous part).

☐ Make sure the slot-1 instruction uses the correct value ☐ and that both instructions execute correctly.

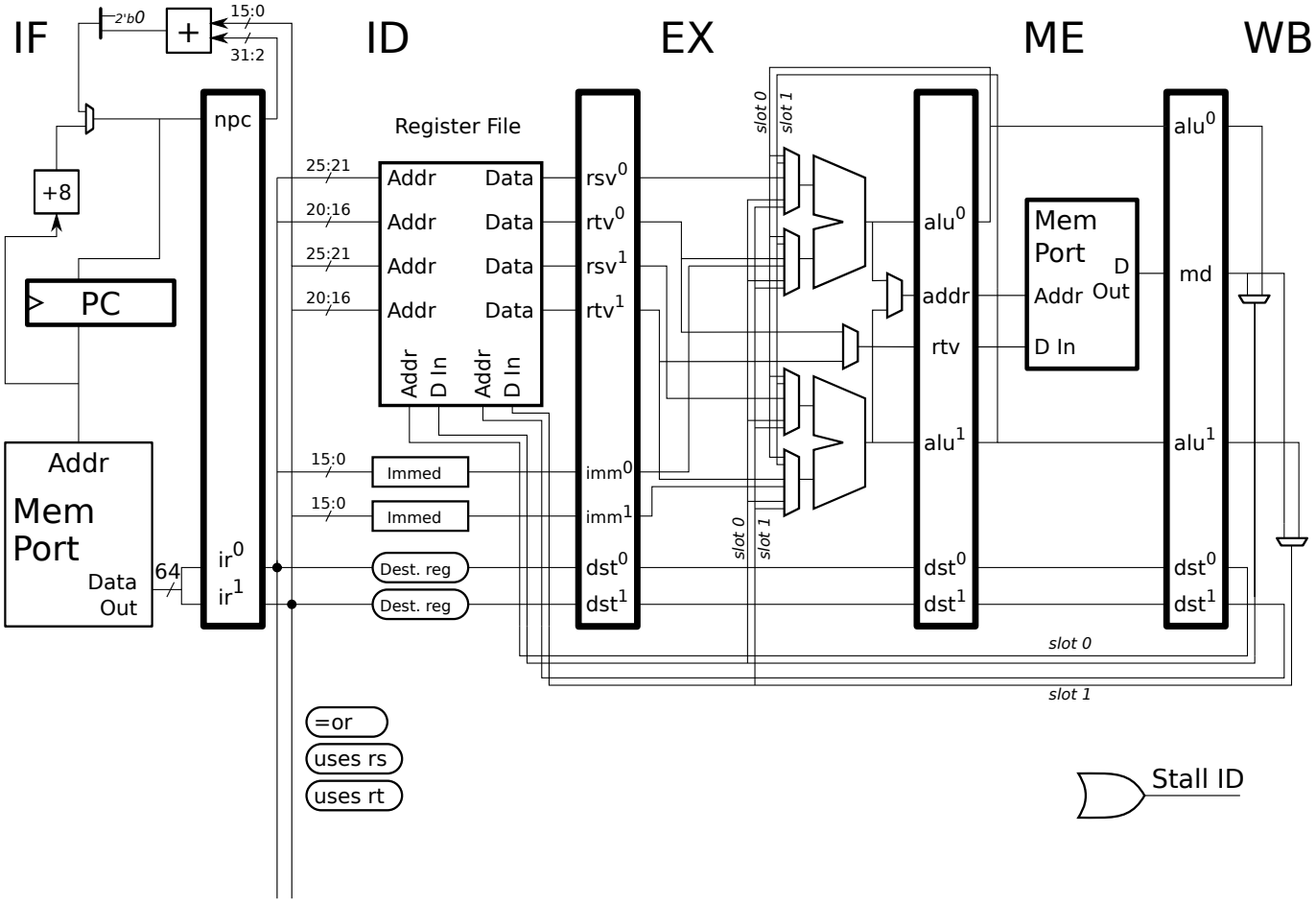
☐ Of course, pay attention to cost. Nothing added for this problem should touch 32 bits.

(*d*) *The following is a bonus question that did not appear on the original exam. Bonus for whom you ask? Definitely a bonus for those who took the class in the Spring 2019 semester and took a look at the posted*

exam. Those (you) will have an opportunity to make connections between concepts learned in the class and that will provide a deeper understanding and longer retention. Yes, the substitution hardware eliminates a stall. Suppose that `r2` had to be copied into `r1`. Provide an argument that substitution hardware is a waste of resources, illustrate with an example. Provide another argument—also with an example—that substitution hardware eliminates a stall that cannot be eliminated in another way. Whether substitution is a good idea will depend on whether the example illustrating its utility is representative of realistically compiled actual code.

☐ Argument against substitution hardware. ☐ Code example.

☐ Argument for substitution hardware. ☐ Code example.



Problem 2: (22 pts) Appearing below is our MIPS FP pipeline with the comparison units added.

(*a*) Show the execution of the following fragment on this hardware.

☐ Show execution up to second fetch of `lwc1`. ☐ Pay attention to dependencies, including the FP condition.

```
# Cycle:            0
LOOP:
 lwc1 f1, 0(r1)

 add.s f2, f1, f2

 add.s f4, f1, f3

 c.lt.s f2, f6

 bc1t LOOP # Taken

 addi r1, r1, 8

 swc1 f2, 4(r1)

 and r1, r1, r9
```

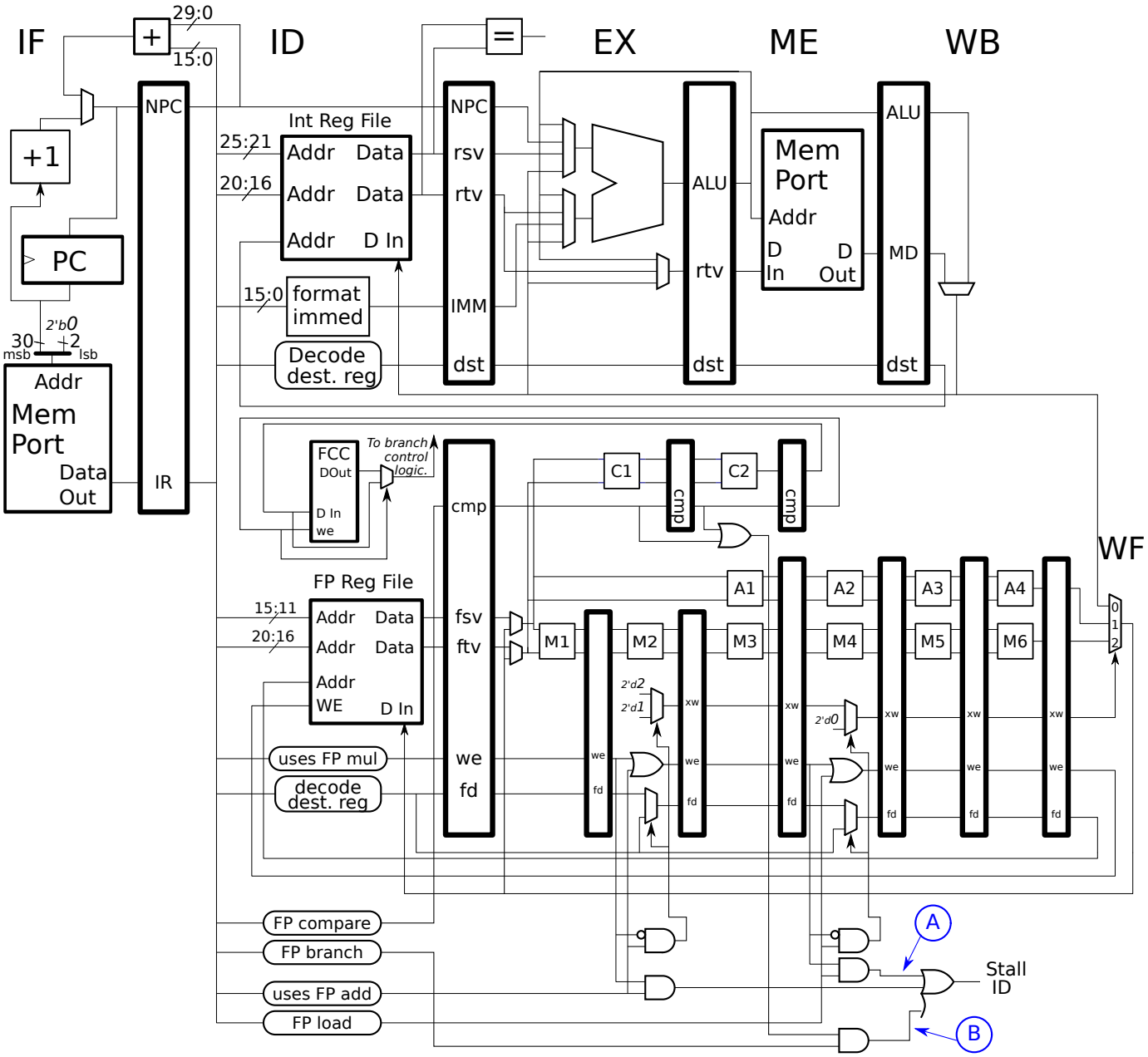
(*b*) Notice that there are two circled letters (in blue) in the lower part of the diagram. For each letter provide a code fragment that causes the labeled wire to go to logic 1.

☐ Code fragment that makes A logic 1.

☐ Show its execution and ☐ indicate cycle at which A is 1.

☐ Code fragment that makes B logic 1.

☐ Show its execution and ☐ indicate cycle at which B is 1.


IF
ID
EX
ME
WB
WF
29:0
15:0
+
=
+1
PC
NPC
IR
Addr
Mem
Port
Data
Out
2'b0
30
msb
2
lsb
Int Reg File
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
FCC
DOut
D In
we
To branch
control
logic.
cmp
C1
C2
cmp
cmp
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr
WE D In
fsv
ftv
we
fd
uses FP mul
decode
dest. reg
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
FP compare
FP branch
uses FP add
FP load
A
B
Stall
ID

Problem 3: (21 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B1` has a repeating pattern, two repetitions are shown. Branch `B2` repeatedly and randomly emits three sequences, `a`, `b`, and `c`. Sequence `a` is `NT`, sequence `b` is `NNNTT` (five outcomes), and sequence `c` is `NNNN NTTT` (eight outcomes). After finishing one sequence, a new one is started. Sequence `a` is chosen with probability .4, sequence `b` with probability .5, and `c` with probability .1.

Here are some examples of `B2` outcomes, with spaces placed between the sequences for clarity. Example 1: `NT NNNTT NT NNNNNTTT` (that's `a`, `b`, `a`, `c`). Example 2: `NNNNNTTT NT NT NNNTT NNNTT` (that's `c`, `a`, `a`, `b`, `b`).

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   T  N  N  N  N  T  T    T  N  N  N  N  T  T

B2:    (a, p=.4): NT    (b, p=.5): NNNT T    (c, p=.1): NNNN NTTT
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the local predictor on branch `B1`?

☐ What is the accuracy of the local predictor on branch `B2`?

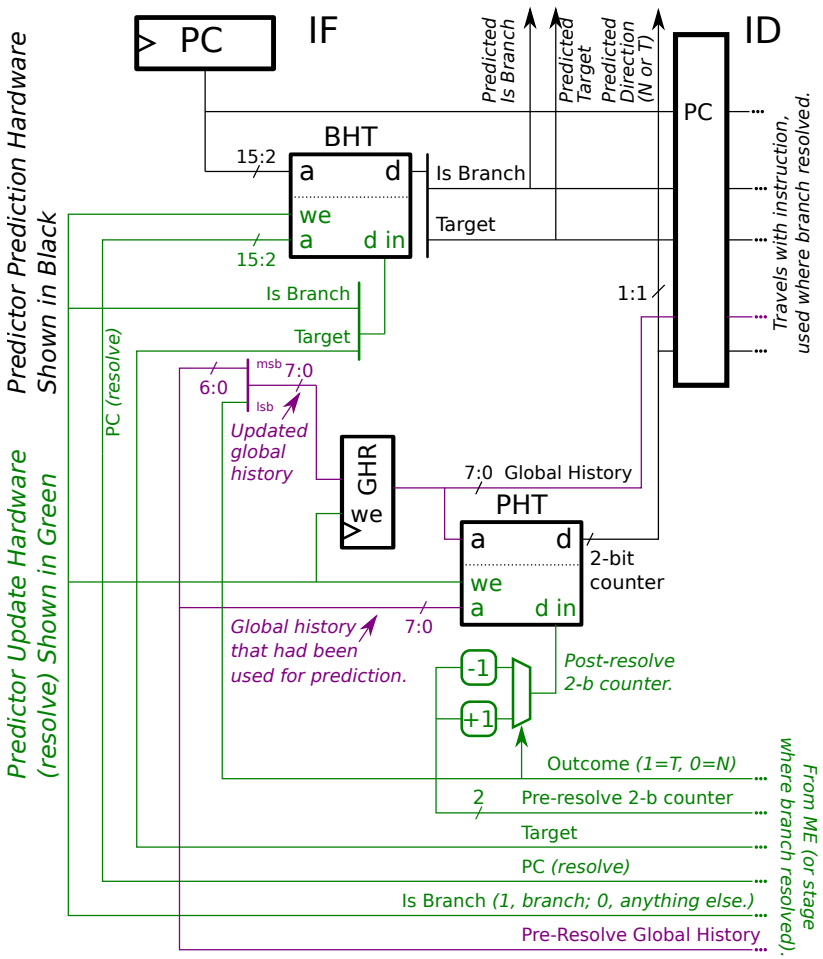
☐ What is the accuracy of the bimodal predictor on branch `B2`?

(*b*) Appearing below is a diagram of our global predictor. Notice that the GHR is not updated until the branch resolves. Modify the predictor so that the GHR is updated when the branch is being predicted (in `IF`) using the *predicted* outcome. When the branch resolves check whether the prediction was correct, and if not (if it was mispredicted) write the correct history into the GHR.

The following is interesting background material omitted from the original exam. The importance of updating the GHR using the predicted outcome increases with the number of post-branch instructions that are in the pipeline at the time a branch resolves. Consider our five-stage pipeline with branches resolving in `ME`. In that case there are just three post-branch instructions. For an 8-way superscalar pipeline there would be $3 \times 8 = 24$ instructions. One or more of those 24 instructions could itself be a branch. In the unmodified design below those branches would have been predicted with a GHR that lacked the outcome of the resolving branch and those that followed. The problem is much greater in dynamically scheduled systems where over 100 instructions can be *in flight*. For that reason global-like predictors in dynamically scheduled systems use designs like the one requested for this problem.

☐ Add hardware to detect whether the resolving branch has been mispredicted.

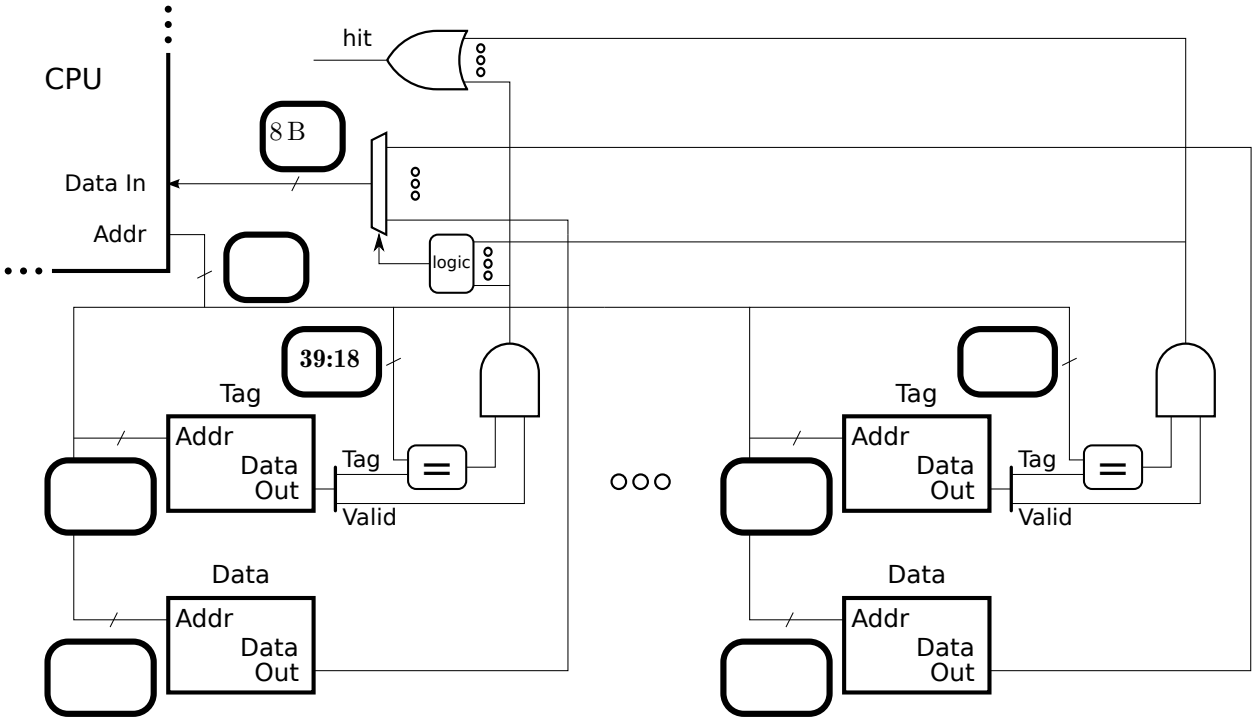
☐ During prediction write GHR based on prediction, ☐ during resolve apply corrected GHR if branch mispredicted.



Problem 4: (10 pts) The diagram below is for a 4 MiB set-associative cache with a line size of 32 B. The character size is the usual 8 bits. Other information about the cache can be deduced using hints in the diagram. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $32 = 2^5$.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: ☐ ☐ ☐

0

☐ Associativity:

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

Address: ☐ ☐ ☐

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 B (which is 2^6 B). The code fragment starts with the cache empty; consider only accesses to the array.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int ILIMIT = 1 << 11;    // = 2^11

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 5: (25 pts) Answer each question below.

(*a*) Appearing below are simple C routines and corresponding MIPS assembler code. C variable names match the MIPS registers to which they were assigned. Register `v0` is used for the return value. The first C routine, `proc1`, operates on 32-bit signed integers. Further below are two similar C routines, `proc2` and `proc3`, each followed by the MIPS routine written for `proc1`—which is wrong because the MIPS routine is only correct for `proc1`. Rewrite those MIPS routines for `proc2` and `proc3`. Note that `int16_t` is a signed 16-bit integer and `uint8_t` is an unsigned 8-bit integer.

```
int32_t proc1(int32_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Code below is correct for proc1.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☐ Modify MIPS code for `proc2`. Pay attention to ☐ size and ☐ sign. ☐ Eliminate any unneeded instructions.

```
int16_t proc2(int16_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc2.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☐ Modify MIPS code for `proc2`. Pay attention to ☐ size and ☐ sign. ☐ Eliminate any unneeded instructions.

```
uint8_t proc3(uint8_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc3.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

(*b*) The statement below is based on a lack of understanding of how compilers work. Explain the misunderstanding and otherwise correct the statement.

> *It takes a great deal of effort to write a correct and effective compiler optimizer. Therefore optimizers are written for popular high-level languages such as C++11 but not for less popular languages such as COBOL.*

☐ The misunderstanding about compilers is:

☐ How does that change the conclusion about which languages get better optimization?

(*c*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock.

☐ Compute the peak execution rate in units of instructions per second of ☐ Chip A and ☐ Chip B.

☐ Why would Chip A run faster on simple code, such as the routines used in the homework assignments?

☐ Which chip might be less expensive? Explain.

(*d*) Answer the following questions about ISAs.

☐ Implementations of VLIW ISAs are supposed to be less costly and have higher performance than superscalar implementations of conventional ISAs. Is Intel Itanium a good example of that? Explain

☐ What important concept came out of the development IBM System/360?

☐ VAX is a good example of which ISA type?

☐ True or false: IA-32 (a.k.a. x86) was widely adopted because of its elegant design? Explain.

(*e*) The SPECcpu benchmarks can be run at two tuning levels, *base* and *peak*. Base scores are useful to those running software developed using typical practices.

☐ What kind of computer buyers should use peak scores?

☐ How do the SPEC rules for preparing base and peak runs differ?

8 Spring 2018

Name _____

Computer Architecture
EE 4720
Midterm Examination

Monday, 19 March 2018, 9:30–10:20 CDT

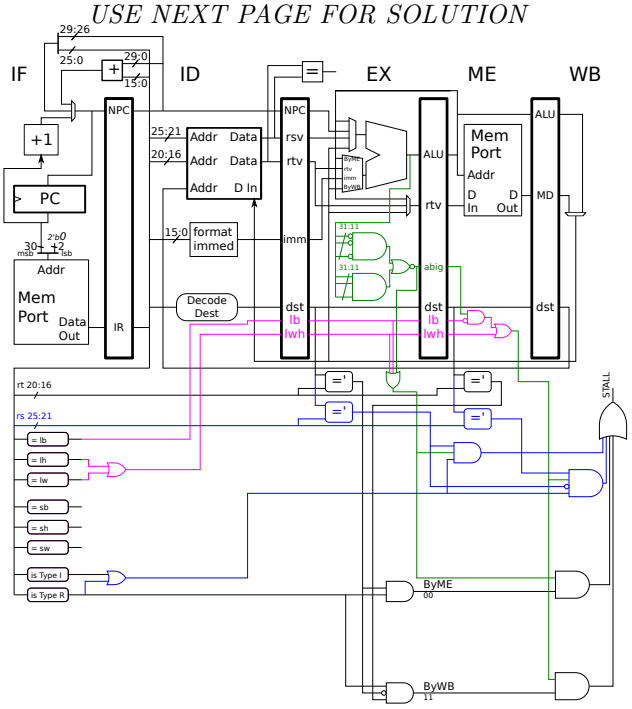
Problem 1	_____	(25 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below is the solution to Homework 4, in which additional control logic is added for the 12-bit bypass paths. The illustrated hardware generates stall signals for a load/use dependence and for cases in which the value that needs to be bypassed is unknown or too wide for the 12-bit bypass paths.

USE NEXT PAGE FOR SOLUTION



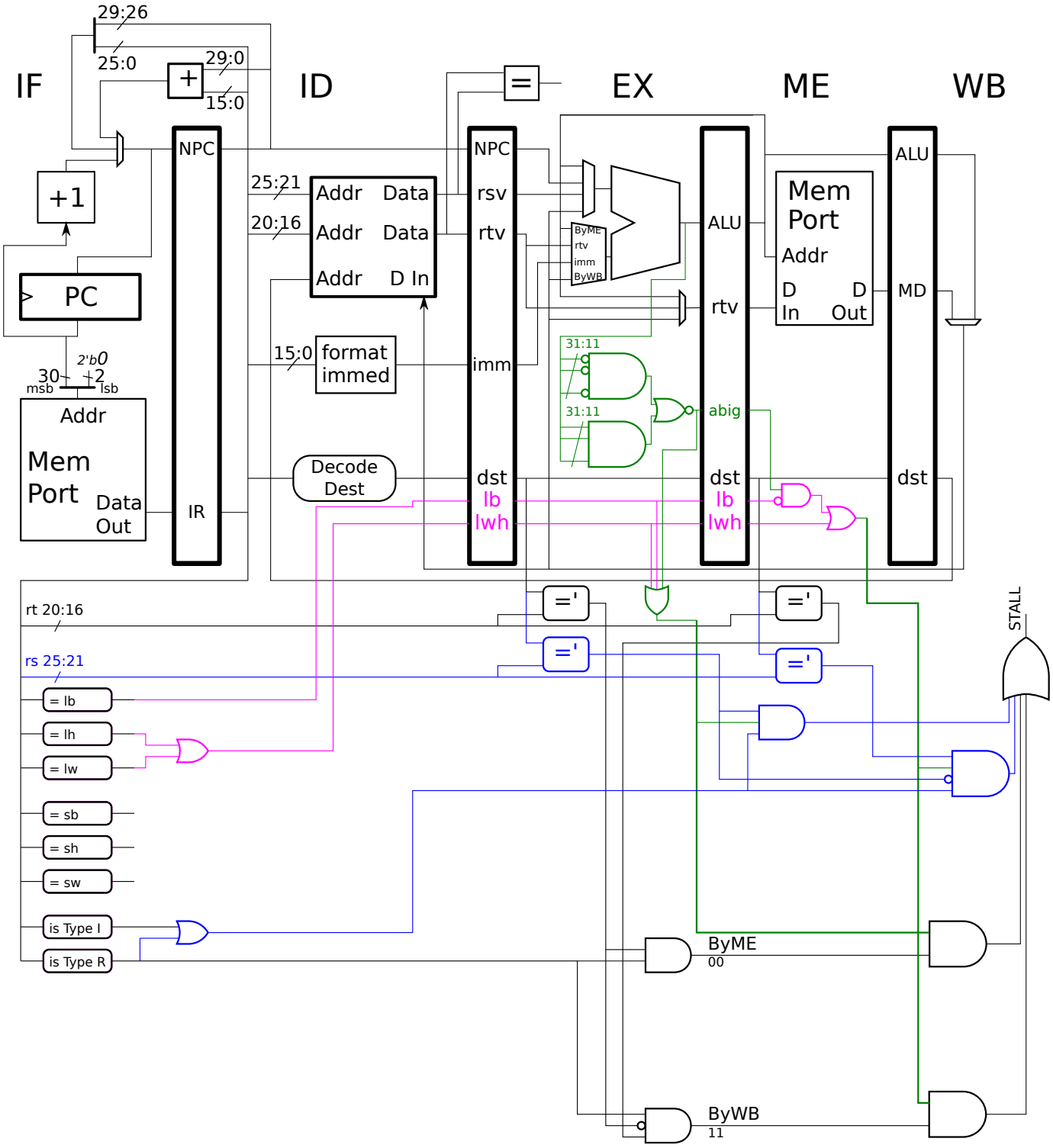
USE NEXT PAGE FOR SOLUTION

(*a*) Suppose, on further consideration, it was decided that full-sized, 32-bit bypass paths to the upper ALU mux were needed. Elsewhere 12-bit bypass paths would be retained. Modify the hardware so that a stall signal would no longer be generated for such too-big values to the upper ALU mux. Do so **without** affecting stalls for the 12-bit bypass paths to the lower ALU mux and without affecting stalls for load/use dependencies.

```
# Should not stall anymore, regardless of size of r1.
add $r1, $r2, $r3
lw $r4, 0($r1)

# Should still stall.
lw $r5, 0($r6)
addi $r7, $r5, 9

# Should still stall if r1 too big for 12-bit bypasses.
add $r1, $r2, $r3
sub $r5, $r6, $r1
```

☐ Remove hardware generating stall due to values too large for upper ALU bypass paths.

☐ **Do not** change stalls for load/use cases and bypasses to lower ALU mux.

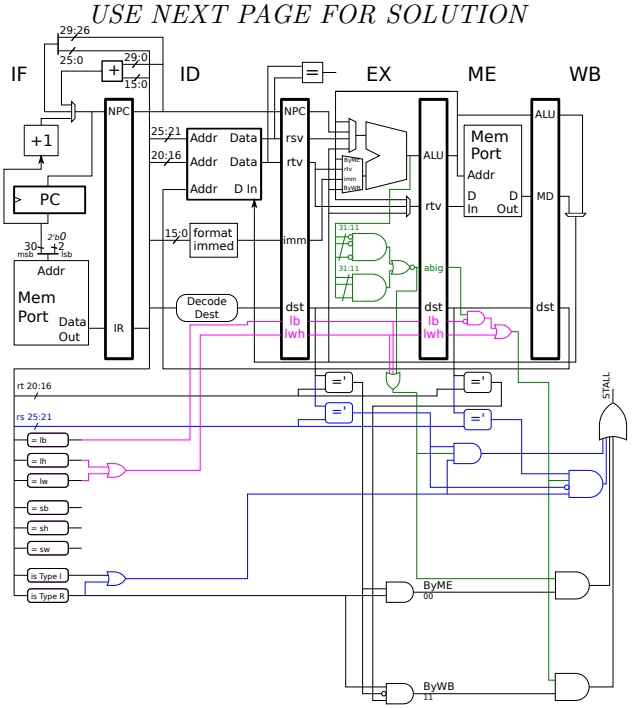


Problem 1, continued:

(*b*) Suppose that the bypass paths to the `EX`-stage `rtv` mux were also just 12 bits wide. Modify the control logic on the next page so that a stall signal would be generated when appropriate for store instructions.

Note that a `lb` produces a value small enough for the 12-bit bypass paths and that the store value needed by a `sb` is always small enough for the 12-bit bypass paths. See the examples below.

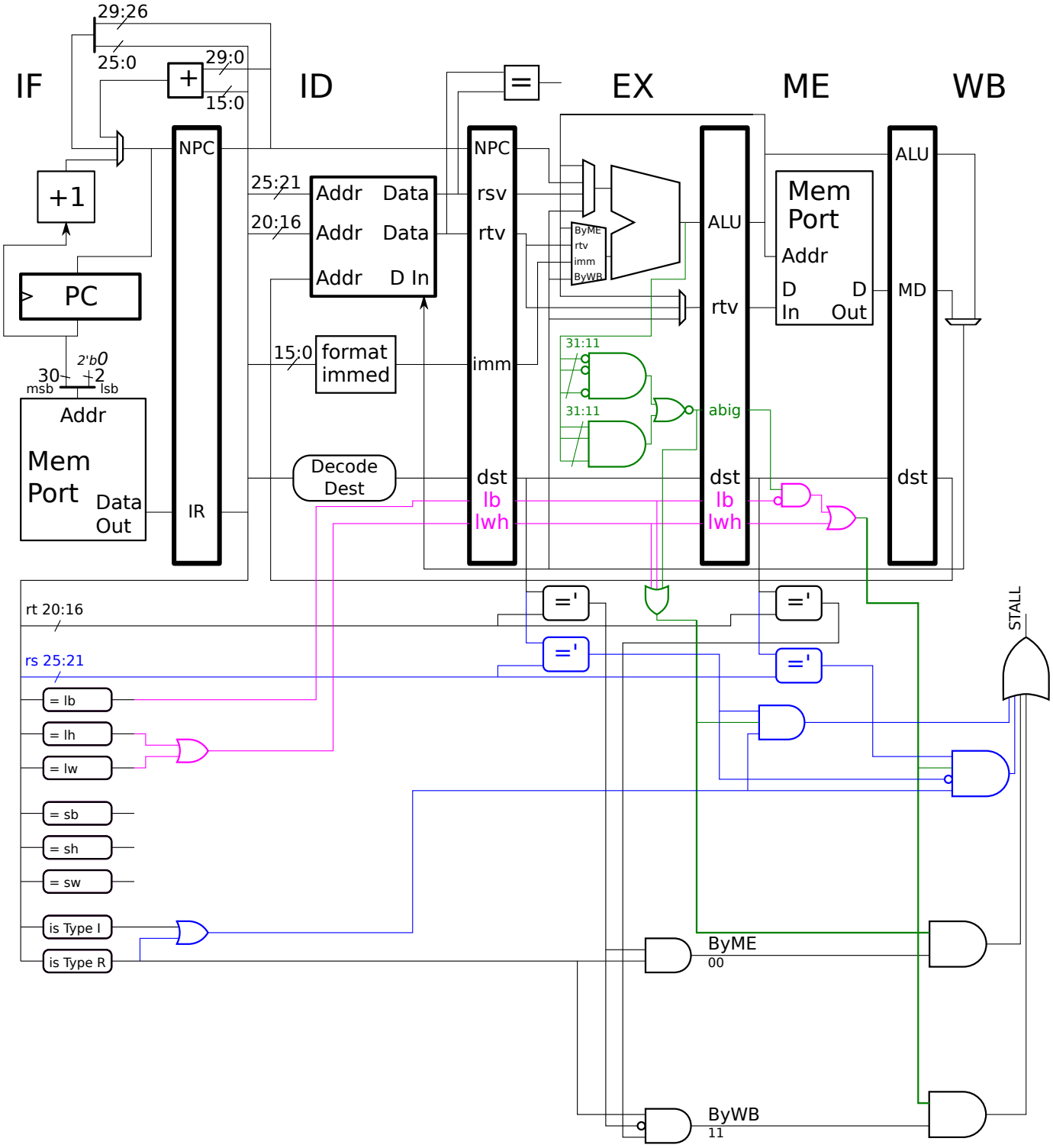
USE NEXT PAGE FOR SOLUTION



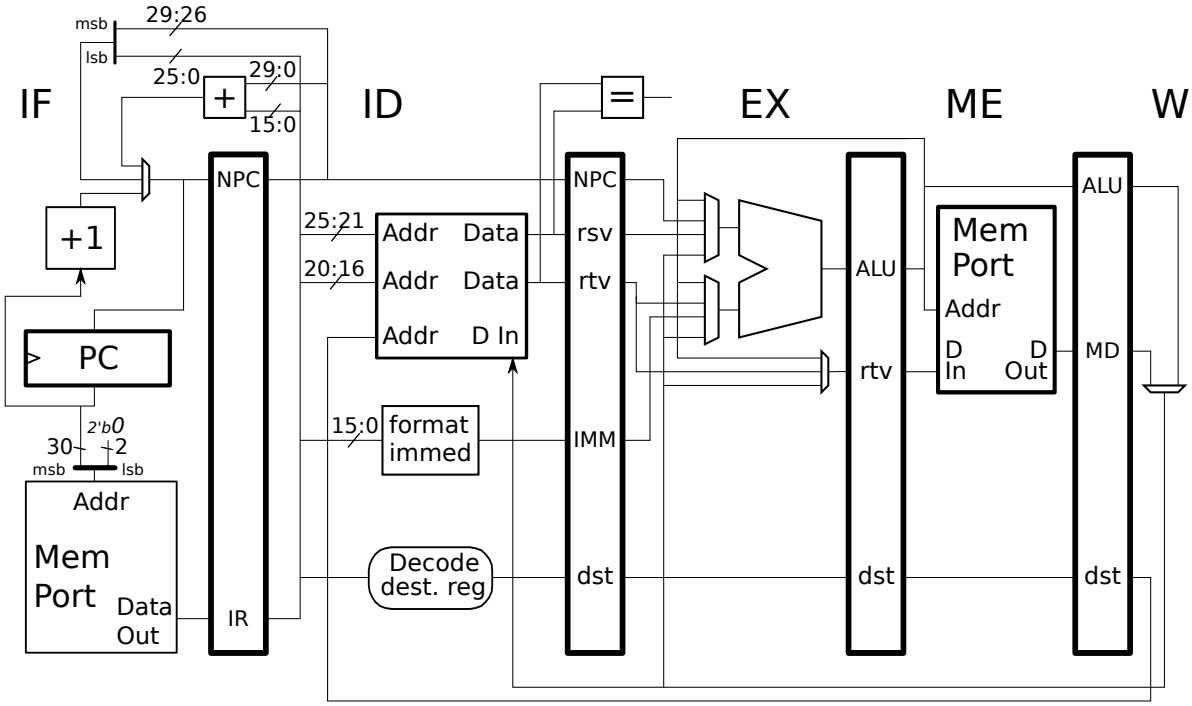
USE NEXT PAGE FOR SOLUTION

```
# sb should not stall for this dependency.
add $r1, $r2, $r3  IF ID EX ME WB
sb $r1, 0($r4)        IF ID EX ME WB

# sw should stall for one cycle.
lb $r1, 0($r5)     IF ID EX ME WB
sw $r1, 0($r4)        IF ID -> EX ME WB

# sh should stall only if r1 is too large.
add $r1, $r2, $r3  IF ID EX ME WB
sh $r1, 0($r4)        IF ID ----> EX ME WB
```

☐ Generate stall for unbypassable value from prior instructions to store value (not store address). ☐ Consider store size and any load size. (See examples on previous page.)



Problem 2: [20 pts] Answer the following questions about two versions of our bypassed MIPS implementation.



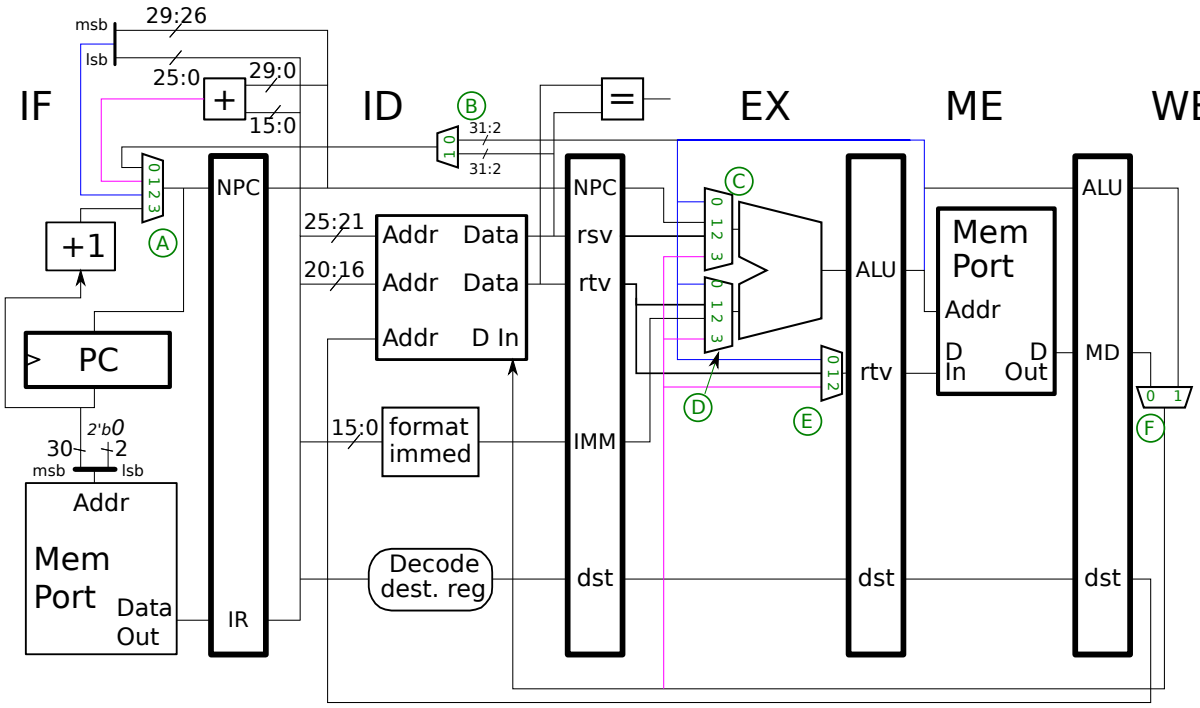
(*a*) Show the execution of the code below on the implementation illustrated above when the branch is taken.

☐ Show Execution.

☐ Check the code for dependencies.

```
lw r2, 0(r4)

addi r1, r2, 3

bne  r1, r3  TARG

ori r1, r9, 10

andi r11, r1, 14

TARG:
xor r20, r11, r22
```

(*b*) Each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.



☑ Use C0 to carry `r1`, elsewhere `r1` not used.

```
# Cycle          0  1  2  3  4  5   SAMPLE SOLUTION
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☐ Use D3 to carry `r1`, elsewhere `r1` not used.

☐ Use E0 to carry `r1`, elsewhere `r1` not used.

☐ Use A1.

☐ Use B0.

☐ Use C1.

Problem 3: [15 pts] The floating point code fragment below computes `f2 = f3 * f0 + f1`, where `f3` is a call argument and `f0` and `f1` are loaded from a table. A total of eight instructions are used to load the constants into `f0` and `f1`. Re-write the code so that fewer instructions are used. It is possible to load both registers using a total of two instructions.

☐ Load constants into `f0` and `f1` using two or three instructions.

☐ The constants must be loaded from the table.

```
        .data
famous_constants: # 0x10010300
        .float 2.718282818
        .float 3.141592654
        .text
pie:
        # Load f0 with first element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x300
        lw $t1, 0($t0)
        mtc1 $t1, $f0

        # Load f1 with second element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x304
        lw $t1, 0($t0)
        mtc1 $t1, $f1

        # Don't modify the code below this line.
        mul.s $f2, $f3, $f0
        add.s $f2, $f2, $f1
```

Problem 4: [40 pts] Answer each question below.

(*a*) In class we said that MIPS-I lacks an instruction like `bgt` (branch greater-than) because the magnitude comparison would take a little too long. MIPS-I does have `beq` and `bne` instructions that compare two registers. However, SPARC v8 has a `bgt` instruction, but it is done in such a way that here is no risk of critical path impact.

☐ How is the actual SPARC `bgt` different than a hypothetical MIPS `bgt`?

☐ How does that difference avoid critical path impact in resolve-in-ID implementations?

☐ Explain why a `bgt r1, r2, TARG` would not have a big critical path impact if it were resolved in EX.

(*b*) In the MIPS `add` instruction the `sa` field must have a value of zero. Consider a future version of MIPS in which the `sa` field would hold a scale factor, `s`. The result of the add would be `rsv + rtv * s`. Suppose that analysis of users' programs found that such an instruction would be **very useful** and that it could **easily be implemented in hardware**. Should the `add` be extended in that way? If not, suggest another way of providing the scaled add.

☐ Should `add` be extended to compute `rsv + rtv*s`?

☐ Explain a possible objection and suggest an alternative way of including the instruction.

(*c*) The MIPS-I assembly instruction below is invalid. Explain why and replace it with one that correctly adds two double-precision values.

```
add.d $f0, $f1, $f2
```

(*d*) What is the difference between a dependency and a hazard?

☐ The difference between a dependency and a hazard is:

(*e*) Identify the type of dependence between each pair of instructions below, and indicate the corresponding hazard.

☐ The type of dependence is:_____. The corresponding hazard is:_____

```
add r1, r2, r3
sub r1, r5, r6
```

☐ The type of dependence is:_____. The corresponding hazard is:_____

```
add r1, r2, r3
sub r4, r1, r6
```

☐ The type of dependence is:_____. The corresponding hazard is:_____

```
add r1, r4, r3
sub r4, r2, r6
```

(*f*) In class we said that the lifetime of an ISA can be decades and so it must be carefully designed to take into account current and future implementation technologies. Is IA-32 (80x86) a good example of this rule? Explain.

☐ IA-32 ◯ *is* ◯ *is not* a good example of this rule because ….

(*g*) Indicate the most appropriate ISA family for each ISA below.

☐ Intel 64 is considered ◯ *RISC* ◯ *CISC* ◯ *VLIW*

☐ Itanium is considered ◯ *RISC* ◯ *CISC* ◯ *VLIW*

☐ MIPS is considered ◯ *RISC* ◯ *CISC* ◯ *VLIW*

☐ SPARC is considered ◯ *RISC* ◯ *CISC* ◯ *VLIW*

☐ VAX is considered ◯ *RISC* ◯ *CISC* ◯ *VLIW*

Name _____ *Formatted For 2-Sided Printing*

Computer Architecture
EE 4720
Final Examination

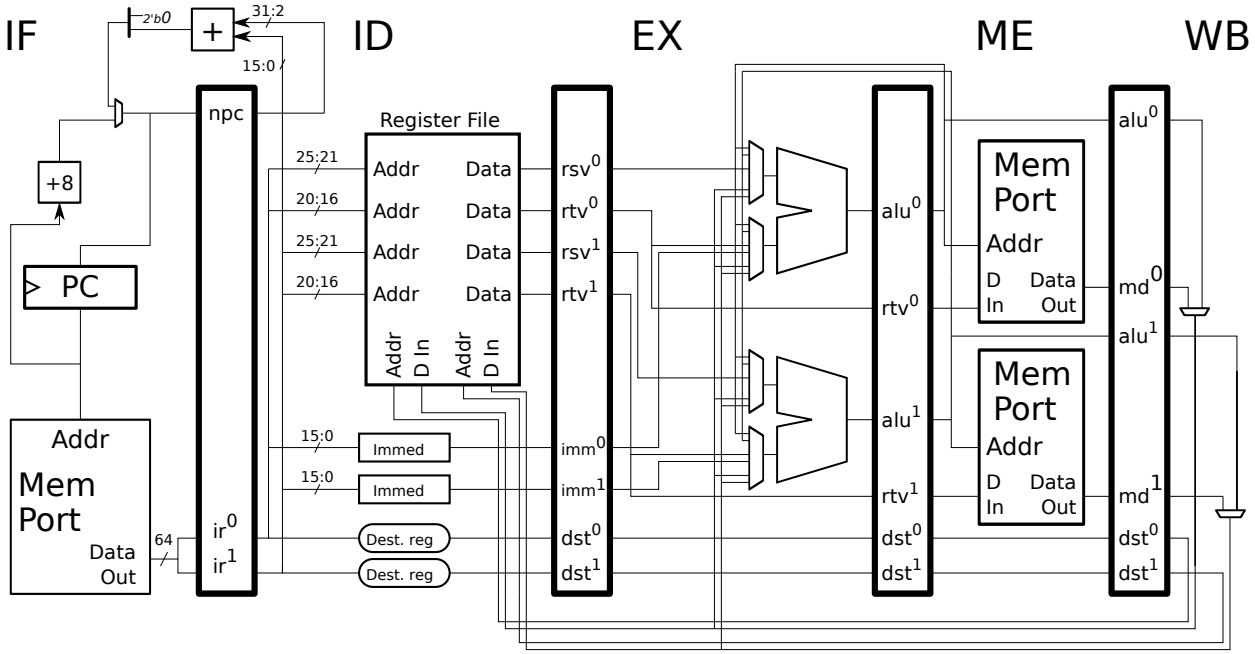
2 May 2018, 15:00–17:00 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(16 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(8 pts)
Problem 7	_____	(21 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (15 pts) Appearing below is the 2-way superscalar implementation used in class. As we usually assume, fetch groups are aligned and stalls must keep instructions within a stage in order.



(*a*) Show the execution of the code below on the implementation above.

☐ Show execution. ☐ Check for dependencies!!

```
LINE1: # Address of the first lw insn below is 0x1000

lw r1, 0(r2)

lw r3, 0(r1)

lw r4, 4(r1)

lw r5, 8(r1)

sw r5, 12(r1)

sw r4, 16(r1)
```

(*b*) Show the execution of the code fragment below on the illustrated implementation.

☐ Show execution and ☐ check for dependencies here too.

☐ Don't overlook the fact that the branch is taken.

☐ Pay attention to fetch groups and the aligned fetch restriction.

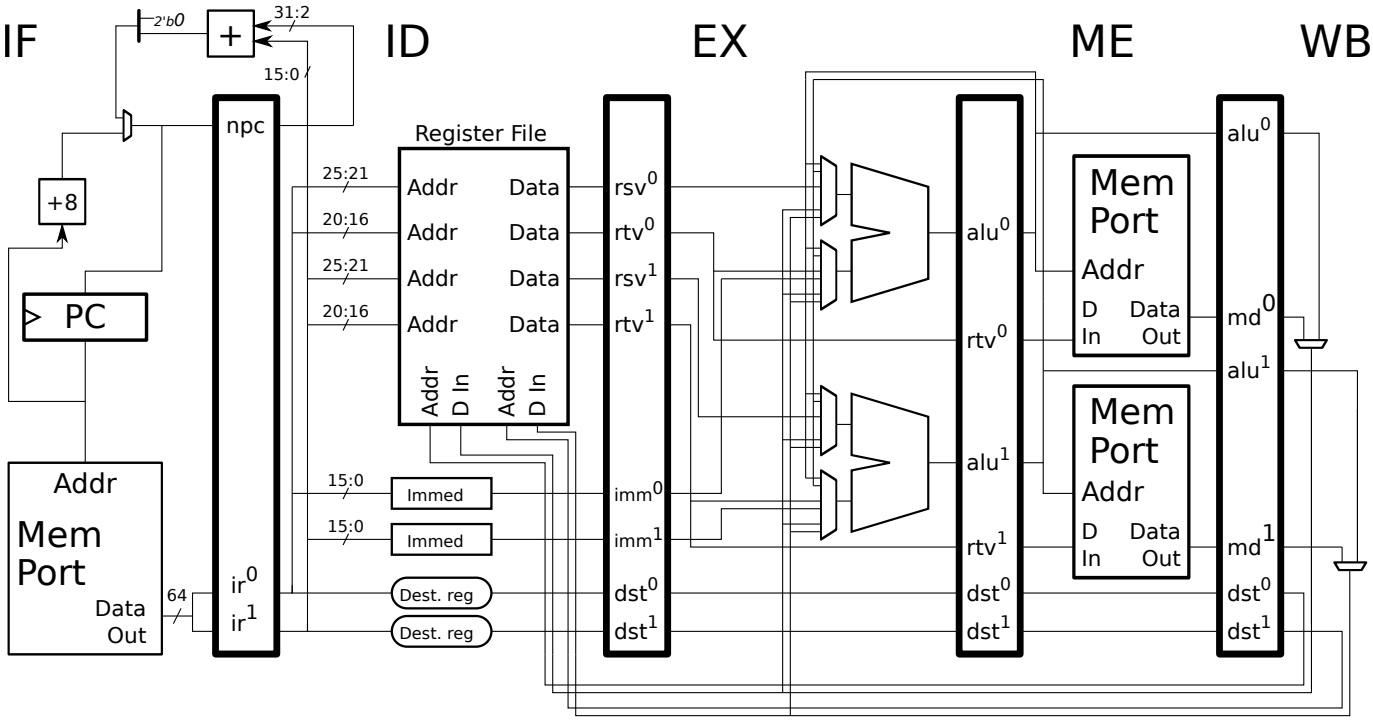
```
 # Address of beq is 0x1004

 beq r1, r1, SKIP1

 lw r2, 0(r3)

 sw r4, 0(r3)

 addi r3, r3, 4

SKIP1:  # Address of andi r2 is 0x2008

 andi r2, r2, 0xfff

 andi r6, r2, 0xff0

 add r7, r2, r6

 sub r8, r2, r6
```

(*c*) Appearing below is again our 2-way superscalar MIPS. Notice that the branch hardware shown can only provide the target for a branch in slot 1. Add hardware for providing the branch target of a branch in slot 0. **Do not** add hardware for checking the branch condition. **Do not** add control logic.

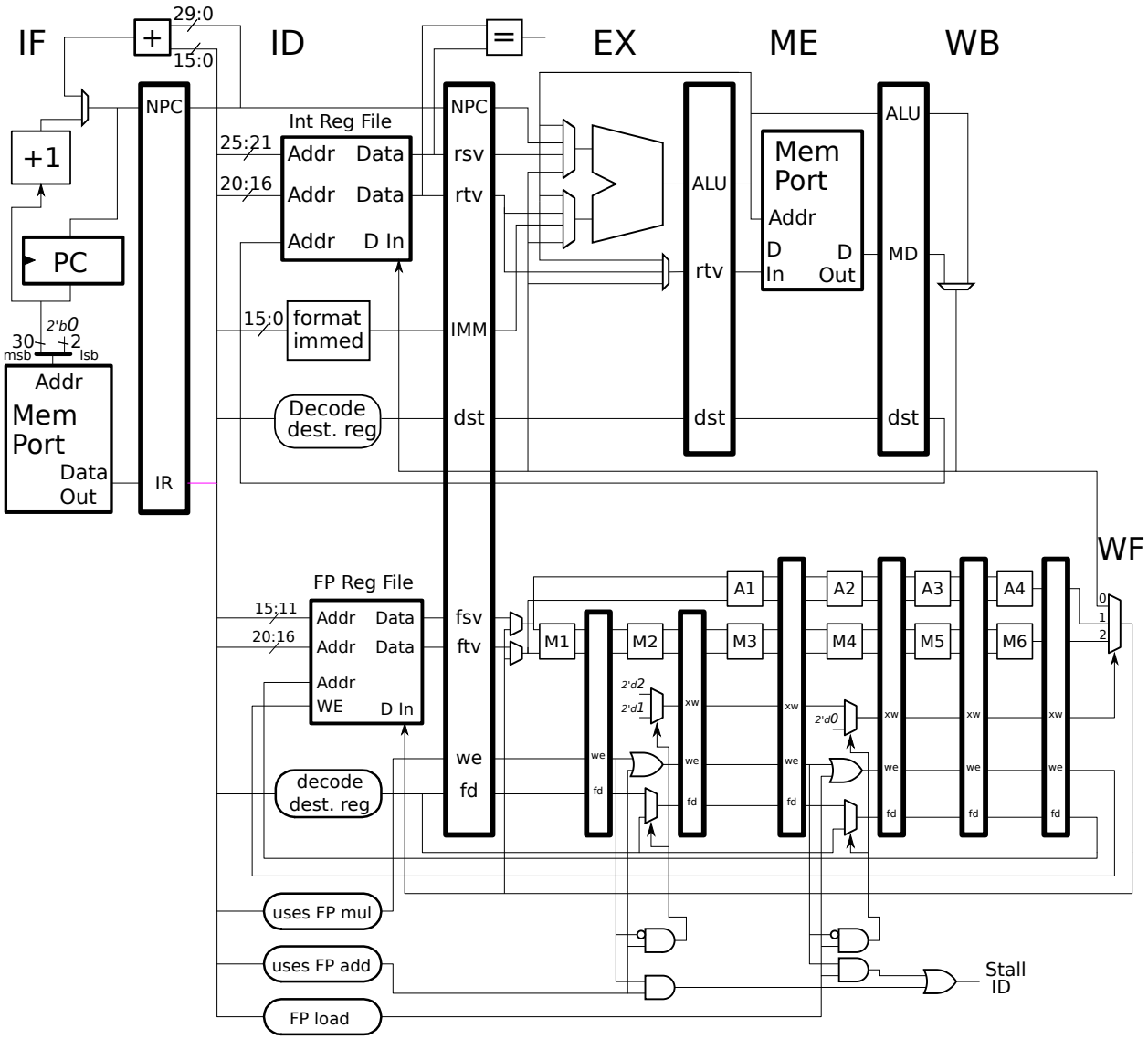
☐ Add hardware for a slot-0 branch.

☐ **Pay attention to cost**.

☐ Be sure the hardware computes the correct target address.


IF
ID
EX
ME
WB
2'b0
+
31:2
15:0
npc
+8
PC
Addr
Mem
Port
Data
Out
64
ir0
ir1
Register File
25:21
20:16
25:21
20:16
Addr
Data
Addr
D In
15:0
Immed
Dest. reg
rsv0
rtv0
rsv1
rtv1
imm0
imm1
dst0
dst1
alu0
rtv0
alu1
rtv1
dst0
dst1
Mem
Port
Addr
D
In
Data
Out
alu0
md0
alu1
md1
dst0
dst1

Problem 2: (10 pts) Appearing below is the execution of a bit more than two iterations of a loop on the illustrated MIPS implementation. The execution shows the use of a two-stage FP compare unit, `C1-C2`, by the `c.lt.s` instruction, but the unit isn't shown.




```
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2  IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3        IF ID -------------> C1 C2 WF
bc1f LOOP               IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                          IF ----> ID A1 A2 A3 A4 WF
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2                             IF ID -------> M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3                                   IF -------> ID -------------> C1 C2 WF
bc1f LOOP                                                   IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                                                              IF ----> ID A1 A2 A3 A4 WF
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2                                                                    IF ID -------> M1 M2 M3 M4 M5 M6 WF
```

(*a*) Compute the CPI of the execution of the loop above for a large number of iterations.

☐ Compute the CPI.

☐ Clearly show how the time for an iteration was determined, perhaps using the pipeline diagram.

(*b*) Reschedule the instructions to reduce the time needed to execute a large number of iterations of the loop. Add a `nop` if that helps. A correct solution will still have many stalls.

☐ Re-schedule to improve performance.

☐ Don't change what the loop is computing.

Problem 3: (20 pts) The MIPS implementation on the next page shows the two stages of the comparison units, C1 and C2, but they are not connected to anything. The illustration also shows an FCC register that will hold the floating-point condition code value computed by compare instructions such as c.lt.s. Connect the comparison units and the FCC register so that they operate correctly and as described by the check items below. Notice that logic to detect FP branch instructions and FP compare instructions has been added to the ID stage near the bottom.

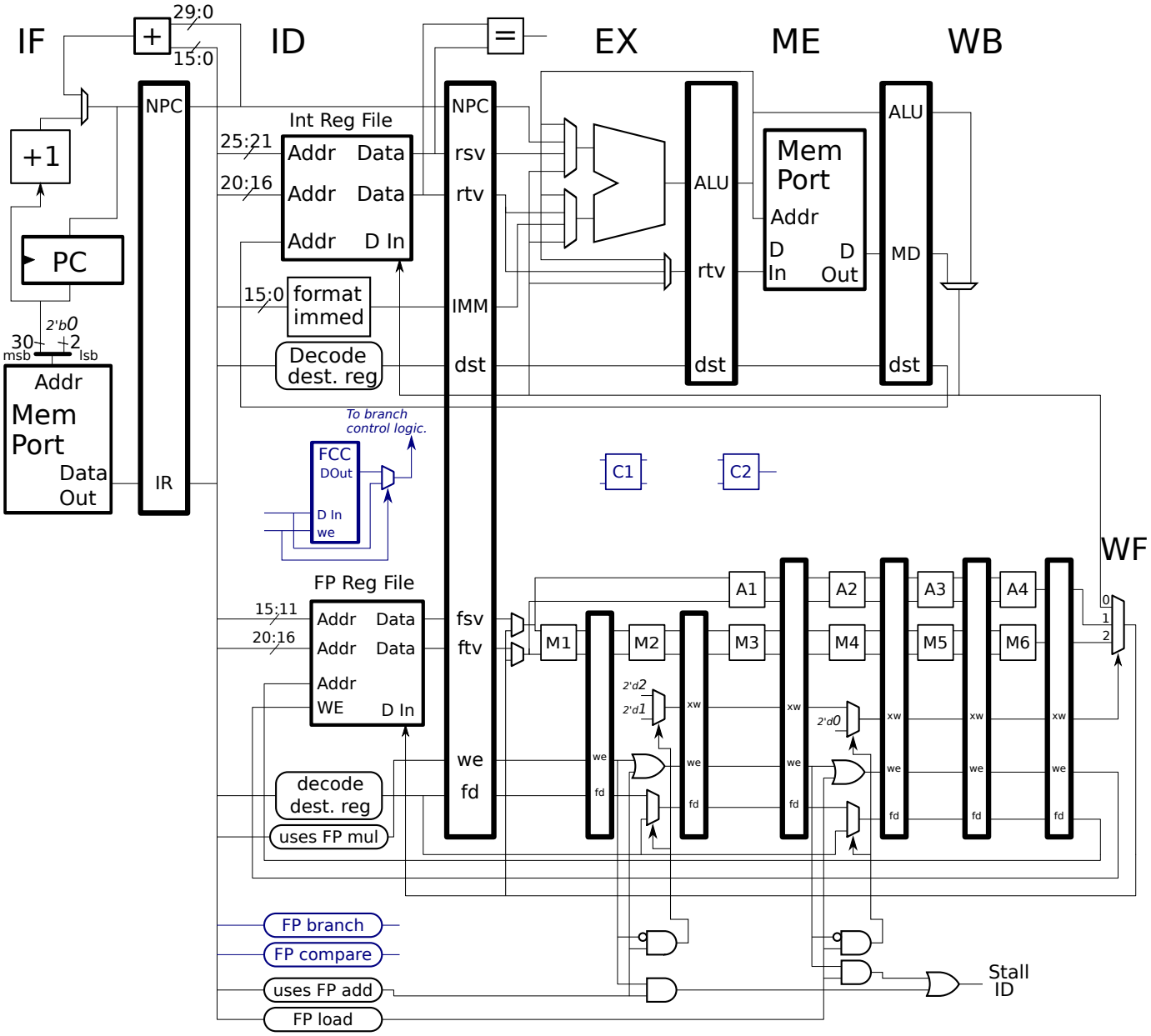
```
LOOP:     # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
mul.s f1, f1, f2   IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3         IF ID -------------> C1 C2 WF
bc1f LOOP                IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                           IF ----> ID A1 A2 A3 A4 WF
          # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

☐ Provide connections to C1, C2, and the two FCC inputs so that the code above executes as shown.

☐ Modify the control logic so that a compare does not arrive in WF in the same cycle as other FP instructions. (This is despite the fact that compares do not write the FP register file.)

☐ Modify the control logic so that the Stall ID signal is asserted for dependencies from compare to branches, such as occurs above with the bc1f.

☐ As always, pay attention to cost and performance and ☐ don't break existing functionality.


IF
ID
EX
ME
WB
WF
29:0
15:0
+
=
+1
PC
NPC
IR
Addr
Mem
Port
Data
Out
msb
lsb
30
2
2'b0
Int Reg File
25:21
20:16
Addr
Data
D In
15:0
format
immed
Decode
dest. reg
rsv
rtv
IMM
dst
ALU
MD
D
In
Out
To branch
control logic.
FCC
DOut
D In
we
C1
C2
FP Reg File
15:11
20:16
WE
fsv
ftv
we
fd
decode
dest. reg
uses FP mul
FP branch
FP compare
uses FP add
FP load
Stall
ID
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
xw
2'd2
2'd1
2'd0
0
1
2

Problem 4: (16 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B2` starts with a random outcome, then repeats that same outcome two more times, followed by another random outcome followed by two more repeats of that. The random outcome is `T` with probability .3 and is independent of other outcomes. The following are possible `B2` outcome sequences: `TTT NNN NNN TTT TTT`. Note that the number of consecutive `T`'s or `N`'s must be a multiple of 3 and so the following **is not** a possible sequence of outcomes for `B2`: `TT NN T NNNN T`.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  T  N  T  T  T  N  N   T  T  N  T  T  T  N  N  ...

B2:   r  r  r  q  q  q  s  s  s ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the bimodal predictor on branch `B2`?

☐ What is the accuracy of the local predictor on branch `B1`?

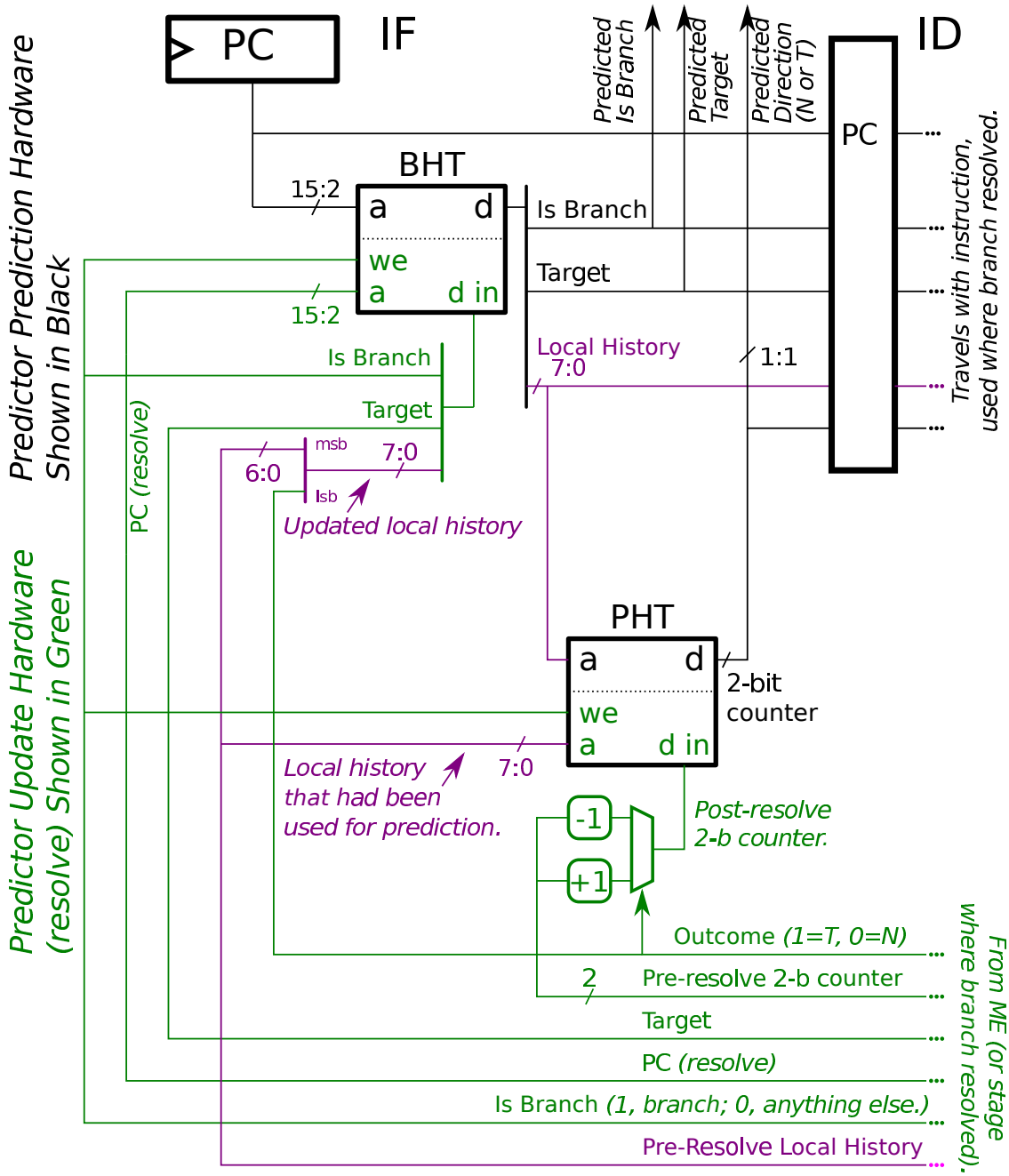
☐ What is the accuracy of the local predictor on branch `B2`?

(*b*) Appearing below is a diagram of a local predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a global predictor with an 8-outcome global history.

☐ Modify so that it is a global predictor.

☐ Remove hardware that's no longer needed.

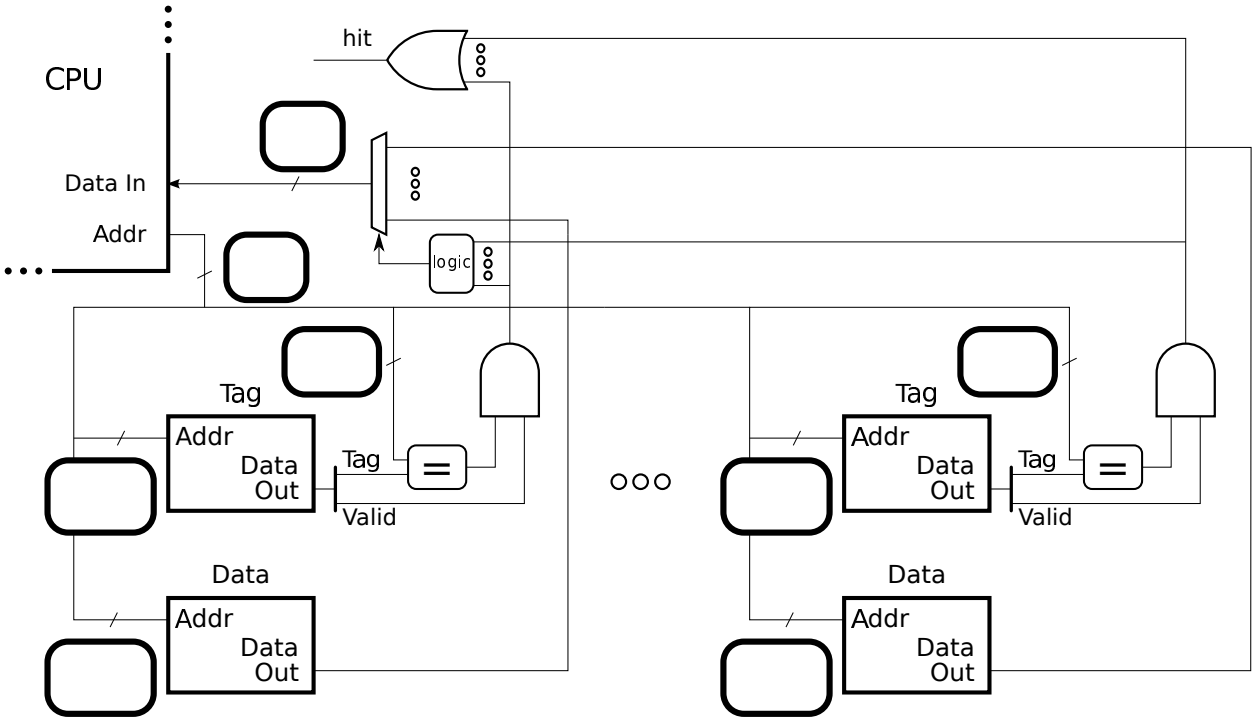
☐ Be sure to show the GHR (global history register).



Problem 5: (10 pts) The diagram below is for a 64 MiB, 4-way set-associative cache with a line size of 256 B, a bus width (w) of 8 B, for a 64 b address space. Helpful facts: $64\,\text{MiB} = 64 \times 2^{20}\,\text{B} = 2^{26}\,\text{B}$ and $256 = 2^8$.

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: ☐ ☐ ☐

0

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

Address: ☐ ☐ ☐

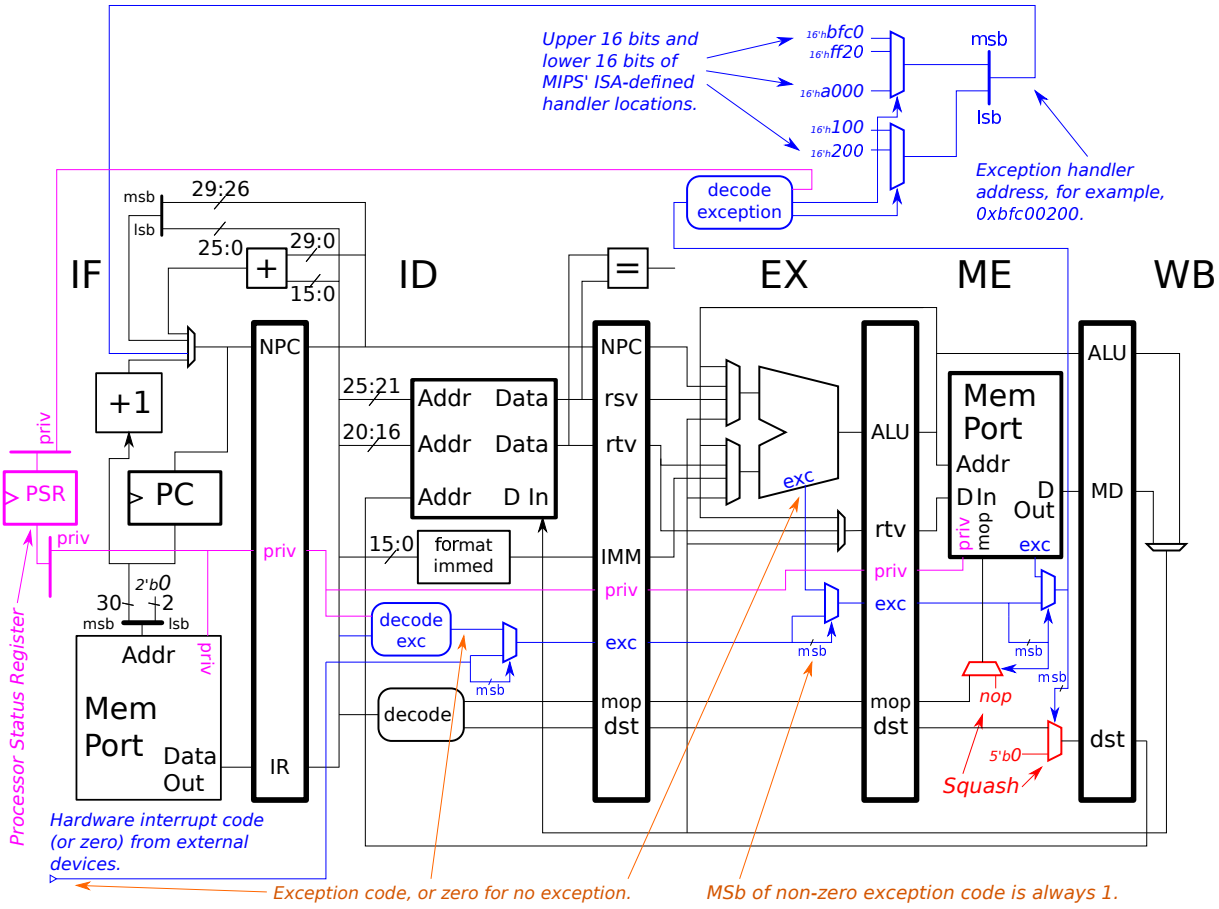
The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 256 B (which is 2^8 B). Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
short *a = 0x2000000; // sizeof(short) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 6: (8 pts) Appearing below is a MIPS implementation that includes hardware for interrupts (hardware interrupts, exceptions, and traps). An exception code, `exc`, is collected and passed down the pipeline. Its value indicates the type of hardware interrupt, exception, or trap that has been encountered, a value of 0 indicates no interrupt of any kind.



(*a*) Notice that the decode exc logic in the `ID` stage examines the opcode of the instruction in `ID` as well as the value of `ID.priv`. *Hint:* `priv` *is an abbreviation for privileged.* Some opcode values raise exceptions only when `ID.priv` is zero, others raise exceptions whether or not `ID.priv` is zero.

☐ Describe an instruction that raises an exception in `ID` only if `ID.priv` is zero.

☐ Why is it important that such an instruction raise an exception?

☐ Describe an instruction that raises an exception in `ID` whether or not `ID.priv` is zero.

(*b*) The illustrated hardware squashes the faulting instruction in `ME`, but no hardware is shown to squash any instructions that may be in the stages before `ME` nor for the stage after `ME`. That hardware may have been omitted for simplicity (the same reason that control logic is omitted) or because it is not needed.

☐ To implement precise exceptions should the instructions in the stages before `ME` be squashed? ☐ Explain in terms of the handler and what would go wrong if instructions were not treated the right way.

☐ To implement precise exceptions should the instructions in the stage after `ME` be squashed? ☐ Explain in terms of the handler and what would go wrong if instructions were not treated the right way.

Problem 7: (21 pts) Answer each question below.

(*a*) Show the encoding of the following MIPS instructions. Write the instruction name in opcode or func field values that cannot be determined.

```
0x1000:  beq r10, r11,  TARG
0x1004:  add r7, r8, r9

TARG:
0x1034:  lw r12, 14(r15)
```

☐ Encoding for `beq` from code fragment above. ☐ Pay attention to the branch target.

☐ Encoding for `add` from code fragment above:

☐ Encoding for `lw` from code fragment above:

(*b*) MIPS has one kind of memory addressing for all load and store instructions, such as in `lw r1, 2(r3)` where the immediate, 2, is added to the value in `r3`. A CISC ISA might have two versions of the load, `lw r1, (r3)`, which lacks an immediate (the immediate would be zero in MIPS), and `lwi r1, 2(r3)` for when an immediate is needed.

☐ What would be the benefit for the CISC ISA of having the no-immediate version of the `lw`?

☐ Why would MIPS and other RISC ISAs not realize the same benefit?

(*c*) A design team is considering removing a bypass connection to the ALU and adding a bypass connection to the branch resolve unit. This won't change the cost, they hope it will improve performance. "Simulation of this design change shows that performance drops by 5%," a sad-faced engineer announces. "We forgot to talk to the compiler people!," another excitedly points out, splashing hope and excitement around the room.

☐ What should they ask the compiler people?

(*d*) Someone preparing the SPECcpu benchmarks for their company's next product (currently under development) decides to replace one of the benchmark programs with an improved version, one which better reflects that company's customers. *Note: the emphasis below was added after the original exam, as was the phrase "not to market."*

☐ Should the company use the SPECcpu benchmarks in this way **to develop (but not to market)** their product?

With the substituted benchmark program the SPEC scores are higher (better). The company decides to release these higher SPEC results without mentioning the substitution.

☐ How does the design of SPECcpu make it likely that they will get caught?

(*e*) Compiler optimization is more important for a supercalar implementation than a scalar implementation.

☐ Optimization is more important for superscalar than scalar because:

☐ The important optimization is:

9 Spring 2017

Name _____

Computer Architecture
EE 4720
Midterm Examination

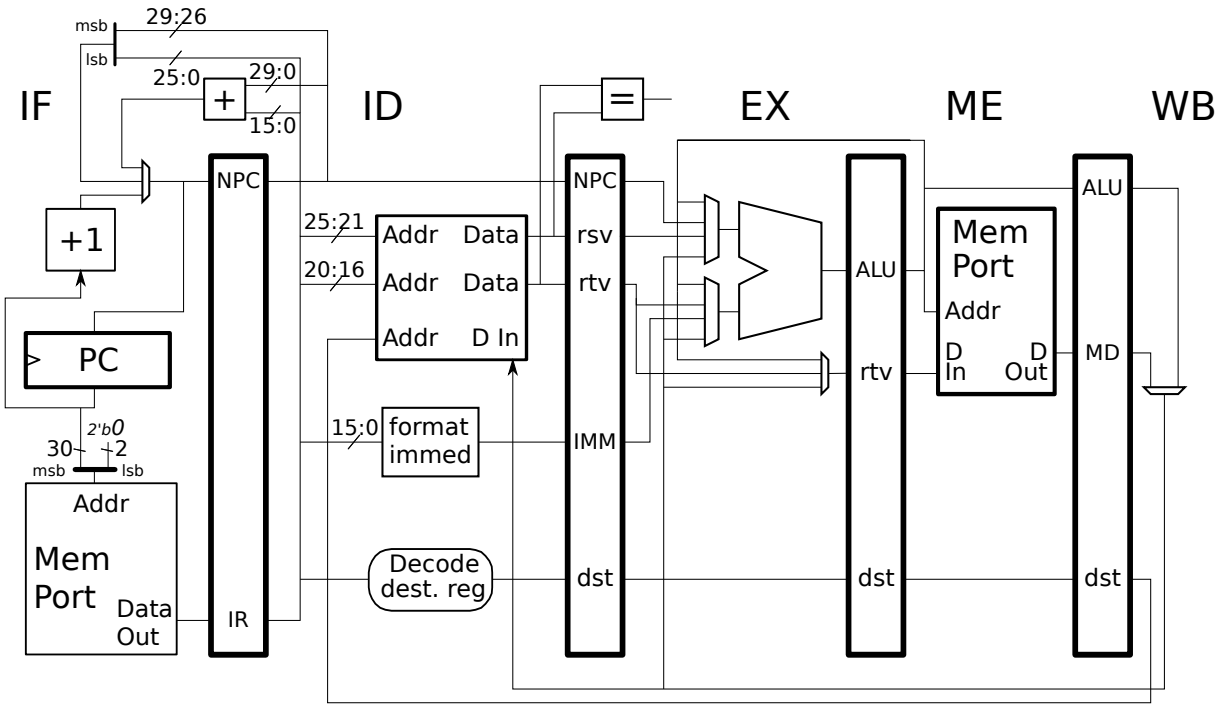
Wednesday, 22 March 2017, 9:30–10:20 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is our familiar MIPS implementation.



(*a*) Show pipeline execution diagrams for the code fragments below executing on the illustrated implementation and label as indicated.

☐ Show pipeline diagram. ☐ Doublecheck dependencies.

☐ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

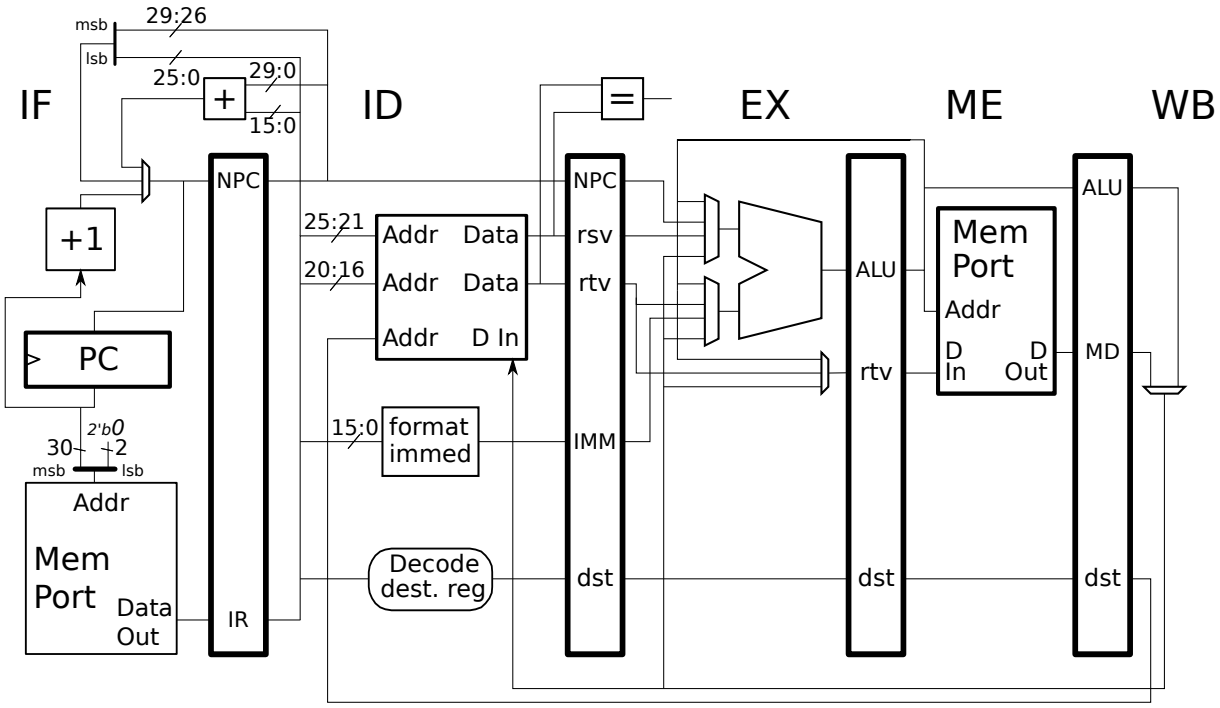
```
add r1, r2, r3

sub r4, r5, r6

sw r4, 8(r1)
```

☐ Show pipeline diagram. ☐ Doublecheck dependencies.

☐ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

```
and r1, r2, r3

lw r4, 16(r1)

ori r5, r4, 7
```

Problem 1, continued:



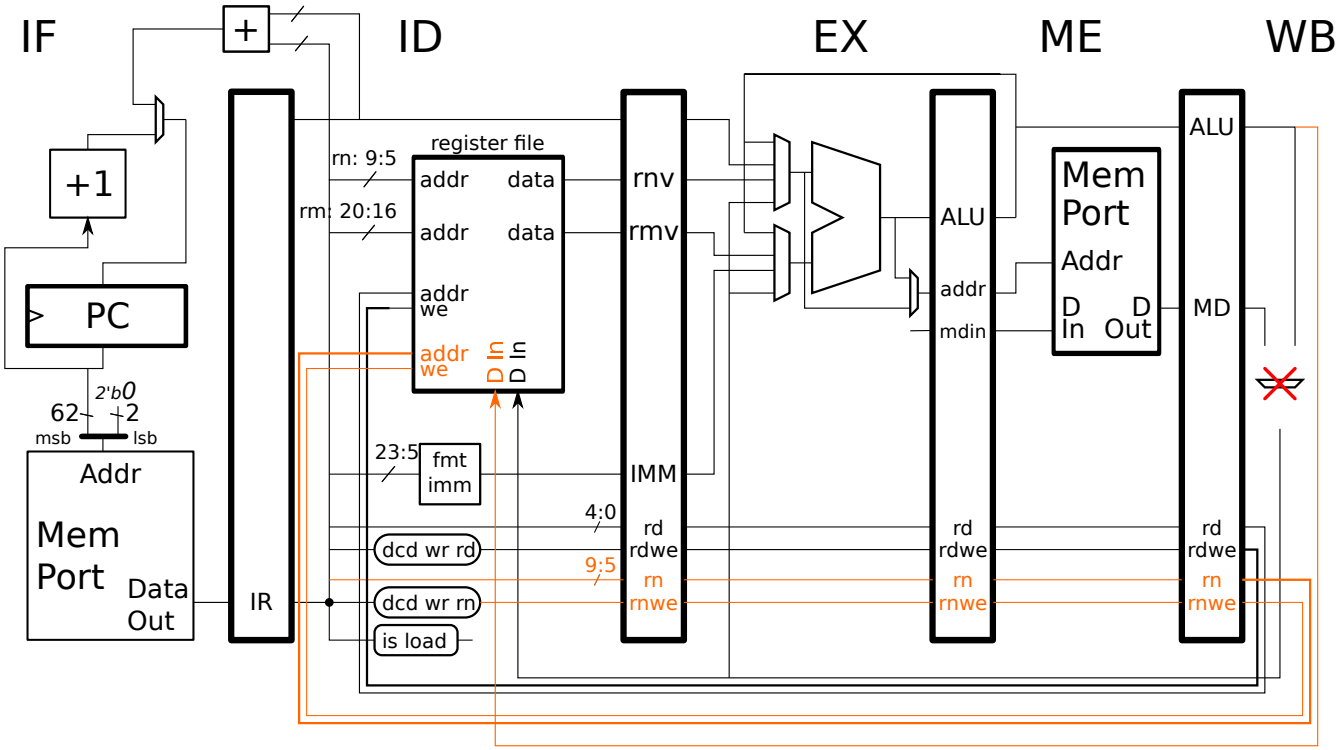
(*b*) Show a pipeline execution diagram for an execution of the code fragment below when the branch is taken. Label the ID-stage unit as indicated.

☐ Show pipeline diagram. ☐ Pay close attention to the branch.

☐ Label the **inputs and outputs** of the ID-stage unit that computes the branch target. Label with $c : v$, where c is the cycle number and v is the value on the input or output.

```
0x4000: addi r1, r2, 3

0x4004: bne r1, r6, TARG

0x4008: lw r4, 0(r5)

0x400c: addi r5, r5, 4

0x4010: xor r8, r9, r10

TARG:
0x4014: add r5, r4, r4
```

Problem 2: [20 pts] Appearing below is a partial implementation of ARM A64 taken from the solution to Homework 4. The WB-stage mux is crossed out because it's wasteful to use a 64-bit mux when the same functionality can be realized using less expensive logic in the ID stage. For reference, some A64 instructions are shown below, the comments show which field registers are encoded in.




```
@   rd   rn        : Writes rd and rn
ldr x1, [x2], #8   @ x1 = Mem[x2];  x2 = x2 + 8

@   rd, rn, rm     : Writes rd
add x3, x4, x1     @ x3 = x4 + x1

@   rd, rn         : Writes rd
and x1, x2, #34    @ x1 = x2 & 34;
```

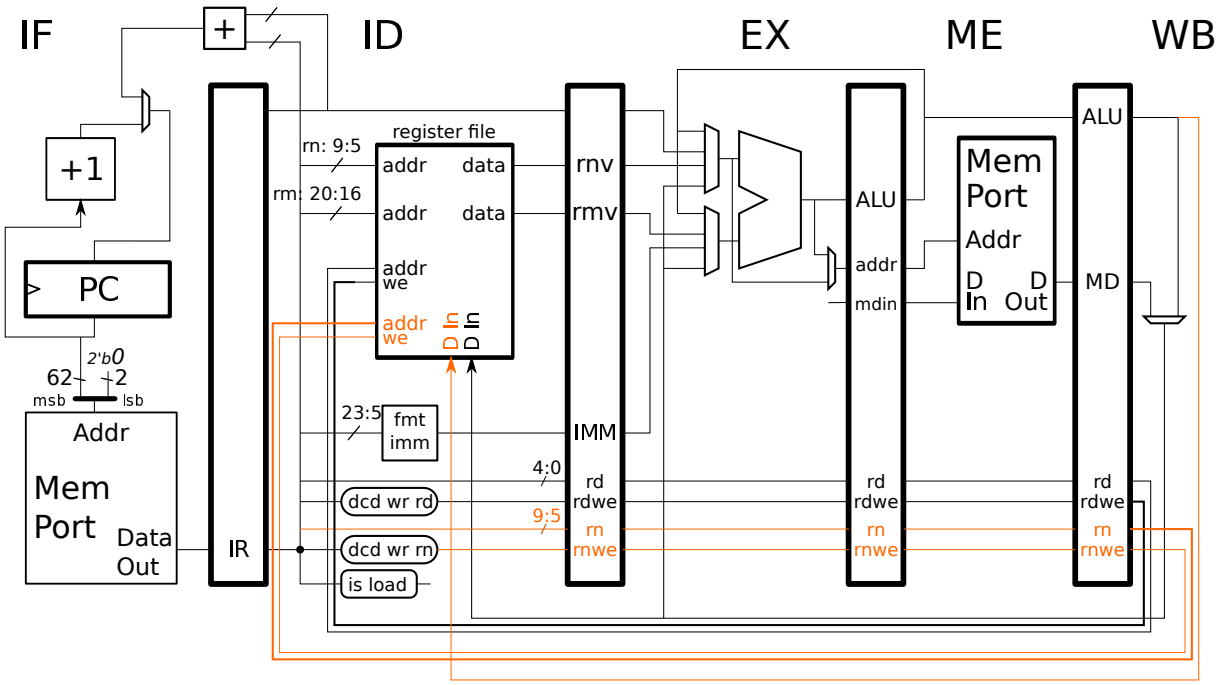
Complete the changes so that instructions such as the ones above can write back their results. Assume that only the load instructions write the rn-field register.

☐ In WB consider re-connecting wires broken by the removal of the mux.

☐ In ID make changes so that instruction results can be written back to the correct registers.

☐ Make the best use of the existing from-WB bypass path.

Problem 3: [20 pts] Appearing below again is the partial implementation of ARM A64 taken from the solution to Homework 4.




```
@   rd   rn         : Writes rd and rn.  Post-index
ldr x1, [x2], #8    @ x1 = Mem[x2];  x2 = x2 + 8

@   rd   rn         : Writes rd and rn.  Pre-index
ldr x1, [x2, #8]!   @ x2 = x2 + 8;  x1 = Mem[x2];

@   rd   rn  rm
str x1, [x2, x3]    @ Mem[x2+x3] = x1
```

(*a*) Make changes needed to implement the store instruction, see the example above. Just show datapath, not control logic.

☐ Changes for the store instruction.

(*b*) Do we really need *both* pre-index and post-index addressing for loads and stores? Eliminating either of them will reduce cost, but one's elimination would reduce cost by more than the other's. Indicate which saves more and show the hardware that can be removed and other needed changes. *Note: The phrase* and other needed changes *was not in the original exam.*

☐ Greater cost reduction by eliminating: ◯ *pre-index* ◯ *post-index* (check exactly one).

☐ Show the hardware that's not needed ☐ and other needed changes.

Problem 4: [20 pts] When MIPS routine `coursei` is called register `a0` will hold an entry number, referring to the table at label `courses`. Complete the routine so that when it returns register `v0` will have the integer representation of entry number `a0` in the table. Note that the table itself holds floats. For example, when called with `a0=0` it should return with `v0=2740`, when called with `a0=2` it should return with `v0=3755`, etc.

☐ Complete so `v0` is integer representation of `a0`th table entry.

☐ Read the table as it is, don't modify it or read a different table.

```
        .data
courses:
        .float 2740
        .float 3750
        .float 3755
        .float 4755
        .float 4720
        .float 7722
        .float 7725
        .text
        # CALL VALUE: $a0: Entry in table to look up.
        # RETURN:     $v0: Table entry #$a0 represented as an integer.
coursei:
        la $t0, courses









        jr $ra
        nop
```

Problem 5: [20 pts] Answer each question below.

(*a*) What kind of implementations were RISC ISAs designed to simplify?

☐ Kinds of implementations that RISC designed to simplify:

(*b*) Describe how the features below simplify RISC ISA implementations.

☐ Fixed-size instructions.

☐ Avoiding arithmetic instructions that access memory.

(*c*) The SPECcpu package contains the source code for the SPEC benchmarks and scripts to compile and run them, but it does not come with compilers. The tester provides his or her own. Consider a *SPECcpu+* package that comes with compilers, and the requirement that those compilers be used. Why would that make SPECcpu+ less useful to computer engineers?

☐ SPECcpu+ less useful because:

(*d*) In class we described some optimizations as high-level, and some as low-level, performed by the back end. What distinguishes high- and low-level optimizations? Provide an example of a low-level optimization that could only be performed by the compiler back end.

☐ Difference between high- and low-level optimization.

☐ Example of an optimization that must be low-level (that can only be done in the back end).

☐ Briefly explain why.

(*e*) MIPS I has instruction `bgtz r1, TARG` in which the branch is taken if `r1>0` but it lacks an instruction like `bgt r1, r2, TARG` that would branch if `r1>r2`. Why?

☐ Why does MIPS lack `bgt r1, r2, TARG`?

☐ What would be the impact on performance of including `bgt r1, r2, TARG` in MIPS on "our" five-stage implementation?

Name _____

Computer Architecture
EE 4720
Final Examination

1 May 2017, 10:00–12:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

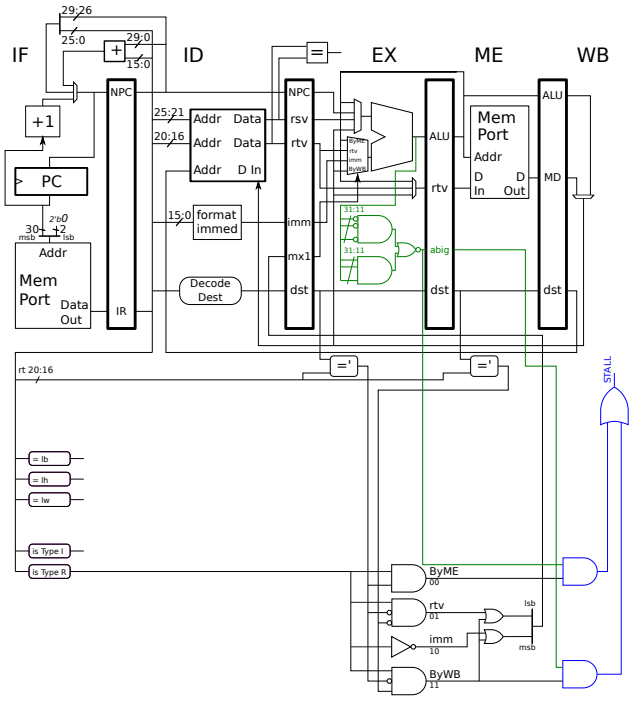
Alias _____

Good Luck!

Problem 1: (20 pts) The diagram below, based on the solution to Homework 5, shows control logic that generates a stall signal when the value to be bypassed is too large for 12-bit bypass paths. The logic only works when the dependency is with the `rt` register of the consuming instruction and when the producing instruction is not a load. Modify the control logic so that it will generate a stall signal for a dependency to an `rs` register (first example below) and dependencies with loads. Pay attention to the load sizes.

```
# Dependency through rs register (r1 in the sub).
add r1, r2, r3
sub r4, r1, r5

# Producing instruction is a load word.
lw r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load half.
lh r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load byte.
lb r1, 2(r3)
and r4, r5, r1
```

Use next page for solution.

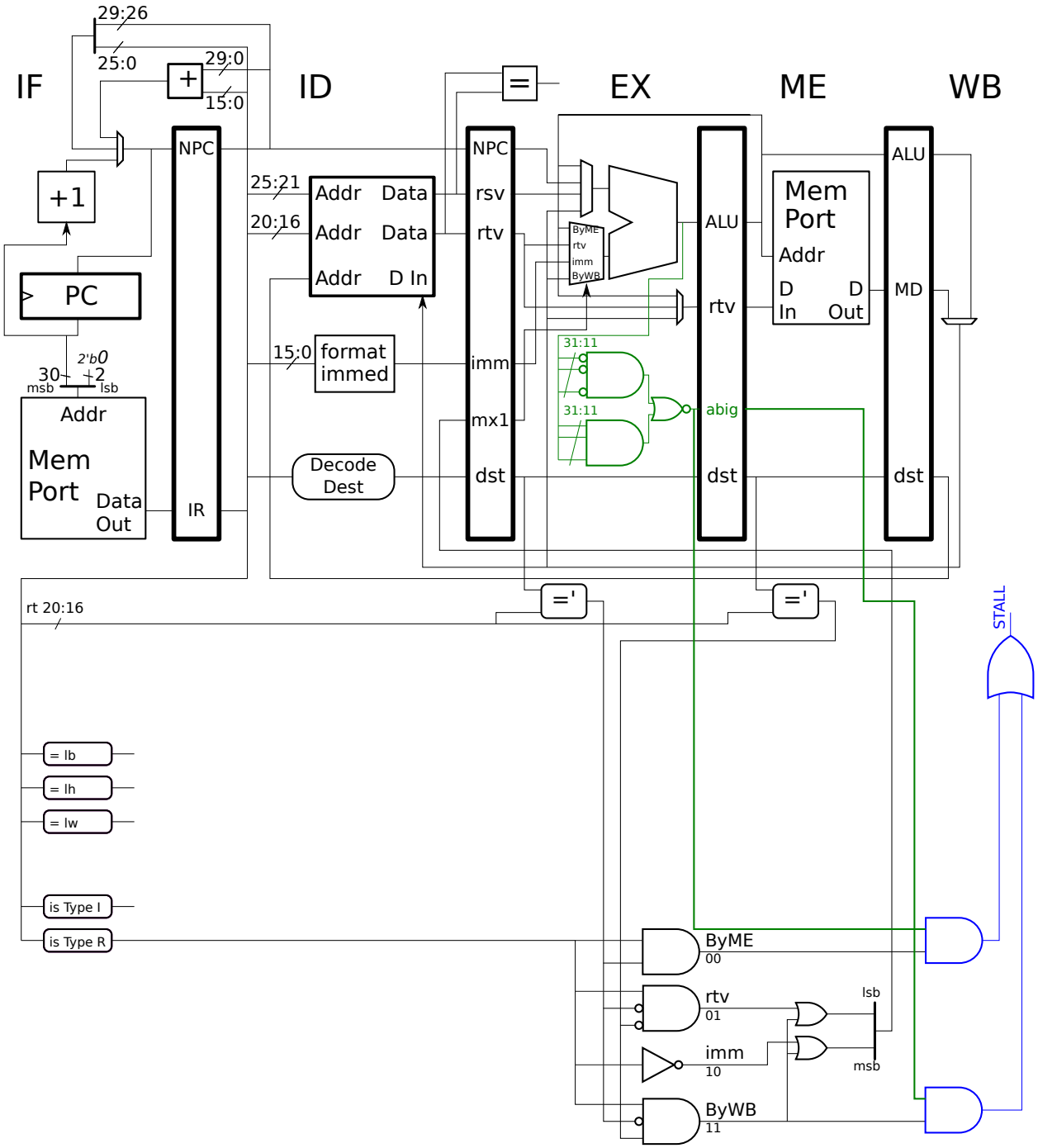


Use next page for solution.

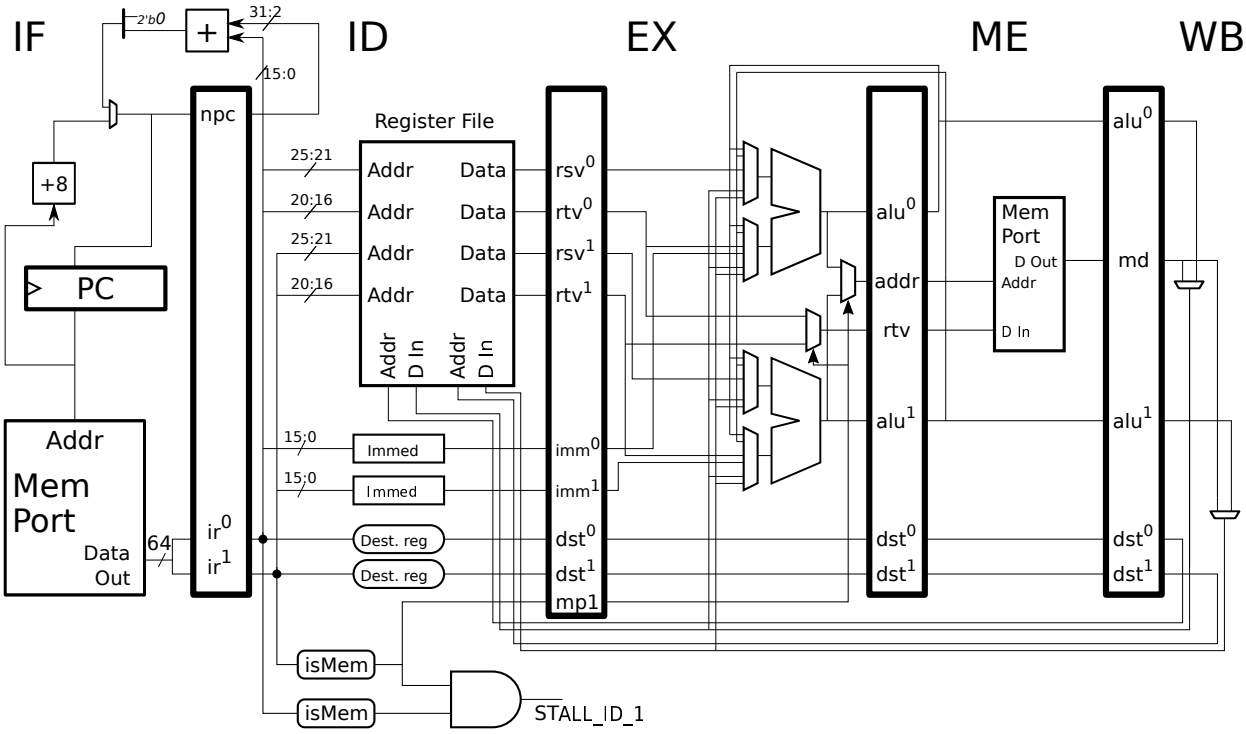
☐ Modify the control logic so that it also generates the stall signal for dependencies through the `rs` register that can't use 12-bit bypasses.

☐ Modify the stall control logic for when loads `lb`, `lh`, and `lw` produce the value to bypass, ☐ take into account whether value can use the 12-bit bypasses and ☐ whether the instructions are too close to bypass.

☐ Do not break existing control logic. As always ☐ consider cost and performance.



Problem 2: (15 pts) Illustrated below is a superscalar implementation taken from the solution to last year's final exam and the subject of this semester's Homework 7. Show the execution of the code sequences below on the illustrated superscalar MIPS implementation. Don't forget to check for dependencies.



(*a*) Show the execution of the code below on this implementation. Note that the address of the first instruction is `0x1000`.

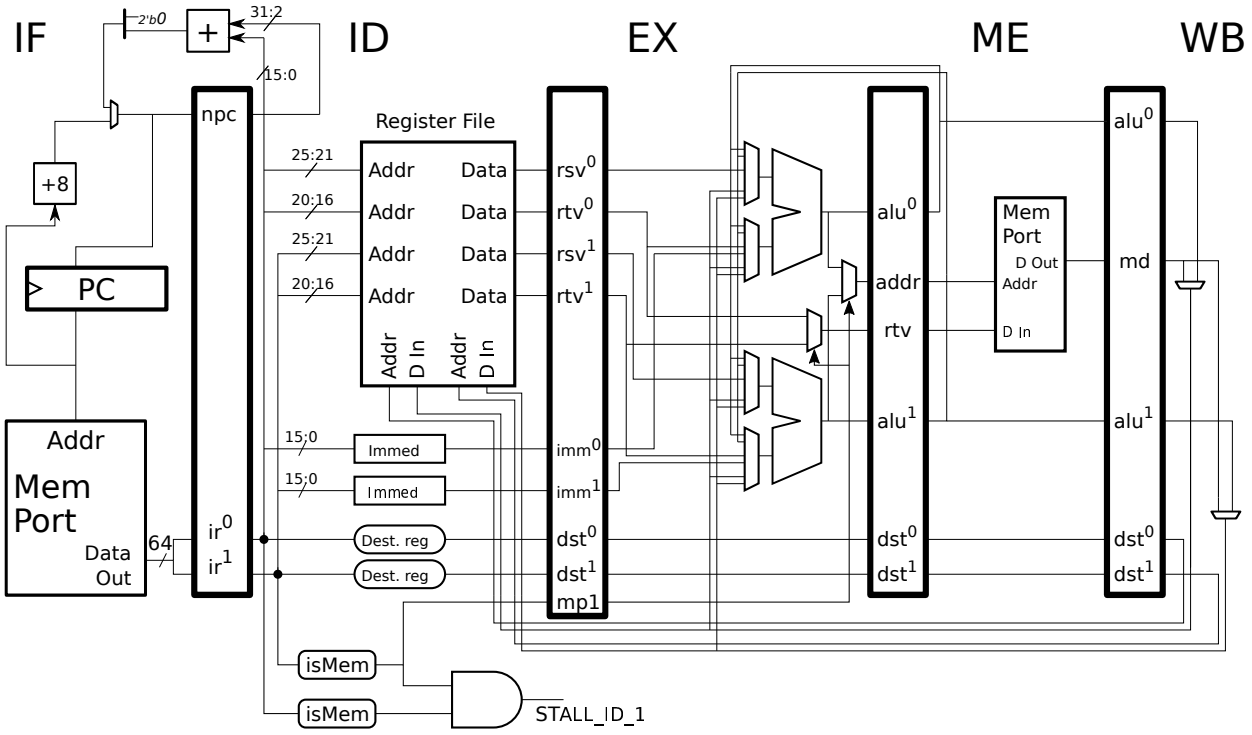
☐ Show execution of the following code sequence. ☐ Pay attention to ME in the diagram.

☐ Check for dependencies.

START: Address is 0x1000.

```
lw r1, 0(r2)

lw r3, 4(r2)

lw r4, 8(r2)

add r5, r1, r5

add r5, r3, r5

add r5, r4, r5
```

Problem 2, continued: The illustration below is the same as the one on the previous page.



(*b*) Show the execution of the code below on the illustrated implementation when the branch is taken. Use the classroom default assumption: fetches are aligned.

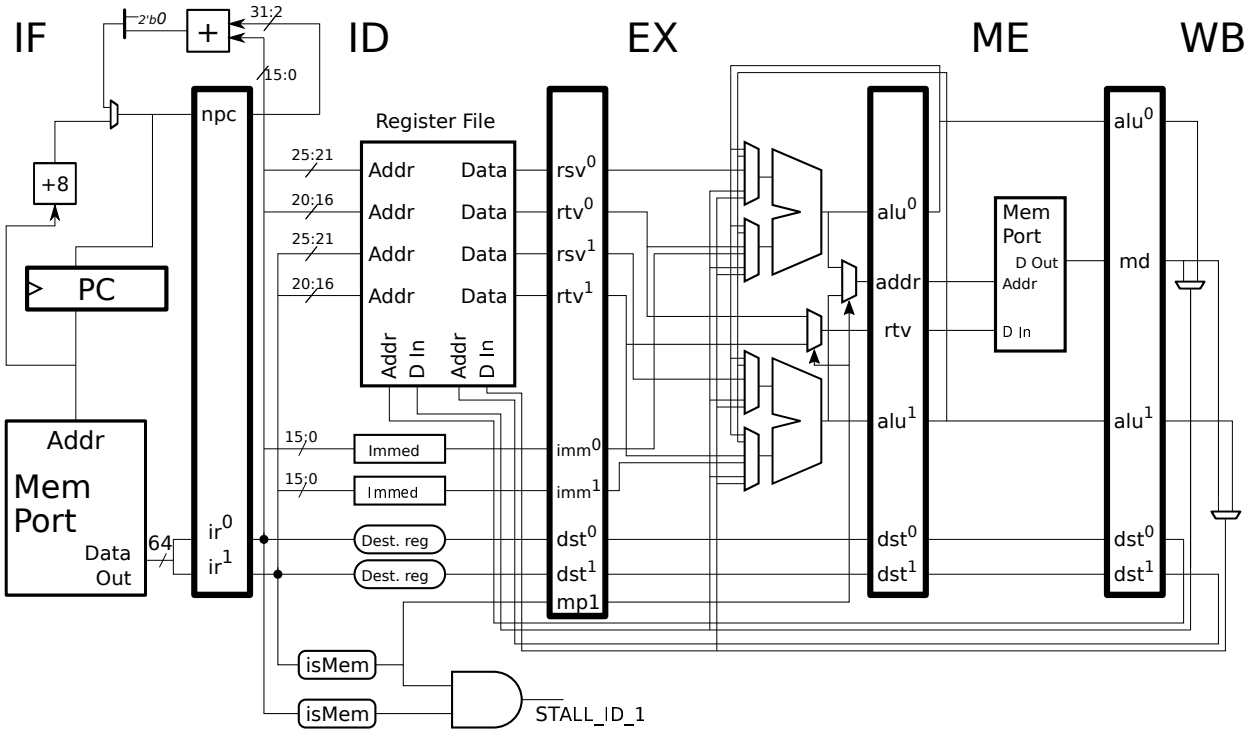
☐ Show execution of the following code sequence. ☐ Check for dependencies.

☐ Show all instructions that enter the pipeline, even those that are squashed in IF or later.

☐ Pay attention to instruction addresses, such as `0x1000`.

```
#       Branch is taken.
0x1000: bne r1, r4  TARG

0x1004: sub  r5, r2, r7

0x1008: xor  r10, r11, r12

0x100c: lbu  r9, 0(r5)

0x1010: andi r8, r9, 12

TARG:
0x1014: or  r11, r5, r12

0x1018: sb  r11, 0(r5)
```

Problem 2, continued: The illustration below is the same as on the previous page.



(*c*) Appearing below is an execution of MIPS code on the illustrated superscalar implementation shown for the first two iterations. Compute the CPI for a large number of iterations. If necessary extend the execution diagram.

```
 lw r1, 0(r2)     IF ID EX ME WB
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4      IF ID -> EX ME WB            # First Iteration
 lw r1, 0(r2)        IF ID -> EX ME WB
 bne  r2, r3  LOOP      IF -> ID EX ME WB
 addi r2, r2, 4         IF -> ID EX ME WB
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4                 IF ID EX ME WB   # Second Iteration
 lw r1, 0(r2)                   IF ID EX ME WB
 bne  r2, r3  LOOP                 IF ID EX ME WB
 addi r2, r2, 4                    IF ID EX ME WB
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
```

☐ CPI for a large number of iterations.

Problem 3: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 8-outcome local history, and one system has a global predictor with a 8-outcome global history. Branch `B2` consists of a repeating pattern that starts with `TNTT` and is either followed by three not-taken outcomes, `nnn`, or four taken outcomes, `tttt`. (They are shown in lower case for clarity.) The `nnn` sequence occurs with probability .4, and is not correlated with anything.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
                          ↓
B1:   T  N  T  T  N       T  N  T  T  N          T  N  T  T  N
B2:    T  N  T  T  n  n  n   T  N  T  T  t  t  t  t
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the bimodal predictor on branch `B2`? ☐ Account for the variable pattern length.

☐ What is the accuracy of the local predictor on branch `B2`? ☐ Account for the variable pattern length.

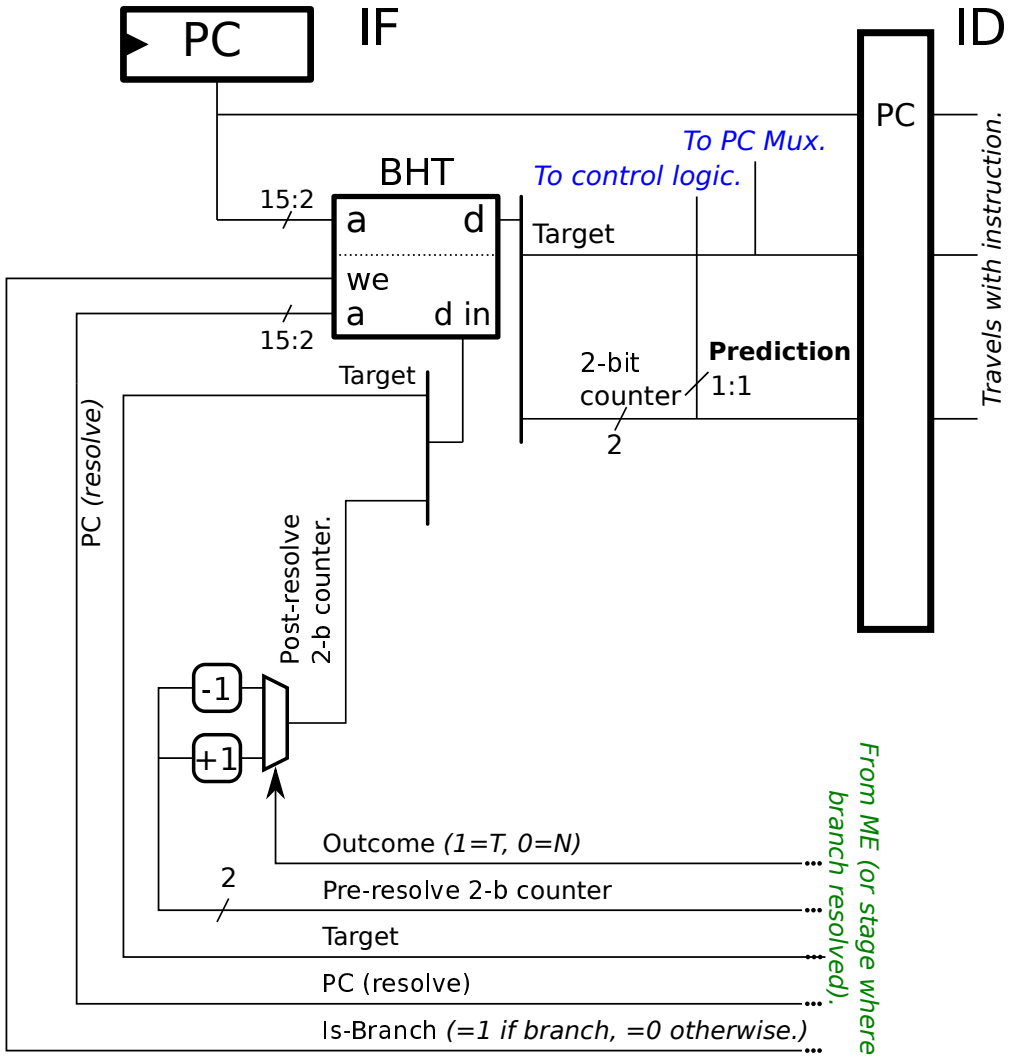
☐ What is the minimum local history size for which branch `B1` and `B2` will not interfere with each other?
☐ Explain.

☐ Note that an arrow (↓) points at an execution of `B1`. Show the value of the GHR at the time that that execution is being predicted.

Problem 3, continued:

(*b*) Appearing below is a diagram of a bimodal predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a local predictor with an 8-outcome local history.

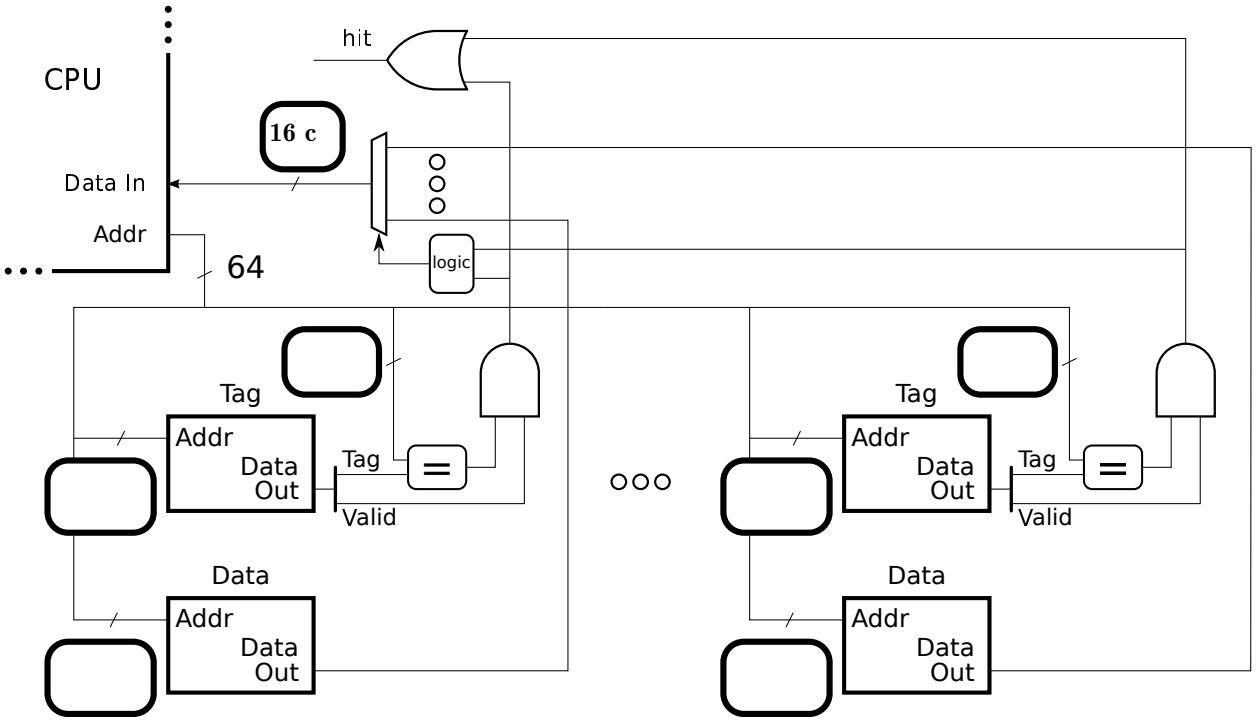
☐ Show the PHT, and connections for ☐ prediction and ☐ update.



Problem 4: (15 pts) The diagram below is for a 32 MiB (2^{25} B) four-way set-associative cache with a line size of 32 B.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: | | | |

0

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

Address: | | | |

Problem 4, continued: The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 1024 B (2^{10} B). Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 5: (30 pts) Answer each question below.

(*a*) Consider a 4-way superscalar system and a scalar system with a 4-lane vector unit. Both can compute arithmetic at a rate of 4 operations per cycle. The vector system is cheaper but the superscalar system is more flexible.

☐ Why is the vector system less costly?

☐ Show something the superscalar system can do that the vector system cannot.

☐ Explain why vector system can't execute equivalent vector code as efficiently.

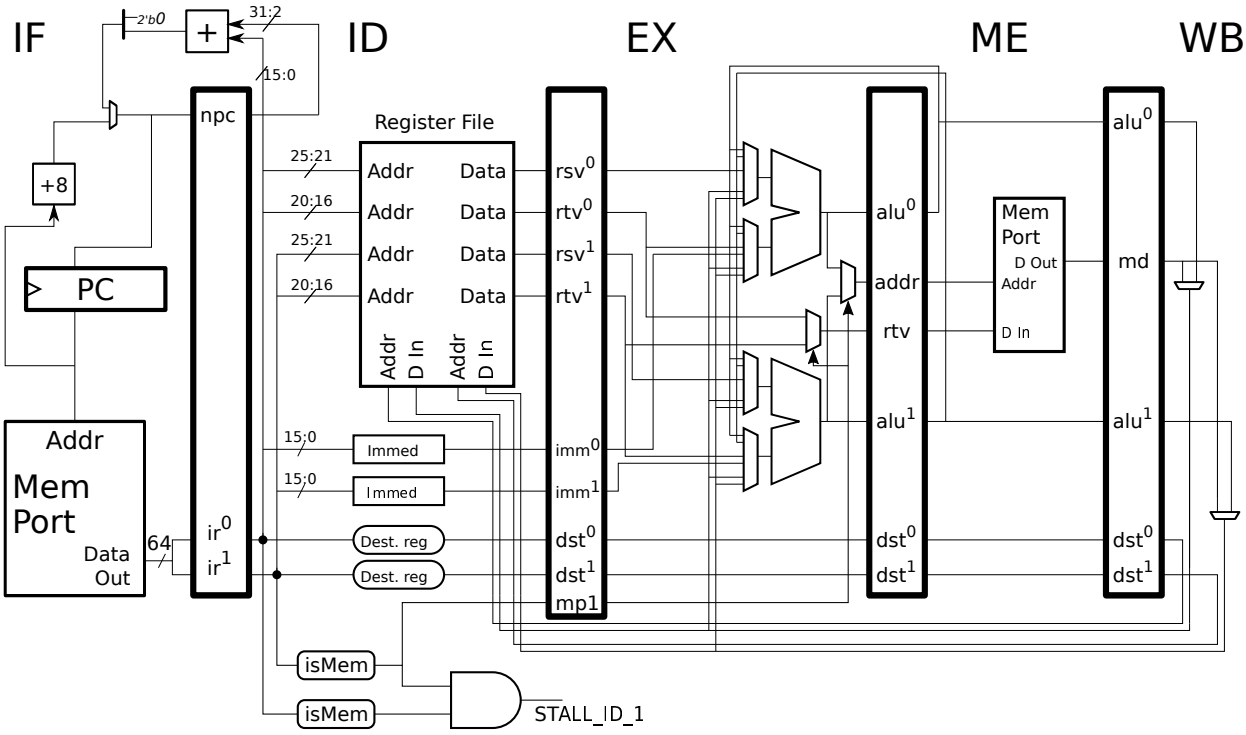
(*b*) Unlike MIPS, ARM A64 has pre-index and post-index load and store instructions. Show two code examples, one in A64 that uses a post-index load, and one in MIPS that does the same thing (but without a post-index load). The exact syntax of the ARM instructions is not important, use comments to clarify instructions.

☐ ARM code and ☐ equivalent MIPS code.

(*c*) What substantial additional hardware is needed to implement ARM A64 pre- and post-index loads when starting with something like our five-stage MIPS implementation. (Think about Homework 4.) *Note: The words "substantial" and "costly" were not included in the original exam.*

☐ Costly additional hardware for pre- and post-index loads.

(*d*) VLIW ISAs are supposed to do for superscalar implementations what RISC ISAs did for pipelined implementations. The diagram below shows our 2-way superscalar MIPS. Show how a 2-slot-bundle VLIW ISA (perhaps one a lot like MIPS) could simplify hardware in this implementation related to the sharing in `ME`.



☐ Modify the hardware above.

☐ Explain the bundle slot restrictions based on modified hardware.

☐ Explain why the control logic driving `STALL_ID_1` would no longer be needed.

(*e*) When an exception occurs (or a trap instruction is executed) the processor switches from user mode into privileged mode (also called system mode). Explain how privileged mode affects instruction execution, including loads, compared to user mode.

☐ Effect of privileged mode on instruction execution including ☐ effect on load instruction execution.

(*f*) It's hard to choose a line size that makes everyone happy. Explain how a long line size might slow down some programs in a small cache in comparison to the right line size (for those programs).

☐ With a small cache large lines can slow some programs because:

☐ Describe the characteristics of code that works well with long lines.

10 Spring 2016

Name _____

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 30 March 2016, 9:30–10:20 CDT

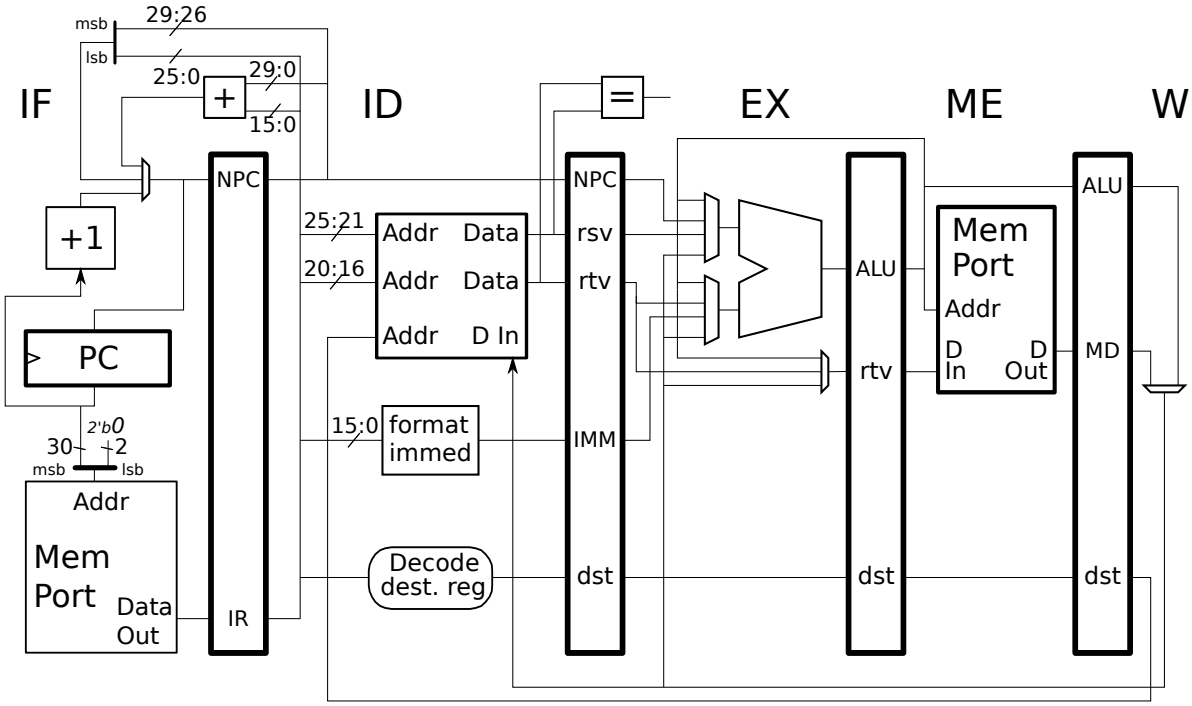
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(12 pts)
Problem 4	_____	(28 pts)
Problem 5	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing below are what are supposed to be pipeline execution diagrams (PEDs) of code fragments executing on the illustrated implementation. The PEDs are incorrect.

(*a*) Correct the PEDs.



☐ Correct the PED below.

```
add r1, r2, r3  IF ID EX ME WB
lw r3, 0(r1)       IF ID -> EX ME WB
```

☐ Correct the PED below.

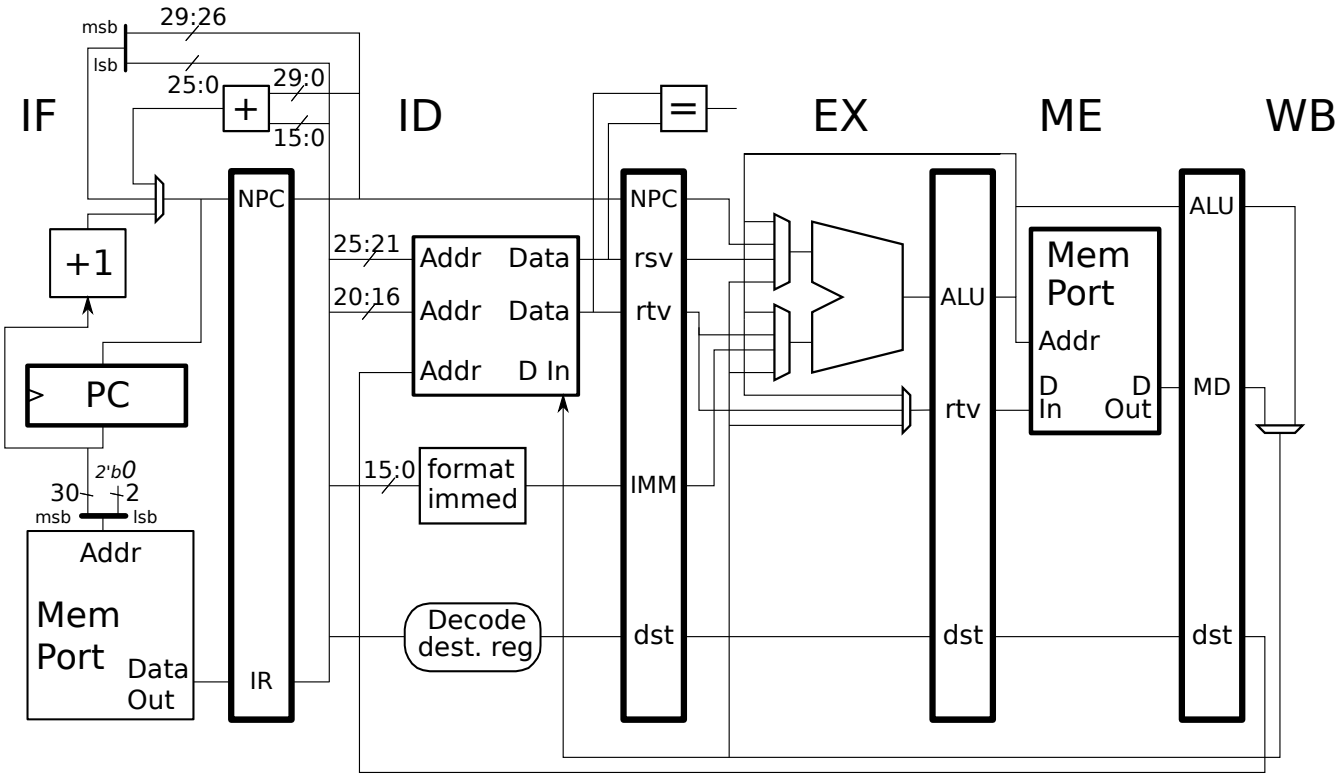
```
lw r3, 0(r1)    IF ID EX ME WB
add r4, r3, r5     IF ID -> EX ME WB
sub r6, r7, r8        IF ID EX ME WB
```

☐ Correct the PED below.

```
# Cycle            0  1  2  3  4  5  6  7
 beq r1, r1  TARG  IF ID EX ME WB   # Branch is taken.
 xor r5, r6, r7       IF IDx
 add r8, r9, r10         IFx
TARG:
 sub r2, r3, r4             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7
```

(*b*) Appearing below are more PEDs which are not correct for the illustrated implementation. This time **modify the implementation** so that the executions are correct. Only make necessary changes.

- Delete a bypass path by showing an × at the **mux input** where it ends.
- Do not delete or add more hardware than is necessary.



☐ Modify the implementation so that the execution below is correct.

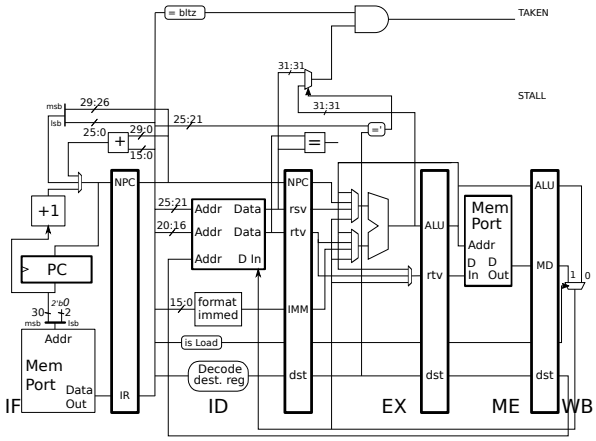
```
add r1, r2, r3   IF ID EX ME WB
sub r3, r1, r5      IF ID ----> EX ME WB
```

☐ Modify the implementation so that the execution below is correct.

```
lw r1, 0(r2)   IF ID EX ME WB
sw r1, 0(r3)      IF ID EX ME WB
```

Problem 2: [25 pts] The implementation below is based on the solution to Homework 2 Problem 2 in which a bypass was added for `bltz` instructions.

Use Next Page for Solution



Use Next Page for Solution

(*a*) The implementation can only bypass values in `EX` to a `bltz`. Modify the implementation on the next page so that values can be bypassed from both `EX` and `ME`. With these changes the two fragments below should run without a stall and of course bypass the correct value.

```
# Example 1
add r1, r2, r3
sub r4, r5, r6
bltz r1, TARG

# Example 2.
add r1, r2, r3
sub r1, r1, r6
bltz r1, TARG
```

(*b*) A bypass from EX isn't possible for the code fragment below, and a bypass from `ME` is problematic too. On the next page add logic to generate a stall signal for these situations (load/`bltz` dependencies) and connect it to the word `STALL` in the upper-right of the diagram. Notice that there is an is Load logic block in `ID`.

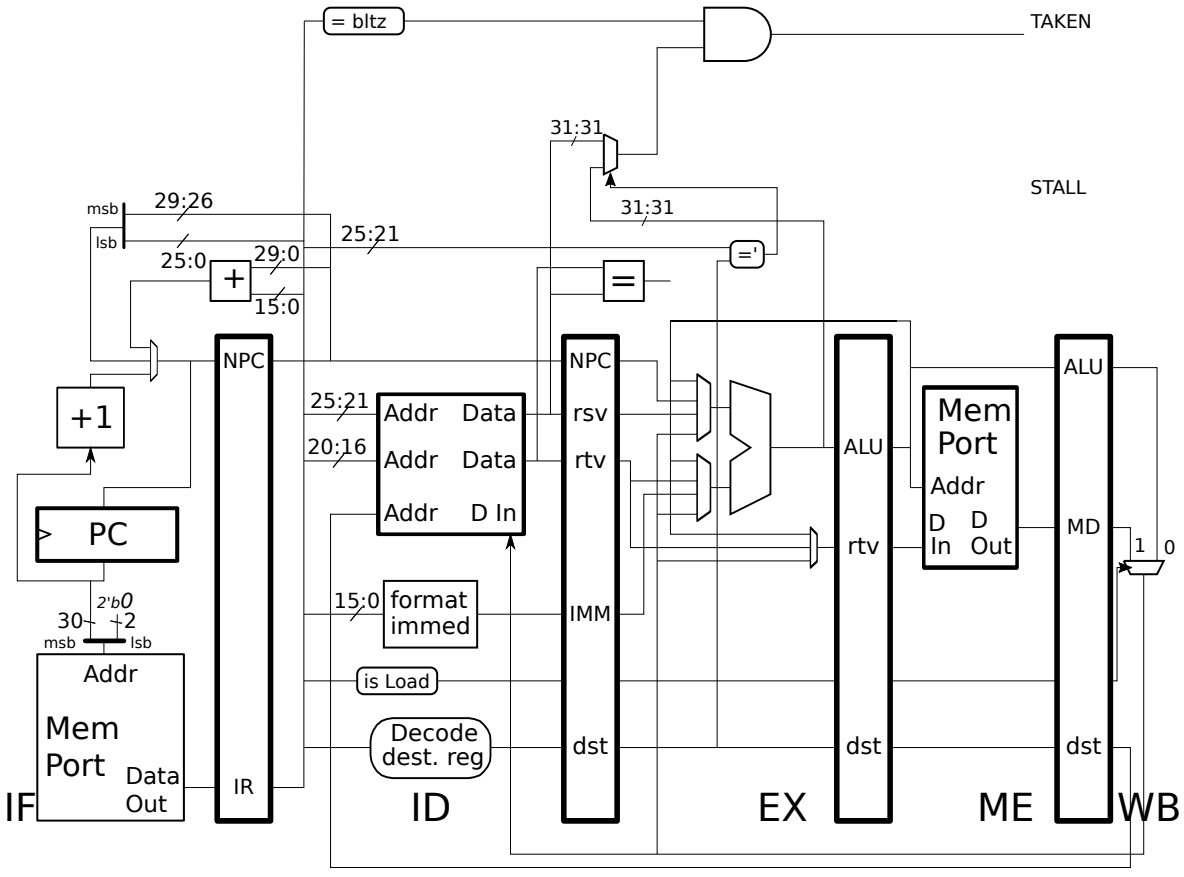
```
lw r1, 0(r2)
bltz r1, TARG
```

(*c*) Explain why it would not be a good idea to bypass the load value to the `bltz` when the load is in `ME`.

☐ Bypassing load from `ME` not a good idea because:

Problem 2, continued:

- ☐ Modify implementation so `bltz` can bypass from `EX` and `ME`.
- ☐ Logic to generate stall signal for `bltz` dependent on load.
- ☐ Answer part c.



Problem 3: [12 pts] Answer each question below.

(*a*) Each code fragment below writes register `f30` with the sum `f2 + 4720`.

```
# Plan A
        addi $t0, $0, 4720
        mtc1 $t0, $f17
        cvt.s.w $f16, $f17
        add.s $f30, $f2, $f16


# Plan B
        lui $t0, 0x4593
        ori $t0, $t0, 0x8000
        mtc1 $t0, $f16
        add.s $f30, $f2, $f16
```

☐ What is the difference between `mtc1` and `cvt`?

☐ Why doesn't Plan B need a `cvt`?

(*b*) All MIPS integer instructions have their source register numbers in the `rs` and, if needed, `rt` fields. But the destination register number can be found in either the `rt` or `rd` fields.

☐ How does limiting integer sources to `rs` and `rt` reduce cost and improve performance?

☐ Why isn't performance hurt by having the destination in either `rt` or `rd`?

Problem 4: [28 pts] Answer each question below.

(*a*) The statement below omits an important reason why customers can be kept by companies that manage an ISA and implementation as two different things.

By separating the ISA from the implementation we can keep our customers by offering them a faster implementation when they are ready to buy a new system.

☐ What is the important reason that has been omitted?

(*b*) To use profiling to improve performance a program is compiled twice.

☐ What is done between the first and second compilation?

☐ Why does the program need to be compiled a second time?

☐ Suppose that taken branches have a penalty. Show how profiling helps.

Problem 4, continued:

(*c*) Consider an instruction such as `add (r1), r2, 4(r3)`. What about it makes it unsuitable for a RISC ISA? Explain why it would be difficult to implement in our pipelined design.

☐ `add (r1), r2, 4(r3)` unsuitable for RISC because:

☐ It would be difficult to implement because:

(*d*) When we compared the un-optimized and optimized versions of the π program we found that the optimized version had many fewer load and store instructions. Why?

☐ The optimized π program had fewer loads and stores because:

(*e*) A tester preparing a run of the SPECcpu suite is responsible for compiling the benchmarks. Why does that make SPECcpu results interesting to computer engineers?

☐ Tester compilation makes SPECcpu interesting to computer engineers because:

Problem 5: [10 pts] Answer the following questions about bypass paths.

(*a*) Consider the two statements below about bypasses in implementations like our five-stage MIPS **running typical programs**.

A: Compiler scheduling makes bypass paths unnecessary.

☐ Explain why the statement above is wrong.

B: Bypass paths make compiler scheduling unnecessary.

☐ Explain why the statement above is wrong.

(*b*) Consider the two above statements (about bypass paths) again as it applies to our MIPS implementation, but this time **running a special set of programs**. We plan to design an implementation for this set of programs. For these programs the two statements are true! *Note: The original exam did not mention the new implementation, and it had an "or both" option below.*

☐ For such programs should we eliminate bypass paths or should we eliminate compiler scheduling?

☐ Explain.

Name _____

Computer Architecture
EE 4720
Final Examination

4 May 2016, 15:00–17:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (20 pts) Appearing below is our two-way superscalar MIPS implementation with a single, unconnected memory port in the `ME` stage. Since there is only one memory port there will have to be a slot-1 `ID`-stage stall whenever `ID` contains two memory instructions, for example:

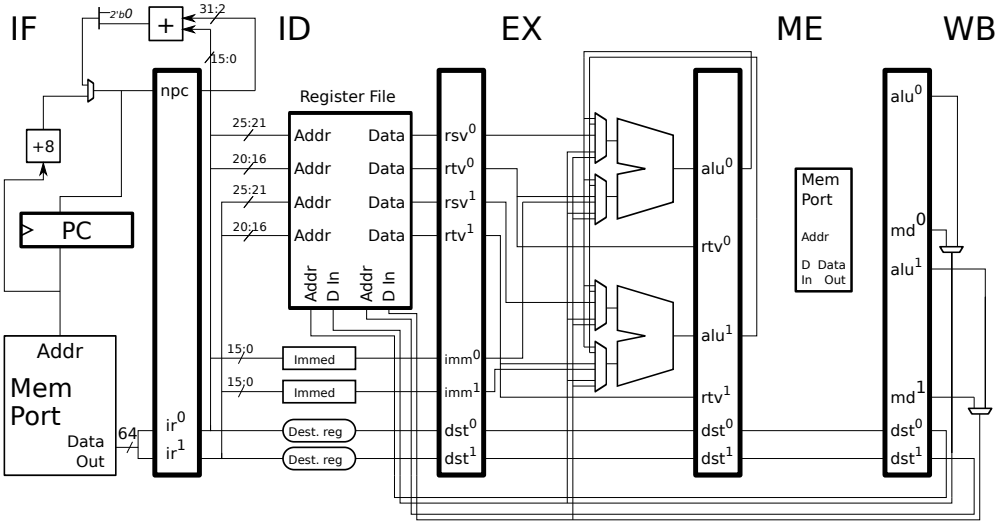
```
# Cycle       0  1  2  3  4  5
lw r1, 0(r2)  IF ID EX ME WB
lw r3, 0(r4)  IF ID -> EX ME WB
```

(*a*) On the next page add datapath to the implementation so that the memory port can be used by a load or store instruction in either slot. Pay attention to the cost of pipeline latches.

(*b*) On the next page add control logic needed for the datapath changes. The control logic should be for any multiplexors that you've added, don't include control logic for the memory port itself.

(*c*) On the next page add control logic to generate a `STALL_ID_1` signal when there are two memory instructions in `ID`, as occurs in cycle 1 to the `lw r3` in the example above.

Use next page for solution.



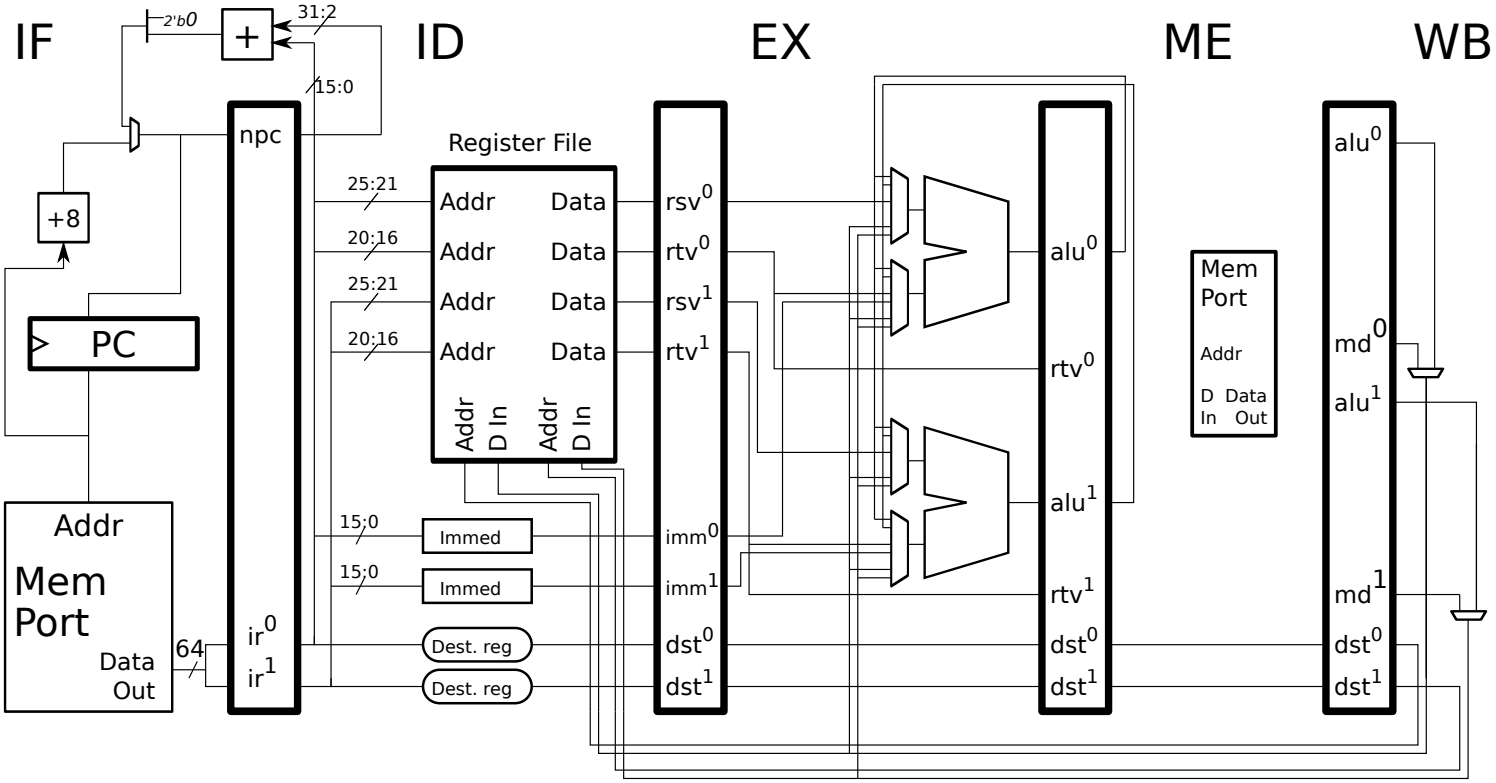
Use next page for solution.

Problem 1, continued:

☐ Datapath so that memory port can be used by a ☐ load or ☐ store in either slot.

☐ Control logic for multiplexors that you've added. ☐ Control logic to generate `STALL_ID_1`.

☐ Pay attention to ☐ **pipeline latch cost** and ☐ the critical path.



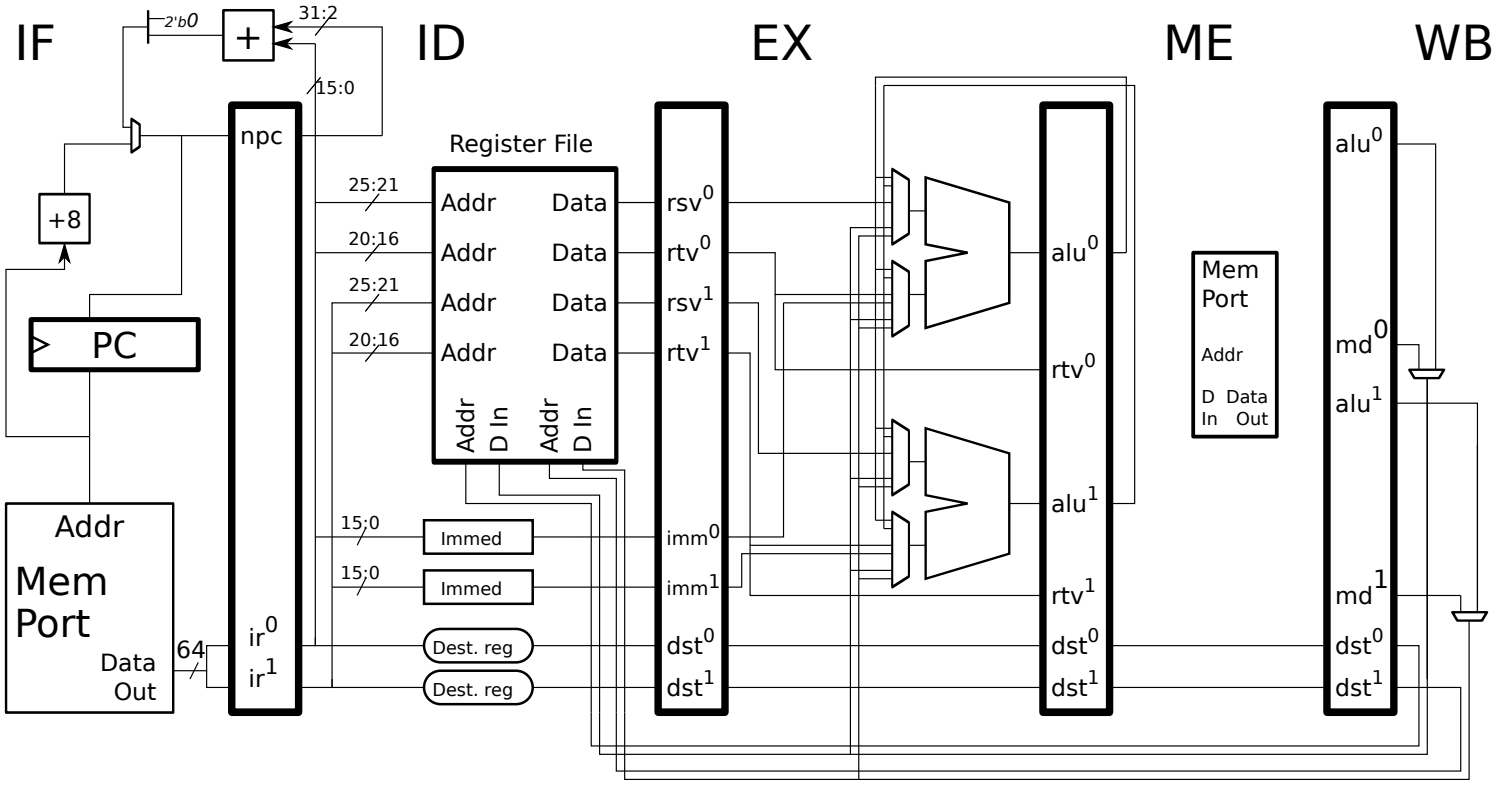
(*d*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Only show `ID`-stage changes. Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values.

☐ Control logic to detect possible shared loads.

☐ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.



The following part was NOT on the final exam as given. It will be assigned as homework in 2017.

(*e*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Here's the plan: In `ID` detect whether a shared load is possible. In `EX` check addresses. In `ME` have memory port perform only one kind of load, 32 b (stores are unaffected). In `WB` have a separate alignment network for each slot.

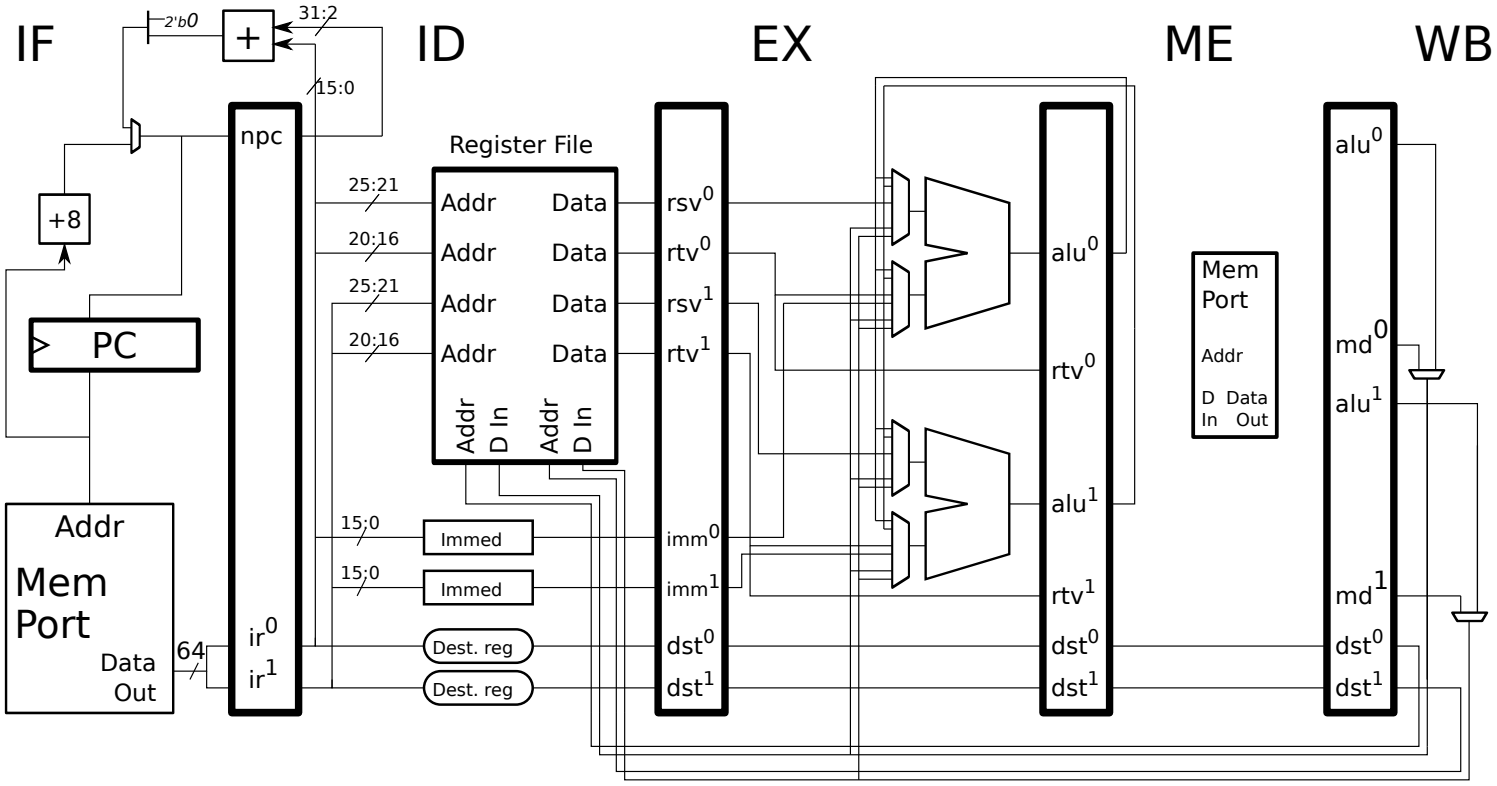
Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values. In `EX` generate a `STALL-EX-1-SL` signal if the shared load cannot be performed. This will stall the pipeline allowing the slot-0 load to proceed normally in the current cycle and the slot-1 load to be executed in the next cycle. Show connections for the alignment units in the appropriate places.

☐ Control logic to detect possible shared loads.

☐ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.

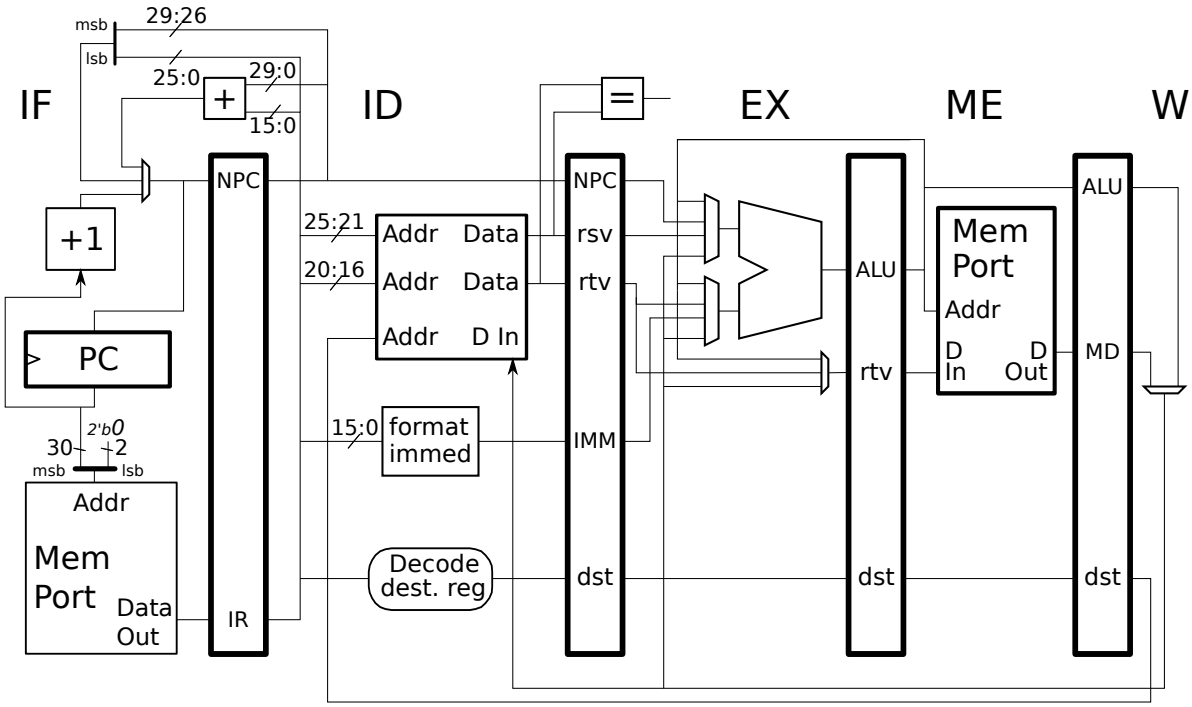
☐ `EX`-stage hardware to generate `STALL-EX-1-SL` if can't do shared load based on addresses.

☐ Add remaining `ME`- and `WB`-stage hardware.



Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken. Don't forget to check for dependencies.

(*a*) Show executions.



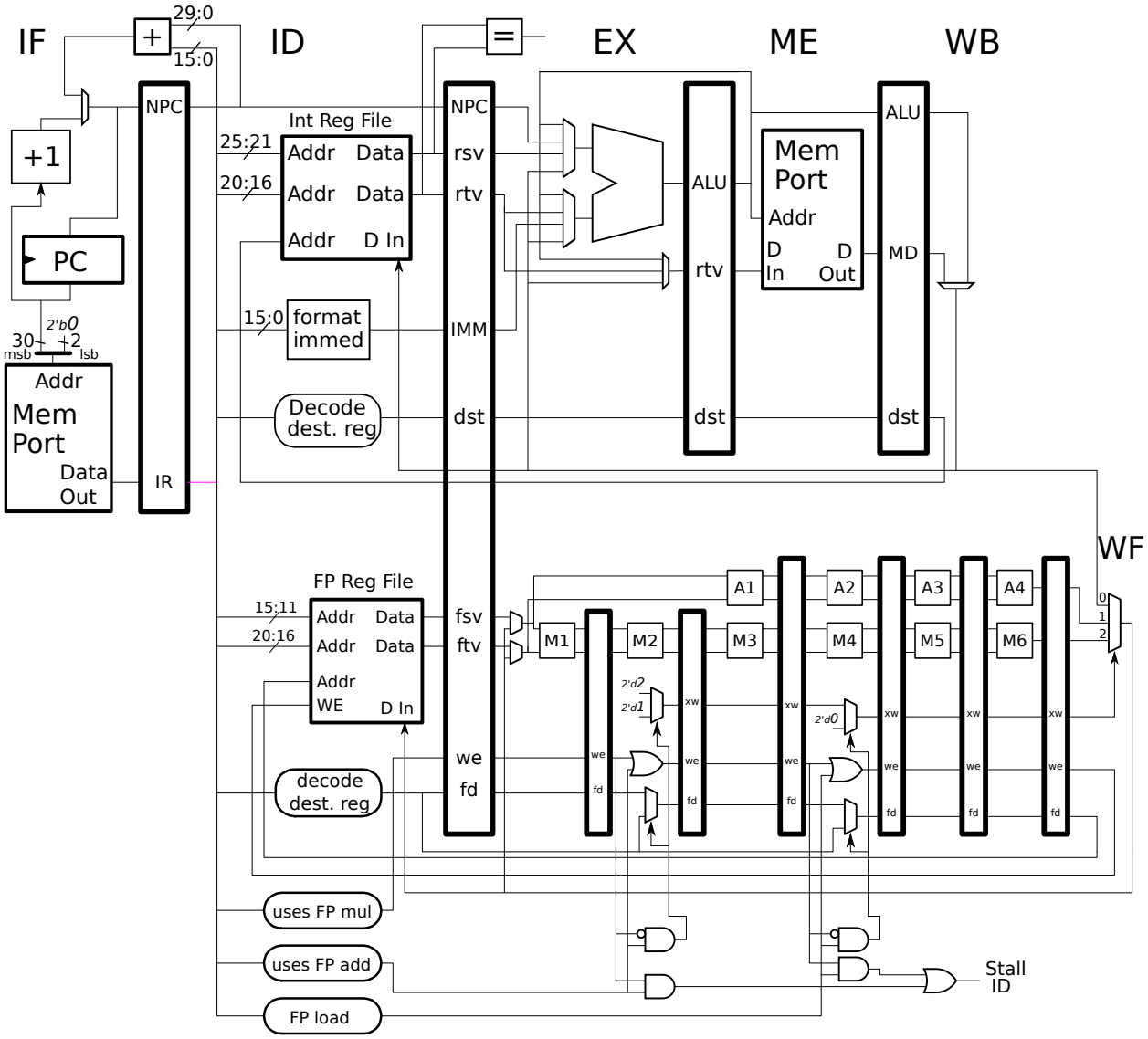
☐ Show execution of this simple code sequence.

```
add r1, r2, r3

sub r4, r1, r5
```

☐ Show execution of the following code sequence.

```
beq r1, r1 TARG

or  r2, r3, r4

sub r5, r6, r7

xor r8, r9, r10

TARG:
lw r10, 0(r11)
```

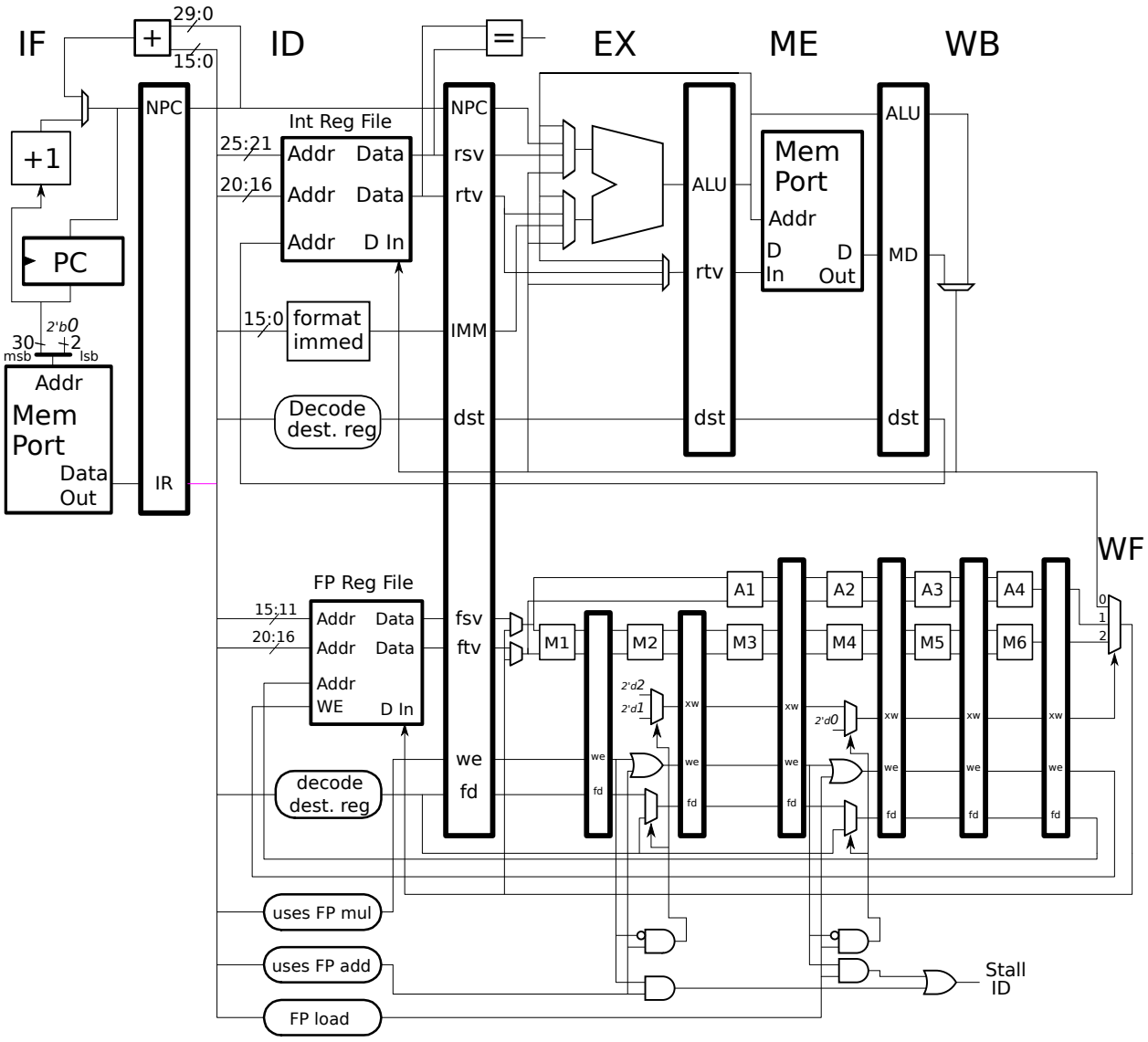
(*b*) Show the execution of the code sequences below on the illustrated MIPS implementation. Don't forget to check for dependencies.



☐ Show execution of the following code sequence.

```
mul.s f5, f6, f7

add.s f1, f2, f3

sub.s f4, f8, f5
```

(*c*) Add any datapath hardware needed to execute the code sequence below, and show its execution. (Control logic is not needed.) Don't forget to check for dependencies.



☐ Add hardware needed by, and ☐ show execution of, the following code sequence.

☐ Consider both stores and, as always, avoid unnecessary costs.

```
add.s f2, f3, f4

swc1 f1, 0(r7)

swc1 f2, 4(r8)
```

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 10-outcome local history, and one system has a global predictor with a 10-outcome global history.

(*a*) Branch behavior is shown below. Two repetitions of a repeating sequence are shown for branches `B1` and `B2`. Branch `B2` generates the following repeating sequence: the eight outcomes `TNTNTNTN` followed by either `NN`, probability .2, or `TT`, probability .8. These last two outcomes are shown below as `r r` and `q q`.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   N   N   T   T   T   N   T   N   N   T   T   T   N   T ···

B2:     T   N   T   N   T   N   T   N   r   r   T   N   T   N T N T N q q ···
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the bimodal predictor on branch `B2`?

☐ What is the accuracy of the local predictor on `B2`?

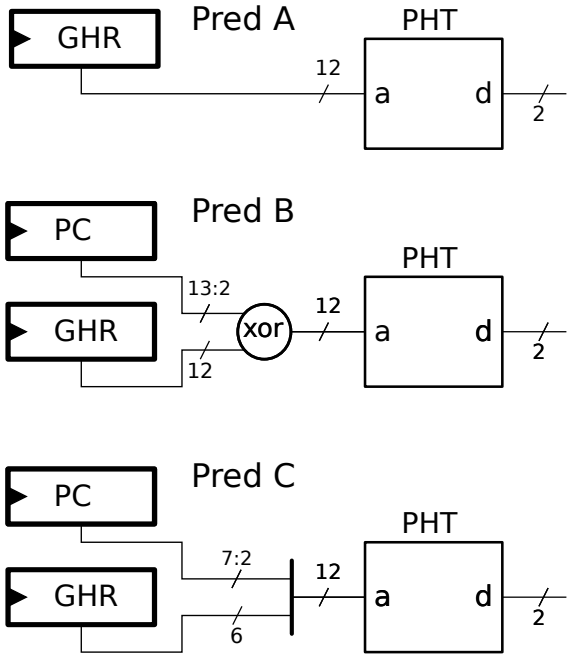
☐ What is the minimum local history size needed to predict `B1` with 100% accuracy?

☐ What is the minimum local history size needed to predict `B2` with maximum accuracy?

Problem 3, continued:

(*b*) The diagram below shows four branches. Branches `B1` and `B3` are loop branches, each in a 30-iteration loop. Branches `B2` and `B4` are perfectly biased branches (`B2` always taken, `B4` never taken). Consider their execution on the three variations of the global predictor shown below.

```
B1:   TT ··· TN                  TT ··· TN                  TT ··· TN
B2:            T                          T                          T
B3:                TT ··· TN                  TT ··· TN
B4:                          N                          N
```



☐ Which variation (A, B, or C) will predict `B2` or `B4` poorly?

☐ Why does the variation do poorly on one or both of these simple to predict branches?

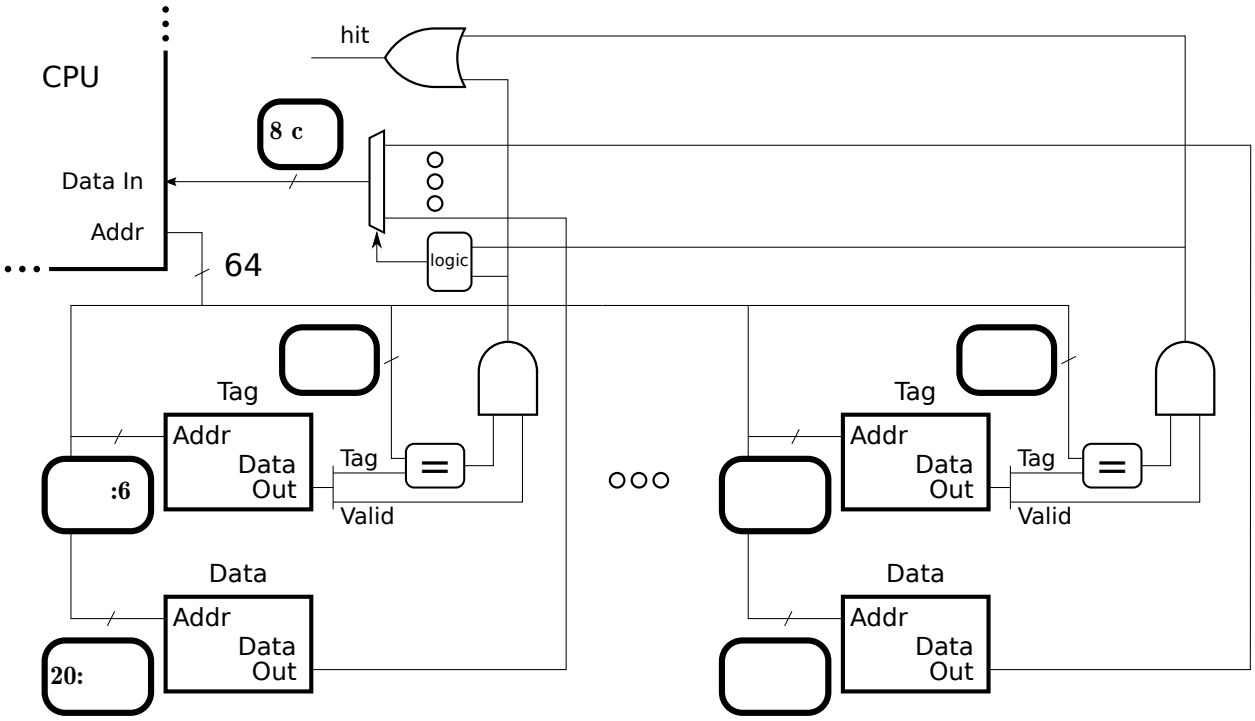
☐ Which variation (A, B, or C) is best for `B2` and `B4` and also for branches with **repeating patterns**?

☐ Show an example that one variation predicts accurately that the other can not.

Problem 4: (20 pts) The diagram below is for a three-way set-associative cache. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

Address: ☐ ☐ ☐

0

☐ Cache Capacity ☐ Indicate Unit!!:

☐ Memory Needed to Implement ☐ Indicate Unit!!:

☐ Line Size ☐ Indicate Unit!!:

☐ Show the bit categorization for the **smallest** direct mapped cache with the same line size and which can hold at least as much data as the cache above.

Address: ☐ ☐ ☐

Problem 4, continued: The problems on the following pages are **not** based on the cache from Part a. The code in the problems below run on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
long double sum = 0;
long double *a = 0x2000000; // sizeof(long double) == 16
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Problem 4, continued: The code in the problem below runs on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*c*) The code below scans through an array (or arrays) twice, once in the `First Loop` and a second time in the `Second Loop`. Three different variations are shown, Plan A, Plan C, and Plan D. For each Plan, find the largest value of `BSIZE` for which the second loop will have a 100% hit ratio.

```
// Note:  sizeof(double) == 8

// Plan A
struct Some_Struct   { double val, norm_val;   double stuff[14];   };

// Plan C
struct Some_Struct   { double val, norm_val; };
struct Some_Struct_2 { double stuff[14];};

// Plan D
double *vals, *norm_vals;
struct Some_Struct_2 { double stuff[14]; };

// Plan A and Plan C
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second Loop.

// Plan D
  for ( int i = 0;  i < BSIZE;  i++ ) sum += vals[i] + norm_vals[i];   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) norm_vals[i] = vals[i] / sum;   // Second Loop.
```

☐ Largest `BSIZE` for Plan A. ☐ Show work or explain.

☐ Largest `BSIZE` for Plan C. ☐ Show work or explain.

☐ Largest `BSIZE` for Plan D: ☐ Show work or explain.

(*d*) What's wrong with the statement below:

In the first iteration of the First Loop under Plan C there will be one miss but with Plan D there will be two misses. Therefore Plan C is better, at least for the First Loop.

☐ What's wrong with the statement?

Problem 5: (20 pts) Answer each question below.

(*a*) Suppose that a new ISA is being designed. Rather than requiring implementations to include control logic to detect dependencies the ISA will require that dependent instructions be separated by at least six instructions. As a result, less hardware will be used in the first implementation.

☐ Explain why this is considered the wrong approach for most ISAs.

☐ What is the disadvantage of imposing this separation requirement?

(*b*) In the execution below the `add.s` instruction encounters an arithmetic overflow when in the `A4` stage and as a result raises an exception in cycle 6.

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
add.s  f1, f2, f3  IF ID A1 A2 A3 A* WFx
lwc1 f3, 0(r2)        IF ID EX ME WB
lw r4, 0(r5)             IF ID EX ME WBx
add r6, r7, r8              IF ID EX MEx

HANDLER:
sw                                     IF ID ..
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```

☐ Explain why this exception can't be precise.

☐ Describe something the handler could do if the exception were precise.

(*c*) Chip A has 60 2-way superscalar cores and Chip B has 20 4-way superscalar cores. Both chips run at 3 GHz, and so Chip A has a higher peak instruction throughput. The cost and power consumption of the two chips are identical. Why might someone prefer Chip B over Chip A?

☐ Chip B preferred, despite lower peak throughput, because:

(*d*) The SPECcpu suite is used to test new chips that might have cost hundreds of millions of dollars to develop, and test results are closely watched.

☐ Why might we trust the SPECcpu selection of benchmarks and rules for building and running the benchmarks?

☐ Why might we trust the SPECcpu scores that Company X publishes?

11 Spring 2015

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 20 March 2015, 9:30–10:20 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Appearing on the next page is the *HW 3 Implementation*, one of the solutions to Homework 3, Problem 2, in which the `bgt` instruction resolves in `EX`. In the HW 3 Implementation there is a 1-cycle branch penalty when `bgt` is taken. (The branch penalty is the number of squashed instructions.) Suppose that based on benchmark analyses we find that `bgt` is mostly taken. For that we would like a New Implementation [tm] in which there is no penalty when `bgt` is taken, and a 1-cycle penalty when it's not taken. The PEDs below show execution examples for the HW3 and New implementations.

```
 # Cycle          0  1  2  3  4  5  6  7   HW 3 Impl, bgt taken, 1-cyc penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF IDx
TARG:
 and r9, r10, r11           IF ID EX ME WB
```

```
 # Cycle          0  1  2  3  4  5  6  7   HW 3 Impl, bgt not taken, no penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF ID EX ME WB
TARG:
 and r9, r10, r11
```

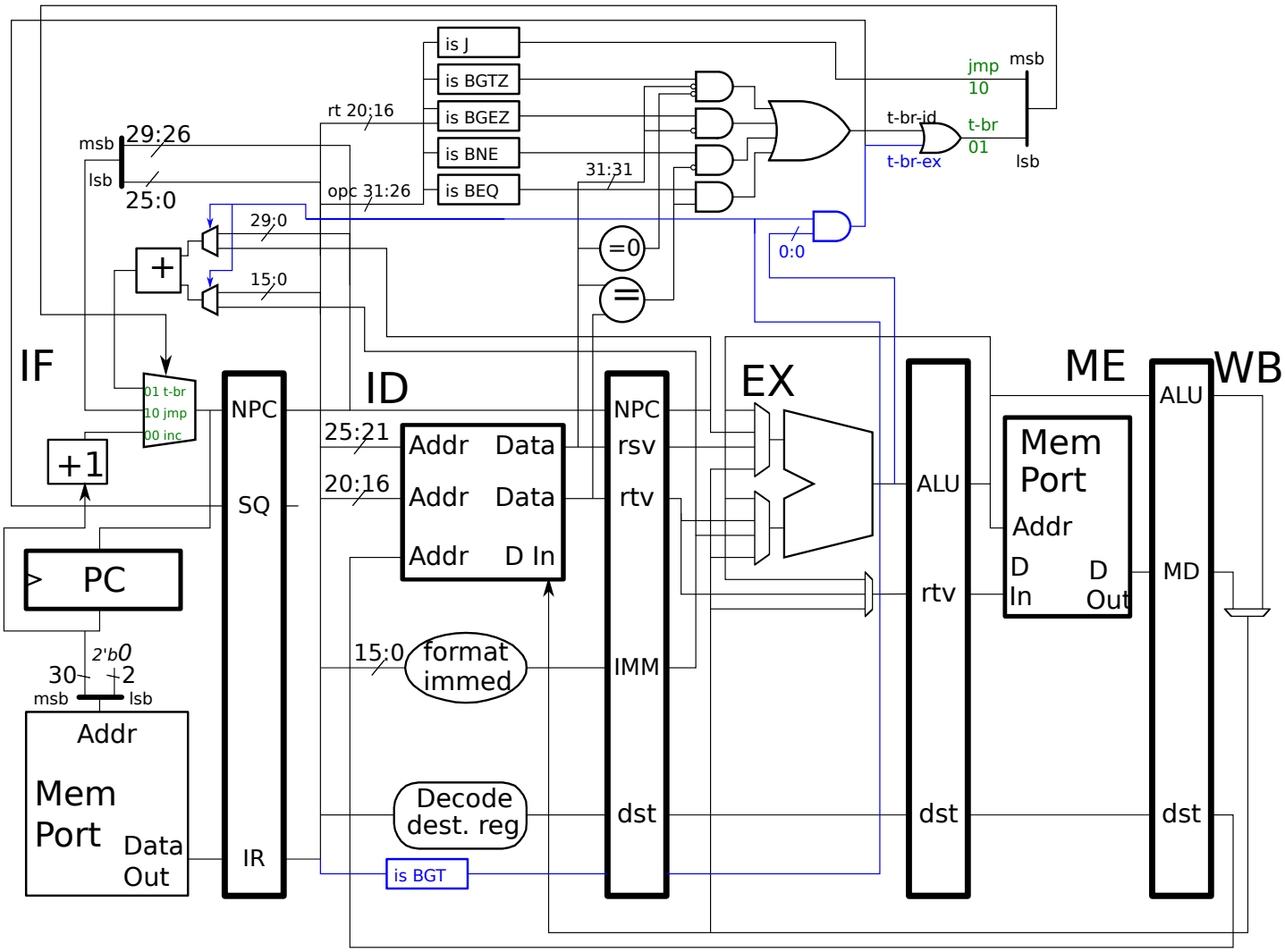
```
 # Cycle          0  1  2  3  4  5  6  7   New Impl, bgt taken, no penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8
TARG:
 and r9, r10, r11        IF ID EX ME WB
```

```
 # Cycle          0  1  2  3  4  5  6  7   New Impl, bgt not taken, 1-cyc penalty.
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF ID EX ME WB
TARG:
 and r9, r10, r11        IF IDx
 # Cycle          0  1  2  3  4  5  6  7
```

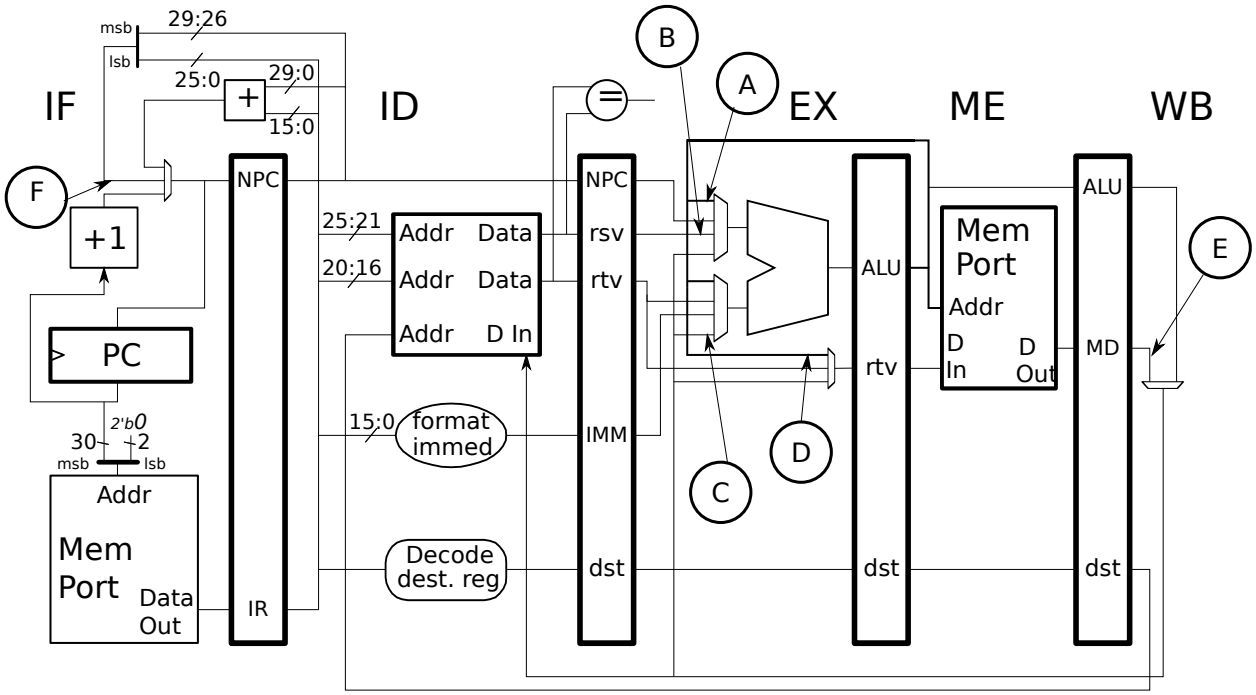
Problem 1, continued: Convert the HW 3 Implementation below into the New Implementation. *Hint: Calm down! A correct solution only requires three minor changes, one of those changes is substituting a mux input with a constant.*

☐ `ID` changes: `bgt` acts like it's always taken.

☐ `EX` changes: if `bgt` resolves not taken, fetch insn after delay slot insn.



Problem 2: [25 pts] In the implementation below several multiplexor inputs are labeled. For each labeled input write a program that uses it. A sample solution is provided for **A**.



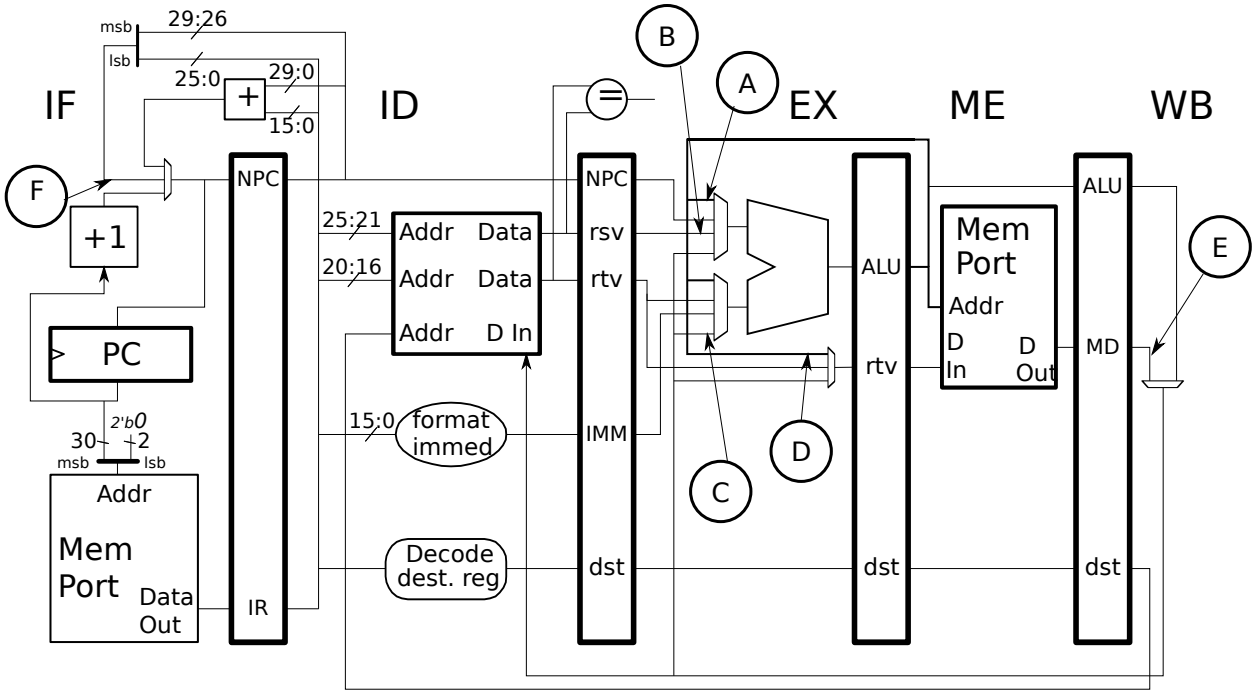
☑ Write a code fragment for mux input **A**. ☑ Mux input used in cycle 3, ☑ for register r1.

```
# SAMPLE SOLUTION -- Mux input A.
# Cycle          0  1  2  3  4  5
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☐ Write a code fragment for mux input **B**. ☐ Mux input used in cycle _____, ☐ for register _____.

☐ Write a code fragment for mux input **C**. ☐ Mux input used in cycle _____, ☐ for register _____.

Problem 2, continued:



☐ Write a code fragment for mux input **D**. ☐ Mux input used in cycle _____, ☐ for register _____.

☐ Write a code fragment for mux input **E**. ☐ Mux input used in cycle _____, ☐ for register _____.

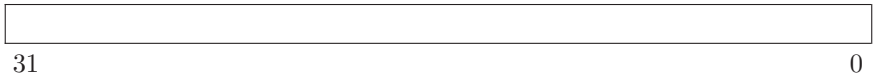
☐ Write a code fragment for mux input **F**. ☐ Mux input used in cycle _____.

Problem 3: [15 pts] Answer each question below.

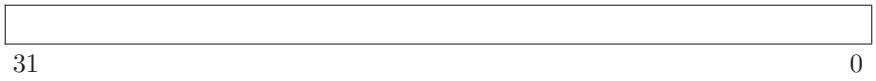
(*a*) Show the encoding of the MIPS instructions below. The opcode for `beq` is `0x4` and the opcode for `lw` is `0x23`. *Hint: For the fields' bit positions see the implementation diagrams in previous problems.*

```
TARG:
 lw r1, 2(r3)
 beq r4, r5, TARG
```

☐ Encoding of `lw r1, 2(r3)`:



☐ Encoding of the `beq` above ☐ with the correct immediate field value.



(*b*) Many MIPS Format-R instructions, such as `add` and `sub`, have an opcode value of 0. Explain why they don't have their own opcode field values, such as `0x11` for `add` and `0x12` for `sub`.

☐ R-format instructions have opcode 0 because ...

☐ If R-format instructions each had their own opcodes that would be a problem because ...

Problem 4: [10 pts] Answer each question below.

(*a*) Describe what the dead-code elimination optimization is and provide an example.

☐ Description of dead-code elimination.

☐ Example.

(*b*) For instruction scheduling optimizations is it necessary or just helpful for the the compiler to know the implementation?

☐ Implementation: Necessary, important, not needed. ☐ Circle one ☐ and explain.

Problem 5: [10 pts] Answer each question below.

(*a*) A CISC ISA might have an instruction like `add (r1), r2, 8(r3)`, in which the source and destination come from memory.

☐ Explain why such an instruction is not suitable for a RISC ISA, ☐ refer to RISC goals in your answer.

(*b*) Describe a feature and goal of VLIW ISAs.

☐ VLIW feature:

☐ VLIW goal:

Problem 6: [15 pts] Answer each question below.

(*a*) SPECcpu benchmark results have two levels of tuning, *peak* and *base*. In one of these tuning levels the same optimization flags must be used for all benchmarks of the same language.

☐ Which tuning level is this for?

☐ What is the purpose for requiring the same flags?

(*b*) Explain how the following corruptions of SPECcpu would be (we hope) prevented.

☐ *The suite excludes benchmarks that perform poorly on Evil Company's products.* This won't happen because . . .

☐ *The SPECcpu results for Evil Company's System X are just made-up numbers.* This won't happen because . . .

Name _____

Computer Architecture
EE 4720
Final Examination

4 May 2015, 10:00–12:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

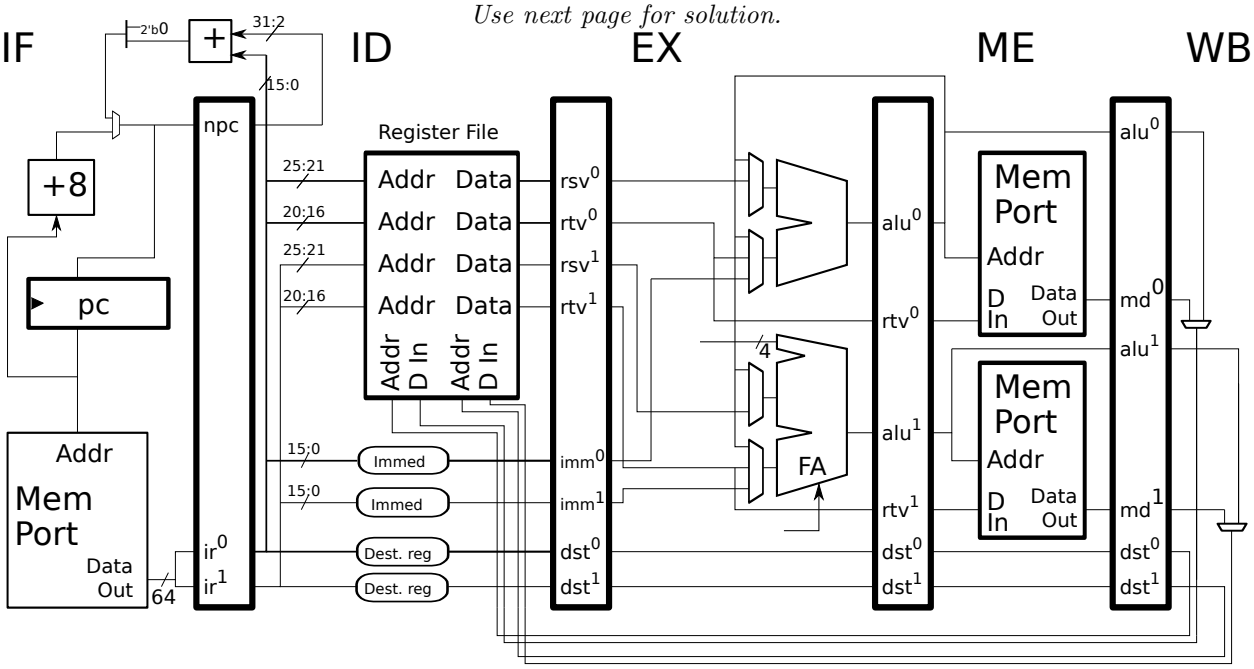
Problem 1: (20 pts) The instruction sequence below performs the calculation `r2+3+r5` and requires two MIPS instructions to do so. If these two instructions were in the same fetch group in a two-way superscalar processor then the second one would stall, this is shown in the execution below.

```
addi r1, r2, 3   IF ID EX ME WB
add  r4, r1, r5  IF ID -> EX ME WB
```

An *FA-ALU* has two regular inputs, and a third input reserved for small quantities, here for values < 16, and it can produce a result in the same time as an ordinary two-input ALU. An FA-ALU can be used to execute instruction pairs like the one above without a stall, call such an execution a *fused add*.

Illustrated below is a 2-way superscalar implementation with an FA-ALU. Most bypass paths have been removed to provide space for the solution to this problem. In addition to a 4-bit input for the third operand, the FA-ALU has a control input, `FA`, which should be set to `1` when the third input is to be used.

Use next page for solution.



Use next page for solution.

(*a*) Add datapath connections to this implementation so that the FA-ALU can be used for fused adds. *Hint: The datapath changes are not just for that new FA-ALU input.*

☐ Add datapath so that the correct operands are delivered to the FA-ALU for fused adds.

☐ Non-fusable sequences must still execute correctly. (Don't break existing functionality.)

☐ Make reasonable cost and clock frequency tradeoffs.

(*b*) Add control logic.

☐ Design logic to generate the `FA` signal for the FA-ALU and deliver it in the correct cycle.

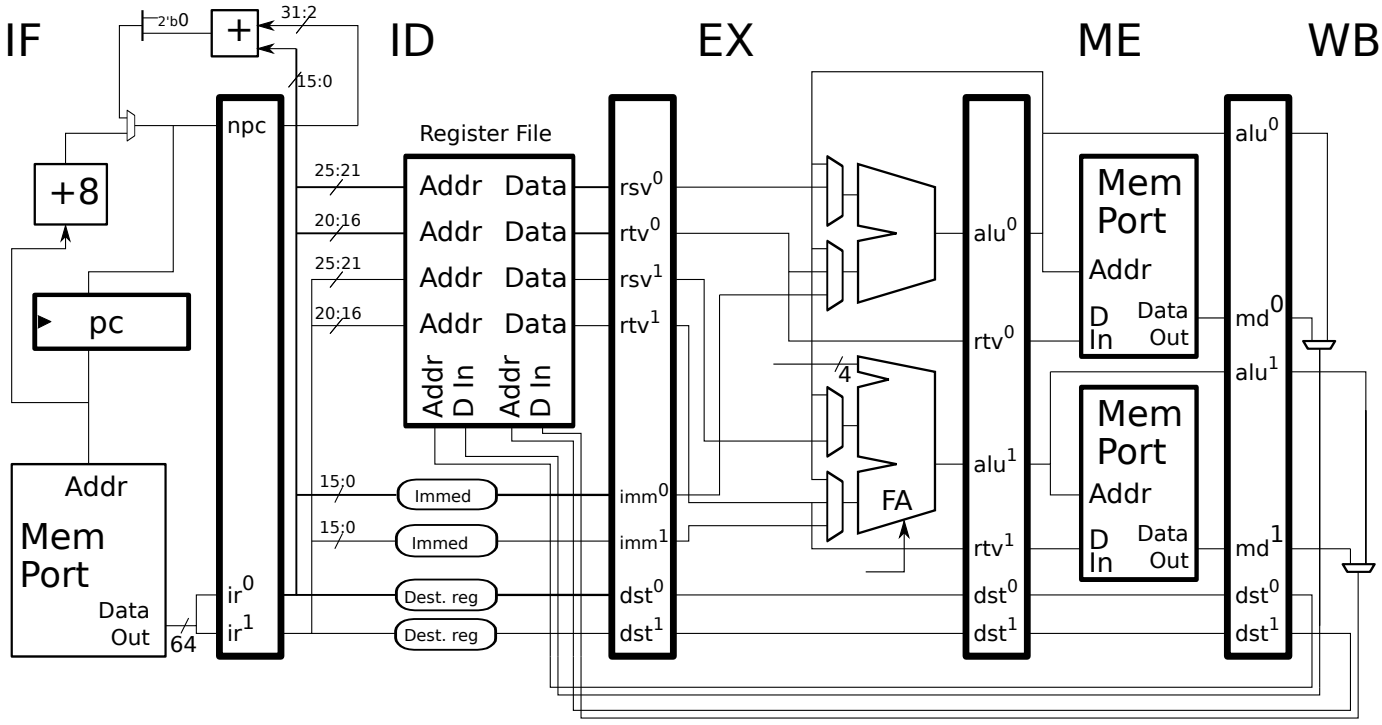
☐ Consider ☐ instruction type, ☐ dependencies, and ☐ operand size.

☐ Add control signals for the datapath added in the previous part.

☐ Make reasonable cost and clock frequency tradeoffs.

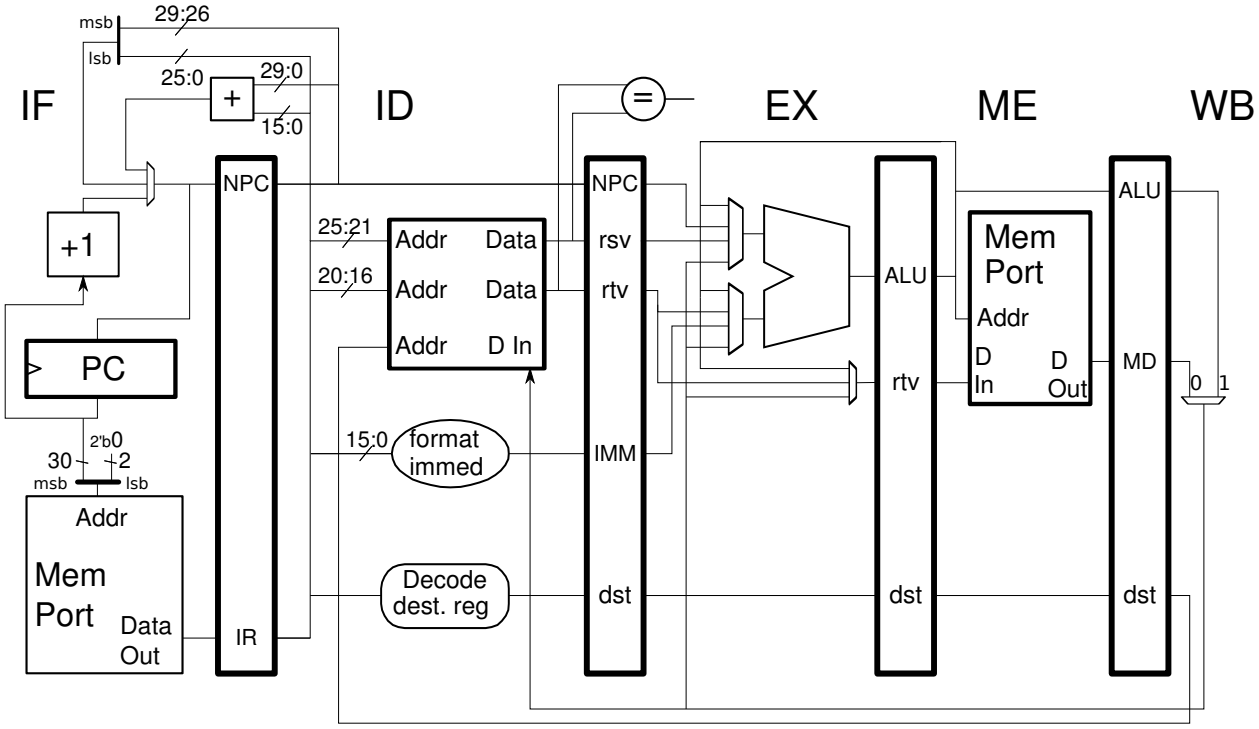
Problem 1, continued: You can use logic blocks such as isADD and isADDI in your solution.

```
addi r1, r2, 3    # Sample fusable sequence.
add  r4, r1, r5
```



Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken.

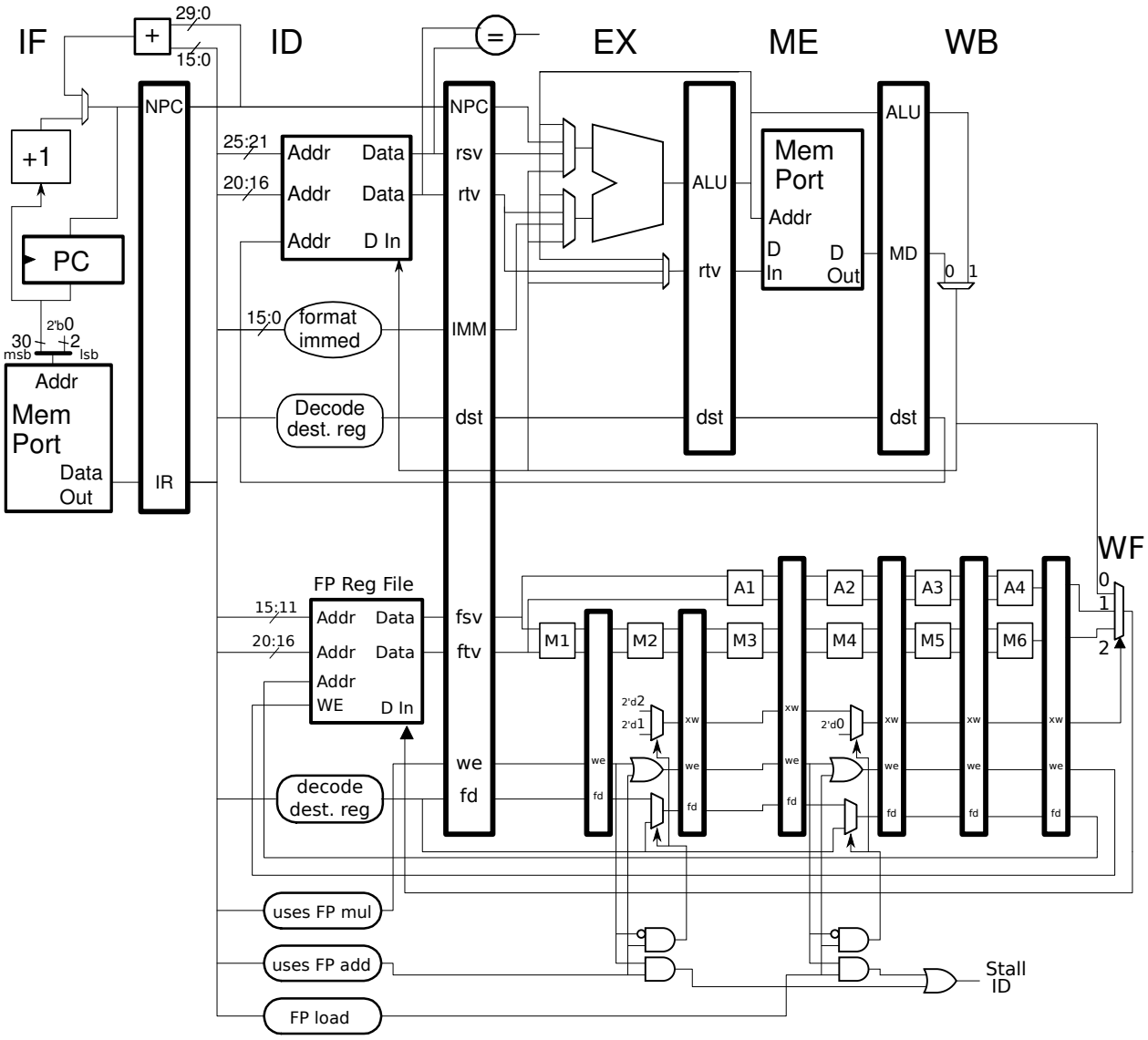
(*a*) Show the execution of the code below on the following implementation. The branch is taken.



☐ Show execution. ☐ Doublecheck for dependencies.

```
 add r4, r2, r3

 lw r6, 8(r4)

 sub r1, r6, r5

 beq r7, r1 TARG

 and r8, r7, r10

 or  r11, r12, r13

 xor r14, r11, r8

TARG:
 sw r1, 0(r2)
```

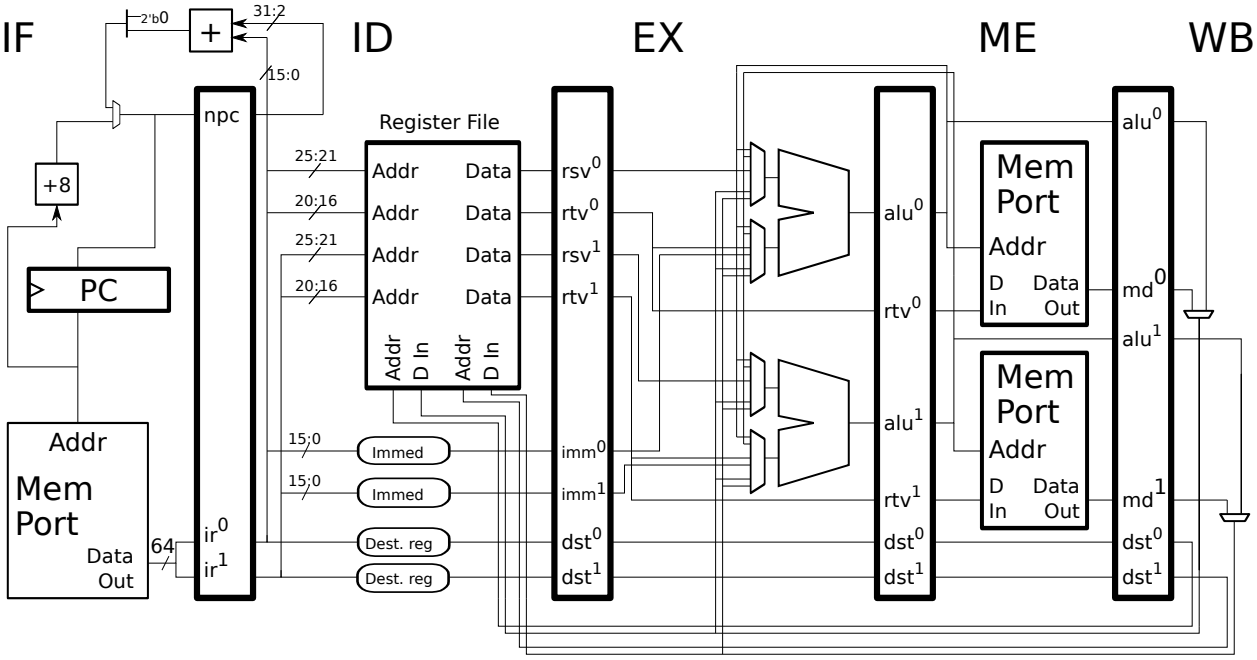
(*b*) Show the execution of the MIPS FP code below on the following implementation. Assume that **there is** a bypass from `WF` to `M1` and `A1`.



☐ Show code execution. ☐ Doublecheck for dependencies.

```
add.s  f6, f2, f3

add.s  f1, f7, f8

lwc1   f9, 0(r10)

mul.s  f4, f1, f9
```

(*c*) Show the execution of the MIPS code below on the following 2-way superscalar implementation. The branch is taken. Fetch groups in this implementation must be 8-byte-aligned.



☐ Show code execution. ☐ Show squashed instructions with an `x`.

☐ Doublecheck for dependencies. ☐ Account for aligned fetch groups.

```
START: Instruction address of add is 0x1000

 add r4, r2, r3

 lw r6, 8(r4)

 sub r1, r6, r5

 beq r7, r15 TARG

 and r8, r7, r10

 or  r11, r12, r13

 xor r14, r11, r8



 sll r16, r17, 8

TARG: Instruction address of sw is 0x2004

 sw r1, 0(r2)

 lui r15, 0x1234
```

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{30} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(*a*) Branch behavior is shown below. Branch `B2` consists of a pattern in which there will be either 2, 4, or 6 `T` outcomes followed by exactly four `N` outcomes. After the fourth `N` the probability of exactly two `T`s is $\frac{1}{3}$ and the probability of exactly four `T`s is $\frac{1}{3}$.

Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  N    T    T    N    N    N      N    T    T    N    N    N   ...
B2:   T{2,4,6}  N    N    N    N      T{2,4,6}  N    N    N    N  ...
B3:    T    T    T    T    T    T      T    T    T    T    T    T ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the accuracy of the bimodal predictor on branch `B2`? ☐ Explain.

☐ What is the minimum local history size needed to predict `B1` with 100% accuracy?

☐ What is the accuracy of the local predictor on branch `B2`, after warmup. ☐ Explain.

☐ What is the minimum GHR size to predict `B1` with about 100% accuracy. ☐ Explain, and state any assumption about `B2`'s behavior.

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries.

Branch `B1` below is perfectly biased not taken. It executes r_1 times over a short time interval, followed by a big time gap before its next bunch of r_1 executions. Branch `B2` behaves similarly, except that it is biased taken and executes in bunches of length r_2 executions.

```
0xee4720  B1:   N..N r1                          N..N r1
# ...
          B2:             T..T r2                          T..T r2
```

(*b*) Choose an address for branch `B2` that will result in a BHT collision with branch `B1`. (Branch `B1` is at address `0xee4720`.)

☐ Address for `B2` that results in a collision.

(*c*) Assume that branch `B1` and `B2` collide in the BHT. For what positive values of r_1 and r_2 will this collision be most harmful? For what range of positive values of r_1 and r_2 will collisions be least harmful?

☐ Worst values of r_1 and r_2 for collision. ☐ Explain.

☐ Favorable range of positive values of r_1 and r_2 for collision. ☐ Explain.

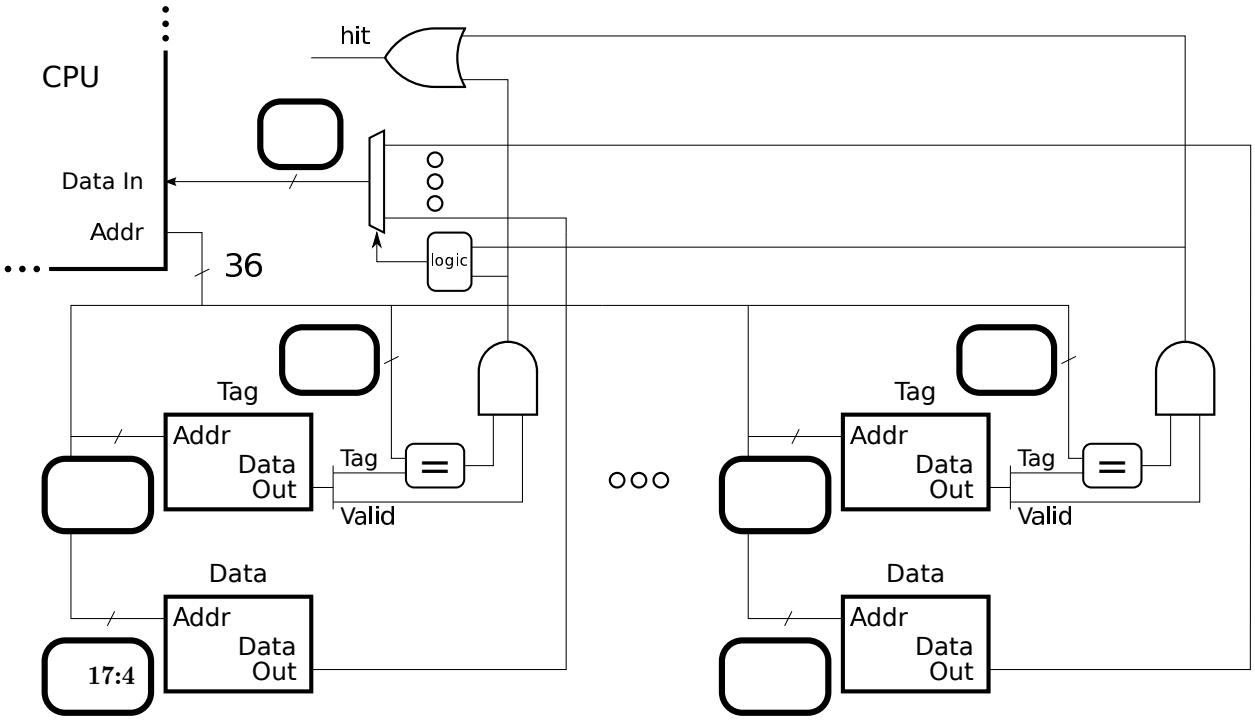
(*d*) A tag can be placed in the BHT to detect collisions. Which branch will this help?

☐ Branch that would benefit from tag: `B1` or `B2`. ☐ Explain.

Problem 4: (20 pts) The diagram below is for a 4 MiB (2^{22} B) set-associative cache with a line size of 128 bytes. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

Address: ☐ ☐ ☐

0

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a **direct mapped** cache with the same capacity and line size.

Address: ☐ ☐ ☐

Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 64 bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
half *a = 0x2000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

(*c*) The code appearing below is the same as the code from the 2014 Final Exam and 2015 Homework 6, except that two possibilities for `Some_Struct` are shown, Plan A, and Plan B. The caches too are identical except that here the line size is 64 bytes whereas in the prior problems it was 128 bytes. The value of `BSIZE` has been chosen to the largest value for which the hit ratio of the second loop will be 100% when using the Plan B struct. *Note: In the original exam* `norm_val` *was omitted from the first loop.*

```
struct Some_Struct { // Plan A
  double val;        // sizeof(double) = 8
  double norm_val;
  double a1[14];   };

struct Some_Struct {  // Plan B
  double val;         // sizeof(double) = 8
  double a1[7];
  double norm_val;
  double a2[7];    };

  const int BSIZE = 1 << 15;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;  // First loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second loop.
```

☐ Explain why the Plan A struct will result in better performance on the first loop than the Plan B struct.

☐ Suppose that the value of `BSIZE` is to be chosen based on the Plan A struct, to a maximum value for which the second loop enjoys a 100% hit ratio. Explain why that value is the same as for Plan B.

Problem 5: (20 pts) Answer each question below.

(*a*) Which kind of implementations were RISC ISAs originally designed to simplify? Explain how a RISC ISA feature achieves this goal.

☐ Type of implementation RISC designed to simplify.

☐ Choose a feature and ☐ explain how it achieves the goal.

(*b*) What is the difference between a trap instruction and an instruction that raises an exception? Give an example of how each is used in a system with bug-free code.

☐ Difference between trap and exception.

☐ Example of what trap instruction used for.

☐ Example of what instruction exception used for.

Problem 5, continued:

(*c*) Consider a $5n$-stage scalar implementation and an n-way superscalar implementation, both derived from our 5-stage design with the goal of improving performance by as much as $n\times$. As n increases the clock frequency of the $5n$-stage implementation increases but the frequency of the n-way superscalar implementation decreases. Explain why. Assume that both types of system are well designed.

☐ Clock increases in the $5n$-stage system because ...

☐ Clock decreases in the n-way system because ...

(*d*) A company is preparing a run of the SPEC benchmarks for their new system. While trying to tweak compiler flags to get the best performance they discover a new optimization technique and implement it in their compiler. They re-test with that compiler and disclose the test results. When is the use of the modified compiler allowed under SPEC rules? When isn't that allowed under SPEC rules?

☐ Modified compiler allowed provided that ...

☐ This is a beneficial because ...

☐ Modified compiler isn't allowed if ...

☐ The modified compiler is not beneficial because ...

12 Spring 2014

Name _____

Computer Architecture
EE 4720
Midterm Examination

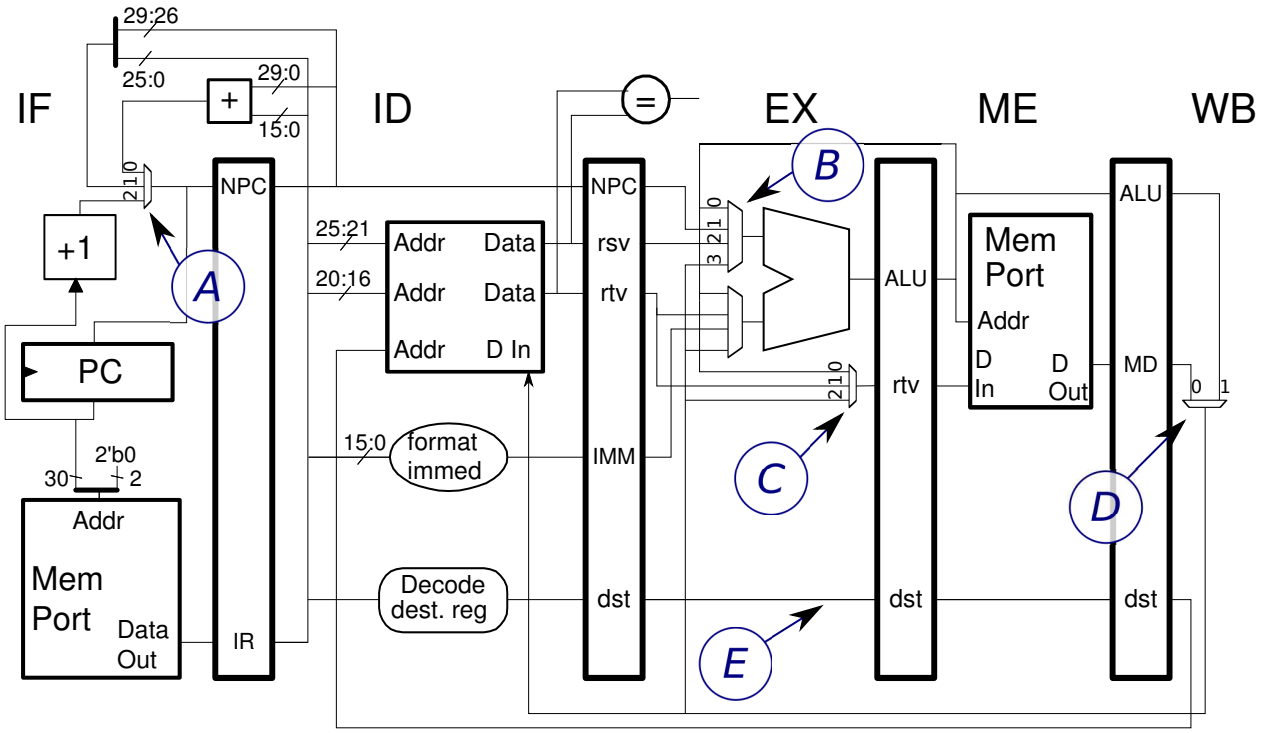
Monday, 31 March 2014, 9:30–10:20 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(18 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(6 pts)
Problem 6	_____	(12 pts)
Problem 7	_____	(12 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The MIPS code fragment below executes on our familiar five-stage scalar MIPS implementation.



(*a*) Show the execution of the code for enough iterations to determine CPI, and determine the CPI assuming a large number of iterations.

(*b*) There are five labels in the diagram, labels A through D point at multiplexor control inputs and label E points at the output of the `EX.dst` pipeline latch. Show the values of these signals up until the second execution of `lw r1`. For A show only values $\neq 2$, for E show only values $\neq 0$, for the others only show values at cycles in which the multiplexor is in use.

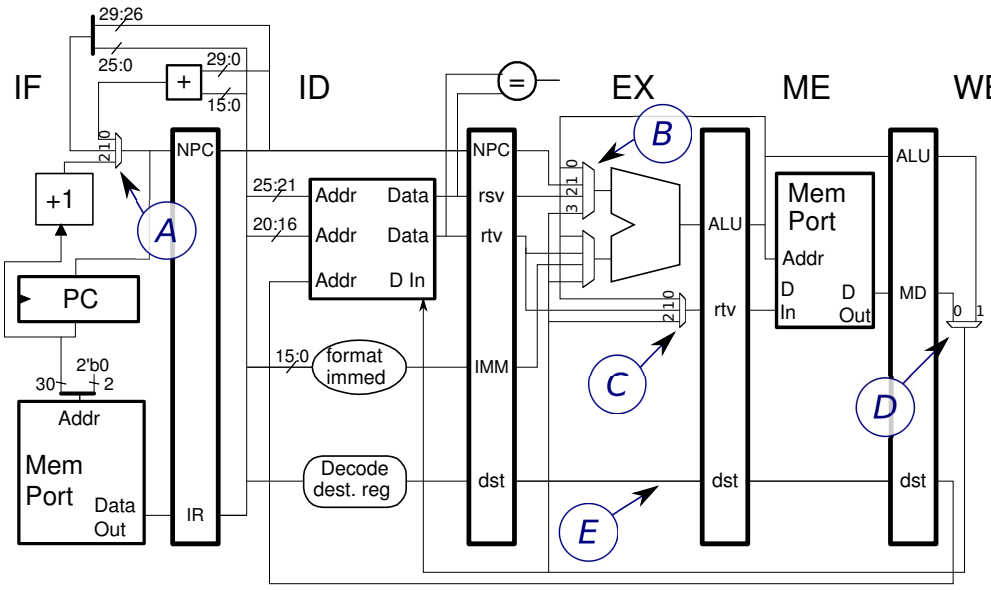
USE NEXT PAGE FOR SOLUTION

```
LOOP: Cycle        0
 A                 2
 B
 C
 D
 E                 0
 lw r2, 0(r5)      IF
LOOP:
 lw r1, 0(r2)
 sh r1, 16(r2)
 bne r1, r0, LOOP
 add r2, r2, r3
 xor r4, r5, r2
```

USE NEXT PAGE FOR SOLUTION

Problem 1, continued:

- ☐ Check code for dependencies.
- ☐ Show pipeline execution diagram for enough iterations to determine CPI.
- ☐ Determine the CPI.
- ☐ Show values for *A* through *E*.

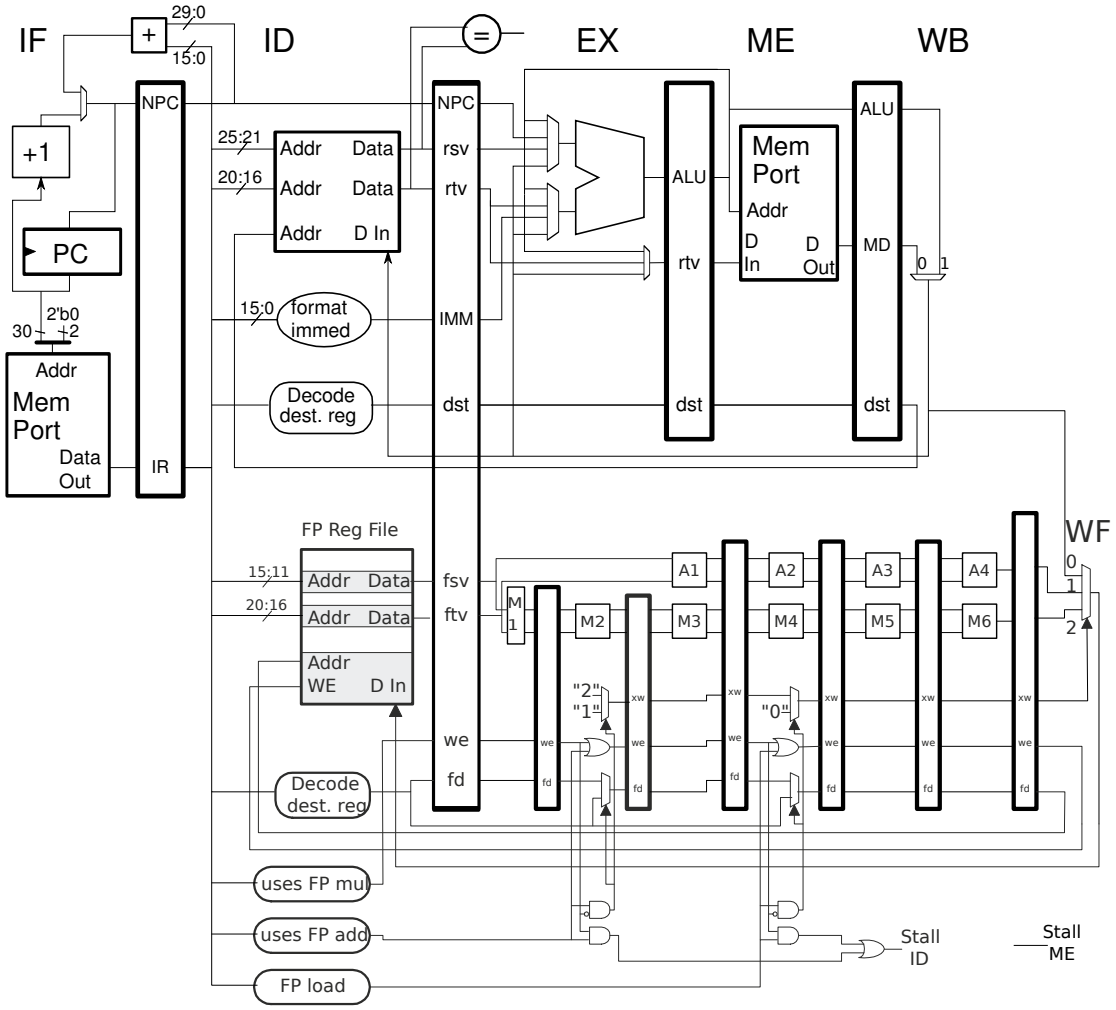



```
LOOP: Cycle       0

 A                2

 B

 C

 D

 E                0

 lw r2, 0(r5)     IF
LOOP:
 lw r1, 0(r2)

 sh r1, 16(r2)

 bne r1, r0, LOOP

 add r2, r2, r3

 xor r4, r5, r2
```

Problem 2: [20 pts] The pipeline execution diagram below shows the *incorrect* execution of the MIPS code for the illustrated pipeline. That is, in the pipeline below the stall to avoid the structural hazard would have occurred when `lwc1` was in the `ID` stage.

```
# Cycle            0  1  2  3  4  5  6  7
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
addi r1, r1, 4        IF ID EX ME WB
lwc1  f4, 0(r1)          IF ID EX ME -> WF
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

As described below, modify the pipeline so that FP load instructions, such as `lwc1`, will stall when they are in `ME` rather than `ID` to avoid a `WF` structural hazard.

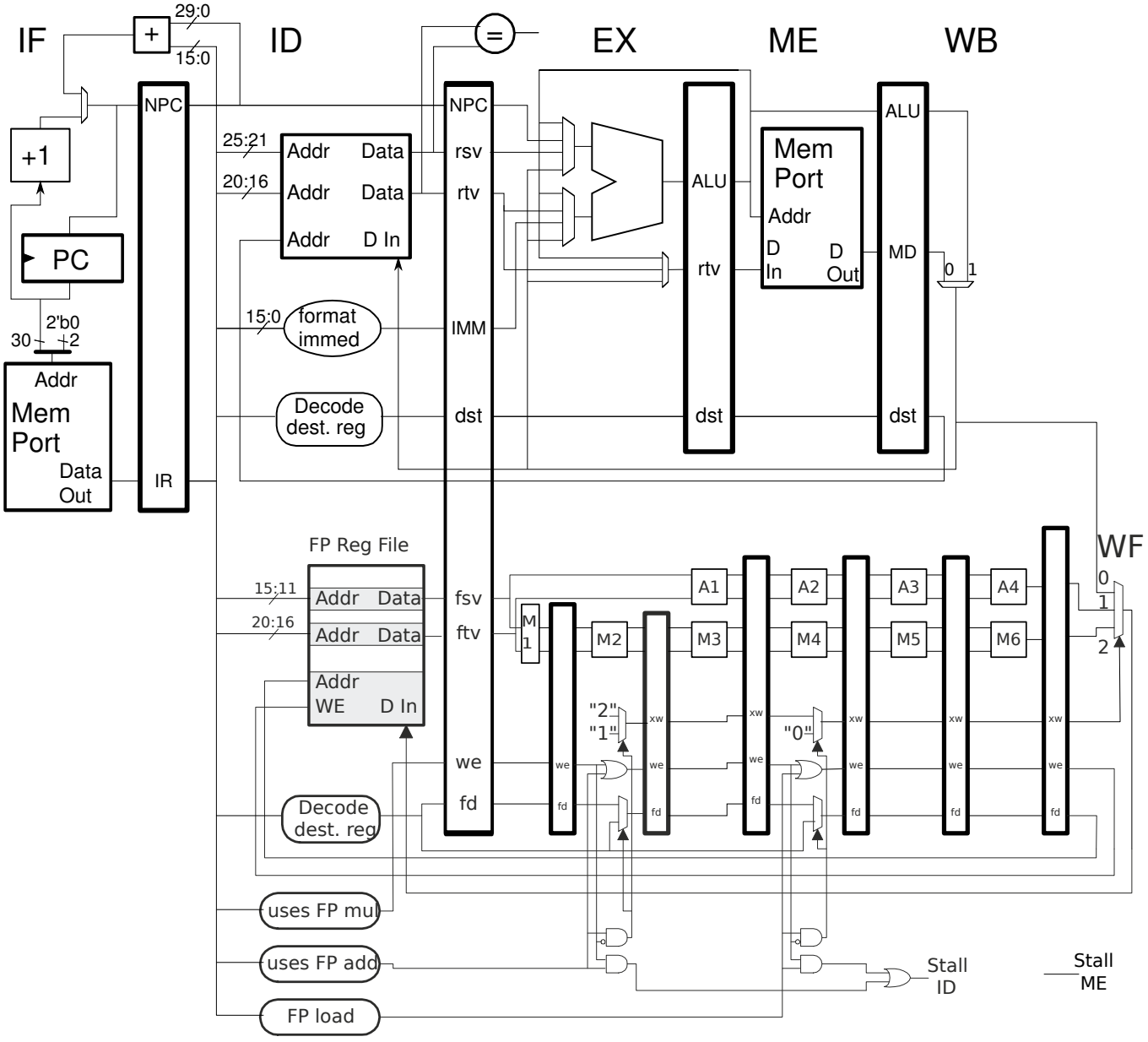
(*a*) Show logic to generate a signal `Stall ME` in the correct cycle. When `Stall ME` is logic `1` stages `ME` and earlier will stall.

(*b*) Cross out the logic generating the `Stall ID` for the `lwc1` that is no longer needed.

(*c*) Make other changes, including control logic, so that the correct destination register number, destination value, and write-enable signal are delivered to the FP register file. *Hint: Some solutions are simpler than others.*

Problem 2, continued:

- ☐ Generate `Stall ME` to avoid `lwc1 WF` structural hazard. ☐ Make sure signal is `1` in the right cycle.
- ☐ Cross out `Stall ID` logic for FP loads, but leave logic for other instructions.
- ☐ Make sure that correct `fd`, `we`, and value delivered to FP register file.



Problem 3: [18 pts] Answer each question below.

(*a*) Re-write the SPARC code fragment below in MIPS. Pay attention to the load instruction and the instructions related to the branch. Don't forget that in SPARC assembly language the last register is the destination.

```
!
 addcc %g1, %g2, %g3
 ld [%g5+%g3], %g4
 bg TARGET          ! Branch greater than zero.
 add %g5, 22, %g6   ! g6 = g5 + 22
```

☐ MIPS equivalent of SPARC code. ☐ Pay attention to load and branch-related insn.

(*b*) Show a MIPS equivalent of the ARM A32 code below. The MIPS code will use more than one instruction. (This is based on Homework 3.) *Note: A32 added in 2017 to avoid confusion with the A64 instruction set.*

```
add r1, r2, r3, LSL #4
```

☐ MIPS equivalent of ARM code above.

(*c*) Explain why the MIPS code fragment below is certain to execute with an error.

```
lw r1, 0(r2)
lw r3, 1(r2)
```

☐ Error is certain because:

Problem 4: [12 pts] Answer each question below.

(*a*) In the execution below the `ant` instruction raises an illegal instruction exception and the handler is fetched, note that the handler starts execution in cycle 4. Explain why this exception (as executed) cannot be precise and show how execution should proceed for our five-stage pipeline.

```
# Cycle             0  1  2  3  4  5  6
 add r10, r11, r12  IF ID EX ME WB
 sw r3, 4(r5)          IF ID EX x
 ant r1, r2, r5           IF*ID*x
 or  r5, r6, r7              IF x

HANDLER:
 sw                             IF ID ..
```

☐ Reason why this execution not for a precise exception.

☐ Show execution that could be precise when `ID` stage discovers the bad `ant` opcode.

```
# Cycle
 add r10, r11, r12

 sw r3, 4(r5)

 ant r1, r2, r5

 or  r5, r6, r7

HANDLER:
 sw
```

(*b*) An interrupt mechanism will switch the processor from user mode to privileged mode when the handler starts. What is the difference between user and privileged mode and why are they necessary to implement an operating system?

☐ Difference between user and privileged mode? ☐ Importance for OS?

Problem 5: [6 pts] In class we described three broad families of ISAs: CISC, RISC, and VLIW.

(*a*) In which ISA family is it easy to have instructions with long (say, 32 bit) immediates? Explain how.

☐ ISA family with long immediates is:

☐ What about the ISA family enables this?

(*b*) In which ISA family are dependencies between instructions explicitly given? How is the dependence information supposed to help?

☐ ISA family in which dependence information is part of program is:

☐ Reason for providing dependence information.

(*c*) Which ISA family tends to have the most compact programs? (That is, the size in bytes of the program is smallest.) What makes them compact?

☐ ISA family with most compact programs is:

☐ Program sizes are small because.

Problem 6: [12 pts] Answer each question below.

(*a*) Consider a 2-way superscalar implementation and a 10-stage implementation, both derived from our 5-stage MIPS used in class. All implementations include reasonable bypass paths.

Explain how the instruction sequence below would always result in a stall on the 10-stage pipeline but might not result in a stall on the 2-way system. (The compiler cannot separate the two instructions.) Illustrate your answer with a pipeline execution diagram.

```
add r1, r2, r3
sub r4, r1, r5
```

☐ Reason for maybe stall on superscalar and certain stall on 10-stage.

☐ Illustrate using a pipeline diagram.

(*b*) Consider two MIPS implementation, one is an n-way superscalar of MIPS-I, with n sets of FP units, the other is a scalar implementation of a hypothetical MIPS-I-vec, which includes an n-lane vector unit for the vector instructions in MIPS-I-vec. Both the superscalar and vector MIPS implementations can sustain execution at n FP operations per cycle. *Note: The phrase "In terms of n" did not appear in the original exam.*

☐ In terms of n how does the cost of bypass paths compare in the two systems?

☐ In terms of n how does the cost of registers compare in the two systems?

Problem 7: [12 pts] Answer each question below.

(*a*) The SPECcpu benchmarks are designed to evaluate the CPU and memory system in new implementations and ISAs. To realize this testers are responsible for providing a compiler and for compiling the benchmarks.

Suppose SPEC required testers to use a compiler that they provide. How much would this impact the goal of testing new ISAs? New implementations of existing ISAs?

☐ Degree of impact of SPEC-provided compilers for testing new ISAs. ☐ Explain.

☐ Degree of impact of SPEC-provided compilers for testing new implementations of old ISAs. ☐ Explain.

(*b*) A company releases a SPEC disclosure in which they have substituted the bzip2 benchmark with another version of bzip2 which does the same thing but runs faster. Since it does the same thing it provides the correct output to the SPEC verification script. As a result they get very good scores. How will they get caught? How is it against the goal of SPECcpu?

☐ How will the benchmark substitution get caught?

☐ How does the substitution undermine what SPECcpu is supposed to measure?

Name _____

Computer Architecture

EE 4720

Final Examination

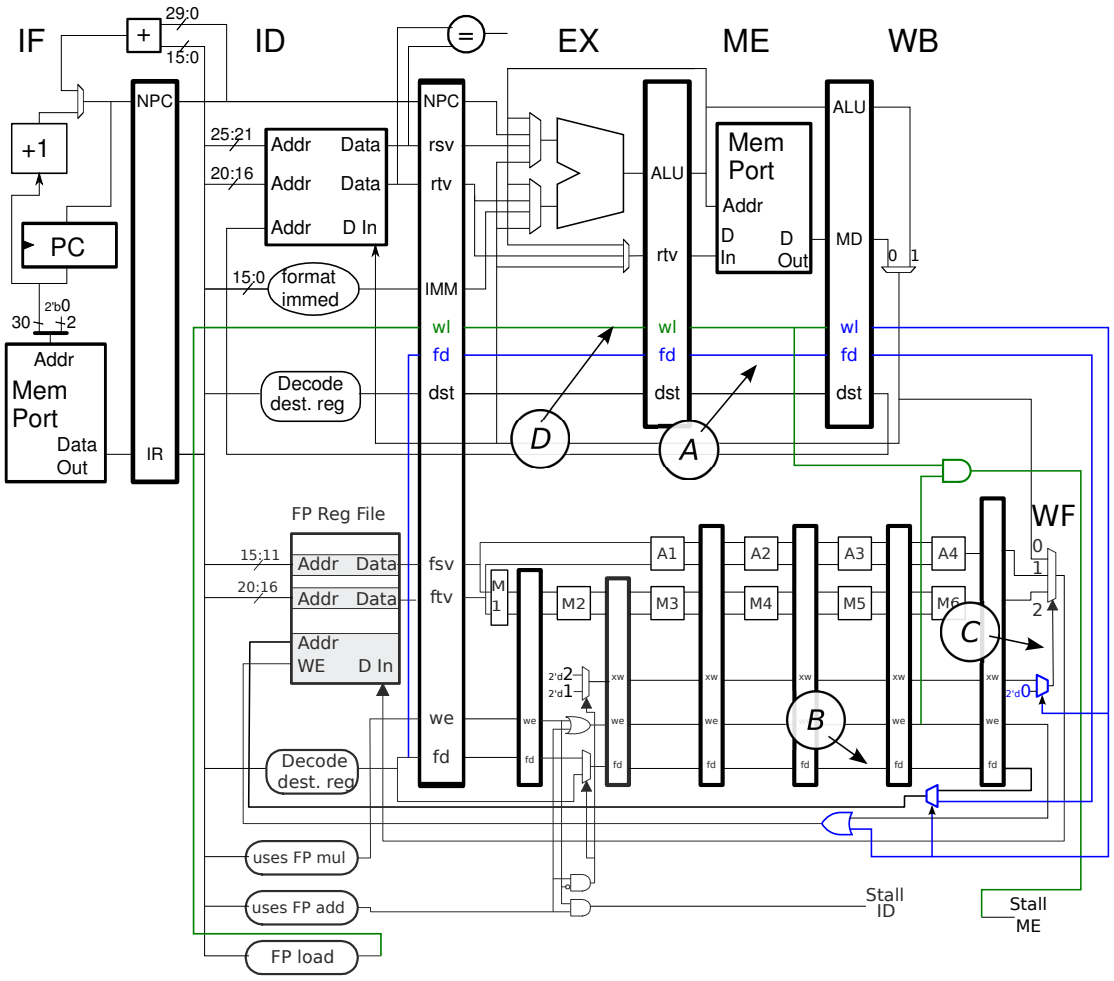
10 May 2014, 10:00–12:00 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(5 pts)
Problem 6	_____	(10 pts)
Problem 7	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (15 pts) Illustrated below is the **stall-in-ME** version of our MIPS implementation, taken from the solution to the midterm exam.



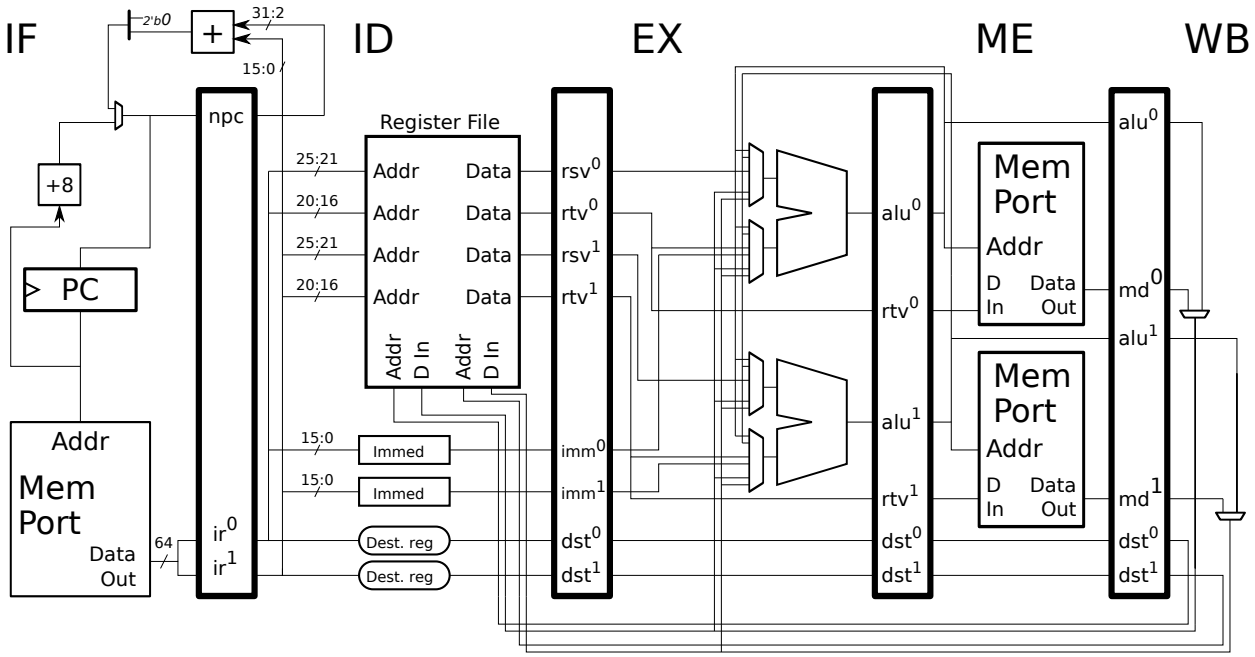
(*a*) Show a pipeline execution diagram of the code below on this pipeline.

(*b*) Wires in the diagram are labeled A, B, C, and D. Under your pipeline execution diagram show the values on those wires when they are in use.

☐ Show pipeline execution diagram. ☐ Show values of A, B, C, and D.

```
add.s f4,f5,f6

sub.s f7,f8,f9

lwc1 f1, 0(r2)

A:

B:

C:

D:
```

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block isALU is 1 if the instruction's result is computed by the integer ALU. The output of logic block rtSrc is 1 if the instruction uses the `rt` register as a source. The output of logic block isStore is 1 if the instruction is a store.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☐ Control logic to detect true dependence in `ID` and assign `STALL`.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular instruction pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Note: There was a similar-sounding problem in last year's final, but the solutions are different.*

```
0x1000: add r1, r2, r3
0x1004: sw r1, 0(r5)
```

☐ Add the bypass path(s) needed so that the code executes without a stall.

☐ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

Problem 3: (15 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{30} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(*a*) Branch behavior is shown below. Notice that `B2`'s outcomes come in groups of three, such as 3 `q`s. The first outcome of each group is random and is modeled by a Bernoulli random variable with $p = .5$ (taken probability is .5). The second and third outcomes are the same as the first. For example, if the first `q` is `T` the second and third `q` will also be `T`. If the first `s` is `N`, the second and third `s` will also be `N`. Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T   T   T   N   N     T   T   T   N   N    ...
B2:   r   r   r   q   q     q   s   s   s   u    ...
B3:    T   T   T   T   T     T   T   T   T   T   ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the approximate accuracy of the bimodal predictor on branch `B2`? ☐ Explain.

☐ What is the minimum local history size needed to predict `B1` with 100% accuracy?

☐ What is the accuracy of the local predictor on branch `B2`, after warmup. ☐ Explain.

☐ What is the best local history size for branch `B2`, *taking warmup into consideration.* ☐ Explain.

☐ How many different GHR values will there be when predicting `B3`?

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries. Also, consider the same branch patterns, they are repeated below, along with the address of branches `B1` and `B2`. The branch predictors are part of a MIPS implementation.

```
0x1234: B1:  T   T   T   N   N     T   T   T   N   N    ...
0x1242: B2:   r   r   r   q   q     q   s   s   s   u    ...
        B3:    T   T   T   T   T     T   T   T   T   T   ...
```

(*b*) Choose an address for branch `B3` that will result in a BHT collision with branch `B1`.

☐ Address for `B3` that results in a collision.

(*c*) How does the collision change the prediction accuracy of the bimodal predictor on the two branches?

☐ Change in `B1` and ☐ Change in `B3`.

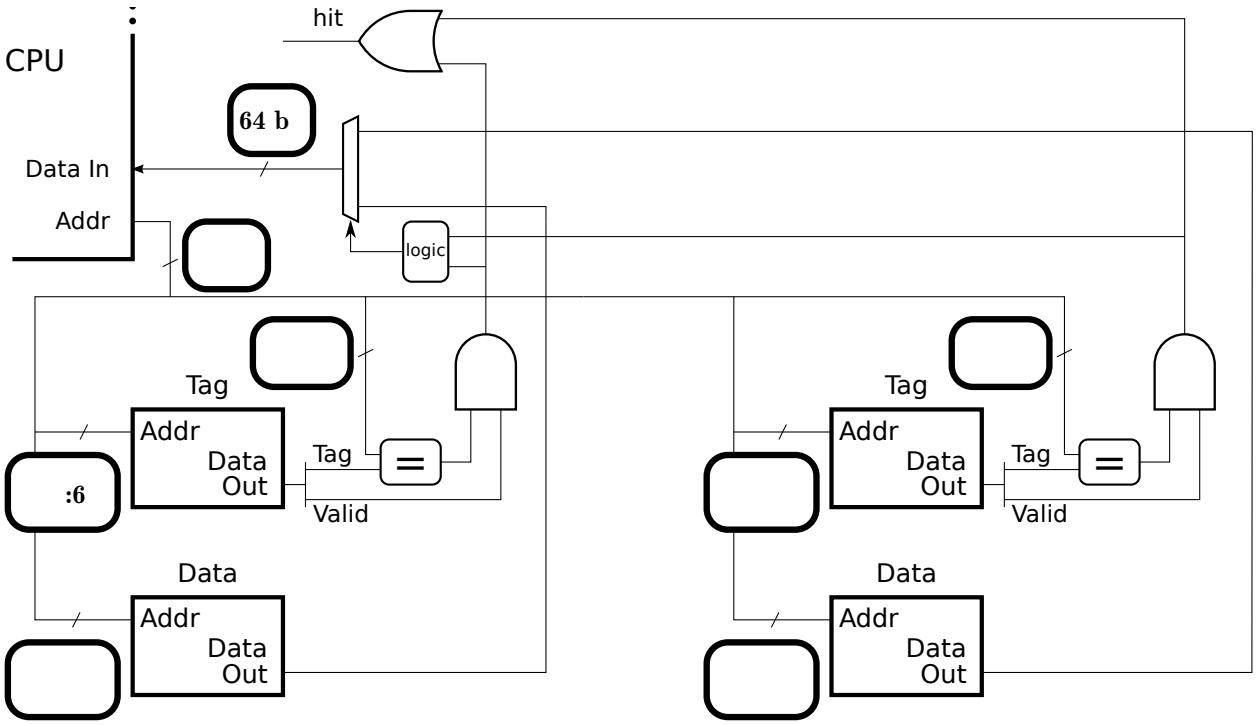
(*d*) (The answer to the following question does not depend on the sample branch patterns above.) Suppose we detect a BHT collision (perhaps by using tags). Why should we predict not taken?

☐ Reason for predicting not-taken for a collision.

Problem 4: (15 pts) The diagram below is for a 256 kiB (2^{18} B) set-associative cache. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

Address: ____ ____ ____

31 0

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Address: ____ ____ ____

Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 128 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
double sum = 0;
double *a = 0x20000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

(*c*) Find the largest value for `BSIZE` for which the second `for` loop will enjoy a 100% hit ratio.

```
struct Some_Struct {
  double val;        // sizeof(double) = 8
  double norm_val;
  double a[14];
 };

  const int BSIZE =                          ; // <-  FILL IN
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val;
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;
```

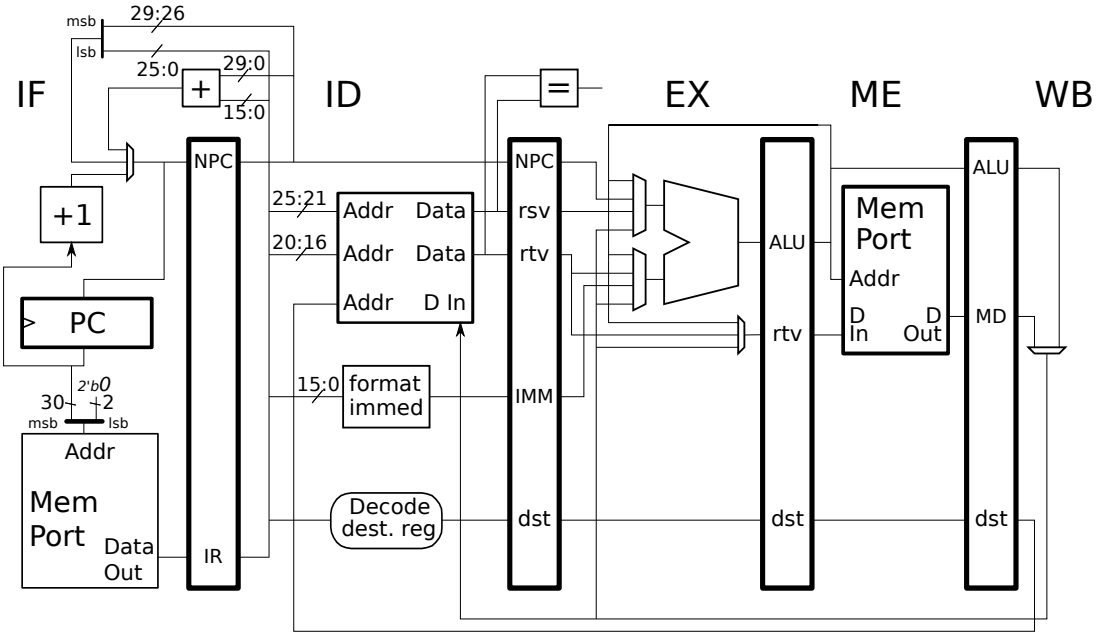
Problem 5: (5 pts) The displacement in MIPS branches is 16 bits. Consider a new MIPS branch instruction, `bfeq rsn, rtn` (branch far), where `rsn` and `rtn` are 2-bit fields that refer to registers 4-7. As with `beq`, branch `bfeq` is taken if the contents of registers `rsn` and `rtn` are equal. With six extra bits `bfeq` can branch 64 times as far.

(*a*) Show an encoding for this instruction which requires as few changes to existing hardware as possible.

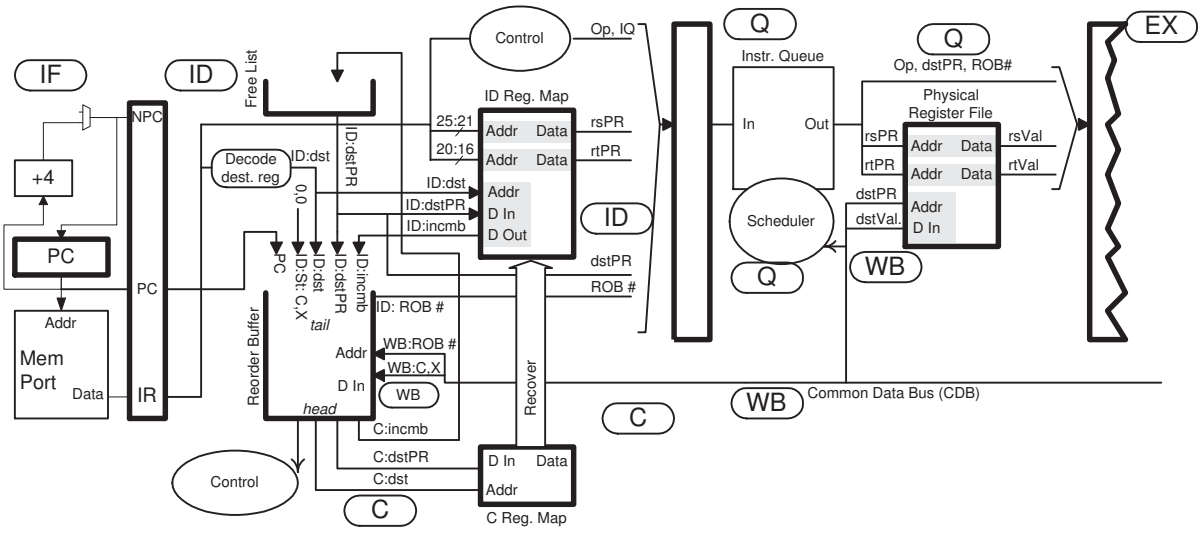
☐ Encoding for `bfeq`. ☐ Explain how minimizes changes.

(*b*) Modify the pipeline below to implement the new instruction. Use as little hardware as possible.

☐ Briefly show changes.



Problem 6: (10 pts) Illustrated below is a dynamically scheduled four-way superscalar MIPS implementation and the execution of code on that implementation.




```
LOOP: #   Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f2, 0(r1)     IF ID Q  RR EA ME WB
 add.s f4, f4, f2   IF ID Q        RR A1 A2 A3 A4 WB
 bne  r1, r2, LOOP  IF ID Q  RR B  WB
 addi r1, r1, 4     IF ID Q  RR EX WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f2, 0(r1)        IF ID Q  RR EA ME WB
 add.s f4, f4, f2      IF ID Q                 RR A1 A2 A3 A4 WB
 bne  r1, r2, LOOP     IF ID Q  RR B  WB
 addi r1, r1, 4        IF ID Q  RR EX WB
#         Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

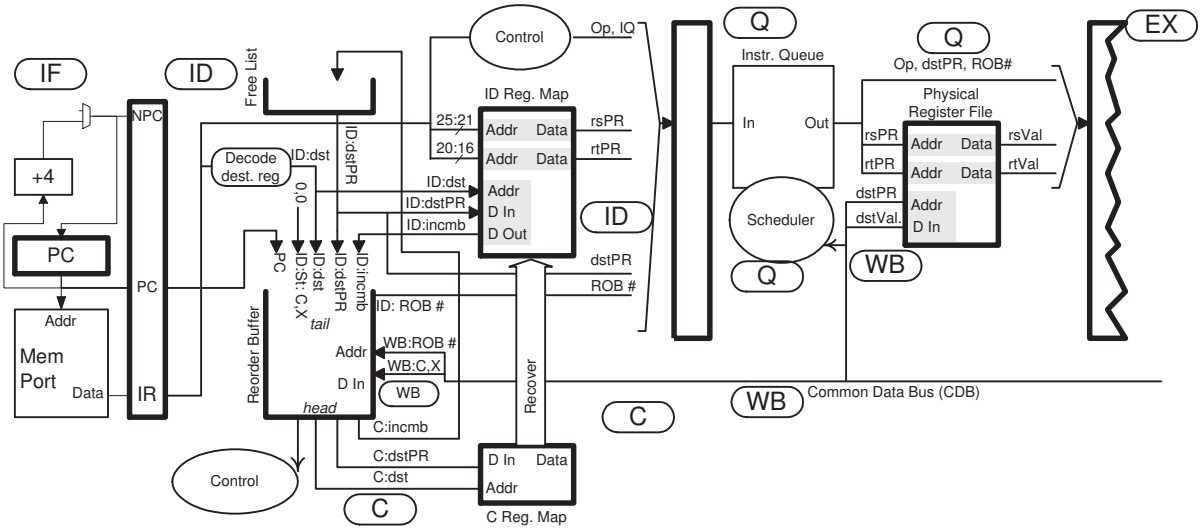
(*a*) On the diagram above indicate when each instruction will commit.

☐ Show commits on diagram above.

(*b*) What is the execution rate, IPC, for the code above for a large number of iterations assuming perfect branch prediction. Note that the system is dynamically scheduled.

☐ IPC for code above for large number of iterations.

(*c*) On the next page there is a table showing the values of selected signals during the execution of the code, the signals are related to register renaming. Show values where indicated on the table. Note that `ID:incmb` is already shown in cycle 1, show its values for later cycle(s).



☐ Show values where indicated.

```
LOOP: #   Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)      IF  ID  Q   RR  EA  ME  WB
 add.s f4, f4, f2    IF  ID  Q           RR  A1  A2  A3  A4  WB
 bne  r1, r2, LOOP   IF  ID  Q   RR  B   WB
 addi r1, r1, 4      IF  ID  Q   RR  EX  WB
LOOP: #   Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)          IF  ID  Q   RR  EA  ME  WB
 add.s f4, f4, f2        IF  ID  Q                       RR  A1  A2  A3  A4  WB
 bne  r1, r2, LOOP       IF  ID  Q   RR  B   WB
 addi r1, r1, 4          IF  ID  Q   RR  EX  WB
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID:dstPR 0 (lwc1)       65  62
ID:dstPR 1 (add.s)      97  69
ID:dstPR 3 (addi)       60  79
# Show values for signals below, including incmb.
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14

ID:incmb 0 (lwc1)       83
ID:incmb 1 (add.s)      20
ID:incmb 3 (addi)       67

ID:rsPR 0 (lwc1)

ID:rsPR 1 (add.s)

ID:rsPR 3 (addi)

ID:rtPR 1 (add.s)                                                   <- rt, not rs
#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
WB:dstPR 0 (lwc1)

WB:dstPR 1 (add.s)

WB:dstPR 3 (addi)

#         Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

Problem 7: (25 pts) Answer each question below.

(*a*) Describe how cost and performance limit the practical largest value of width (value of n) in an n-way superscalar implementation.

☐ Cost limiter.

☐ Performance limiter.

(*b*) What is the most important factor in determining the size of a level 1 cache?

☐ Most important factor in L1 cache size.

(*c*) Suppose the 16-bit offset in MIPS `lw` instructions was not large enough. Consider two alternatives. In alternative 1 the offset in the existing `lw` instruction is the immediate value times 4. So, for example, to encode instruction `lw r1, 32(r2)` the immediate would be 8. In alternative 2 the behavior of the existing `lw` is not changed but there is a new load `lws r1, 32(r2)`, in which the immediate is multiplied by 4. Note that alternative 2 requires a new opcode. Which instruction should be added to a future version of MIPS?

☐ Should choose alternative 1 or alternative 2? ☐ Explain.

(*d*) The SPECcpu suite comes with the source code for the benchmark programs. How does that help with the goal of measuring new ISAs and implementations?

☐ Source code helps with testing new implementations because:

(*e*) What's the difference between a 4-way superscalar implementation (of say MIPS) and a VLIW system with a 4-slot bundle?

☐ Difference between superscalar and VLIW.

13 Spring 2013

Name _____

Computer Architecture
EE 4720
Midterm Examination

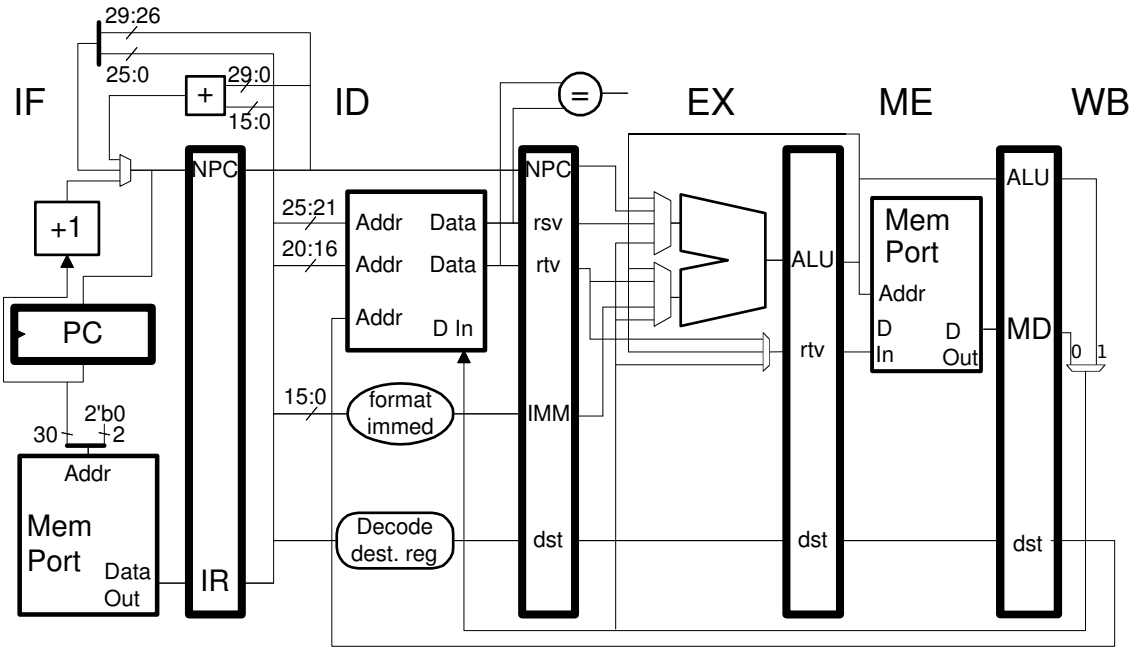
Wednesday, 13 March 2013, 9:30–10:20 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(9 pts)
Problem 5	_____	(11 pts)
Problem 6	_____	(20 pts)
Problem 7	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The code below executes on the illustrated MIPS implementation, which is the one usually used in class.



(*a*) Show the execution of the code below until the start of the third iteration.

- **Check** and **double-check** the code for **dependencies.**

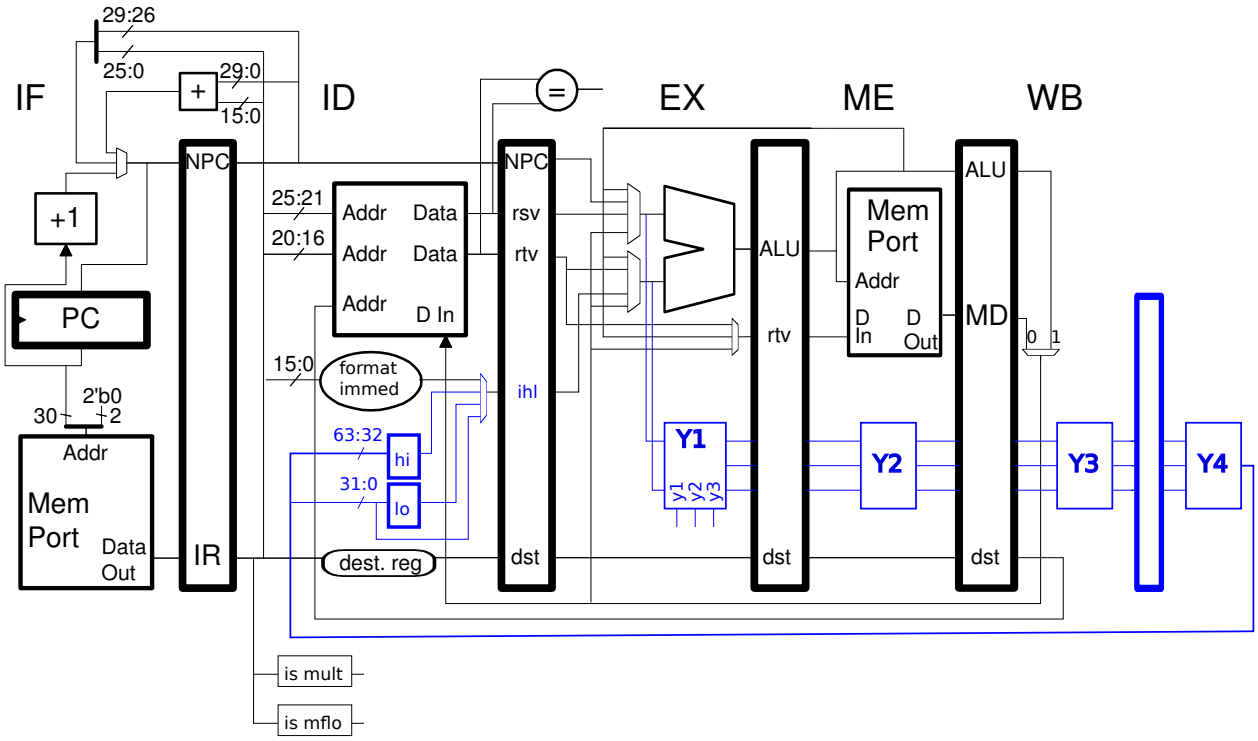
☐ Show execution to start of third iteration.

```
LOOP:
 add r1, r2, r3

 sw r4, 5(r1)

 bne r1, r6, LOOP

 lw r2, 7(r8)
```

(*b*) Compute the CPI assuming a large number of iterations. *Hint: It's important to consider up to the third iteration.*

☐ CPI is:

Problem 2: [15 pts] Illustrated below is our familiar five-stage MIPS implementation with a multiply unit added. Unlike the multiplier in Homework 4, this unit has four stages.



(*a*) Show the execution of the code below on the illustrated implementation.

☐ Show execution of code.

```
mult r1, r2

mflo r3

mfhi r4
```

(*b*) Explain why execution of the following code would be different.

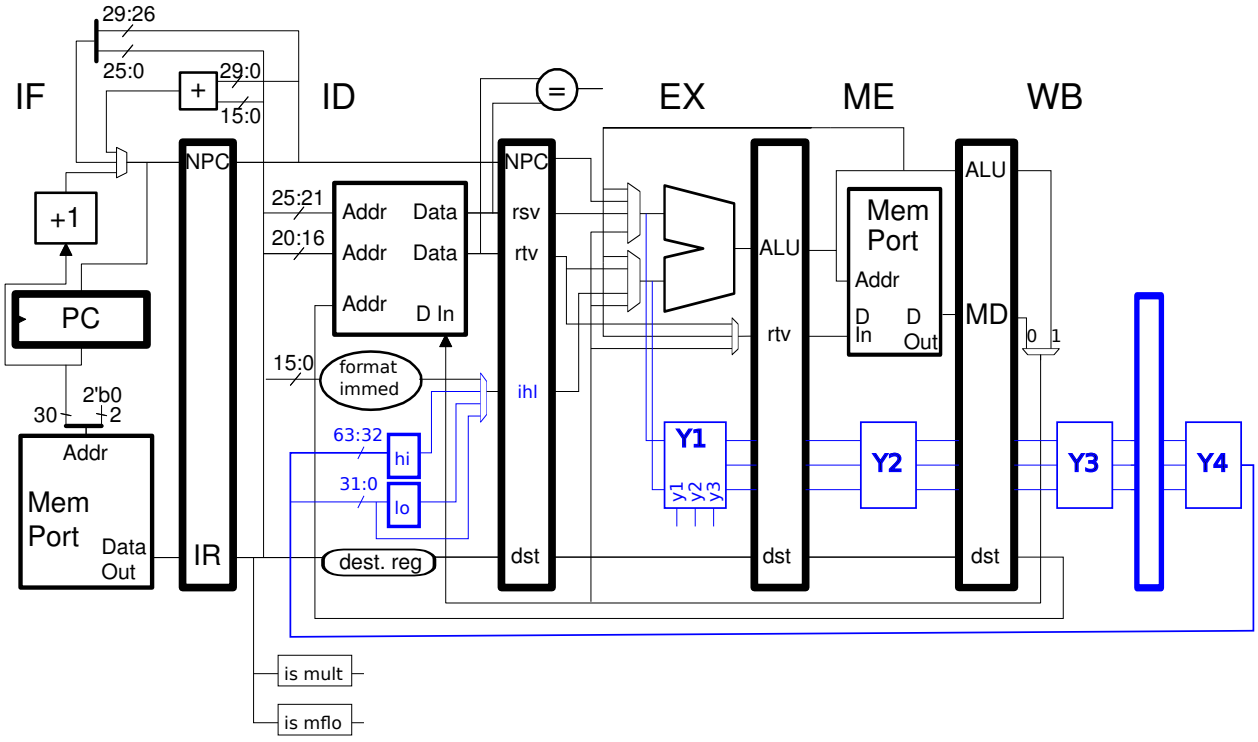
```
mult r1, r2
mfhi r4
mflo r3
```

☐ Execution of the code above would be different because:

(*c*) Design the control logic needed to generate a stall signal for dependencies between a `mult` and a `mflo`, something the execution above should have experienced. For this problem do not assume any bypass paths for the product other than those illustrated. Note that logic to identify the `mult` and `mflo` instructions has already been added to the `ID` stage.

☐ Control logic to generate stall signal.

Problem 3: [15 pts] In this problem the multiply unit has been designed to produce a product sooner for special cases, such as when one of its inputs is zero. To indicate when the product will be available `Y1` has three one-bit outputs along the bottom, `y1`, `y2`, and `y3`. Signal `y1` is `1` if the center output of `Y1` will have a completed product at the end of the cycle, `y2` is `1` if the center output `Y2` will have a completed product at the end of the cycle in which the `mult` is in `Y2`, output `y3` works similarly. If `y1` is `1`, then `y2` and `y3` are also `1`, if `y2` is `1` then `y3` is also `1`. In other words, if an early product is available at the output of `Y1`, it will be at the outputs of `Y2` and `Y3` in subsequent cycles.




```
# Cycle             0  1  2  3  4  5  6  7
    mult r10, r11   IF ID Y1 Y2 Y3 Y4
a:  mflo r1            IF ID EX ME WB
b:  mflo r2               IF ID EX ME WB
c:  mflo r3                  IF ID EX ME WB
# Cycle             0  1  2  3  4  5  6  7
```

(*a*) Add bypass paths so that the code executes above without a stall assuming an early product is available.

☐ Show bypass paths for each of the `mflo` instructions above.

☐ Label the bypass paths with `a`, `b`, and `c`, based on the instruction (above) that uses them.

Problem 4: [9 pts] Show the encoding of each of the instructions below. Be sure to handle the branch target correctly. Fill in as many fields as you can. Use the value "lookup" for any field who's value can't be determined by inspection.

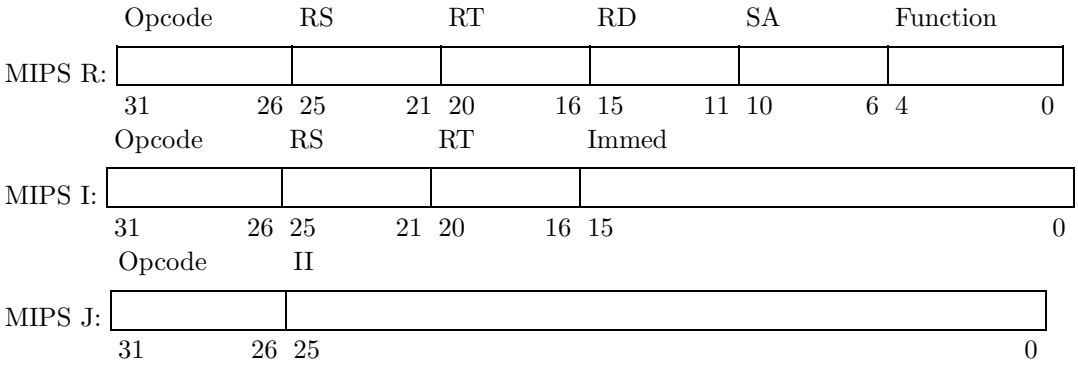
```
 0x1000 beq r1, r2  TARG
 0x1004 lui r1, 0x1234
TARG:
 0x1404 sw r3, 4(r5)
```

☐ Encoding for `beq` as used above:

☐ Encoding for `lui` as used above:

☐ Encoding for `sw` as used above:

The formats appearing below are for reference.



Problem 5: [11 pts] Answer each question below.

(*a*) MIPS and many other RISC ISAs have a special zero register, `r0` in MIPS. VAX, a CISC ISA, lacks a zero register, but VAX has an `mneg` (negate) instruction (`dest = -source`), which MIPS and other RISC ISAs lack.

☐ Why does MIPS' zero register make it unnecessary to have a negate instruction?

☐ What can VAX use in place of a zero register? (The answer is not `mneg` or any other instruction.)

☐ If VAX can use its zero register replacement in the same way MIPS uses its zero register, what is the advantage for VAX in having a negate instruction? A quantitative answer will get full credit.

(*b*) Why might it be possible to have a higher clock frequency in an implementation of an ISA that uses condition code registers (such as SPARC) compared to one which allows branch instructions to compare two registers? Consider the types of implementations covered in class so far.

☐ Higher clock frequency possible with condition code registers because:

Problem 6: [20 pts] Answer each question below.

(*a*) What is the difference between a dependency and a hazard?

☐ Difference between dependency and hazard:

(*b*) How do we know that the benchmarks in SPECcpu were not unfairly chosen to favor an influential company's processors.

☐ Can trust choice of SPECcpu benchmarks because:

(*c*) When using profiling for optimization a program needs to be compiled twice. Explain what happens in each compilation related to profiling.

☐ During the first compilation:

☐ During the second compilation:

(*d*) Who would benefit the most if each new processor implemented a new, one-of-a-kind, ISA: computer engineers, compiler back-end writers, software developers, computer buyers? Explain.

☐ Group that would benefit most (circle): computer engineers, compiler back-end writers, software developers, computer buyers.

☐ Reason that the group would benefit the most:

Problem 7: [10 pts] Compiler and microarchitecture experts at a computer company are preparing the optimization flags for a SPECcpu run. They have come up with a different set of optimization settings for each benchmark in the suite.

(*a*) Based on the SPEC rules it is okay to use these individualized sets for the peak tuning runs but not the base tuning runs. Why?

☐ Reason individualized settings are not allowed for base:

(*b*) The compiler writer figures out a way to have the compiler itself, given only the `-O3` flag, choose the same individualized sets as the expert did. The compiler expert makes these changes to the compiler, which the company will start selling. It is easy to do this in a way which goes against the spirit of the SPECcpu rules. It would require great skill to do this in a way consistent with the spirit of the SPECcpu rules.

☐ Easy way to have `-O3` pick correct set of optimizations, but against spirit (goals) of rules. Explain.

☐ Hard way to have `-O3` pick correct set of optimizations, but in spirit (goals) of rules. Explain.

Name _____

Computer Architecture
LSU EE 4720
Final Examination

10 May 2013, 12:30–14:30 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

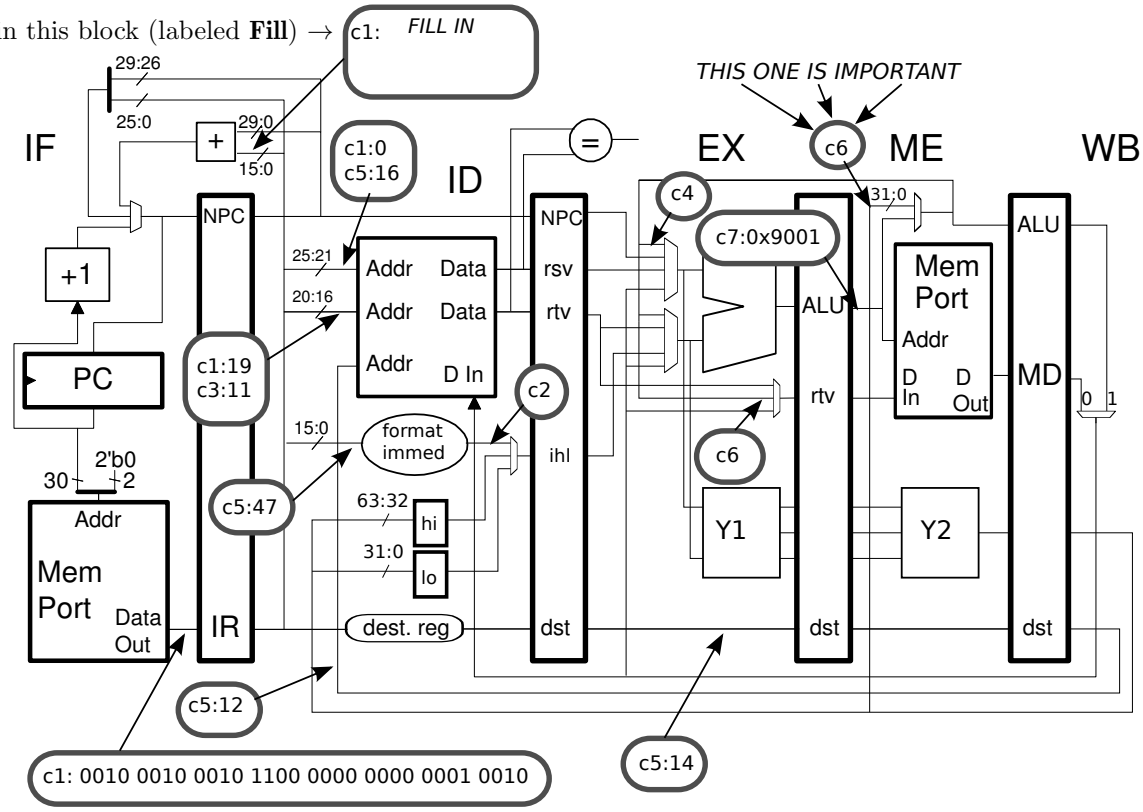
Good Luck!

Problem 1: (20 pts) The MIPS implementation below includes a two-stage integer multiply unit (`Y1` and `Y2`) for the `mult` instructions, similar to the MIPS implementation covered in Homework 4. Some wires are labeled with cycle numbers and values that will then be present. For example, c5:14 indicates that at cycle 5 the pointed-at wire will hold a 14. Other wires just have cycle numbers, indicating that they are used in that cycle. *Note that instruction addresses are provided.*

☐ Write a program consistent with these labels.

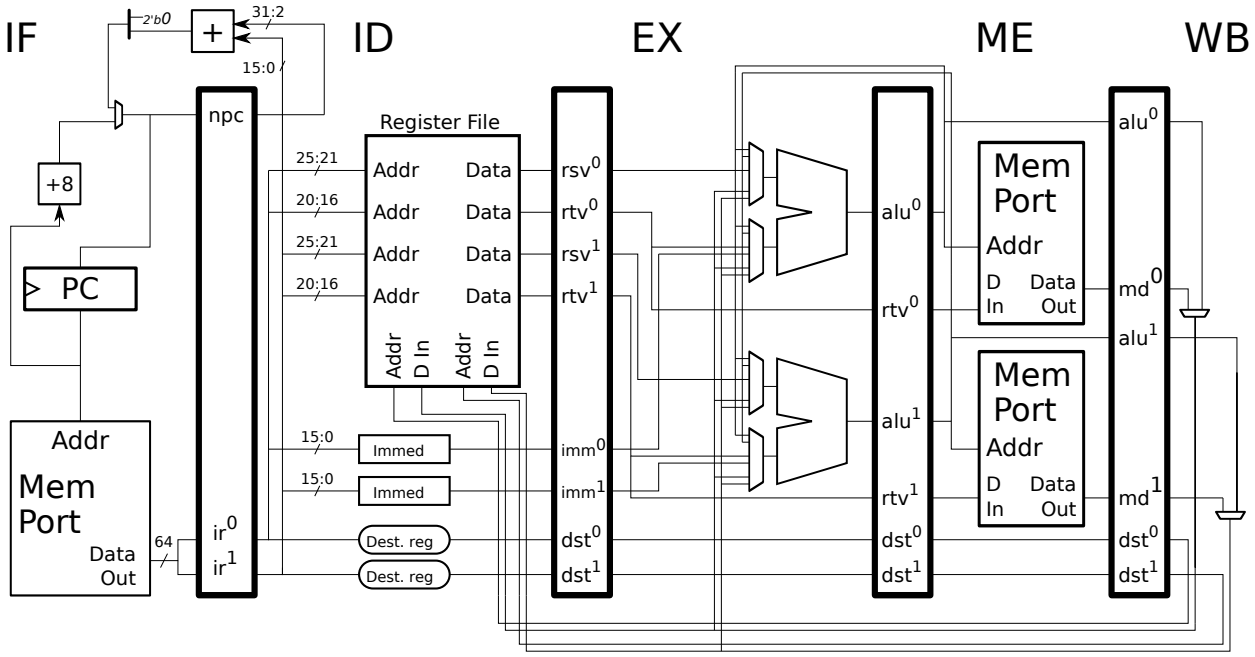
☐ All register numbers and immediate values can be determined.

☐ Fill in this block (labeled **Fill**) →



#	Cycle:	0	1	2	3	4	5	6	7	8
0x1000		IF	ID	EX	ME	WB				
0x1004			IF	ID	EX	ME	WB			
0x1104				IF	ID	EX	ME	WB		
0x1108					IF	ID	EX	ME	WB	
0x110c						IF	ID	EX	ME	WB
#	Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block isALU is 1 if the instruction's result is computed by the integer ALU. The output of logic block rtSrc is 1 if the instruction uses the `rt` register as a source. The output of logic block isLoad is 1 if the instruction is a load.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☐ Control logic to detect true dependence in `ID` and assign `STALL`.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular arithmetic/load pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Hint: Something about the* `lw` *makes it a special case.*

```
0x1000: add r1, r2, r3
0x1004: lw r4, 0(r1)
```

☐ Add the bypass path(s) needed so that the code executes without a stall.

☐ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a *12-outcome local history*, and one system has a global predictor with a *12-outcome global history*.

(*a*) Branch behavior is shown below. The `r` outcomes for branch `B2` are random and are modeled by a Bernoulli random variable with $p = .25$ (taken probability is .25). Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T   N   T   T   T   N    T   N   T   T   T   N    ...
B2:   T   N   T   r   T   N    T   N   T   r   T   N    ...
B3:    T   T   T   T   T   T    T   T   T   T   T   T   ...
```

☐ What is the accuracy of the bimodal predictor on branch `B1`?

☐ What is the approximate accuracy of the bimodal predictor on branch `B2`? ☐ Explain.

☐ What is the minimum local history size needed to predict `B1` with 100% accuracy? Ignore branch `B2` for this question.

☐ What is the accuracy of the local predictor on branch `B2`, ignoring the effect of `B1`? ☐ Explain.

☐ Describe a situation in which branch `B2` is mispredicted using the local predictor due to the effect of `B1`. The low bits of the address of `B1` and `B2` are different so there is no chance of a BHT collision. How frequently do such mispredictions occur? *Grading Note: The original exam and the Spring 2014 homework question did not include the statement about the low address bits.*

☐ How many possible GHR values will be present when predicting branch `B3`? It's okay to show patterns that include `r`'s, but indicate how many of each pattern there are.

Problem 3, continued: Recall that the local predictor uses a 12-outcome local history and a 2^{14}-entry BHT.

(*b*) Draw a diagram of the local predictor hardware used for making a prediction but omit the hardware for updating the predictor. The input to your hardware should be `PC`, and output should be a 1-bit quantity (`0` for not-taken, `1` for taken).

- Be sure to show the BHT and PHT.
- Assume that `PC` is the address of a branch. (That is, don't worry about detecting non-branch instructions.)
- Don't predict the target, just the direction (taken or not-taken).

☐ Local predictor hardware for predicting branch direction.

☐ Label wires with bit ranges (*e.g.* 31:26) or number of bits, as appropriate.

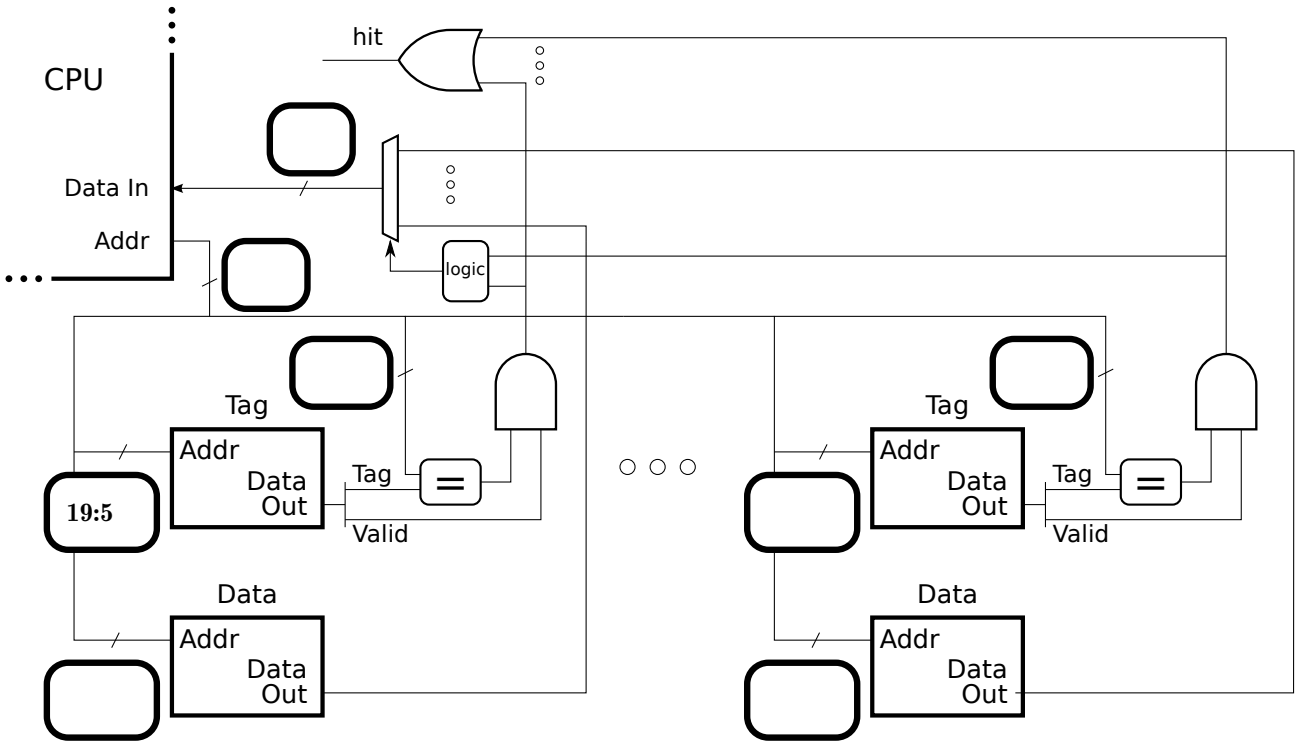
(*c*) How much memory is needed to implement the local predictor. Only include storage for predicting the branch direction, do not include storage for predicting the branch target.

☐ Storage needed to implement local predictor. ☐ Indicate unit.

Problem 4: (20 pts) The diagram below is for an 8-way set-associative cache. Hints about the cache are provided in the diagram and also in the address bit categorization just below the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)

Address: 

☐ Cache Capacity (Indicate Unit!!):

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Address: 

Problem 4, continued: The problems on this page are **not** necessarily based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 256 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
Complex sum = {0,0};
Complex *a = 0x2000000; // sizeof(Complex) == 16  Each element loaded with 1 load insn.
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

(*c*) Find the smallest value for `ASIZE` that will minimize the hit ratio (make things as bad as they can get) of the code below.

```
struct Some_Struct {
  double val;        // sizeof(double) = 8
  double norm_val;
  double a[ASIZE];  };

  const int BSIZE = 1 << 10;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val;
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;
```

☐ Smallest value of `ASIZE` to minimize hit ratio:

Problem 5: (25 pts) Answer each question below.

(*a*) Indicate whether each feature below is a feature of an ISA or the feature of an implementation.

☐ Number of stages in pipeline. *Circle One*: ISA or Implementation

☐ Number of integer registers. *Circle One*: ISA or Implementation

☐ Number of bits in an integer register. *Circle One*: ISA or Implementation

☐ Whether an adjacent pair of instructions, such as `add r1,r2,r3; sub r4,r1,r2` will generate a stall. *Circle One*: ISA or Implementation

(*b*) Describe the problem with each of the following MIPS code fragments.

☐ Problem with fragment below:

```
addi r1, r2, 0x123456
```

☐ Problem with code execution on our FP pipeline.

```
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
sub.s f4, f1, f5      IF ID A1 A2 A3 A4 WF
```

☐ Problem with code execution on our 5-stage pipeline.

```
lw r1, 2(r3)      IF ID EX ME WB
beq r1, r4 TARG      IF ID EX ME WB
xor r6, r7, r8          IF ID EX ME WB
TARG:
add r9, r10, r11           IF ID EX ME WB
```

(*c*) Consider the following two equal-cost design options:

A dual-core chip, each core is 4-way superscalar and dynamically scheduled.

A 16-core chip, each core is 2-way superscalar and statically scheduled.

Both have the same clock frequency.

☐ What is the peak execution rate of each chip, in IPC?

☐ Describe a situation in which the dual-core chip is a better choice than the 16-core chip.

☐ Describe a situation in which the 16-core chip is a better choice than the dual-core chip.

(*d*) Consider a $5n$-stage implementation created by splitting each stage of a 5-stage implementation into n pieces.

☐ How important is it to have a higher clock frequency in the $5n$-stage design (compared to the 5-stage design)? ☐ Explain.

(*e*) When an instruction raises an exception execution will switch to a trap handler.

☐ How is the trap handler address determined for MIPS?

☐ How is the trap handler address determined for SPARC?

14 Spring 2012

Name _____

Computer Architecture
EE 4720
Midterm Examination

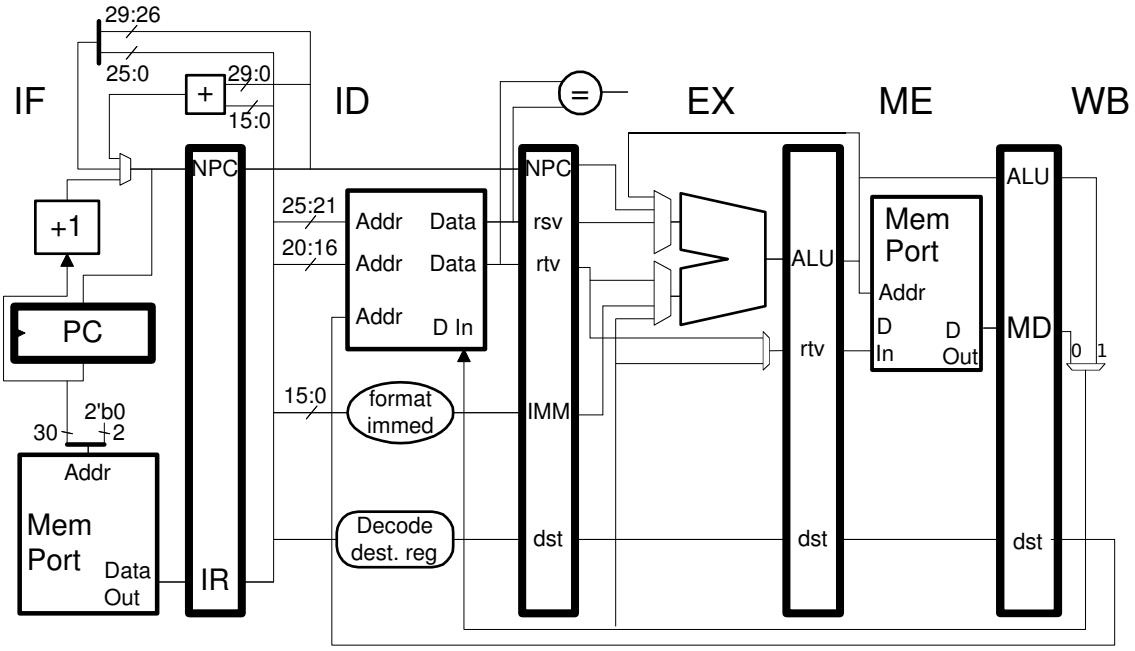
Friday, 23 March 2012, 9:40–10:30 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The MIPS code below executes on the illustrated **seemingly** familiar implementation. If you look closely you'll notice that certain bypass paths that are present in the five-stage implementation usually used in class are missing from the diagram below.



(*a*) Show the execution of the code below on the illustrated implementation for enough iterations to determine CPI. Determine the CPI.

☐ Execution diagram.

☐ Double-check for bypass paths, stall when necessary.

☐ Compute CPI.

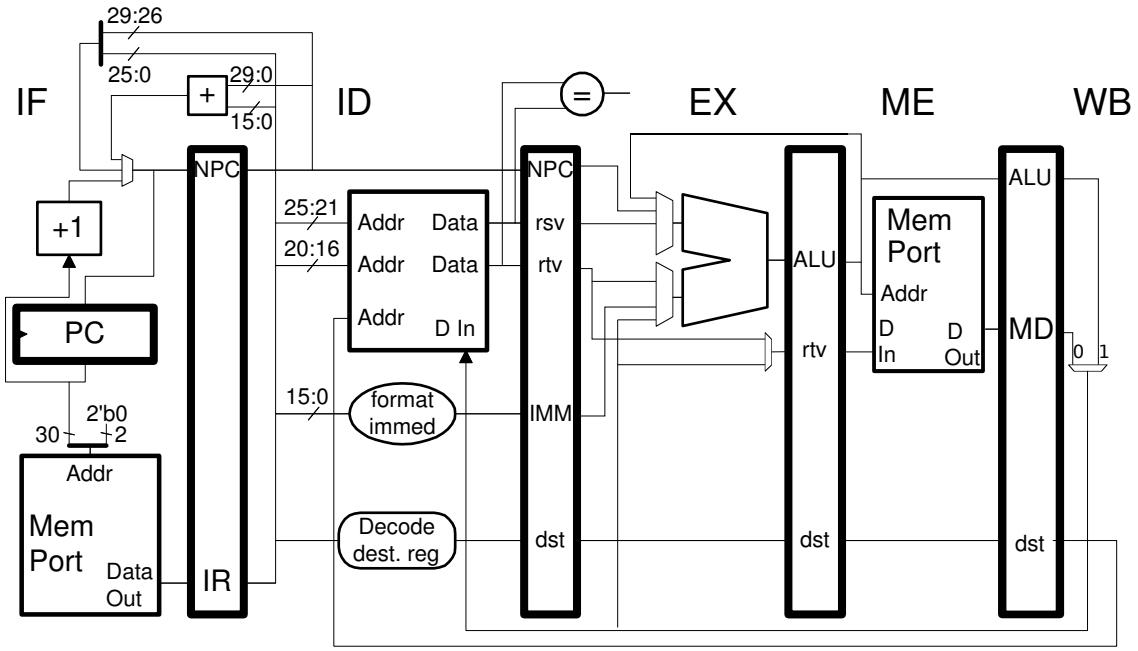
```
LOOP:
 lw  r2, 0(r4)

 add r1, r2, r3

 sw r1, 4(r4)

 bne r4, r5  LOOP

 addi r4, r4, 8
```

Problem 1, continued:

(*b*) The code above should have encountered at least one of the missing bypass paths. Choose one of those missing bypass paths and design control logic for it. The `ID`-stage control logic should generate a 1-bit signal `STALL` that will be logic `1` when the instruction in `ID` will have to stall because of the missing bypass path.

☐ Show which missing bypass path the control logic is for.

☐ Control logic for missing bypass.



Problem 2: [15 pts] Consider a MIPS implementation in which the memory port is split into two stages, `Ma` and `Mb`. With this change the clock frequency can increase from 1 GHz to 1.2 GHz; with this added stage our implementation is now six stages. A sample execution appears below.

```
add r2, r3, r4 IF ID EX Ma Mb WB
lw r1, 4(r2)      IF ID EX Ma Mb WB
```

(*a*) Ignoring the cost of `Ma` and `Mb`, how is the cost of the pipeline changed in this design? Consider bypass paths.

☐ Cost change, ignore `Ma` and `Mb`, consider bypass, etc.

(*b*) Provide an example of a code fragment which will execute more slowly with the six-stage pipeline. Assume that all reasonable bypass paths are provided.

☐ Code fragment that will run more slowly.

(*c*) Provide an argument that this change is good.

☐ This change is good because:

Problem 3: [15 pts] Answer the following questions about a `blt` instruction.

(*a*) MIPS lacks an instruction such as `blt r1, r2 TARG` (branch if `r1` less than `r2`). For the questions below, which ask about the suitability of such an instruction, consider implementations similar to the five-stage pipeline covered in class.

☐ Explain why adding such an instruction would slow such implementations even though `beq r1,r2 TARG` is okay.

☐ Using a PED, explain why there would be no problem adding a `blt` instruction if the ISA had two delay slots.

(*b*) The code fragment below includes the hypothetical MIPS instruction, `blt`.

```
# Hypothetical MIPS Instruction
blt r1, r2, targ
xor r4, r5, r6
```

☐ Show an equivalent MIPS code fragment without using `blt`.

☐ Show an equivalent SPARC V8 code fragment.

Problem 4: [20 pts] For the ISA design tradeoff questions below consider the following data: A typical program executes 10^{10} dynamic instructions, of these 1.38×10^9 are branches (half of which are taken), 9.12×10^7 are indirect jumps (`jr` and `jalr`), 2.22×10^8 are direct jumps (`j` and `jal`).

Consider a debate by those developing the MIPS ISA: don't include format J, which means no `j` and no `jal` instructions.

(*a*) Which MIPS instructions can be used to replace `j` and `jal` in the code below, paying attention to the hints in the target names?

☐ Replacement for `j` and `jal` in code below, paying attention to target names.

```
j NOT_TOO_FAR_AWAY
xor r1, r2, r3


jal A_FAR_AWAY_PROCEDURE
xor r1, r2, r3
```

(*b*) Determine how much longer (as a percentage or absolute time) it would take to run the typical program described above without the format J instructions. Assume execution is always at 1 CPI. (A formula is okay.)

☐ Percent increase in execution time without format J:

(*c*) What parts of out implementation would be unnecessary without format J? Approximately how many bypass paths could you add with the cost saved by not having the format J instructions?

☐ What parts would be eliminated?

☐ How many bypass paths could be added with cost savings?

Problem 5: [15 pts] Answer the following ISA design questions.

(*a*) Suppose that *one of the first* modern computer designers, working in the 1940s, made the following statement: "Let's define an 'Instruction Set Architecture' [the last three words spoken slowly, for emphasis] and then later design some hardware 'im-ple-men-ta-tions' of that architecture." Would that be a good idea? Explain. *Hint: Pay attention to the slanted words.*

☐ ISA before implementation in 1940s, good or bad? Explain.

(*b*) CISC ISAs have variable-length instructions.

☐ Benefit of variable-length instructions for CISC.

☐ Benefit of fixed-length instructions for RISC.

Problem 6: [15 pts] Answer each question below.

(*a*) In our π program demo we found that turning optimization on reduced execution time by half (yay!) but reduced the instruction count even further, from 325 million to 100 million dynamic instructions. This means that optimization *increased* (made worse) CPI from 2.77 to 4.93. *Note: The data is from an earlier semester, so the numbers won't exactly match.*

☐ Optimization is supposed to eliminate stalls, why did the CPI go up?

☐ What category of instruction was completely eliminated from the loop body of the π program by optimization.

(*b*) SPECcpu currently has two sets of results, *base* and *peak*. Consider a third set, *shrink-wrap*, for people who buy mass-marketed software (say, buying a word processor at an office supply store).

☐ Who should use the peak numbers and who should use the base numbers?

☐ Explain how the run and reporting rules might be modified for the new shrink-wrap results.

Name _____

Computer Architecture
EE 4720
Final Examination

8 May 2012, 12:30–14:30 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(30 pts)
Exam Total	_____	(100 pts)

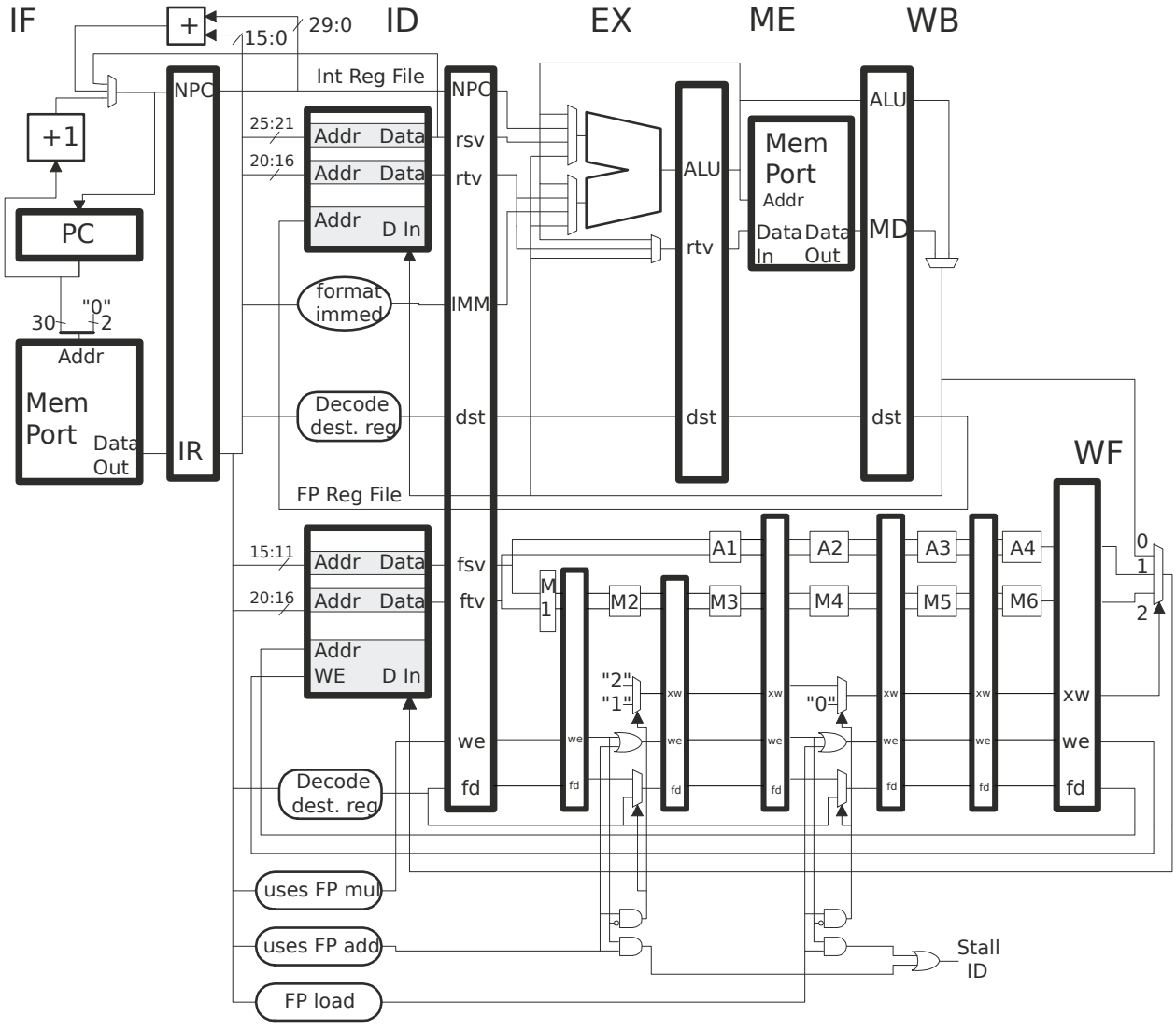
Alias _____

Good Luck!

Problem 1: (15 pts) Consider the following method for implementing the MIPS32 integer multiply instruction `mul` (the one that writes ordinary registers, not to be confused with `mult`) on the implementation below. The full set of stages `M1` to `M6` perform floating point multiply, but stages `M2` to `M4` perform integer multiplication. The integer multiply `mul` will read and write registers from the integer register file but will use `M2` to `M4` to perform the multiplication. A sample execution appears below. *Grading Note: In the original exam there was an ID-stage stall in cycle 6, implying that there was no* `WB` *to* `EX` *bypass for the* `mul`*.*

```
# Cycle          0  1  2  3  4  5  6  7  8
add r1, r2, r3   IF ID EX ME WB
mul r4, r1, r5      IF ID M2 M3 M4 WB
xor r9, r10, r11       IF ID -> EX ME WB
sub r6, r4, r7            IF -> ID EX ME WB
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

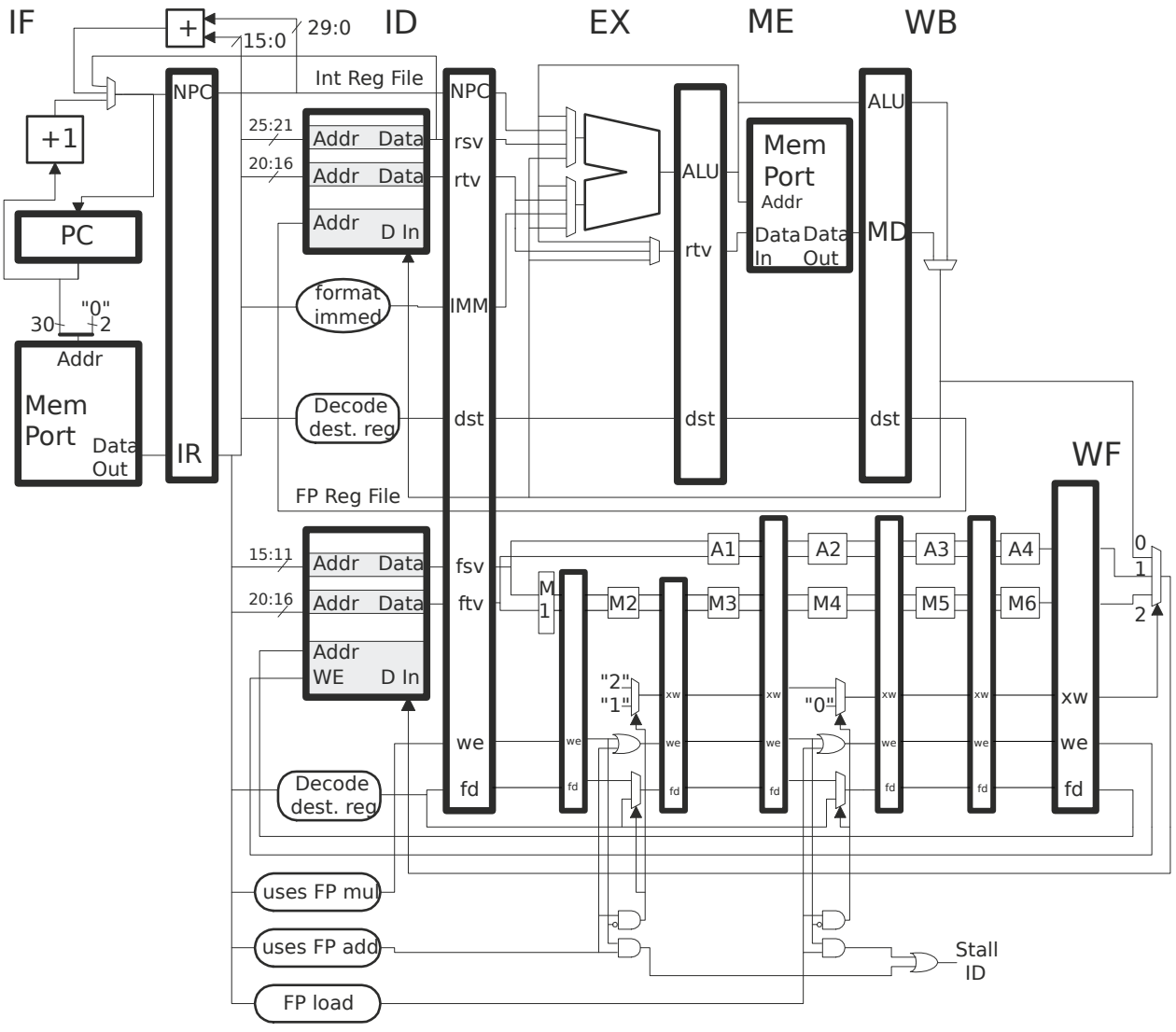
CONTINUED ON NEXT PAGE

(*a*) Modify the implementation below so that `mul` uses `M2` to `M4`. Provide any necessary bypass paths so that the code above executes as shown. For this part do not consider control logic. *Hint: Pay attention to source and destination registers.*

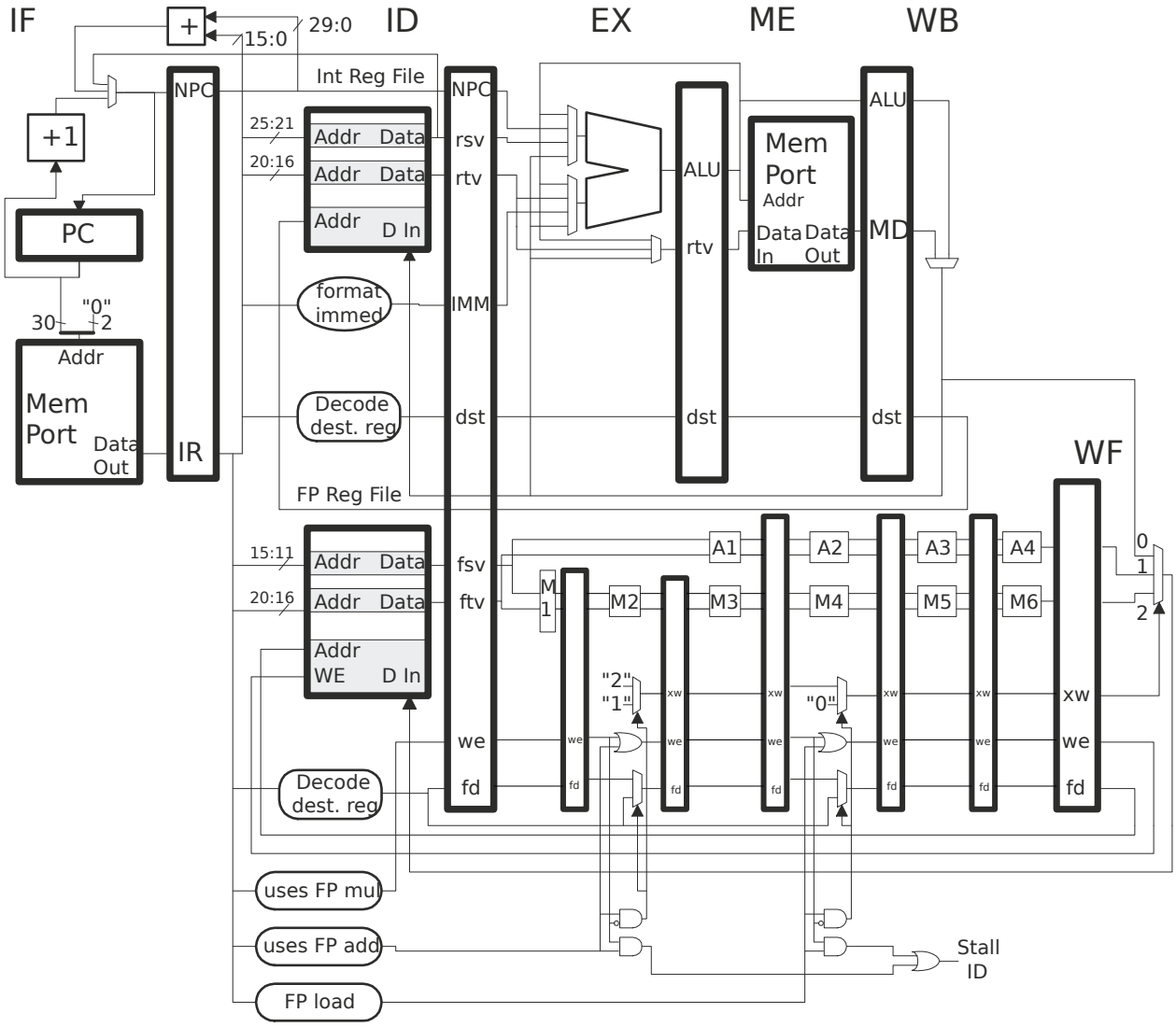
☐ Non-control logic modifications for integer `mul`.

(*b*) Add control logic related to this implementation of `mul` to detect the structural hazard on `M2` and on `WB`. Also add control logic needed for a data dependence between `mul` and an immediately following instruction. All those signals should connect to the existing *Stall ID* signal and use a new uses int mul logic block.

☐ Control logic for `WB` structural hazard, ☐ `M2` structural hazard, ☐ and the data dependence.



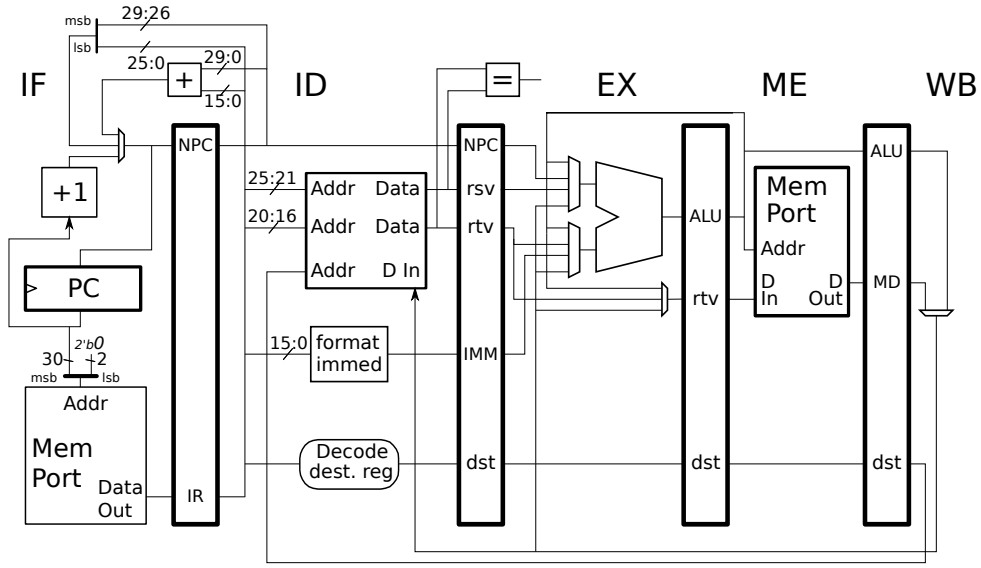
Problem 2: (10 pts) Show the execution of the instructions below on the illustrated implementation. Add any needed datapath and reasonable bypass paths.



☐ Show execution. ☐ Add datapath and reasonable bypass paths. ☐ Double-check for dependencies.

```
add.s f2, f4, f6

sub.s f8, f10, f12

add r1, r2, r3

mul.s f14, f2, f8

swc1 f14, 0(r1)
```

Problem 3: (10 pts) The code fragments below execute on several different MIPS implementations. In all cases the loop iterates many times. A five-stage scalar system is shown for reference.



(*a*) Show the execution of the code below on our familiar scalar pipeline, above. Show enough iterations to compute the CPI, and compute the CPI.

- ☐ Execution for enough iterations to determine CPI.
- ☐ Find the CPI.
- ☐ Doublecheck for dependencies.
- ☐ Note that the first instruction in not part of the loop body.

```
 lw r2, 0(r10)
LOOP:

 lw r1, 0(r2)

 addi r2, r2, 4

 bne r2, r4 LOOP

 add r5, r5, r1
```

Problem 3, continued:

(*b*) Show the execution of the code below on a 4-way superscalar statically scheduled system without branch prediction. The superscalar system has five stages, aligned fetch, and can bypass between the same stages as can our scalar system. This means there are no bypass paths to the branch condition. Compute the CPI.

- ☐ Execution for enough iterations to determine CPI.
- ☐ Find the CPI.
- ☐ The code below is *different* than the previous part.
- ☐ Doublecheck for dependencies.
- ☐ Note that first instruction not part of loop body.

```
 addi r1, r0, 0

LOOP: # Address of insn below is 0x1000

 add r5, r5, r1

 lw r1, 0(r2)

 bne r2, r4 LOOP

 addi r2, r2, 4
```

(*c*) Show the execution of the code below on a four-way superscalar system with perfect branch prediction.

- [ ] Execution for enough iterations to determine CPI.
- [ ] Find the CPI.
- [ ] The code below is *different* than the first part.
- [ ] Doublecheck for dependencies.
- [ ] Note that first instruction not part of loop body.

```
 addi r1, r0, 0
LOOP: # Address of insn below is 0x1000
 add r5, r5, r1
 lw r1, 0(r2)
 bne r2, r4 LOOP
 addi r2, r2, 4
```

Problem 4: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{10}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 12-outcome local history, and one system uses a global predictor with a 12-outcome global history.

```
Insn        Branch
Addr        Outcomes
0x1000: B1  r  r  r  r  r  r  r  r  r  r  r  r  r  r  r  r
0x1010: B2   T  N  N  N  N  T  N  N  T  N  N  N  N  T  N  N
0x1020: B3    R  R  R  R  R  R  R  R  R  R  R  R  R  R  R  R
0x2000: B4     T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
```

Branch `B1` is random, and can be described by a Bernoulli random variable with $p = .5$ (models a fair coin toss). The outcome of branch `B3` is the same as the most recent execution of `B1` (perhaps they are testing the same condition). For the questions below accuracy is after warmup.

☐ What is the accuracy of the bimodal predictor on `B2`?

☐ What is the accuracy of the bimodal predictor on `B3`?

☐ Considering BHT size, what is the approximate accuracy of the bimodal predictor on `B4`? ☐ Explain.

☐ What is the accuracy of the local predictor on `B2`?

☐ What is the minimum local history size needed to predict `B2` with 100% accuracy?

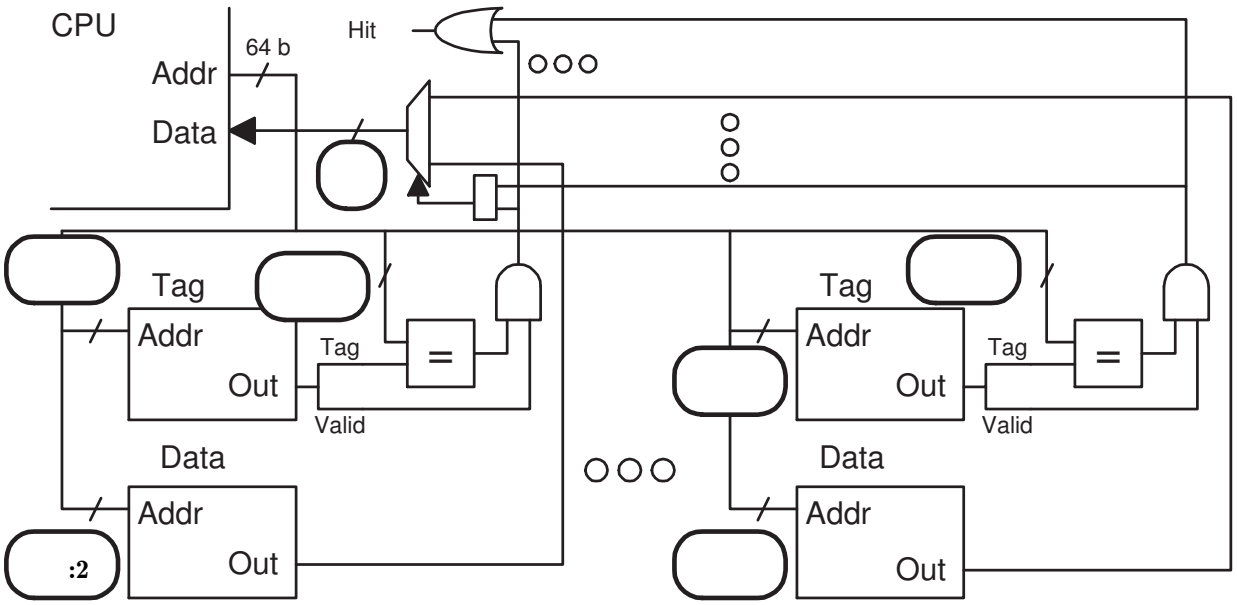
☐ What is the accuracy of the global predictor B3? ☐ Explain

☐ How many PHT entries are used by the global predictor to predict B2?

Problem 5: (15 pts) The diagram below is for an eight-way set-associative cache. The size of a tag is 42 bits and the size of a line is $128 = 2^7$ bytes.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Cache Capacity (Indicate Unit!!):

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 5, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 128-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

(*b*) Find the hit ratio executing the code below.

☐ What is the hit ratio running the code below? Show formula and briefly justify.

```
int sum = 0;
half *a = 0x2000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) Find the minimum positive value of `OFFSET` needed so that the code below experiences a hit ratio of 0% on accesses to `a`. Explain.

☐ Minimum positive `OFFSET` to achieve 0% hit ratio. ☐ Explanation.

```
int sum = 0;
half *a = 0x2000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;

int OFFSET =                                          <--  FILL IN

for ( i=0; i<ILIMIT; i++ ) sum += a[ i ] + a [ i + OFFSET ];
```

Problem 6: (30 pts) Answer each question below.

(*a*) For the SPECcpu benchmark suite results can be reported using two tuning levels, *base* and *peak*. Consider a new level, *no-opt* in which the code is compiled without optimization. How valuable would no-opt tuning scores be to the people that care about SPECcpu scores? How different is no-opt tuning from base tuning?

☐ Value of no-opt tuning level?

☐ Difference between no-opt tuning and base tuning?

(*b*) The `lw` below will experience a TLB miss exception when it first executes, remember that this exception is considered routine and is not due to any kind of error. The word in memory at location `0x12345678` is `0x222`. Show the execution of this code on our five-stage static pipeline until the `add` instruction reaches writeback, and show the value in register `r1` when the handler starts (see code).

☐ Show execution from `lui` to when `add` reaches `WB`.

☐ `FILL IN` the value that will be in `r1` when the handler starts.

```
lui r1, 0x1234

lw r1, 0x5678(r1)

add r2, r2, r1


HANDLER:
# Value of r1 is                          <- FILL IN
sw ...
...
```

(*c*) The instruction below is not a good candidate for a RISC ISA. Explain why in terms of hardware, not just in terms of a rule the instruction would violate. `add (r1), r2, (r3)`

☐ Instruction above unsuitable for RISC because the implementation ...

☐ Provide a quick sketch to illustrate your answer.

(*d*) With the aid of a diagram, explain why a $5n$-stage pipeline would need fewer bypass paths than a 5-stage, n-way superscalar implementation. (Both implementations are statically scheduled.)

☐ Diagram showing why there are fewer bypass paths in $5n$-stage pipeline than 5-way superscalar.

(*e*) Why is it more important to have a good compiler for a superscalar statically scheduled system than a scalar statically scheduled system?

☐ Compiler more important for superscalar because ...

(*f*) Consider the executions of the MIPS code below on a 4-way superscalar dynamically scheduled system of the type discussed in class (method 3). The first execution is correct, the others, though they would run the program correctly, have problems. Describe the problems by completing the statements below.

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q        RR EX WB C
lh r1, 0(r6)      IF ID Q  RR EA ME WB    C
sub r7, r1, r3    IF ID Q           RR EX WB C
```

☐ The one-commit-per-cycle execution below is silly because . . .

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q        RR EX WB C
lh r1, 0(r6)      IF ID Q  RR EA ME WB       C
sub r7, r1, r3    IF ID Q           RR EX WB    C
```

☐ The stalls shown below would be necessary on a statically scheduled pipeline because . . .

☐ . . . but should not occur on a dynamically scheduled one because . . .

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q  ----> RR EX WB C
lh r1, 0(r6)      IF ID Q  ----> RR EA ME WB C
sub r7, r1, r3    IF ID Q  ----------> RR EX WB  C
```

15 Spring 2011

Name _____

Computer Architecture
EE 4720
Midterm Examination

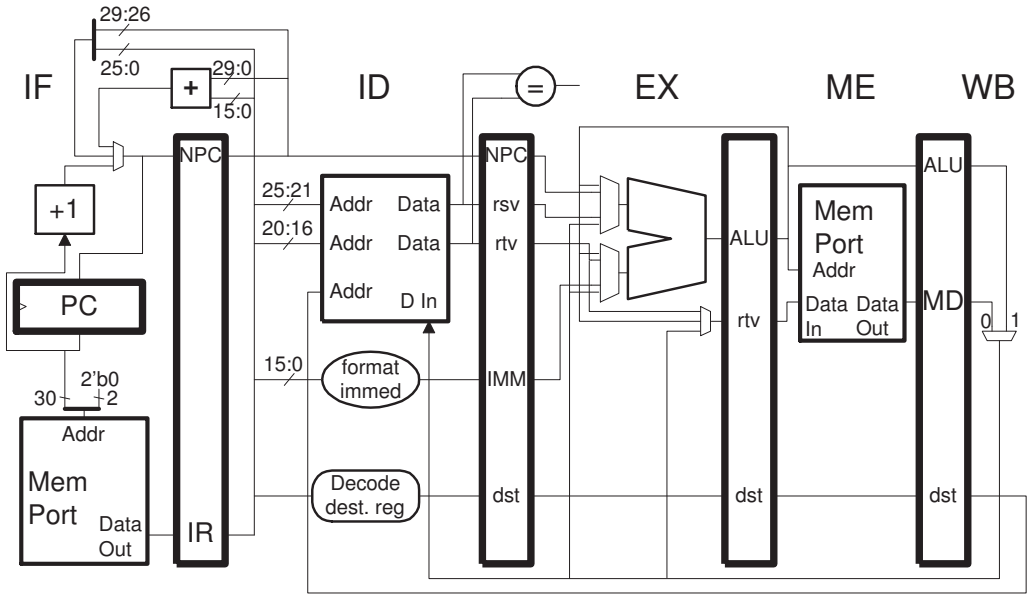
Wednesday, 30 March 2011, 9:40–10:30 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(15 pts)
Problem 7	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts] The MIPS code below executes on the illustrated hopefully familiar implementation.




```
LOOP:
lh r1, 0(r2)

addi r4, r4, 4

andi r3, r1, 0xf0f0

bne r3, r0 LOOP

lw r2, 4(r2)
```

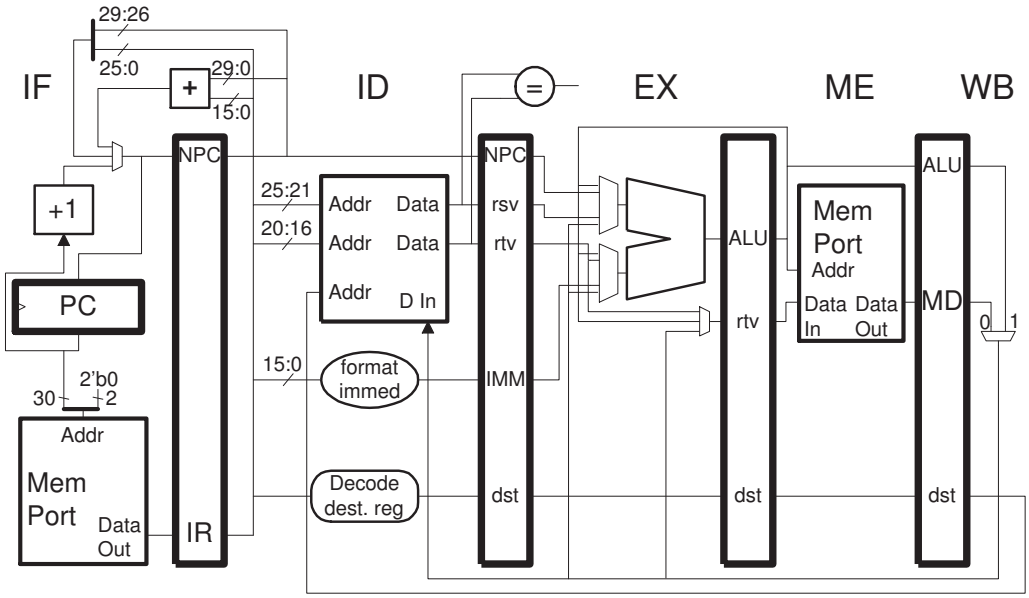
(*a*) Show a pipeline execution diagram for the code below running on the illustrated MIPS implementation for enough iterations to determine CPI.

☐ Show a pipeline execution diagram.

(*b*) Determine the CPI.

☐ Find the CPI.

Problem 2: [20 pts] In the implementation below the datapath for the `jalr` and `jr` instructions is not shown.



(*a*) Add datapath (but not control logic) needed for the `jr` instruction to the diagram above.

☐ Add datapath (but not control logic) for `jr`.

(*b*) Show the execution of the code below based upon the answer above. Show execution starting from the first instruction (as usual) and continue until `ori` executes or eight instructions have executed whichever is longer. (In the correct answer `ori` is the eighth instruction to execute.) Note that the `jalr` will jump to `ROUTINE` the first time it executes.

☐ Show execution consistent with previous answer.

```
 addi r1, r1, 16
 # r1 = ROUTINE
 jalr r31, r1

 addi r4, r31, -8

 ori r6, r6, 0xff

ROUTINE:
 addi r4, r4, 4

 jr r4

 lw r31, 0(r7)
```

(*c*) If your answer to the previous part was correct the code above would suffer stall(s). Add bypasses to avoid as many of the stalls as possible without significant critical path impact.

☐ Bypass paths to eliminate stalls.

Problem 3: [15 pts] In the code fragment below the `lw` raises an exception due to a TLB miss. Remember that a TLB miss is something that happens all the time to almost any load.

```
 # Cycle          0   1   2   3   4   5   6   7
 lw r1, 0(r2)     IF
 add r3, r5, r6
 xor r4, r3, r9
...
HANDLER:
 sw r10, ..
```

(*a*) Based on the exception hardware presented in class, at which cycle will the first instruction of the handler be in `IF`?

☐ Handler in `IF` in cycle:

For the next parts of this problem don't consider the exception hardware presented in class, instead the `lw` will trigger a deferred exception. That is, the `lw` will raise a TLB miss exception but `add` will successfully complete `WB` before the handler is called. Register `r1` will have some garbage value but the `add` will execute correctly.

(*b*) This is **not** a good example for why precise exceptions are needed for loads. Why not?

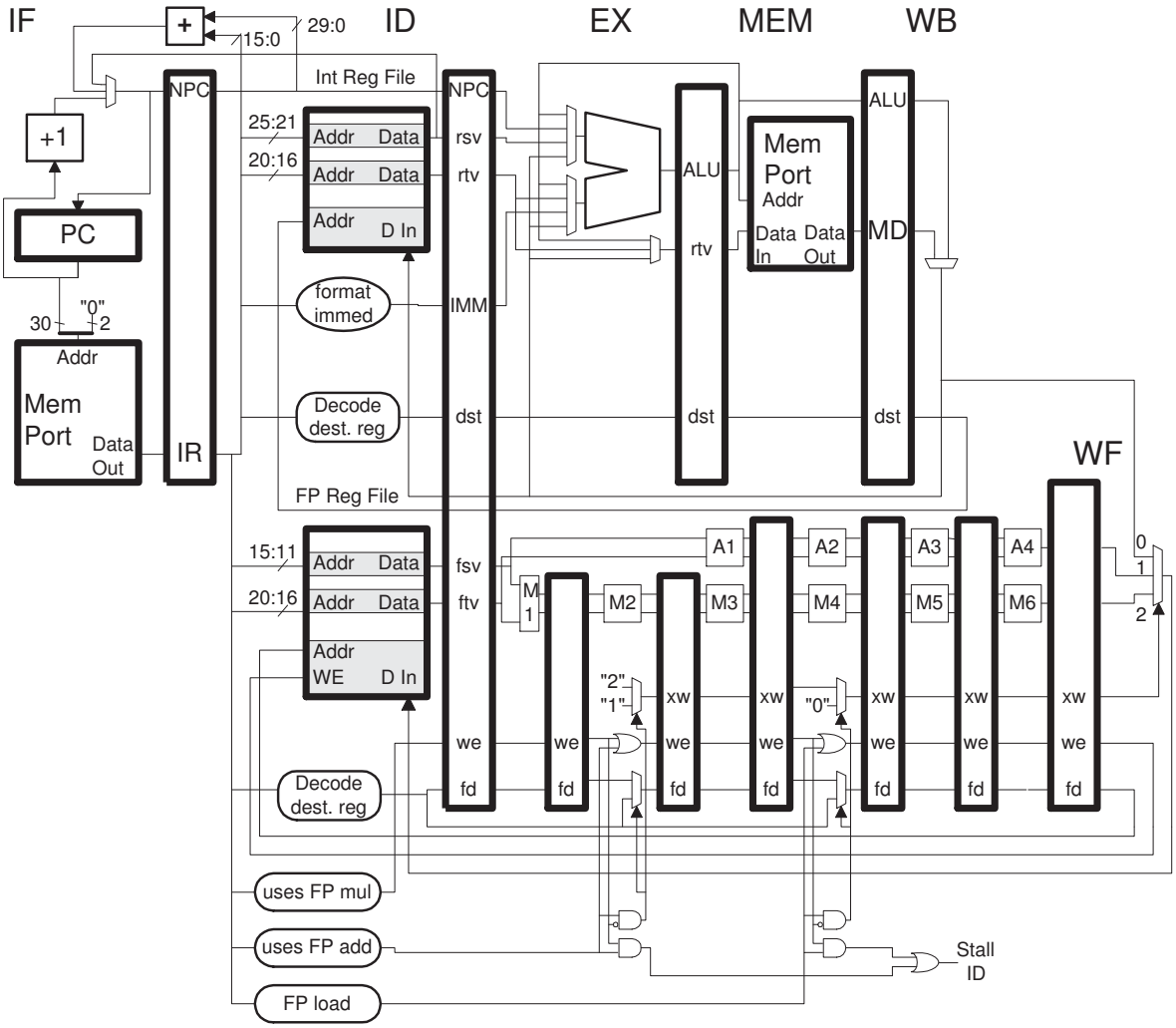
☐ Not a good example because:

(*c*) Modify the code after the `lw` so that this *is* a good example of why precise exceptions are needed. Briefly explain why.

☐ Modify code.

☐ Explain why it is a better example.

Problem 4: [10 pts] Show the execution of the MIPS code below on the illustrated implementation. Assume that all needed bypass paths are available.



☐ Execution of code.

```
add.d f2, f4, f6

sub.d f8, f4, f10

mul.d f12, f2, f14

addi r1, r1, 8
```

Problem 5: [10 pts] Answer each question about the SPECcpu suite.

(*a*) With SPECcpu the tester is responsible for providing compilation tools and choosing optimization switches. Which feature or goal of the SPEC benchmarks does that support?

☐ Compilation tools and optimization choices supports:

(*b*) Part of the SPEC rules requires that compilation tools be actively marketed as products, not just available to someone who knows exactly how to ask for them. What sort of abuse does that prevent?

☐ Actively marketed compilation tools prevents:

Problem 6: [15 pts] Answer each question below.

(*a*) ISAs are frequently updated, examples mentioned in class include MIPS I, MIPS II, SPARC v8, SPARC v9, and the multimedia extensions such as MMX, SSE, SSE 2, etc. for IA-32. With all these updates it sounds like an ISA is not really frozen for All Time. Does that mean the main advantage in separating ISA design from implementation design has been lost?

☐ Has main advantage of ISA/implementation separation been lost? Explain.

(*b*) Why is the typical RISC program larger than an equivalent CISC program?

☐ RISC larger than CISC because:

(*c*) Explain how the approaches to encoding immediate arithmetic instructions differ in the SPARC and MIPS ISAs.

☐ How approaches differ.

Problem 7: [15 pts] Answer each question below.

(*a*) Provide the following optimization examples:

☐ An optimization that can be performed well without knowing the particular implementation being targeted.

☐ An optimization that can be performed well only if the compiler knows the particular implementation being targeted.

(*b*) In an alternate universe, when designing the ALT-MIPS ISA, computer engineers insisted, perhaps for cost reasons, that the data memory port and ALU had to be in the same stage (that is, the `EX` and `ME` stages would be combined), resulting in a four-stage pipeline. How would the matching ALT-MIPS be different from our MIPS ISA? Note that elegance and performance are important for both ALT-MIPS and MIPS. Provide two code samples, one for which the four-stage ALT-MIPS is better and one for which the five-stage MIPS is better.

☐ ALT-MIPS differences with MIPS.

☐ Code sample better for ALT-MIPS.

☐ Explain.

☐ Code sample better for MIPS.

☐ Explain.

Name _____

Computer Architecture
EE 4720
Final Examination

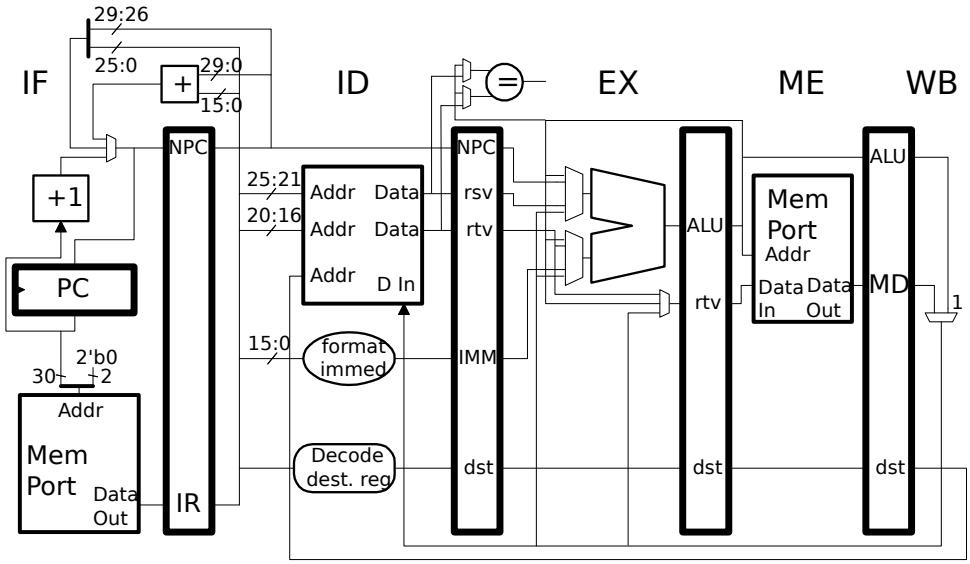
9 May 2011, 15:00–17:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (20 pts) The MIPS implementation below is similar to the one frequently used in class, except that it has bypass paths to the branch condition comparison unit.



(*a*) Show the execution of the code fragments below on the illustrated implementation up until the fetch of the first instruction of the second iteration. Be sure to account for the dependence on the branch condition.

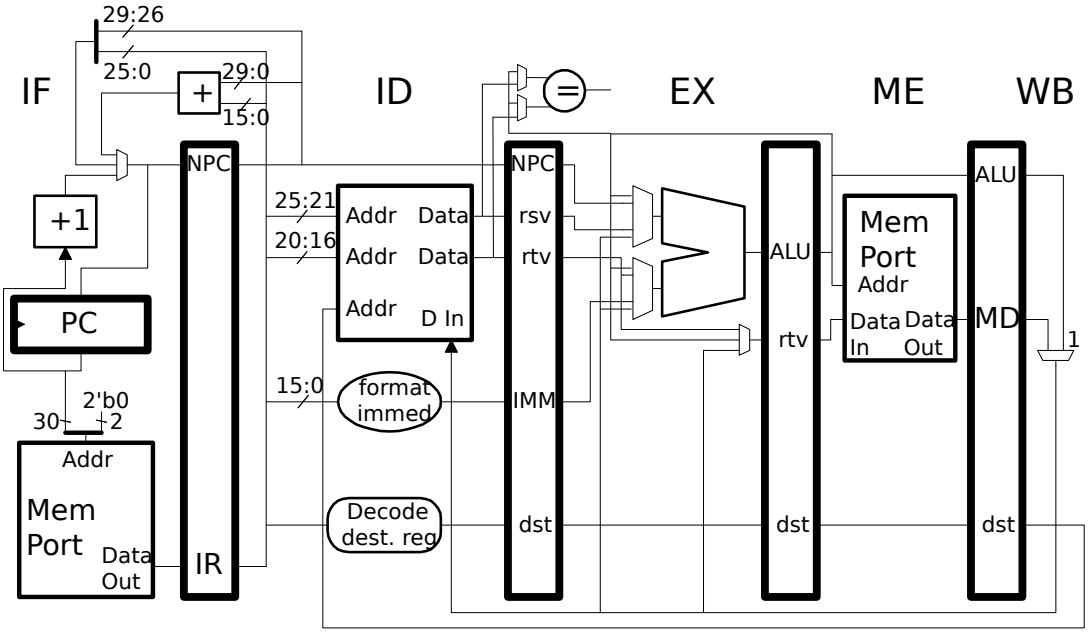
☐ Pipeline execution diagram of code below.

```
LOOP1:
 lw r1, 0(r2)

 addi r2, r2, 4

 bne r2, r3 LOOP1

 add r4, r4, r1
```

☐ Pipeline execution diagram of code below.

```
LOOP2:
 lw r1, 0(r2)

 lw r2, 4(r2)

 bne r2, r0 LOOP2

 add r4, r4, r1
```

(*b*) The implementation below has hardware in `IF` (not shown) to predict branches. Since it has prediction hardware the branch resolution hardware (the hardware that computes the branch condition and branch target) could be moved to the `ME` stage. Move the branch resolution hardware. Note that the resolution hardware will be used only if the branch is mispredicted.



☐ Move resolution hardware to the `ME` stage.

☐ Add any bypasses to resolution hardware needed by code samples in this problem.

☐ Cross out unused hardware in `ID`.

☐ Avoid adding unnecessary hardware.

Problem 1, continued: Consider the implementation from the previous part.

(*c*) Resolving the branch in `ME` with prediction can hurt performance with the code below compared to resolving in `ID` with prediction when the branch is hard to predict. By how much will it hurt performance? *Note: An exact number was not required on the original exam.* Explain, use a diagram if necessary. *Hint: Resolve is where recovery starts for a misprediction. Another hint: pay attention to dependencies on the branch condition.*

```
LOOP1:
 lw r1, 0(r2)
 addi r2, r2, 4
 bne r2, r3 LOOP1
 add r4, r4, r1
```

☐ Resolving in `ME` hurts performance here by ____ cycles.

☐ Explanation.

(*d*) Resolving the branch in `ME` with prediction should help performance with the code below compared to resolving in `ID` with prediction. Explain why, preferably with an execution diagram. *Hint: Same hints as previous part.*

```
LOOP2:
 lw r1, 0(r2)
 lw r2, 4(r2)
 bne r2, r0 LOOP2
 add r4, r4, r1
```

☐ Resolving in `ME` helps performance because...

Problem 2: (10 pts) The diagrams below show the execution of code on two-way superscalar dynamically scheduled systems of the type described in class. Each diagram **has mistakes**. Identify and fix them. Also explain why code should not execute as illustrated. The answer should specify what would go wrong, or why the system would be particularly inefficient, as appropriate.

(*a*) Find and fix the two problems with commit in the execution below, and explain why execution would be incorrect or inefficient.

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
sub r4, r5, r6   IF ID Q  RR EX WB C
or  r7, r4, r8      IF ID Q  RR EX WB    C
# Cycle          0  1  2  3  4  5  6  7  8
```

☐ Fix the two problems above.

☐ Describe one error or inefficiency with execution.

☐ Describe another error or inefficiency with execution.

(*b*) Find and fix the problem with the code below. Note that the processor has two FP adders, named A and B. *Hint: Check dependencies.*

```
add.d f0, f2, f4       IF ID Q  RR A1 A2 A3 A4 WB C
sub.d f6, f0, f8       IF ID Q              RR A1 A2 A3 A4 WB C
add.d f10, f11, f12       IF ID Q           RR B1 B2 B3 B4 WB C
addi r1, r1, 4            IF ID Q              RR EX WB         C
```

☐ Fix the problem above.

☐ Describe one error or inefficiency with execution.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history. The outcomes of branch `B1` form a repeating pattern, the outcome of `B2` can be modeled by a Bernoulli random variable with $p = .7$ (the branch is taken with probability .7), and `B3` is always taken.

```
B1:  N   T   T   T   T   T   N   N     N   T   T   T   T   T   N   N
B2:    R   R   R   R   R   R   R   R     R   R   R   R   R   R   R   R
B3:     T   T   T   T   T   T   T   T     T   T   T   T   T   T   T   T
```

For the questions below accuracy is after warmup.

☐ What is the accuracy of the bimodal predictor on `B1`?

☐ What is the approximate accuracy of the bimodal predictor on `B2`? Explain. *Hint: A correct answer would be slightly more or less (pick one) than.... For the exact answer one would need the state probability formula for a Markov chain.*

☐ What is the accuracy of the local predictor on branch `B1`?

☐ What is the warmup time of the global predictor on branch `B1`? Be sure to consider the effect of `B2`.

☐ What is the minimum local history size needed for a local predictor to predict `B1` with 100% accuracy?

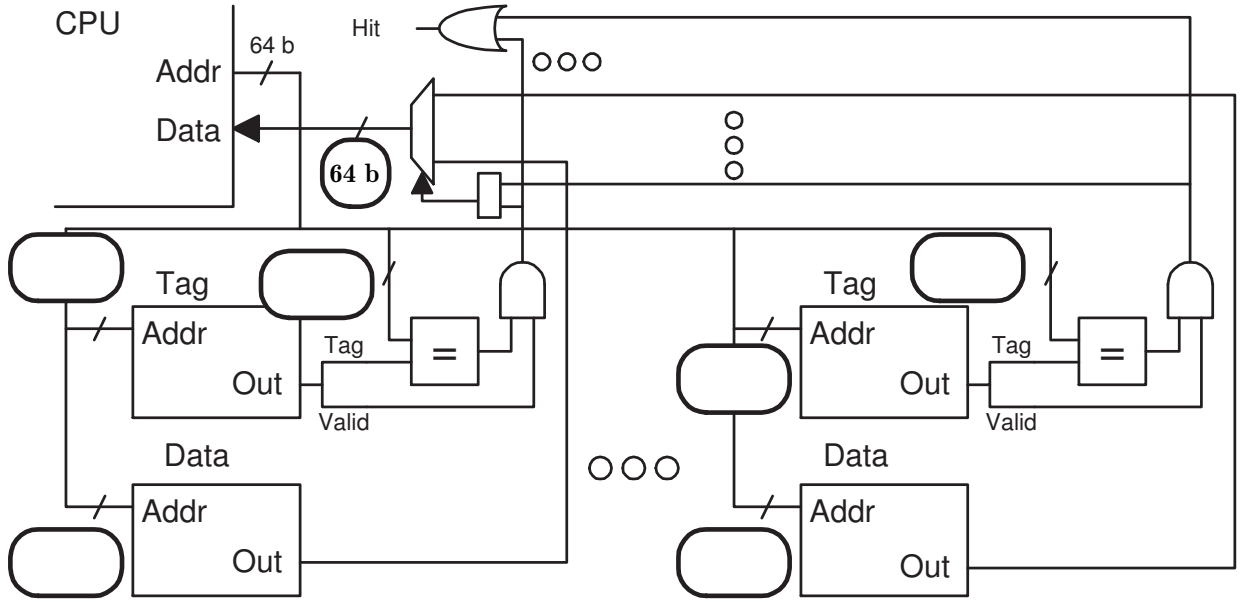
☐ What is the accuracy of a global predictor with a three-outcome global history on branch `B1`?

☐ What is the minimum global history size needed for a global predictor to predict `B1` with 100% accuracy?

Problem 4: (20 pts) The diagram below is for a set-associative cache with a capacity of 16 MiB (2^{24} bytes), and a line size of 64 bytes. The system has the usual 8-bit characters. Each data store below has a capacity of 2^{20} bytes.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 256-byte line size. Each code fragment starts with the cache empty; consider only accesses to the arrays, `a`, `ax`, and `ay`.

(*b*) Find the hit ratio executing the code below.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
double *a = 0x20000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) The two code fragments below, labeled Method 1 and Method 2, perform the same computation but organize the data differently.

```
// Method 1
 struct Our_Struct { int x;  int y; };
 Our_Struct a[SIZE];

 for (int i=0; i<SIZE; i++) sum += a[ i ].x;
 for (int i=0; i<SIZE; i++) myfunc(sum,a[ i ].y);

// Method 2
 int ax[SIZE];
 int ay[SIZE];

 for (int i=0; i<SIZE; i++) sum += ax[ i ];
 for (int i=0; i<SIZE; i++) myfunc(sum,ay[ i ]);
```

☐ Which method has a higher hit ratio if the cache is small (or `SIZE` is large)? Explain

Problem 5: (30 pts) Answer each question below.

(*a*) Eliminating bypass paths from an n-way superscalar statically scheduled processor would have a much larger effect than eliminating them from a scalar statically scheduled processor (like our 5-stage MIPS implementation).

☐ Describe a positive benefit of eliminating the bypass paths that's much larger for the n-way superscalar.

☐ Describe a disadvantage of eliminating the bypass paths that's a much bigger disadvantage for the n-way superscalar.

(*b*) Consider a deeply pipelined system with $5n$ stages without bypass paths and that runs programs in which dependent instructions are far apart. Name two factors that will limit clock frequency for larger values of n. *Note: bypass paths were not mentioned in the original exam.*

☐ One frequency limiter.

☐ Another frequency limiter.

(*c*) Consider the BHT of a bimodal branch predictor. An entry would contain a CTI type, a two-bit counter, a branch target, and perhaps a tag. The tag would be used like a cache tag, but does not need to be as large. For this problem only consider branches.

☐ How large would the target need to be to predict MIPS branches? *Hint: The answer is not 32 bits.*

☐ Explain what the predictor can do with the tag to improve prediction accuracy.

(*d*) A four-way statically scheduled processor is much less expensive than a four-way dynamically scheduled processor. Why would the dynamically scheduled processor run some code faster than the statically processor even when the code was compiled with a good compiler that targeted the specific four-way implementation?

☐ Specific advantage of dynamically scheduled processor for some code.

☐ Describe properties of code that would run at the same rate on the two processors.

(*e*) In many VLIW ISAs the bundle size is 3 slots. Describe a disadvantage of making a VLIW ISA with a larger bundle size.

☐ Disadvantage of larger bundle size.

(*f*) A tester measuring the performance of a system using SPECcpu has the source code to SPECcpu programs. Why is that important to the goals of the suite?

☐ Source code important to SPECcpu goals?

16 Fall 2010

Name _____

Computer Architecture
EE 4720
Midterm Examination

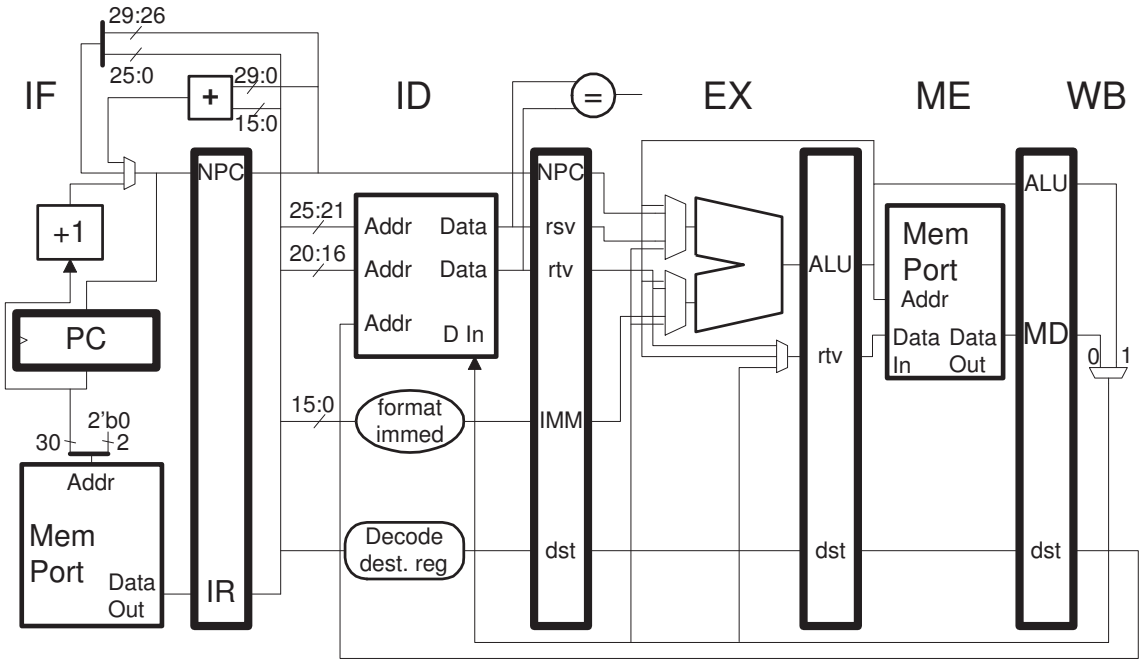
Wednesday, 3 November 2010, 10:40–11:30 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(23 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(7 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: Show the execution of the following code fragments on the illustrated MIPS implementations.



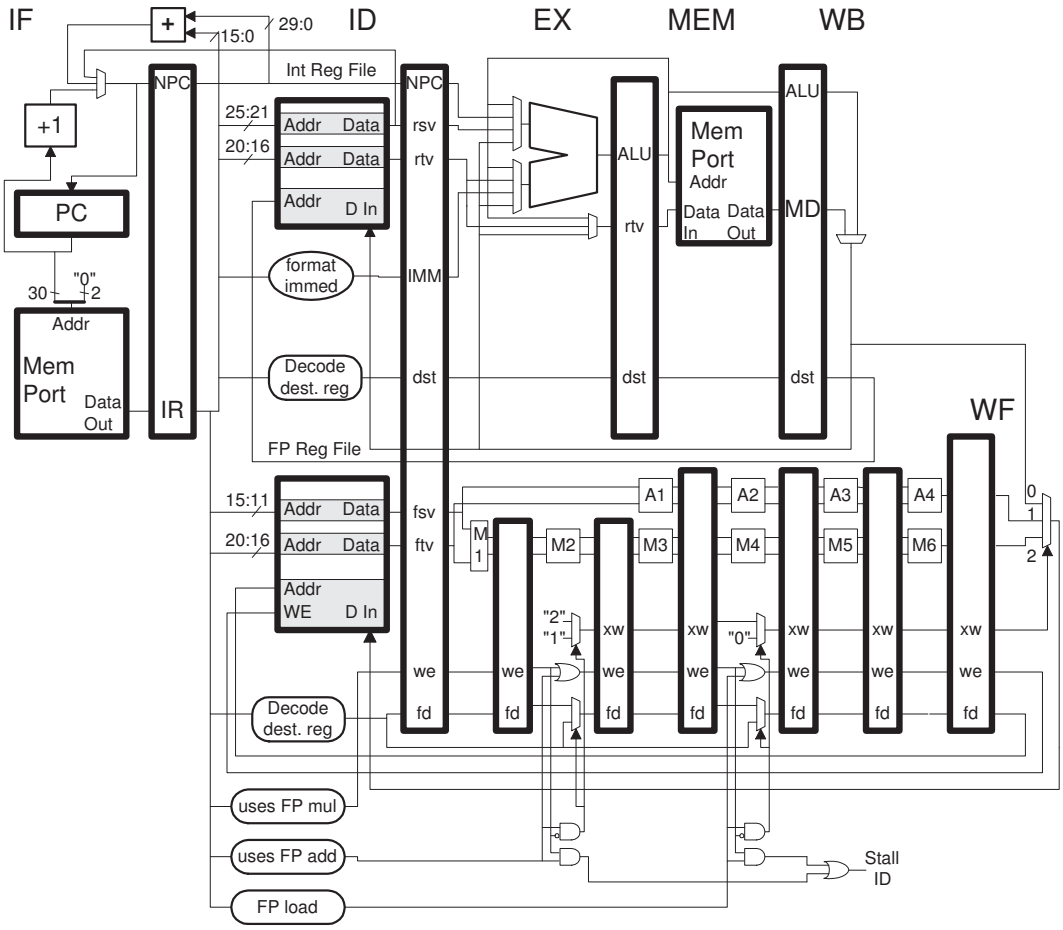
(*a*) [20 pts] The code below executes for many iterations. Show a pipeline execution diagram for the execution of the code on the implementation above for enough iterations to determine the CPI, and determine the CPI.

```
LOOP:
 lw r1, 0(r2)

 lw r3, 0(r1)

 bne r3, r4, LOOP

 lw r2, 8(r1)
```

☐ Pipeline diagram of execution. Don't forget to check for dependencies!

☐ CPI for a large number of iterations.

Problem 1, continued:



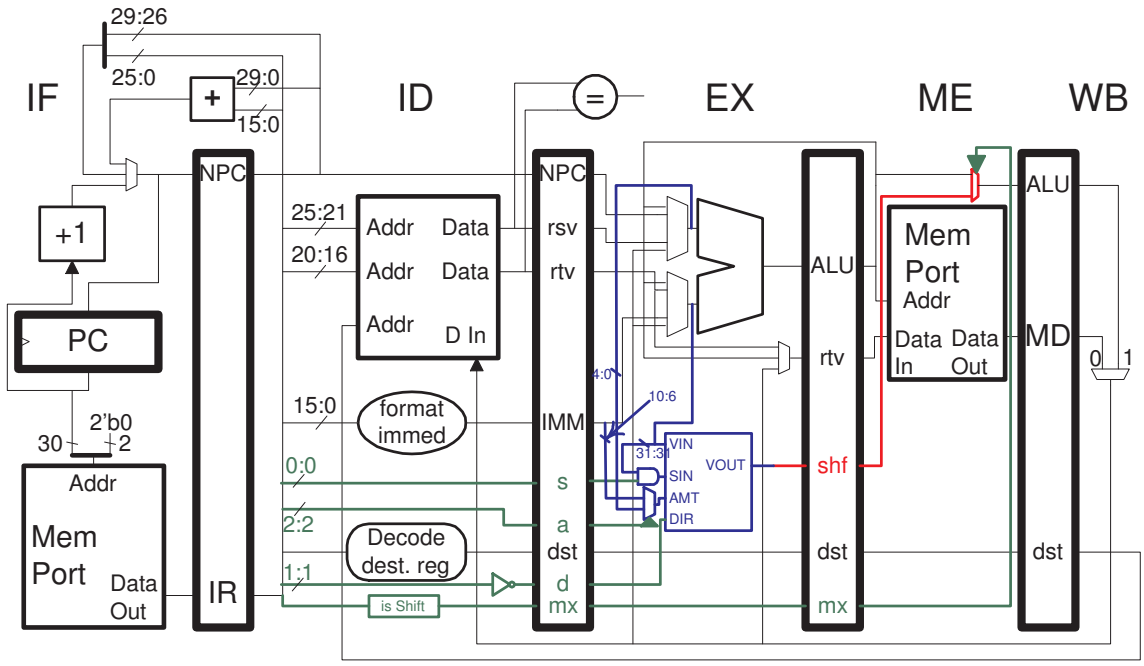
(*b*) [10 pts] Show the execution of the code below on the implementation above.

```
mul.s f1, f2, f3

add.s f4, f5, f6

add.s f7, f8, f9

lwc1 f10, 0(r1)
```

☐ Pipeline diagram of execution.

Problem 2: The MIPS implementation below includes the shift unit (taken from the Homework 3 solution). Notice that it is not possible to bypass a shift result to the EX stage. For that reason the `add` in the code below stalls:

```
# Cycle          0  1  2  3  4  5  6
sll r1, r2, r3   IF ID EX ME WB
add r4, r1, r5      IF ID -> EX ME WB
```



(*a*) [10 pts] Design control logic to generate a signal named `STALL` which will be logic `1` when a stall is needed due to a shift result that cannot be bypassed, as in the example above. (Generate the signal, but don't do anything with it.)

☐ Logic to generate stall signal.

(*b*) [5 pts] The stall above can be avoided by disconnecting the `ME`-to-`EX` bypass from the `EX/ME.ALU` latch and instead connecting it to the output of the `ME`-stage mux. What was the reason for **not** doing something like that in Homework 3.

☐ Problem with `ME`-to-`EX` bypass for shift results.

Problem 3: Answer the following questions about interrupts.

(*a*) [7 pts] An important part of an ISA's interrupt mechanism is a separate privileged mode (also called system or supervisor mode) and user mode.

☐ Why is it necessary to have these two modes?

☐ Explain the difference between privileged and user mode in how the CPU operates.

(*b*) [8 pts] In class our 5-stage implementations resolved exceptions only in `ME`, and by doing so all integer-pipeline exceptions were precise. *Note: In the original exam the phrase "and by doing so..." was not present.* Suppose instead we resolved exceptions in `WB`. Show a code fragment in which an exception could not be precise on such a system.

☐ Simple code fragment.

☐ What can't the handler do, and why can't it do it?

Problem 4: Answer each question below.

(*a*) [8 pts] Consider the distinction between an ISA and its implementation.

☐ What was to be achieved by separating computer design into separate ISA design and implementation design?

☐ Provide a reason not to have separate ISA and implementation design, consider the point of view of a conservative computer engineer from the 1960s.

(*b*) [8 pts] One reason to not compile a program with optimization turned on is because you plan to debug the program. Explain why stepping through a program in a debugger can be confusing when the code has been optimized.

☐ Debugging optimized code is confusing because ...

☐ Name a particular type of optimization and explain how it can cause confusing results when single stepping through a program.

(*c*) [7 pts] Do you agree or disagree with the statement below regarding the rules for building the SPECcpu benchmarks? Answer with respect to the goals for the SPECcpu benchmarks.

"Testers should not be allowed to use their own compilers because the SPECcpu benchmarks are supposed to test CPUs, not compilers. All testers of a particular ISA should use the same compiler." *Grading note: the phrase "of a particular ISA" was not in the original exam.*

☐ Agree or disagree? Explain.

Problem 5: [10 pts] CISC programs generally are smaller than RISC programs.

(*a*) Show how the instruction below is encoded in MIPS and in VAX. For MIPS the name and bit position of each field should be known, and the value of all but one field should be known. For VAX one should know the fields and their sizes, but not every field value needs to be known. *Hint: VAX has 16 general-purpose registers. The size of the two instructions should be the same.*

```
add r1, r2, r3
```

☐ Encoding in MIPS.

☐ Encoding in VAX.

(*b*) Identify unused field(s) in the MIPS instruction.

☐ Unused MIPS field.

(*c*) Since VAX instruction sizes can vary they should be able to use space more efficiently. Yet even though the MIPS instruction has unused field(s) the two instructions are the same size. Explain why the VAX instruction is larger than one might expect and explain what advantage that provides.

☐ Why is VAX larger than one might expect?

☐ What advantage does this larger size provide?

Problem 6: Answer each question below.

(*a*) [7 pts] Unlike MIPS, SPARC integer branches use condition codes.

☐ Why might this allow SPARC branch targets to be further away than MIPS branch targets?

☐ Why might this enable certain SPARC implementations to have a higher clock frequency than comparable MIPS implementations?

Name _____

Computer Architecture

EE 4720

Final Examination

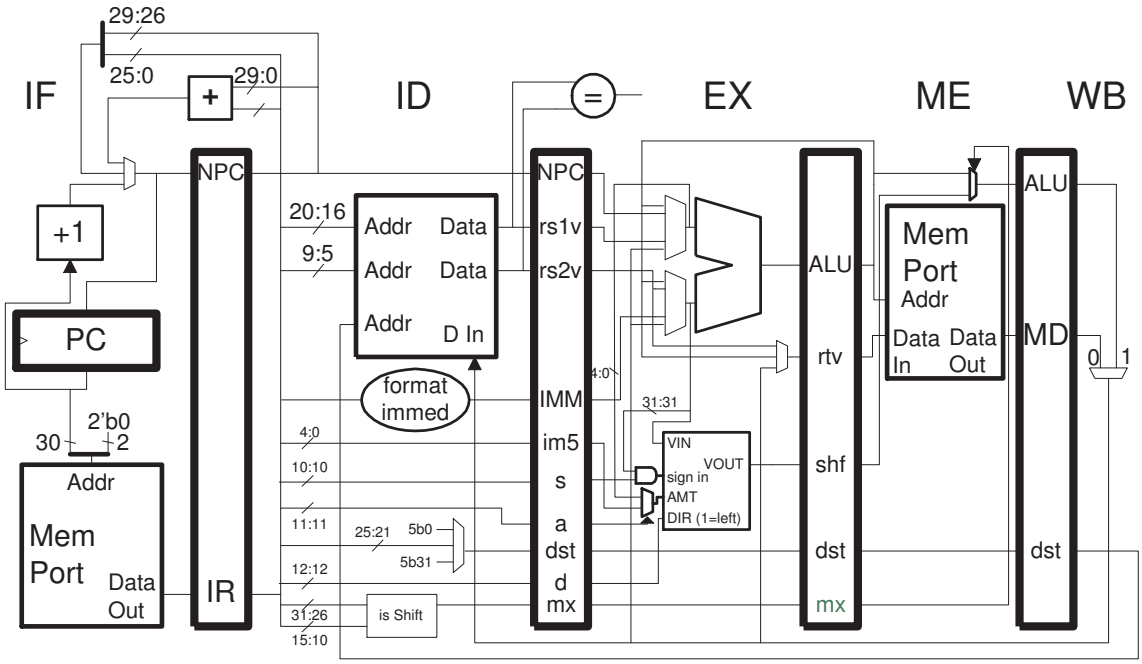
6 December 2010, 7:30–9:30 CST

Problem 1	_____	(10 pts)
Problem 2	_____	(14 pts)
Problem 3	_____	(14 pts)
Problem 4	_____	(14 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(15 pts)
Problem 7	_____	(18 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

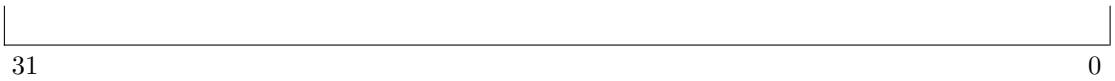
Problem 1: (10 pts) The diagram below shows a 5-stage pipeline that looks alot like our familiar MIPS implementation but it's actually an implementation of ISA X. (The diagram is based on the solution to Homework 3, in which a shift unit was added to MIPS.)



(*a*) ISA X instruction format T encodes the shift instructions and others, it is the equivalent of format R in MIPS. Based on the diagram above show the encoding for ISA X format T.

☐ Format T encoding, including bit positions and field names.

Encoding:



(*b*) Consider the shift instructions `sll`, `sllv`, `srl`, `srlv`, `sra`, and `srav`. Suppose that the encoding of one of these instructions is zero (meaning that every field value is zero). Show the opcode field value(s) for each of these instructions based on the diagram above. *Hint: The control signal for each top mux input is 0, etc.*

☐ Opcode field value(s) for: `sll`, `sllv`, `srl`, `srlv`, `sra`, and `srav`.

(*c*) Explain why the implementation of instructions such as `sw r1,2(r3)` and `beq r1, r2 TARG` would be less elegant for ISA X than for MIPS. *Hint: It has something to do with registers.*

☐ `sw` and `beq` less elegant because...

Problem 2: (14 pts) Answer the questions below.

(*a*) Complete the execution diagram for the MIPS code below on a two-way superscalar statically scheduled implementation of the type described in class.

```
add r1, r2, r3

add r4, r5, r6

add r7, r4, r8

lw  r9, 0(r7)

addi r1, r9, 3

xor r10, r11, r12
```

☐ Complete the diagram above.

(*b*) Show the execution of the code below on an 8-way superscalar statically scheduled processor of the type described in class. Branches are not predicted. Find the CPI for a large number of iterations.

```
LOOP: # Address of first insn is 0x1000

and r1, r1, r5

add r3, r3, r1

lw r1, 0(r2)

bne r2, r4 LOOP

addi r2, r2, 4
```

☐ Complete diagram above for enough iterations to determine CPI.

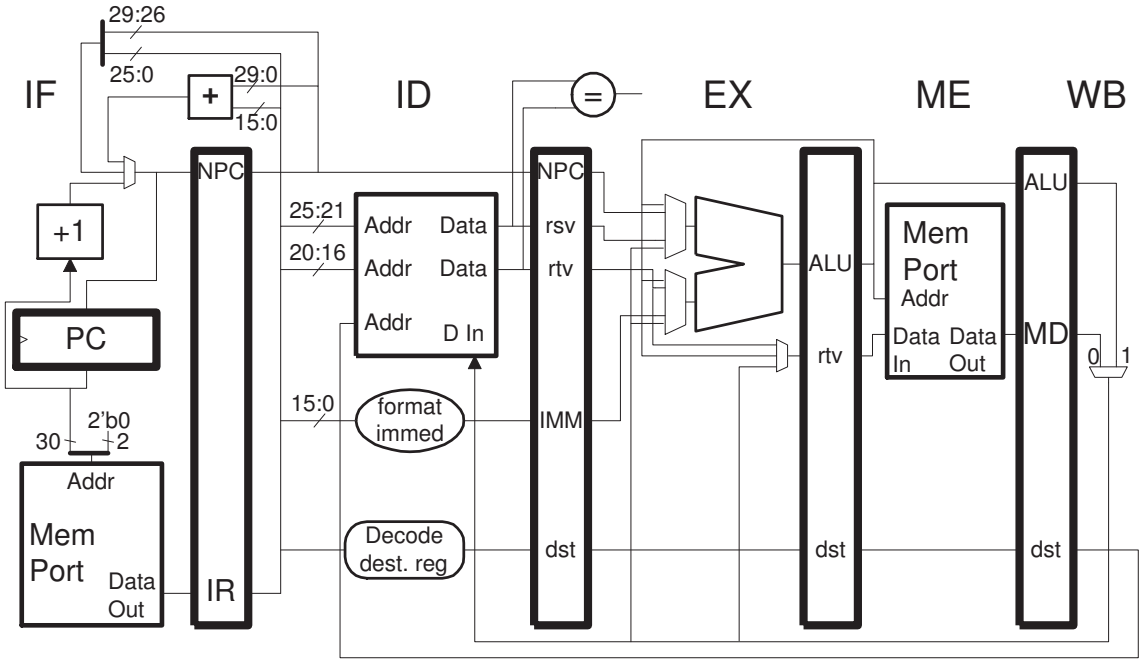
☐ Find the CPI for a large number of iterations.

(*c*) Complete the execution diagram for the MIPS code below on a two-way superscalar dynamically scheduled implementation of the type described in class. The execution of the first instruction is shown. The `lw` instruction uses stages `EA` and `ME` in place of `EX`.

```
add r1, r2, r3          IF   ID   Q    RR   EX   WB   C

add r4, r5, r6

add r7, r4, r8

lw  r9, 0(r7)

addi r1, r9, 3

xor r10, r11, r12
```

☐ Complete diagram above.

Problem 3: (14 pts) A deeply pipelined MIPS implementation can be constructed by dividing some stages of our familiar 5-stage statically scheduled scalar implementation (shown below) into two or more parts. In this problem the technique is applied to construct several 8-stage implementations. All have just one `ID` and `WB` stage, and in all implementations **it takes** 4.4 ns **for an instruction to pass through all 8 stages**, from the beginning of `IF1` to end of `WB1`. The stages are divided without changing what they do. For example, if an "original" MIPS stage, say `EX`, is divided into multiple stages, say `EX1 EX2 ... EXn`, then all values from `ID` and bypass paths are needed when `EX1` starts, and values reach `ME` in the cycle after `EXn`. Our familiar 5-stage implementation is shown below for reference:



(*a*) Consider the 8-stage *baseline* implementation below (indicated by the stage labels). What is the execution rate of a friendly program (one written to maximize performance) on the implementation, in units of instructions per second? The answer can be given as a formula of constants (as opposed to putting down just x as an answer).

`Baseline Implementation: IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1`

☐ Execution rate in instructions per **second**.

(*b*) Call a program *favorable* for an implementation if it runs faster on the implementation than on the baseline (repeated below). If it runs slower, call it *unfavorable*. For each implementation below write a two- or three-instruction favorable program and a two- or three-instruction unfavorable program. Also provide a concise but clear explanation of what it is about the program and implementation that makes it favorable or unfavorable. If a program is favorable on one implementation and unfavorable on another write it once, but provide an explanation for each. *Hint: The three implementations differ in how they are affected by certain dependencies.*

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 1:  IF1 IF2 IF3 ID1 EX1 ME1 ME2 WB1
```

☐ Favorable program. ☐ Explanation.

☐ Unfavorable program. ☐ Explanation.

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 2:  IF1 ID1 EX1 EX2 EX3 ME1 ME2 WB1
```

☐ Favorable program. ☐ Explanation.

☐ Unfavorable program. ☐ Explanation.

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 3:  IF1 ID1 EX1 EX2 ME1 ME2 ME3 WB1
```

☐ Favorable program. ☐ Explanation.

☐ Unfavorable program. ☐ Explanation.

Problem 4: (14 pts) Answer the following branch predictor questions.

(*a*) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 16-outcome local history, and one system uses a global predictor with a 16-outcome global history.

```
0x1000: B1:  T N  T T N  T T T N   T N  T T N  T T T N     ...
0x1010: B2:     T      T        N     T      T        N    ...
0x1020: B3:                       T                    T   ...
```

For the questions below accuracy is after warmup.

☐ What is the accuracy of the bimodal predictor on `B1`?

☐ What is the accuracy of the local predictor on branch `B1`?

☐ What is the warmup time of the local predictor on branch `B1`?

☐ What is the minimum local history size needed for a local predictor to predict `B1` with 100% accuracy?

☐ What is the accuracy of a global predictor with a three-outcome global history on branch `B2` (not `B1`)?

☐ What is the minimum global history size needed for a global predictor to predict `B2` (not `B1`) with 100% accuracy?

Problem 4, continued:

(*b*) Consider code producing the patterns below running on two systems, one using a global predictor and the other using a gshare predictor. Both systems use a 2^{16}-entry BHT and a 12-outcome global history. The hexadecimal numbers indicate the address of the branch instruction. For example, `B5: 0x1100` indicates that the instruction we call `B5` is at address `0x1100`.

```
Pattern L:  T T T ... N    (A hundred iteration loop.)

B5: 0x1100:  L       L       L       L
B6: 0x1110:    N       N       N       N
B7: 0x1200:      L       L       L       L
B8: 0x1210:        T       T       T       T
```

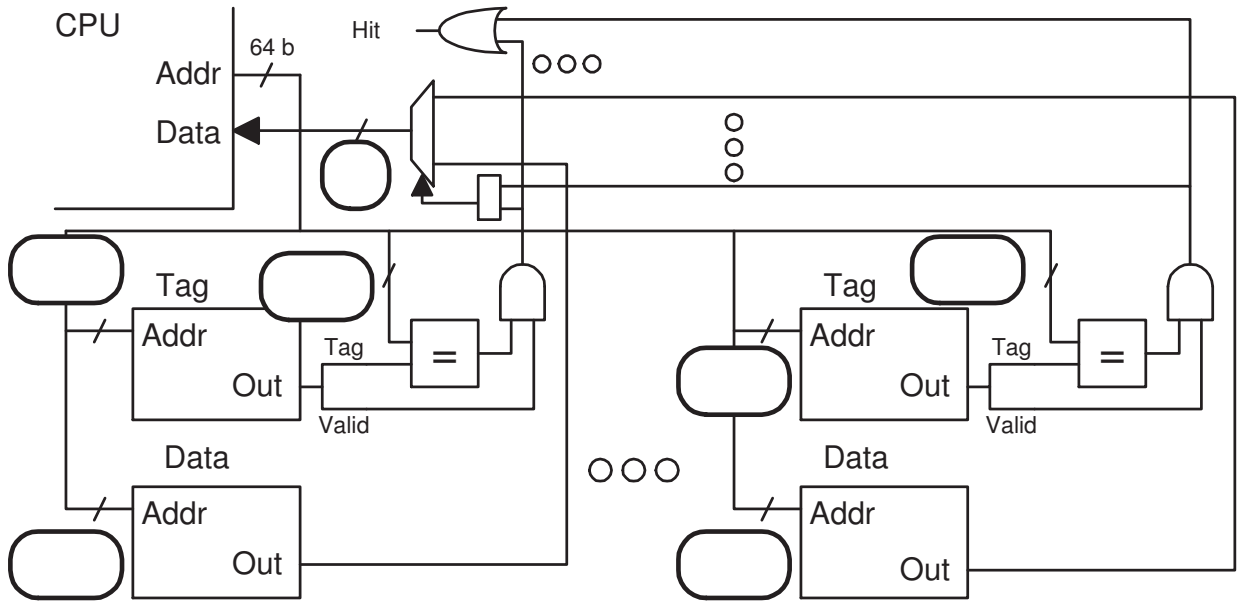
☐ Why is the accuracy of the gshare predictor so much better than the global predictor on this example?

☐ What is the minimum number of BHT entries for which the gshare predictor will outperform global on this code? *Hint: Look at the branch addresses.*

Problem 5: (15 pts) The diagram below is for a **4-way** set-associative cache with a capacity of 8 MiB (2^{23} bytes). The system has the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

7 **4**

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 5, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 64-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

(*b*) Find the hit ratio executing the code below.

☐ What is the hit ratio running the code below? Explain

```
float sum = 0.0;
float *a = 0x2000000; // sizeof(float) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) Find the smallest value of `STRIDE` for which the cache hit ratio in the program below will be zero.

☐ Fill in smallest value for `STRIDE` for which hit ratio 0.

☐ Briefly explain.

```
float sum = 0.0;
float *a = 0x2000000; // sizeof(float) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11

int STRIDE =                                          ;    //   <---- FILL IN

for (i=0; i<ILIMIT; i++) sum += a[ i ] + a[ i + STRIDE ];
```

Problem 6: (15 pts) Answer each question below.

(*a*) What is the difference between instruction-level parallelism (ILP) and explicit parallelism?

☐ ILP and explicit parallelism difference.

(*b*) A company has a large customer base for its ISA Y products, the most advanced of which is a 4-way superscalar dynamically scheduled implementation. The company is considering three possible next-generation systems: Develop an 8-way superscalar dynamically implementation of ISA Y, develop a chip with 16 scalar implementations of ISA Y, or develop a VLIW ISA and implementation. For each strategy indicate how much effort it will take for customers to use the new system.

☐ Customer effort to use 8-way superscalar dynamically scheduled system. Explain.

☐ Customer effort to use chip with 16 scalar implementations. Explain.

☐ Customer effort to use VLIW implementation.

(*c*) Consider the three systems from the previous part. For each system indicate an advantage over the others. The advantage should specify an assumed workload, an answer might start "The advantage, those customers that run programs that are _____, is ....

☐ Advantage of 8-way superscalar dynamically scheduled system.

☐ Advantage of 16 scalar system chip.

☐ Advantage of VLIW implementation.

Problem 7: (18 pts) Answer each question below.

(*a*) Why might a 1% change in branch prediction accuracy have a larger impact on the performance of a 4-way superscalar processor than on a 2-way superscalar system?

☐ Larger impact on 4-way over 2-way because...

(*b*) Dynamically scheduled systems use a technique called register renaming in which an instruction's architected registers are renamed into physical ones. Provide a brief example illustrating why register renaming is necessary.

☐ Example illustrating need for register renaming and explanation.

(*c*) The SPECcpu rules require that the compilers used to prepare a run of the suite be real products that the company (or some other company) makes an honest effort to sell (or give away). Explain how the SPECcpu scores might be inflated if the compilers could be products in a technical sense, but not something the company is trying to put in customer hands.

☐ Scores inflated by unmarketed compilers because...

☐ If the compilers produce faster programs, why not promote them?

(*d*) What is the difference between a trap and an exception?

☐ Difference between trap and exception.

17 Spring 2010

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 26 March 2010, 10:40–11:30 CDT

Problem 1	_____	(40 pts)
Problem 2	_____	(12 pts)
Problem 3	_____	(14 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(24 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

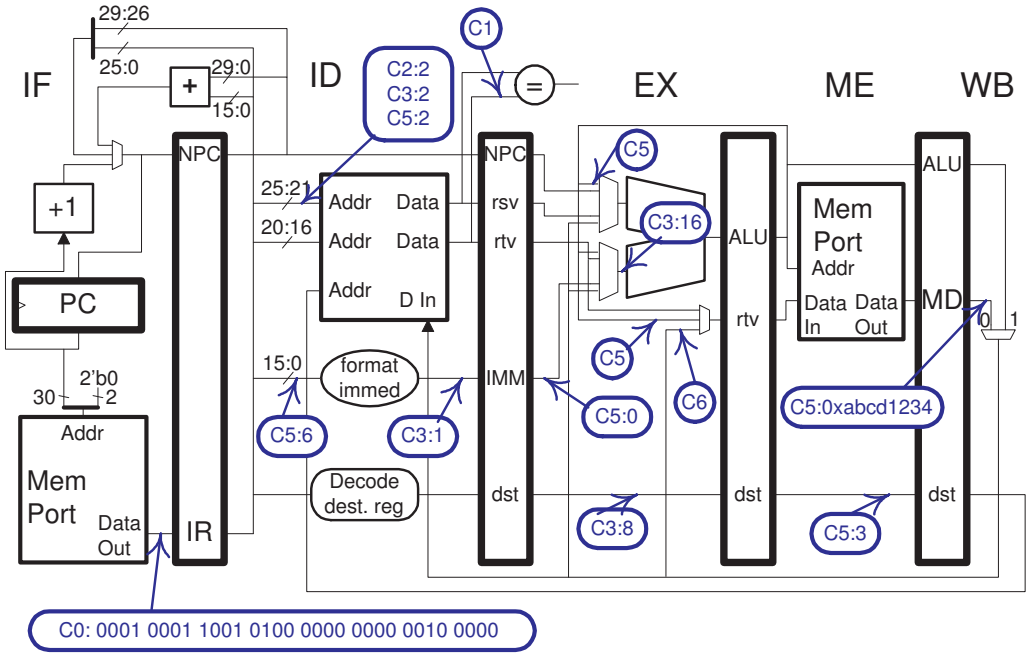
Problem 1: [40 pts] In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c5:6 indicates that at cycle 5 the wire will hold a 6. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Write a program consistent with these labels.

☐ All register numbers and immediate values can be determined.

☐ The first instruction address has been provided, **show the addresses of the remaining four instructions**.

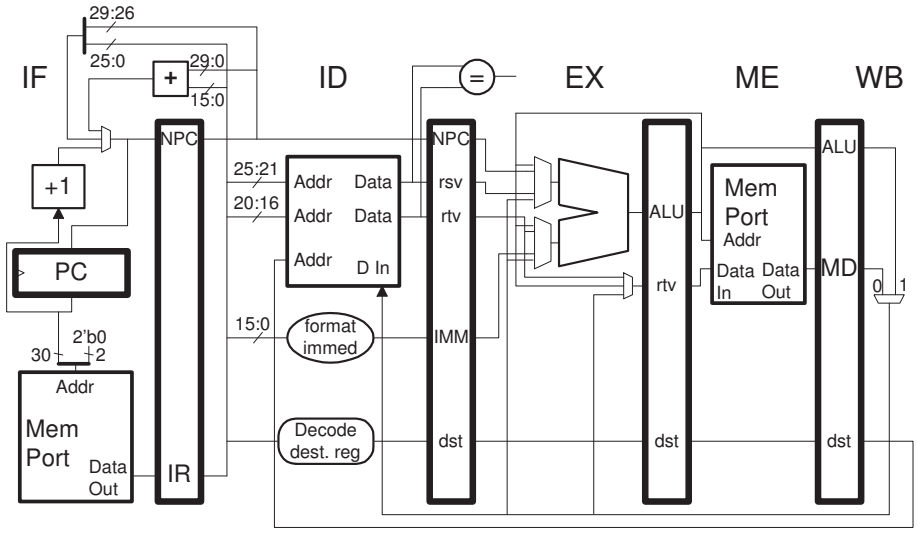
☐ The third instruction is an `addi`, don't forget to show its registers and immediates.

☐ If an instruction is a load or store, show all possible size and sign possibilities. For example, (lw,lh)



	Cycle:	0	1	2	3	4	5	6	7	8
0x1000		IF	ID	EX	ME	WB				
			IF	ID	EX	ME	WB			
addi				IF	ID	EX	ME	WB		
					IF	ID	EX	ME	WB	
						IF	ID	EX	ME	WB
	Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: [12 pts] Consider the following cost-reducing design options for the MIPS implementation shown below. Performance is still important, and a compiler will be able to optimize for this lower-cost implementation, but even with skillful optimization there may be some performance loss.



(*a*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all lower-ALU-input bypass paths. Which would you choose? *Note: The following sentence did not appear on the original exam.* (Only bypass paths are eliminated, other mux inputs remain.)

☐ Eliminate: upper or lower. (Circle one.)

☐ Briefly explain why.

(*b*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all rtv (not to the ALU) bypass paths. Which would you choose (Note: Only bypass paths are eliminated, other mux inputs remain.)

☐ Eliminate: upper or rtv. (Circle one.)

☐ Briefly explain why.

(*c*) Suppose that the mux in the WB stage was moved to the ME stage.

☐ How would that reduce cost?

☐ How might that affect performance?

Problem 3: [14 pts] The branch delay slot ISA feature eliminates the need for the stall or squash following a branch that occurs in implementations such as our 5-stage pipeline.

(*a*) The two MIPS code fragments below do not exactly make a strong case for delay slots. In both cases there is no squash or stall, which sounds good. For each code fragment indicate whether performance is slower, equal to, or faster than corresponding non-delay-slot ISA code on a corresponding implementation (one that will suffer a squash after every branch). If faster indicate if the improvement is full (based on the eliminated squash) or partial. *Hint: it won't be full.*

```
# -------------------- Code Fragment 1 --------------------
beq r1, r2, TARG
nop
lw r3, 0(r1)
```

☐ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☐ Explain.

```
# -------------------- Code Fragment 2 --------------------
beq r1, r2 SKIP
add r9, r4, r5
or r6, r9, r8
SKIP:
sub r3, r6, r5
lw r9, 0(r3)
```

☐ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☐ Explain.

(*b*) How can Code Fragment 2 (above) be improved by profiling? Show the optimized fragment and how profiling led to the optimized fragment.

☐ Optimized version of Code Fragment 2

☐ Specifically, how did profiling help?

Problem 4: [10 pts] The SPECcpu2006 integer suite has 12 benchmarks. Suppose instead it included only four benchmarks, but those benchmarks were chosen fairly, given that only four could be chosen.

(*a*) Considering just the base score (or ignoring the base/peak distinction altogether), what is the disadvantage of having just four benchmarks?

☐ Disadvantage of having four benchmarks.

(*b*) Suppose that with this smaller set of benchmarks the difference between the base and peak scores was smaller than if 12 benchmarks were used. Assume that in both cases the testers were skilled and followed the rules. Also assume that the base score values were about the same with 4 or 12 benchmarks.

☐ Give a reason for the smaller difference that has to do with the different rules for base and peak tests.

☐ Does the smaller difference indicate that base and peak are not measuring what they should?

Problem 5: Answer the questions below.

(*a*) [14 pts] Write the following MIPS code fragments with the minimum number of instructions. If you don't know the exact mnemonic, such as `mtc1` or `cvt.w.f`, make up something that sounds right.

☐ Put constant `0x12345678` into register `r1`.

☐ Jump to address `0x72345678` from address `0x1000`.

☐ Jump to address `0x72345678` from address `0x1000` using a SPARC-like `jmpl` (but in MIPS, like Homework 2).

☐ Registers `r1` and `r2` contain integers. Write register `f4` with the sum as a single-precision floating point number.

(*b*) [10 pts] Typical CISC ISAs have a range of immediate sizes for use in ALU instructions, up to the data size limit. (Say, 8, 16, 24, and 32 bits). MIPS, and other RISC ISAs, have just one immediate size for ALU instructions.

☐ Why can't MIPS have a 32-bit immediate?

☐ Why can a CISC ISA have a 32-bit immediate?

☐ Why would there be no benefit for MIPS to have both 8- and 16-bit immediate sizes for arithmetic instructions?

☐ What is the benefit for CISC ISAs in having 8-, 16-bit, (and more) immediate sizes for arithmetic instructions?

Name _____

Computer Architecture

EE 4720

Final Examination

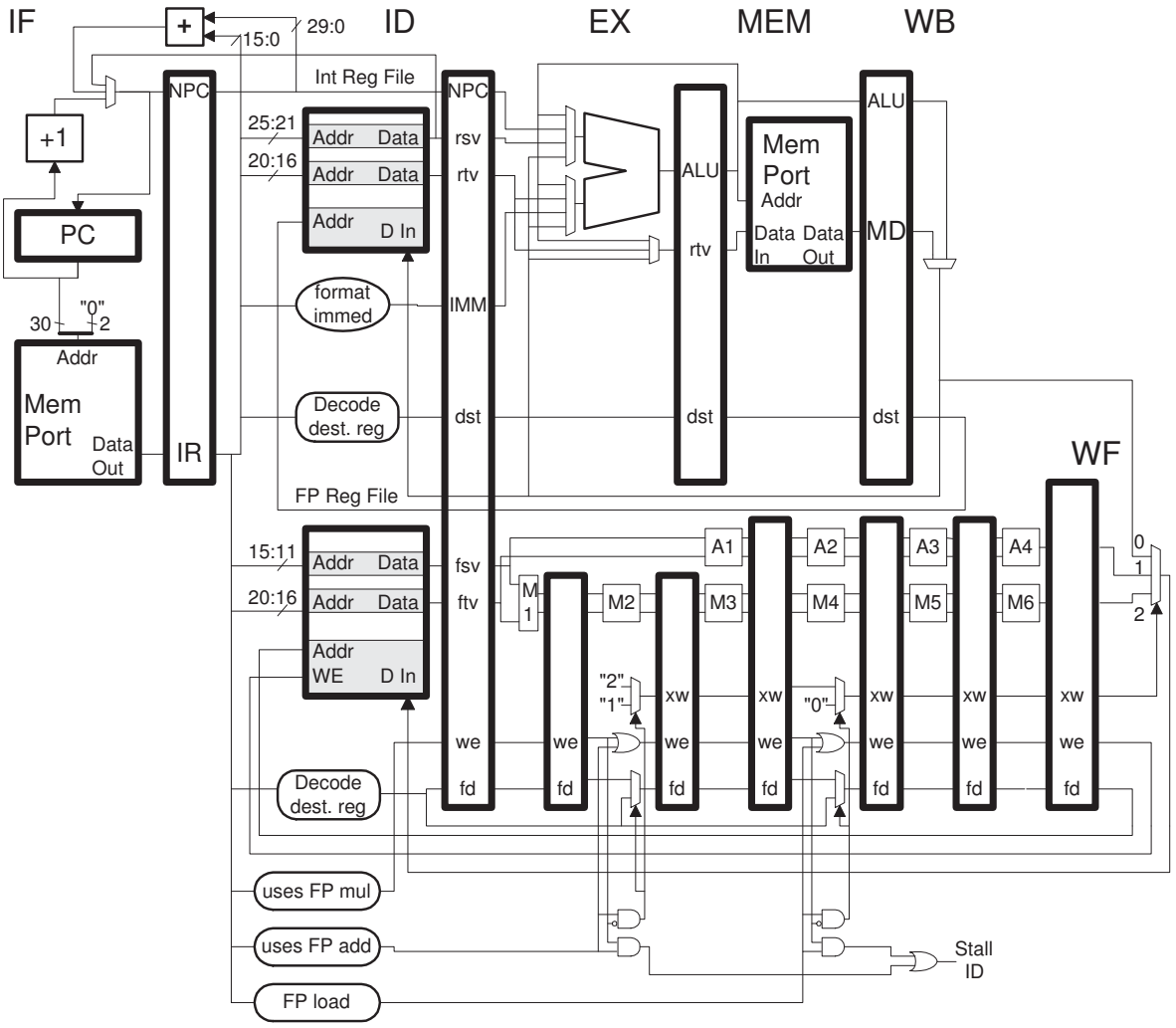
11 May 2010, 12:30–14:30 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(10 pts)
Problem 7	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (15 pts) The statically scheduled MIPS implementation including the floating-point pipeline is illustrated below.



(*a*) Consider the instruction `mtc1 f2, r4`. On the diagram above show the path taken by the data on its trip from `r4` to `f2`.

☐ Show path taken by value using a squiggly line on the diagram above.

(*b*) The control logic for the FP pipeline needs only a small change to handle `mtc1`. Make that change above. (This has nothing to do with the bypass problem below.)

☐ Control logic for `mtc1` in diagram above. (Ignore bypasses.)

(*c*) Add the hardware needed to implement `swc1`. Add only datapath, not control logic.

☐ Datapath for `swc1`.

Problem 1, continued:

```
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13
lui r1, 0x4593      IF ID EX ME WB
ori r1, r1, 0x819c     IF ID EX ME WB
mtc1 f1, r1               IF ID EX ME WF
add.s f2, f2, f1             IF ID A1 A2 A3 A4 WF
mtc1 f4, r4                     IF ID EX ME WF
sub.s f6, f4, f1                   IF ID A1 A2 A3 A4 WF
swc1 f6, 0(r5)                        IF ID -------> EX ME WF
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

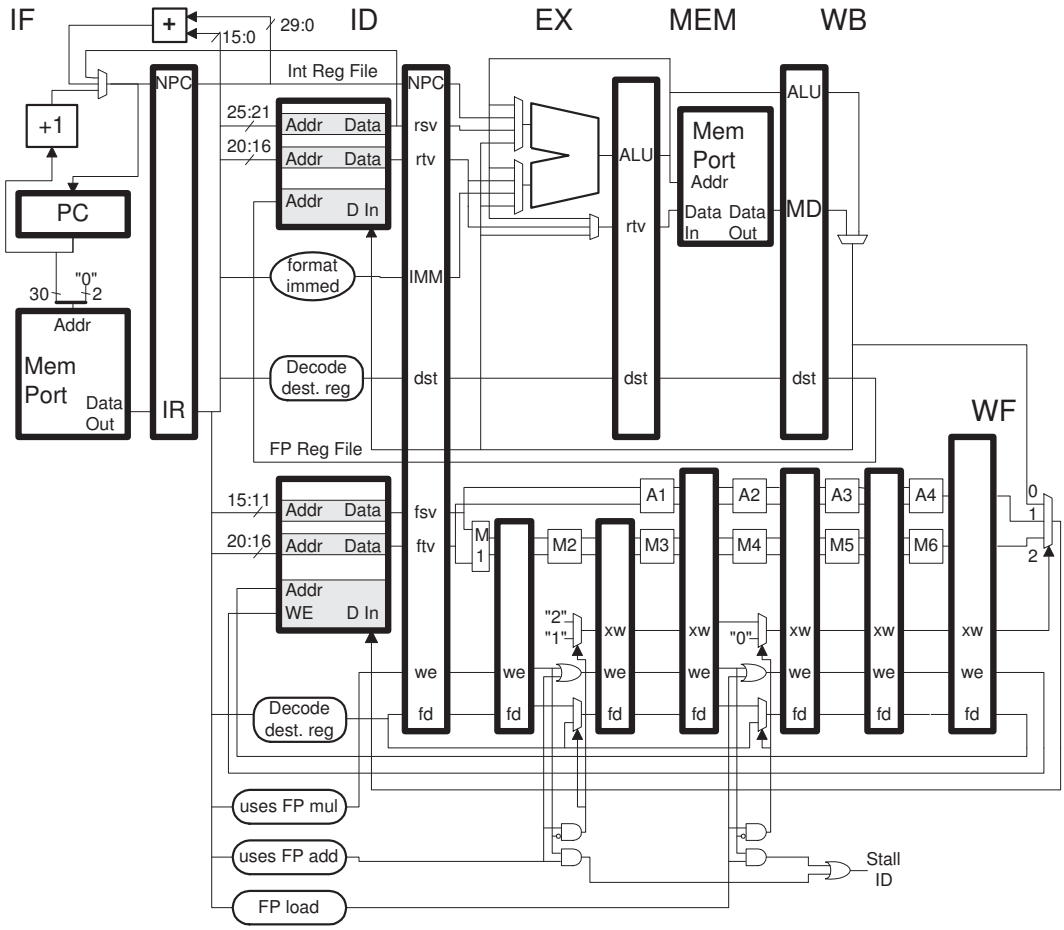
(*d*) The code fragment execution (pipeline diagram) above could not occur on the pipeline above because certain bypass paths are needed.

☐ Add those bypass paths for the code above.

☐ Show the cycle in which each added bypass path is used by the code above.

(*e*) Add control logic needed to detect the bypass used from `mtc1` to `sub.s`. The logic should deliver a signal, `BYPASS`, to the stage containing the bypass multiplexors. The `BYPASS` signal should be true if the bypass is needed.

☐ Logic generating `mtc1` to `sub.s` bypass signal.



Problem 2: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 16-outcome local history, and one system uses a global predictor with a 16-outcome global history.

Branch `B1` is random, and can be described by a Bernoulli random variable with $p = .5$. The outcome of branch `B3` will always be the same as the most recent outcome of branch `B1`. (That is, if an execution of `B1` is taken, the next execution of `B3` will be taken.) Branch `B2` has a repeating pattern, it repeats twice below.

```
B1: r   r       r           r               r   r       r           r
B2:  T   N   T   N   N   T   N   N   N   T    T   N   T   N   N   T   N   N   N   T
B3:   R       R           R               R    R       R           R               R
```

For the questions below accuracy is after warmup.

☐ What is the accuracy of the bimodal predictor on `B2`?

☐ What is the accuracy of the bimodal predictor on `B3`?

☐ What is the accuracy of the local predictor on `B2`?

☐ How small can the local history size be made without affecting the accuracy of branch `B2`? Explain.

☐ What is the accuracy of the local predictor on `B3`?

☐ What is the accuracy of the global predictor on `B3`?

☐ How small can the global history size be made without affecting the accuracy of branch `B3`? Explain.

☐ How many PHT entries are used by `B2` in the system using the local predictor?

☐ What is the warmup time of the global predictor on branch `B3`?

Problem 3: (10 pts) Suppose that in a MIPS-I system using a bimodal predictor there were BHT collisions on 5% of the predictions. (A BHT collision occurs when two branches use the same BHT entry.) A BHT entry stores both a 2-bit counter and the branch's 16-bit displacement. *Grading Note: The contents of a BHT entry was not in the original exam.*

Consider a design alternative in which a tag were used to detect BHT collisions, in the same way a cache uses a tag to detect hits (or misses).

(*a*) How large would the tag have to be to perfectly detect misses on a MIPS-I system using a 2^{14}-entry BHT?

☐ Tag size needed, reason:

(*b*) Suppose that the storage budget for the BHT was fixed at the number of bits in a 2^{14}-entry BHT without tags.

☐ About how many entries would there be in a BHT with tags?

(*c*) Based on the answer to the previous part, could we use a tagged BHT if the benefit of detecting collisions were small, moderate, or large?

☐ Benefit needs to be small, medium, or large. Explain.

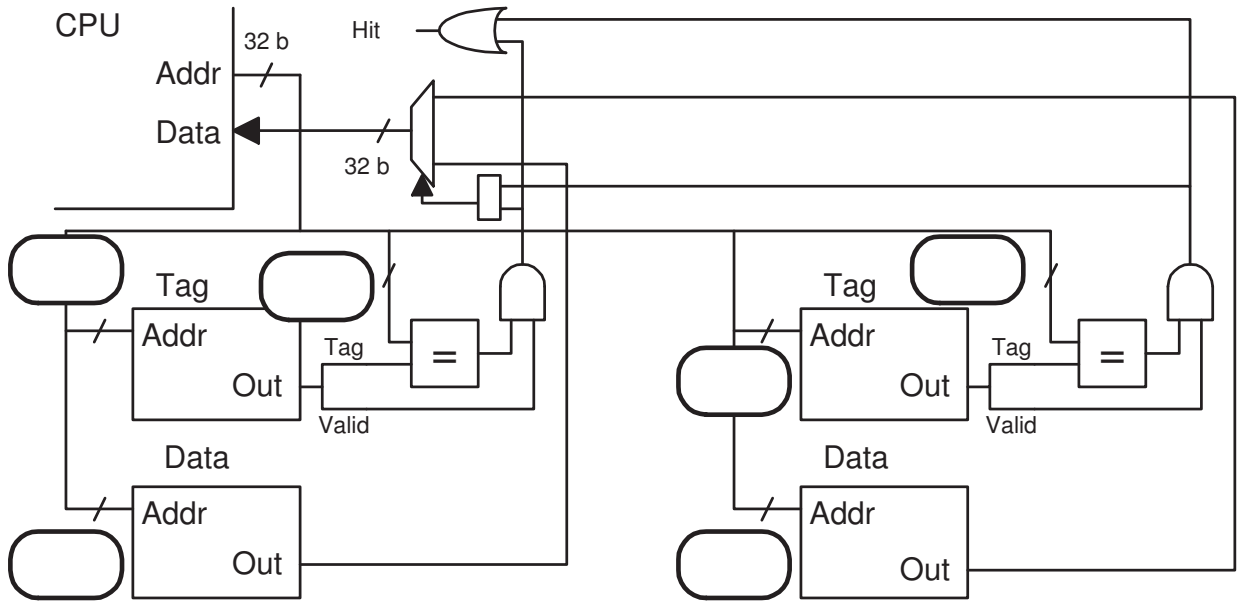
(*d*) What is the benefit of detecting BHT collisions on a four-way superscalar statically scheduled MIPS implementation?

☐ Benefit for 4-way MIPS:

Problem 4: (15 pts) The diagram below is for a set-associative cache with a line size of 64 bytes and a tag size of 11 bits. The system has the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Cache Capacity (Indicate Unit!!):

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 4, continued:

(*b*) The code below runs on a 32 MiB direct-mapped cache with a 256-byte line size. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

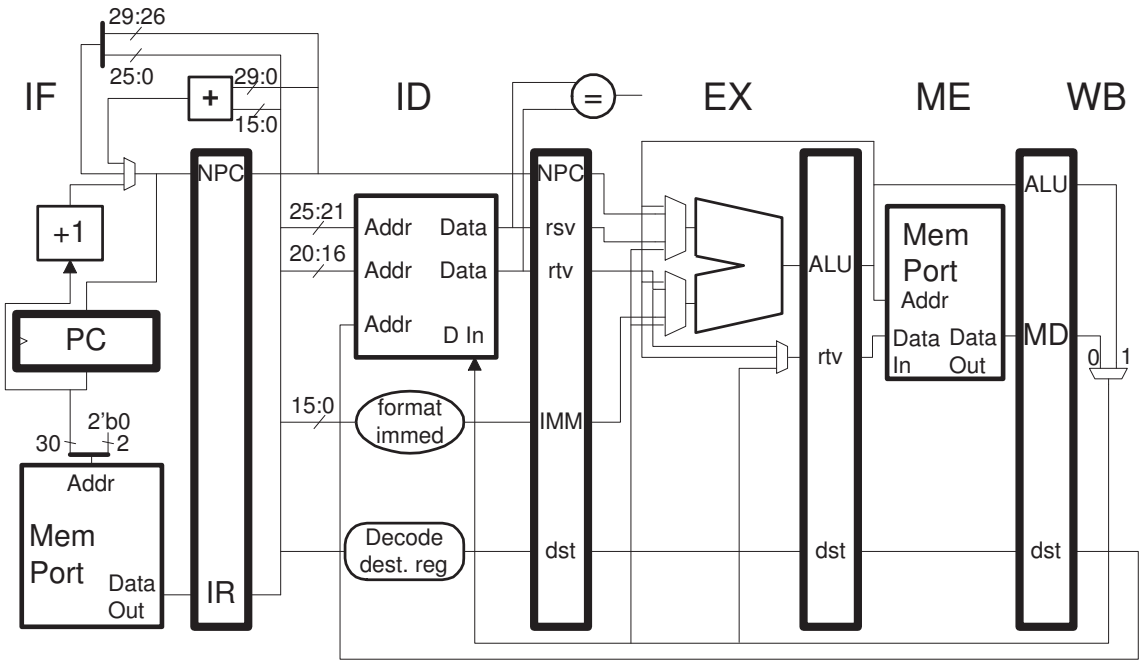
```
double sum = 0.0;
double *a = 0x2000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ 4 * i ];
```

Problem 5: (15 pts) Several possible new MIPS instructions appear below. Show how each instruction can be encoded and show datapath changes needed to implement the instruction.

- The changes cannot break existing instructions.
- The changes can not have a large impact on clock frequency.
- A shift unit is present, but not shown.

Note: The original exam only asked for a summary of datapath changes, and did not mention the shift unit or clock frequency.

Also indicate the relative difficulty of implementing the instruction. If an instruction is deemed moderate or difficult indicate the most important reasons why.



(*a*) The `sllii` instruction is like an `lui` except it can shift the immediate by any amount. For example, `sllii r1, 0x1234,16` would be equivalent to an `lui r1, 0x1234`.

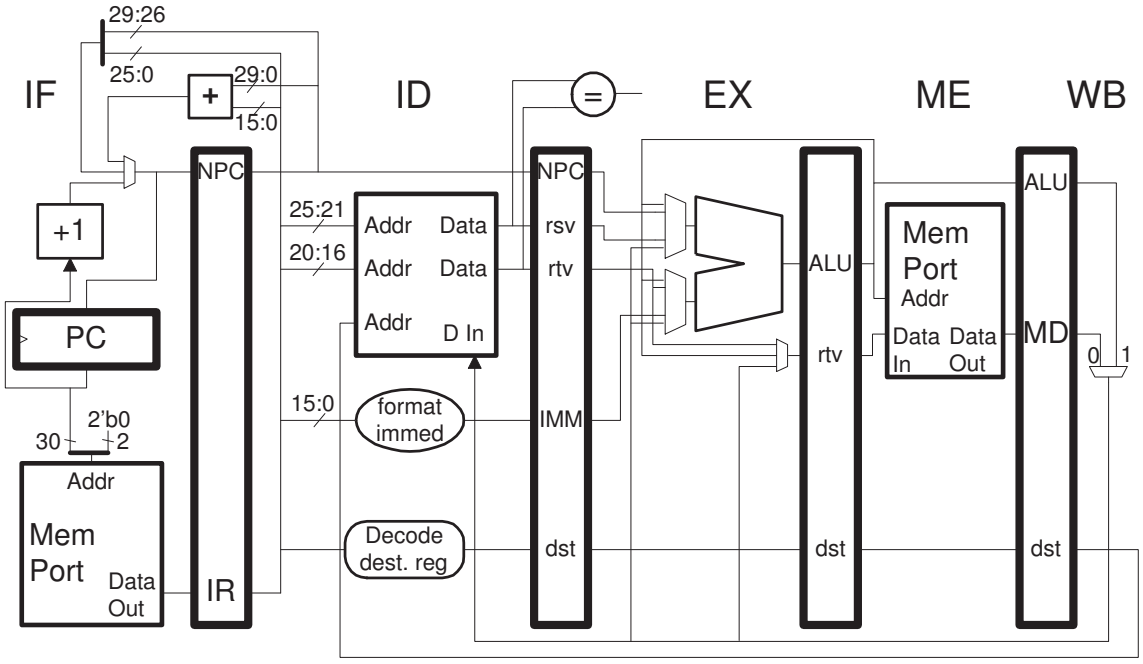
```
sllii r1, 0x1234, 6   ! r1 = 0x1234 << 6
```

☐ Show possible encoding (instruction format (R, I, etc) and field (rs, rt, etc) usage):

☐ Easy, moderate, or difficult to implement:

☐ Show datapath changes.

Problem 5, continued:



(*b*) The `adds` (add scaled) can shift the second operand left by any amount.

```
adds r1, r2, r3, 4   ! r1 = r2 + ( r3 << 4 )
```

☐ Show possible encoding:

☐ Easy, moderate, or difficult to implement:

☐ Show datapath changes.

(*c*) The `addsid` instruction produces a sum like `add` but one operand is obtained from memory.

```
addsid r1, r2, (r3)   ! r1 = r2 + Mem[r3]
```

☐ Show possible encoding:

☐ Easy, moderate, or difficult to implement:

☐ Show datapath changes.

Problem 6: (10 pts) Consider a single four-way superscalar implementation and a chip with four scalar implementations (like our five-stage MIPS). Both are statically scheduled and have similar features.

☐ Why might the clock frequency of the superscalar system be lower than the scalar systems?

☐ Why might the chip area (or cost) of the four scalar systems be less than the superscalar system?

☐ How does the average program run less efficiently on the superscalar system than on one of the scalar systems? (Less efficiently means more stalls and squashes.)

☐ Given the answers above, why does a typical chip have two four-way superscalar implementations rather than eight scalar implementations? (Please do not confuse *eight scalar implementations* with *one eight-way superscalar system.*)

Problem 7: (15 pts) Answer each question below.

(*a*) One use for exceptions is to implement rare or specialized instructions in software (called *emulation*). One example is the SPARC quad-precision arithmetic instructions, such as faddq f4, f8, f12 (floating-point add quad). No existing SPARC implementation can execute these instructions in hardware.

☐ Why would `faddq` have to raise a precise exception in order to be emulated?

(*b*) Consider an implementation similar to our pipelined MIPS in which a floating-point overflow on a `fmuld` did not raise a precise exception (because the hardware to do so would not be worth the trouble).

☐ Does that mean it would not be possible for the `faddq` to raise the precise exception needed for emulation? Explain.

(*c*) Answer the following questions about the SPECcpu benchmarks.

☐ Why are branches in the FP suite easier to predict than branches in the integer suite?

☐ Why is it important that the source code is available for the SPECcpu benchmarks?

(*d*) Answer each ISA question below:

☐ Describe a feature of VLIW ISAs that distinguishes them from RISC and CISC ISAs.

☐ Describe a feature of CISC that distinguishes it from RISC.

18 Spring 2009

Name _____

Computer Architecture
EE 4720
Midterm Examination

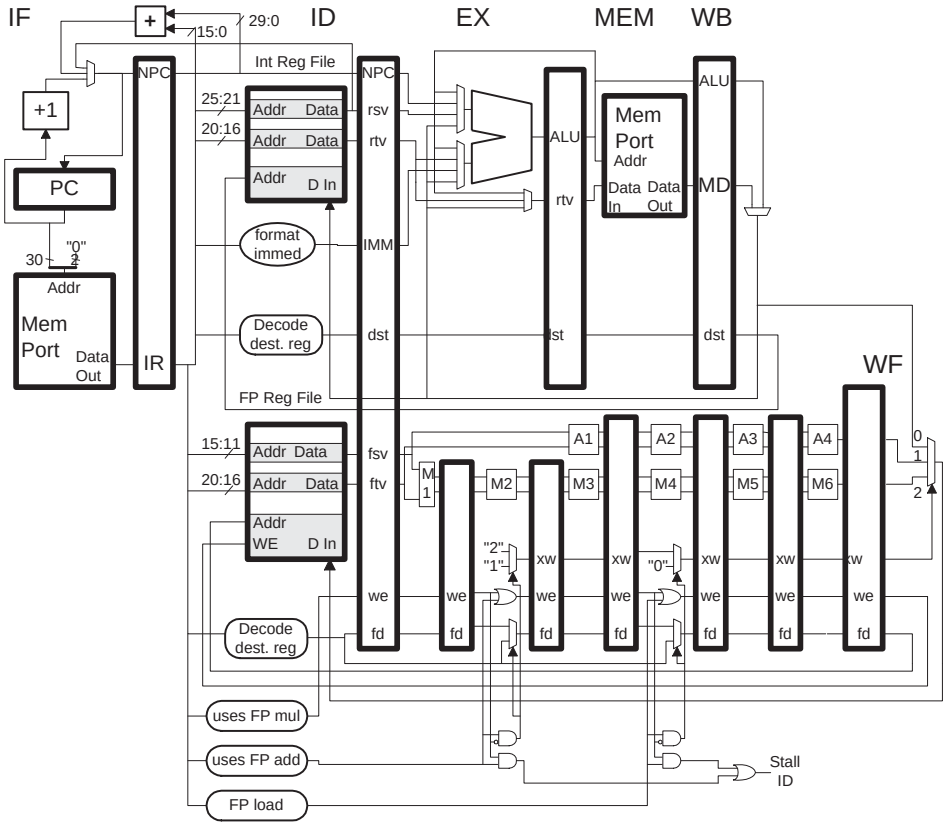
Friday, 27 March 2009, 11:40–12:30 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] The code below executes on the illustrated FP pipeline.




```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

(*a*) Add reasonable bypass paths needed by the code above, for both the integer and FP pipelines.

☐ Add reasonable bypass paths. Don't add unneeded bypass paths.

(*b*) Analyze the performance of the code using your bypasses:

☐ Show a PED for the code above using your bypasses. (Use this page or next page.)

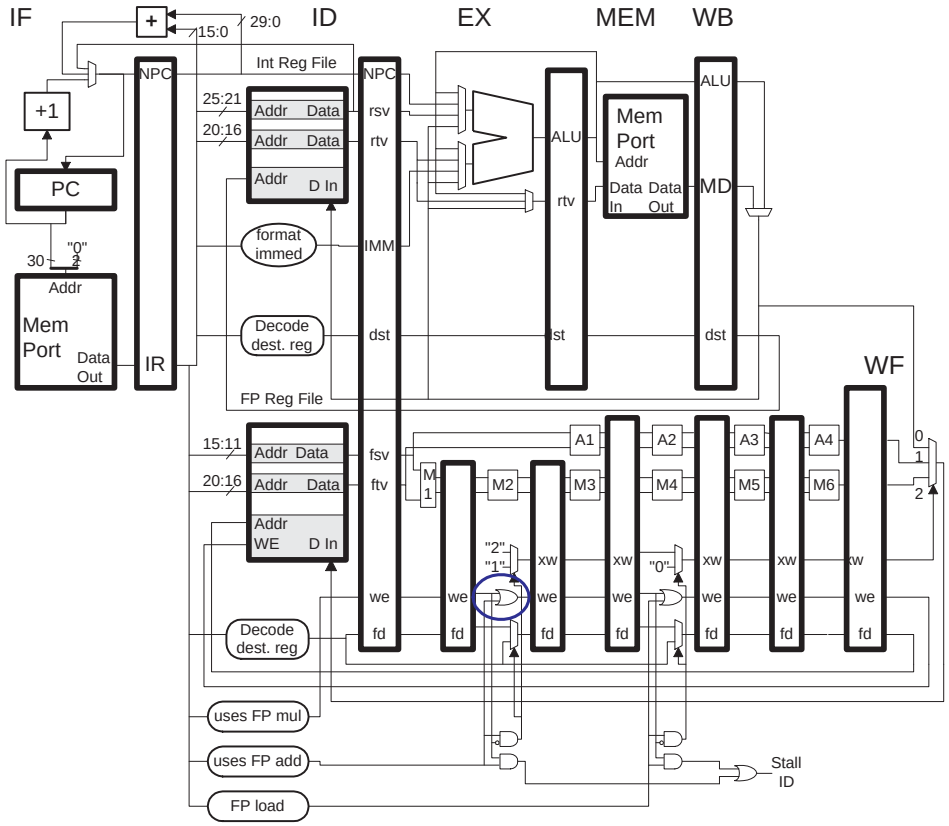
☐ Compute the CPI of the code for a large number of iterations.

Use this page for PED, if needed.

```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

Problem 1, continued:

(*c*) A component failure in the MIPS implementation below has changed the circled OR gate into an AND gate.



☐ Is it still possible to perform a FP multiply? If yes, show how with PED, if no explain.

```
# Original Code. (For answer show modified code with PED.)
mul.d f0, f2, f4
```

☐ Is it still possible to perform a FP add? If yes, show how with PED, if no explain.

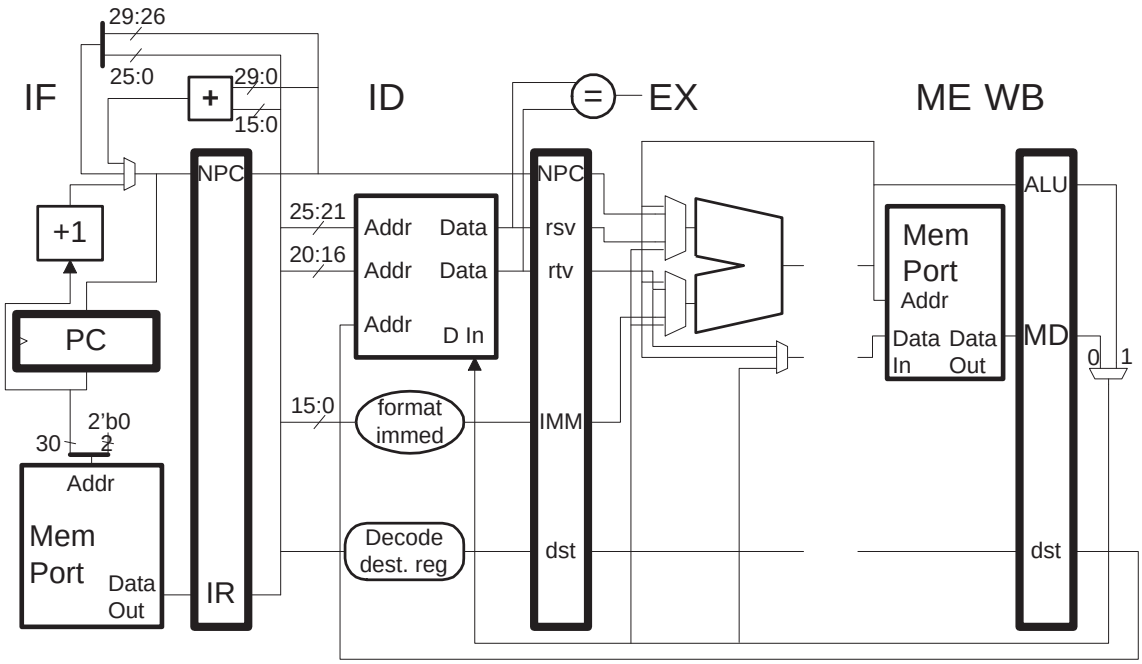
```
# Original Code. (For answer show modified code with PED.)
add.d f0, f2, f4
```

Problem 2: [30 pts] The MIPS-A ISA is like MIPS-I except that load and store instructions use only the `rs` register value for an address, they don't add an offset (or anything else). As a result MIPS-A can be implemented with a four-stage pipeline.

(*a*) Our familiar 5-stage MIPS-I implementation appears below with one of the pipeline latches missing. Make additional changes so that this is a reasonable four-stage implementation of MIPS-A.

☐ Show all connections to the memory port.

☐ Cross out wires and other items that are not needed, add what is needed.



(*b*) Describe two ways in which this MIPS-A implementation costs less than the five-stage MIPS-I.

☐ Two reasons for lower cost.

Problem 2, continued:

(*c*) The four-stage MIPS-A implementation may be faster or slower than the five-stage MIPS-I.

☐ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is faster. *Hint: The MIPS-I code will have a familiar stall.*

☐ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is slower. *Hint: Think about the difference in the ISAs.*

(*d*) A company needs to decide whether to develop MIPS-I or MIPS-A. How does it decide? Assume that at this point software compatibility is not an issue.

☐ How should company decide which to develop?

☐ Will having skilled compiler writers tilt the decision towards MIPS-A or MIPS-I? Explain.

Problem 3: [10 pts] Answer the questions below.

(*a*) Why does MIPS have a `beq` but does not have a `blt` (branch less than), even though a `blt` instruction would be frequently used?

☐ Why `beq` but no `blt`?

(*b*) Explain why a branch delay slot can be though of as a short-sited feature in an ISA.

☐ Delay slots are good when...

☐ ...but delay slots are bad when...

Problem 4: [20 pts] The doing-it-the-hard-way MIPS code below loads the constant $\frac{1}{3}$ in IEEE 754 single-precision format, `0x3eaaaaab`, into register `f2`.

```
addi r10, r0, 1
addi r20, r0, 3
mtc1 f12, r10
mtc1 f22, r20
cvt.s.w f14, f12
cvt.s.w f24, f22
div.s f2, f14, f24
```

(*a*) Describe what the `mtc1` and `cvt.s.w` instructions do.

☐ Explain `mtc1`

☐ Explain `cvt.s.w`

(*b*) The code above uses more instructions than are necessary and also uses a wastefully time-consuming instruction.

☐ What is the wastefully time-consuming instruction?

☐ Re-write the code so it uses fewer instructions without using load instructions. *Hint: A correct answer uses three instructions and a piece of information slipped into the first sentence of the problem.*

Problem 5: [10 pts] Answer the following SPECcpu questions.

(*a*) In the SPECcpu2000 suite profiling was allowed for both base and peak results but in SPECcpu2006 profiling is allowed for peak results but not for base results.

☐ Why isn't profiling allowed for base results in SPECcpu2006? Your answer should say something about the difference between base and peak.

(*b*) Company A has a reputation for reliable compilers, company B has a reputation for buggy compilers. Both companies and their customers are okay with this. (Think Italian sports cars.)

Optimization X results in a substantial improvement in SPECcpu base scores. Company B has it in their shipping compilers (those sold to customers) but company A only has optimization X in their experimental compilers (not available to customers), but they are working hard on it. The optimization achieves the same results for company A and B. Company B's compiler will not run on company A's system (as though they'd want to!). *Grading Note: The last line was not in the original exam, but a student did see the possibility of using B's compiler for A's spec run.*

☐ Can Company A use optimization X to prepare SPECcpu2006? Explain.

☐ Can Company B use optimization X to prepare SPECcpu2006? Explain.

☐ Your answers above should reflect the letter of SPEC's rules. Do you think this achieves SPECcpu's goals or what you would like to see in benchmark results? Explain.

Name _____

Computer Architecture
EE 4720
Final Examination

7 May 2009, 17:30–19:30 CDT

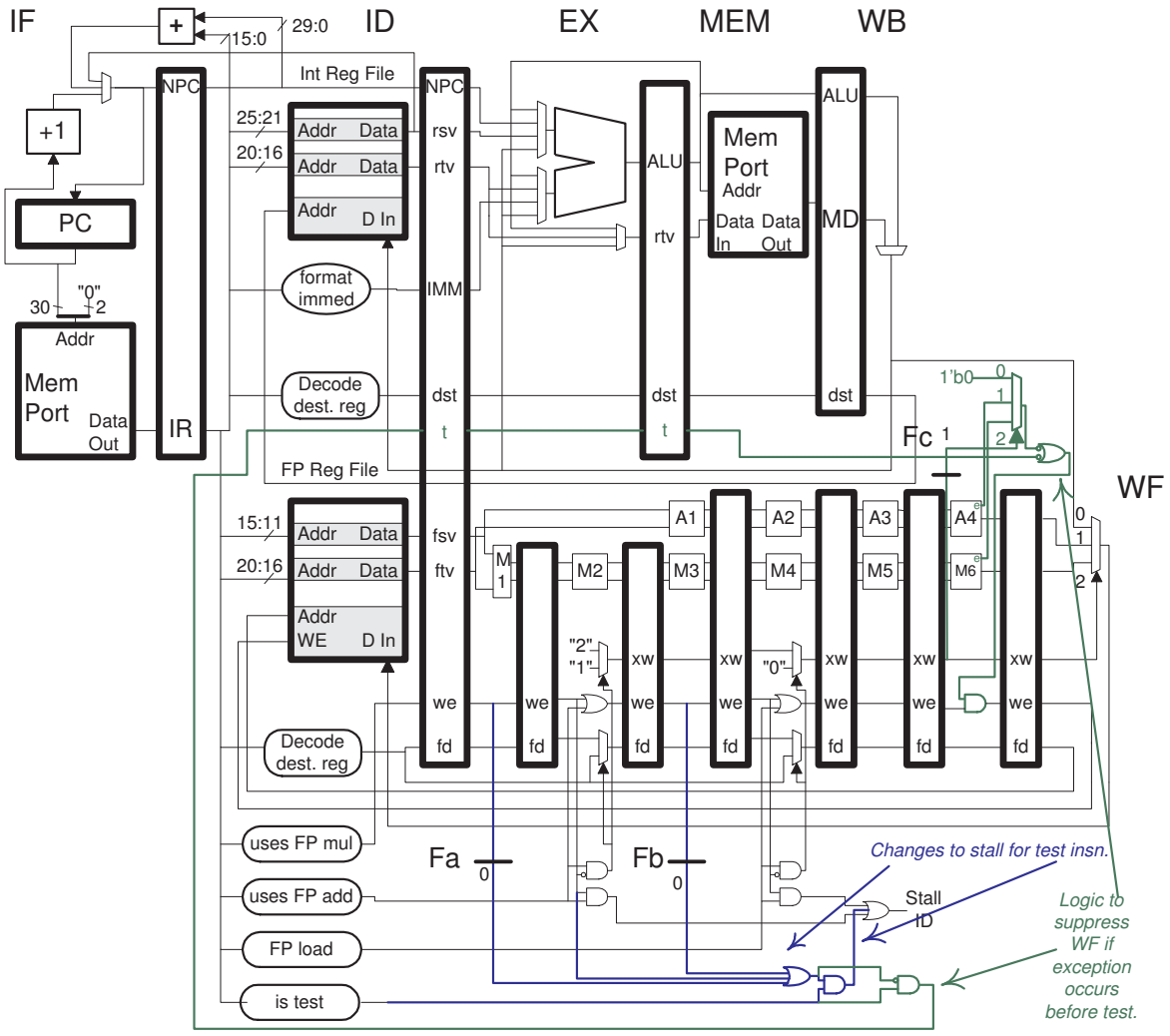
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (20 pts) The MIPS implementation below, taken from the solution to last semester's final, includes hardware to implement an exception test instruction.

Several wires on the diagram are broken with heavy solid lines and marked with `Fx` and a value (mostly `0`). These indicate *potential* fault locations. If there is no fault the wire acts normally, if there is a fault the wire is broken at the heavy line and the free half takes on the indicated value. For example, if fault `Fa` is present the bottom input to the OR gate is always zero however the `ID/EX.we` signal on the other side is unaffected.



The problem here is to detect which fault, if any, is present by running test programs. One test program, and a pipeline diagram, appears below. A handler has been set up that will set a `goodException` variable to 1 if register and memory values are as expected, otherwise it is set to -1. The `goodException` variable is initialized to 0 before each test.

Problem 1, continued:

```
# Test 1: The mul should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 * f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
mul.s f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6x
test                  IF ID -------> EX MEx
sw r1,0(r2)              IF -------> ID EXx
```

(*a*) When a test is run the exception handler is called (because the tests intentionally raise an exception). A handler is shown below, written in C, but the handler does not set `goodException` correctly (not even close). Modify the code so that `goodException` is set correctly based on the information provided by the Test 1 code and comments. That is, `goodException` is set to -1 if the exception could not be precise.

```
int handler_fp(Regs *regs){
 // Code below wrong, but shows how to read registers and memory.
 if ( regs->f10 == regs->f12 && is_nan(f14) && MemW(regs->r31) == 0x1234 )
  goodException = 1; else goodException = -1;
```

(*b*) Suppose Test 1 is run and `goodException` is set to -1 (it would be 1 in the no-fault case). Which of the faults (Fa, Fb, or Fc) could have been responsible for the `goodException` value?

☐ Faults that are definitely present. Explain.

☐ Faults that could be present. Explain.

Problem 1, continued:

(*c*) Develop tests to determine for certain whether each of the faults is present. Test 1 and each of your tests will be run, and based on the `goodException` values from each test one can say for certain which faults are present.

Assume that at most one of the faults is present. Your tests should look similar to Test 1.

☐ Tests (Code like Test 1)

☐ Which combination of `goodException` values conclude `Fa` for certain.

☐ Which combination of `goodException` values conclude `Fb` for certain.

☐ Which combination of `goodException` values conclude `Fc` for certain.

Problem 2: (20 pts) Answer each question below. **Be sure to check each code fragment carefully for dependencies.**

(*a*) The loop below runs on a statically scheduled 4-way superscalar MIPS implementation.

☐ Show a pipeline execution diagram.

```
LOOP:
addi r2, r2, 8

lw r1, 0(r2)

add r3, r1, r4

bneq r2, r5 LOOP

sw r3, 4(r2)
```

☐ Determine the IPC for a large number of iterations and assuming no cache misses.

☐ Comment on the difference between the IPC and the potential IPC of the processor.

☐ Schedule the code to improve execution time.

Problem 2, continued:

(*b*) The code below (same as the previous problem) executes on a 4-way superscalar dynamically scheduled machine. Assume that branch prediction on this machine is perfect. Load instructions use the `EA` and `ME` stages, branch instructions use the `B` stage, store instruction only use `EA` (they write memory when they commit).

Though the machine is 4-way, assume an unlimited number of `WB`, `EX`, and `RR` stages.

☐ Show a pipeline execution diagram for two iterations.

```
LOOP:
addi r2, r2, 8

lw r1, 0(r2)

add r3, r1, r4

bneq r2, r5 LOOP

sw r3, 4(r2)
```

☐ Determine the IPC for a large number of iterations.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system has a bimodal predictor, one system uses a local history predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history.

The code has three branches, `B1`, `B2`, and `B3`. The outcome of `B1` is random, described by a Bernoulli random variable with $p = .5$ and is independent of everything. Branch `B2` has a simple j-iteration loop pattern ($j-1$ T's and an N) and branch `B3` is a repeating pattern of k N's followed by k T's (see diagram below). Also from the diagram notice that `B1` occurs just before `B2` but that a set of j `B1` and `B2s` occur between each `B3`, (similar to a branch in the homework and last semester's final).

```
             A                    A                    A
     ------------------   ------------------   ------------------  ...
B1:  R  R  R  ...  R      R  R  R  ...  R      R  R  R  ...  R
B2:   T  T  T ...T   N     T  T  T ...T   N     T  T  T ...T   N
      !<---  j    --->!

B1&2:   A   A   A       A
B3:       N   N   N  ...  N   T   T   T  ...  T   N   N   N ...  N   T...
          !<---  k Ns  --->!  !<--- k Ts  ---->!  !<--- k Ns  --->!  ...
```

For the questions below accuracy is after warmup.

☐ What is the accuracy of the bimodal on `B1`?

☐ What is the accuracy of the bimodal on `B2` in terms of j?

☐ What is the accuracy of the bimodal on `B3` in terms of k?

☐ What is the accuracy of the local predictor on `B3` when $k \gg 10$ and $j < 5$ in terms of j and k?

☐ What is the accuracy of the local predictor on `B3` when $k \gg 10$ and $j > 10$ in terms of j and k?

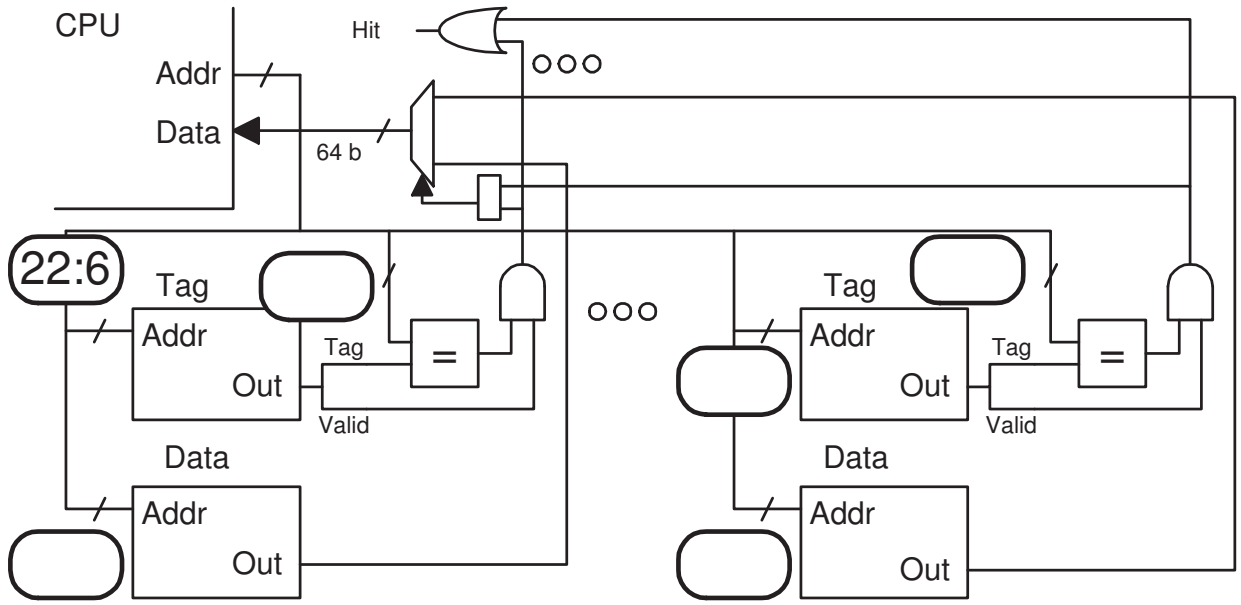
☐ What is the accuracy of the global predictor on `B3` when $j = 10$ and k is very, very large, in terms of j and k?

☐ What is the warmup time for `B3` on the global predictor. (This warmup occurs whenever `B3` switches from Ns to Ts or Ts to Ns.) in terms of j and k?

Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25}-character) set-associative cache with the usual 8-bit characters and a 64-bit address space.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 4, continued:

(*b*) The code below runs on the cache from the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) The code below runs on a direct-mapped cache unrelated to the one above. When the code starts running the cache is cold, for the solution only count accesses to `array`.

```
struct My_Struct {
  double val;
  double something;
  double relative;
  double more_data[29];
};  // Total size: 32 * sizeof(double) = 256 bytes

const int SIZE = 1 << 12;
My_Struct array[SIZE];  // &array[0] = 0x1000000

void tri()
{
  double sum = 0;
  for ( int i=0; i<SIZE; i++ ) sum += array[i].val;
  const double avg = sum / SIZE;
  for ( int i=0; i<SIZE; i++ ) array[i].relative += array[i].val - avg;
}
```

☐ Determine the minimum cache and line size needed so that there are no misses in the second `for` loop.

☐ Explain your answer.

Problem 5: (20 pts) Answer each question below.

(*a*) In a dynamically scheduled machine one would like to be able to have a large number of instructions in flight to, say, find something to do while waiting for data from memory. Which part of the dynamically scheduled machine is most difficult to scale up to support a large number of in-flight instructions?

☐ Part that's difficult to scale, and reason why.

(*b*) How might profiling improve the performance of the following C code:

```
if ( a > b ) { x = d / e; } else { y = q / f; }
```

☐ Explain how profiling is used.

☐ Explain why this profiled code might be faster than code compiled without profile feedback.

(*c*) In the first implementation of a company's ISA the integer multiply instruction was slow. An Engineer working on the second implementation is deciding whether to speed up the multiply instruction. To decide he plans to analyze some benchmark runs, but he can't decide whether the code should be optimized. The compiler was designed for the first implementation.

☐ What is the disadvantage of counting multiply usage in code compiled with optimization?

☐ What is the disadvantage of counting multiply usage in code compiled without optimization?

(*d*) A memory system that can fetch 2^w chars of data is less expensive and faster if the data address is a multiple of 2^w. (That's one reason for the alignment restriction.)

☐ How do VLIW ISAs take advantage of that?

(*e*) Compared to a RISC implementation, say the 5-stage MIPS, what additional logic does a CISC implementation require between the IF-stage memory port output and decode?

☐ Additional logic for CISC.

19 Fall 2008

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 31 October 2008, 10:40–11:30 CDT

Problem 1	_____	(50 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

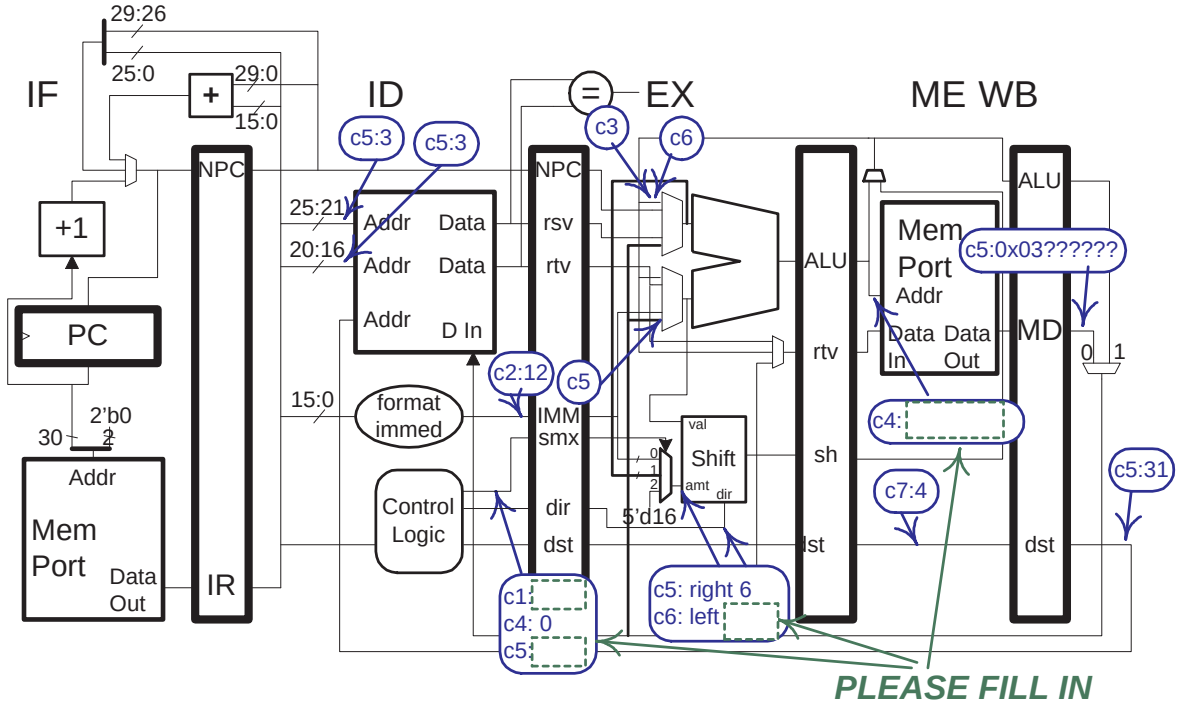
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c5:3 indicates that at cycle 5 the wire will hold a 3. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Instruction addresses and the first instruction have been provided. [50 pts]

(*a*) Finish a program consistent with these labels.

☐ All register numbers and immediate values can be determined.

☐ Be sure to fill the **three** blocks marked *PLEASE FILL IN*.



	Cycle: 0	1	2	3	4	5	6	7	8
0x0b0000 lui r2, 0xb	IF	ID	EX	ME	WB				
0x0b0004		IF	ID	EX	ME	WB			
0x0b0008			IF	ID	EX	ME	WB		
0x0b000c				IF	ID	EX	ME	WB	
0xedd900					IF	ID	EX	ME	WB

Problem 1, continued: Continue referring to the implementation on the previous page.

(*b*) Explain whether each instruction below could be the instruction at address `0xb0008` (on the previous page). If it could be the instruction write "Possible" otherwise write "Impossible because ..." (The grade will be based on the reason.)

☐ `add`

☐ `j`

☐ `jal`

☐ `beq`

Problem 2: [10 pts] Based on the experience of preparing SPECcpu benchmark runs manufacturers A and B each release a new compiler that improves their respective SPECcpu scores.

(a) In the course of preparing a SPECcpu benchmark run manufacturer A discovers a new optimization technique that improves the performance of most of the SPECcpu programs, and other programs. This optimization technique is added to the compiler for use at the `-O1` and higher optimization levels. The manufacturer sells the compiler, proudly boasting about the performance benefits.

☐ Is it in the spirit of the SPECcpu rules to use this new optimization technique for the base results? Explain.

(b) To prepare a SPECcpu benchmark run manufacturer B has its best programmers prepare hand-written assembly code for the most time consuming portion of each benchmark. The compiler will recognize each benchmark based on the source code and substitute the hand-written routines where needed; the hand-written code will only work for these benchmarks.

These optimizations are included in manufacturer B's compiler and used at `-O1` and higher levels. Manufacturer B sells this new compiler.

☐ Is it in the spirit of the SPECcpu rules to use this new optimization technique for the peak results? Explain.

☐ Explain why it is not in the spirit of SPECcpu rules to use such optimizations for base results.

☐ Assuming that the compiler can not be reverse engineered (there is no way to inspect the compiler itself) and assuming that manufacturer B can keep secrets, how might this cheating be discovered?

Problem 3: Answer the following ISA questions.

(*a*) [10 pts] A RISC advocate claims that by having fixed-length instructions and alignment restrictions a branch can reach twice as far for a given displacement field size than would be possible in CISC ISAs.

☐ Explain why.

A CISC advocate responds that branches in CISC programs would take less space anyway.

☐ Provide a reason for small-displacement branches.

☐ Provide a reason for large-displacement branches.

(*b*) [10 pts] Indicate whether each item below is usually an ISA feature or an implementation feature.

☐ Number of bits in immediate.

☐ Clock frequency.

☐ Number of branch delay slots (if any).

☐ Floating-point format.

☐ Minimum distance between load instruction and a dependent instruction to avoid a stall.

Problem 4: Equivalent MIPS and SPARC code fragments appear below.

(*a*) [10 pts] Taking advantage of SPARC's condition code features, modify the SPARC code to use one fewer instruction (without changing what it does).

```
# MIPS Branch Example
 addi $t1, $t1, -1
 bne  $t1, 0  LOOP
 add  $t2, $t2, $t3
...

! Equivalent SPARC Branch Example
 add l1, -1, l1      ! l1 = l1 - 1
 subcc l1, 0, g0     ! g0 = l1 - 0
 bne LOOP
 add l2, l3, l2
...
```

(*b*) [10 pts] For branch-in-ID implementations, why might it be possible to attain higher clock frequencies for condition-code ISAs, like SPARC, than for register-test branche ISAs, like MIPS.

☐ Condition code performance advantage.

Name _____

Computer Architecture
EE 4720
Final Examination

9 December 2008, 17:30–19:30 CST

Problem 1	_____	(10 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(20 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

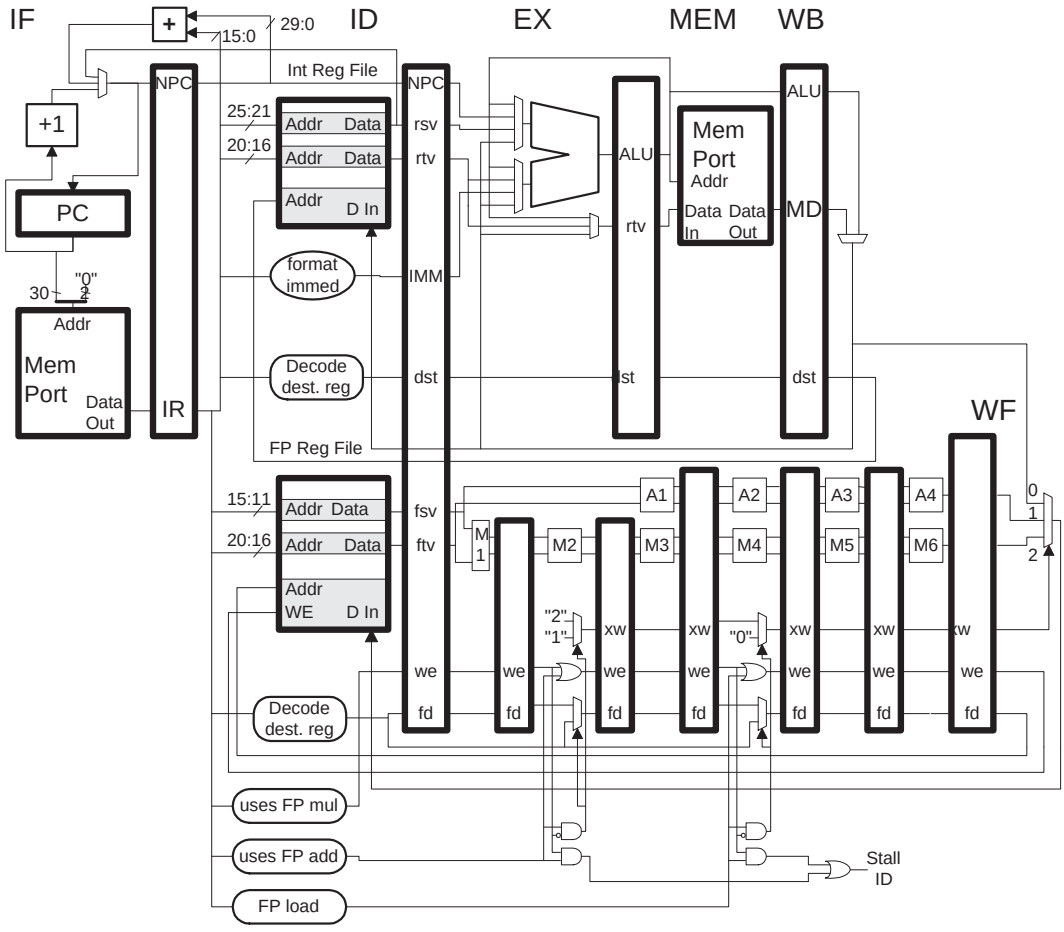
Good Luck!

Problem 1: (10 pts) Consider a new floating point instruction `nin.d` which uses a 5-stage computation unit, N1 - N5 (in contrast to FP add's A1-A4). The format and register usage for `nin.d` is the same as the other FP arithmetic instructions. Modify the implementation below so that it can execute `nin.d`. *Hint: This is an easy question,* `nin.d` *is just like the FP add and multiply, except it's 5 stages.*

☐ Add the datapath components, including the functional unit stages (N1 to N5).

☐ Add control logic for proper write-float and structural hazard detection (as is already present for add and multiply). Be sure not to break existing instructions.

☐ The changes must fit in efficiently with what is already present.



Problem 2: (15 pts) With the MIPS implementation illustrated below floating-point add and multiply instructions do not raise precise exceptions. If a programmer needs a precise exception for a particular FP instruction he or she can follow it with a `test` (test for exception) instruction. In the code below `test` is used to provide a precise exception for `mul.d`.

```
mul.d  f2, f2, f6
test
sub.d  f16, f14, f20
```

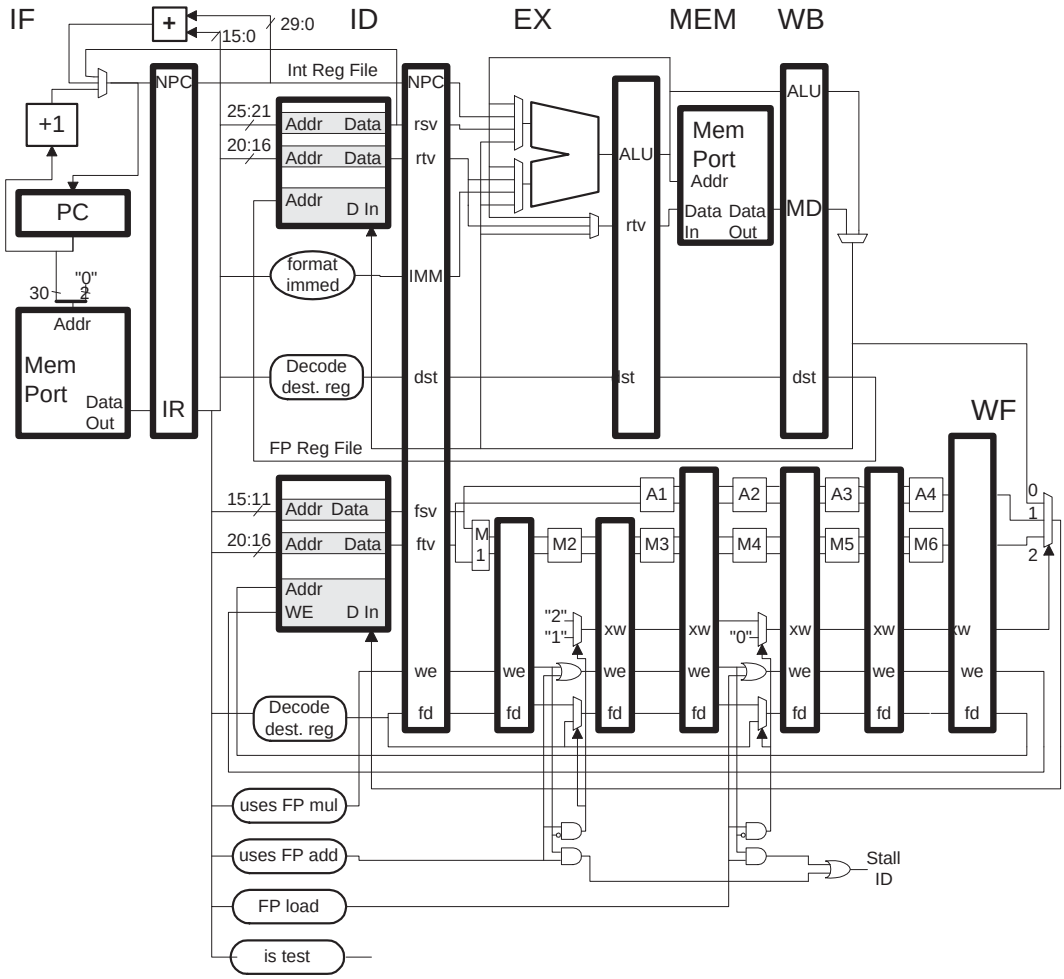
The `test` instruction will stall in ID for **the minimum number of cycles necessary** to ensure a precise exception and will suppress a WF if necessary.

(*a*) Add hardware to implement the `test` instruction. The output of is test is `1` if a `test` instruction is in `ID`. Assume there are A4 and M6 outputs that indicate whether an exception was raised. These can be checked in the middle of the cycle.

☐ Add control logic to stall the pipeline using a new input to the *Stall ID* or gate. *Hint: Very little logic is needed.*

☐ Add logic to suppress WF when necessary.

☐ The hardware should work correctly for floating-point multiplies and adds.



Problem 2, continued:

(*b*) Show an example in which, in a faulty implementation, the test instruction stalls one less than the minimum number of cycles and as a result an exception is not precise.

☐ Example code and pipeline execution diagram, including fetch of first handler instruction.

☐ Explain why the exception is not precise.

Problem 3: (20 pts) The code below has two branches, branch `B1` implements a 3-iteration loop with outcome pattern `T T N T T N`$\cdots$. The outcome pattern for branch `B2` is shown below. **Note that `B1` executes three times for each execution of `B2`.**

The code runs on three systems, the branch predictor on all systems uses a 2^{14}-entry BHT. One system has a bimodal predictor, one system uses a local history predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history.

```
BIGLOOP:    Note: B1 and B2 are the only branches in this code.
T1:
B1:  bne r3, r4 T1   .. 3 iteration loop ..

# .. no CTIs ..
B2:  beq r1, r2 T2   N N N T N N T N N N T N N T N N N T N N T N N N T N N T ...
T2:

j BIGLOOP
```

☐ What is the accuracy of the bimodal predictor on branch `B2` after warmup?

☐ What is the accuracy of the local predictor on branch `B2` after warmup?

☐ What is the size of the BHT and PHT, in bits, needed to implement the local predictor? Only take into account the storage need for the branch predictor, omit things like CTI type.

☐ What is the minimum local history size needed to achieve 100% accuracy on branch `B2` using the local predictor? Explain.

☐ What is the minimum global history size needed to achieve 100% accuracy on branch `B2` using the global predictor? Explain.

Problem 4: (15 pts) Consider two alternatives for improving the performance of our familiar scalar 5-stage implementation: an n-way superscalar implementation or a $5n$-stage superpipelined implementation.

(a) Both systems can *potentially* reduce execution time by a factor of n. Explain how this is achieved for each system in terms of CPI (cycles per instructions), clock frequency, and other relevant factors. The answers might read, "Because of ... the CPI is x^2 times larger which causes ... but ... **and so overall execution is n times faster**."

☐ Explain how the superscalar system is potentially n times faster.

☐ Explain how the superpipelined system is potentially n times faster.

(b) Bypass paths add substantially to the cost of sufficiently large superscalar systems. Provide expressions in terms of n for the cost of bypass paths in both systems. Briefly justify your answers, using a diagram if necessary.

☐ Expression for the cost of bypass paths in superscalar systems, with quick diagram and brief description.

☐ Expression for the cost of bypass paths in superpipelined systems, with quick diagram and brief description.

(c) Ideally the $5n$-stage superpipelined system would have a speedup of n (the scalar system would take n times longer than the superpipelined system to run a program). Consider a program that has no nearby dependencies so that the superpipelined system does not have to stall.

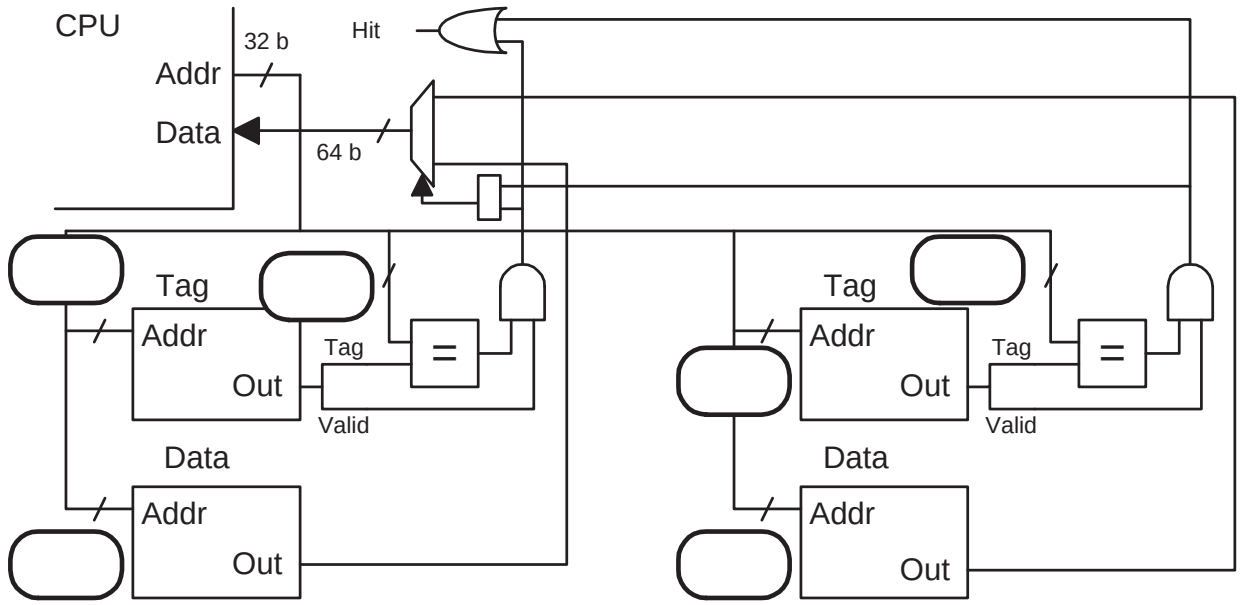
☐ Explain why the speedup of the superpipelined system might be $\frac{t_0+t_1}{t_0+\frac{t_1}{n}}$, where the clock frequency of our familiar scalar system is $\frac{1}{t_0+t_1}$.

☐ Explain what t_0 is likely to be.

Problem 5: (20 pts) The diagram below is for a 4-MiB (2^{22}-character) set-associative cache with a line size of 512 (2^9) characters, with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 5, continued:

(*b*) The code below runs on the cache from the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) The code below runs on a direct mapped (not set-associative) cache with a line size of 512 characters and a capacity of 4 MiB. *Grading note: The cache size was omitted from the original problem.*

☐ Determine an address for `b` that will result in a 0% hit ratio when running the code below.

☐ Briefly explain.

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.
short *b =                                          <-- FILL IN
int i;
int ILIMIT = 1 << 10;    // = 2^10

for (i=0; i<ILIMIT; i++) sum += a[ i ] + b[ i ];
```

Problem 6: (20 pts) Answer each question below.

(*a*) Describe restrictions (or lack of) on instruction addresses and the placement of instructions that distinguish RISC, CISC, and VLIW ISAs.

☐ RISC address and placement restrictions.

☐ Reason for each restriction (if any).

☐ CISC address and placement restrictions.

☐ Reason for each restriction (if any).

☐ VLIW address and placement restrictions.

☐ Reason for each restriction (if any).

(*b*) A feature of the SPECcpu benchmarks is that they come with **source code** and it is the tester's responsibility to **compile** (and run) them. Note: "are these" in the questions below refers to source code and compiling.

☐ Are these necessary, important, or irrelevant for testing an implementation of a new ISA? (This is not a yes-or-no question.) Explain.

☐ Are these necessary, important, or irrelevant for testing a new implementation of an existing ISA? Explain.

(*c*) A corrupt member of the committee choosing benchmarks for the next SPECcpu suite has successfully bribed the other committee members so that the benchmarks that were selected favor the corrupt member's company.

☐ How might this be discovered? Indicate who would discover the problem and what public information draw their suspicion?

☐ Can we expect those involved to be sufficiently motivated to correct the situation? Explain.

20 Spring 2008

Name _____

Computer Architecture
EE 4720
Midterm Examination

Monday, 10 March 2008, 12:40–13:30 CDT

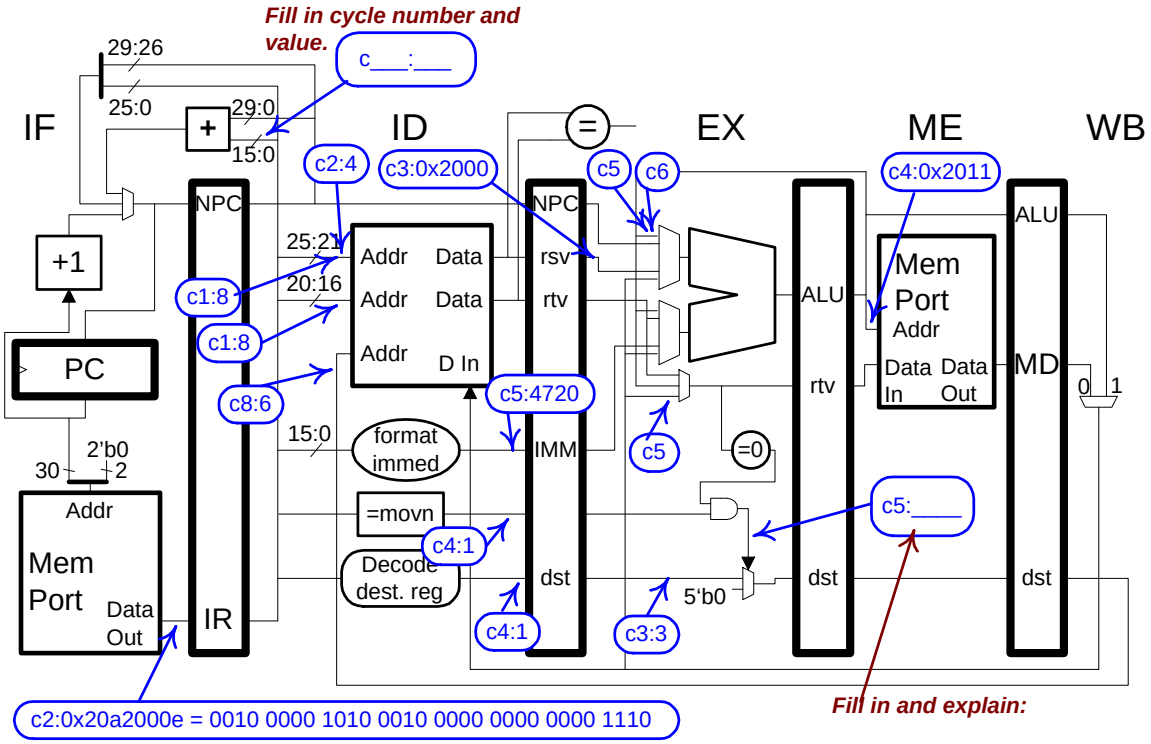
Problem 1	_____	(50 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c2:4 indicates that at cycle 2 the wire will hold a 4. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that instruction addresses have been provided. [50 pts]

- [ ] Finish a program consistent with these labels.
- [ ] All register numbers and immediate values can be determined.
- [ ] Be sure to fill the two blocks marked *Fill In*.
- [ ] Provide an explanation for the EX-stage fill-in block.



Cycle:	0	1	2	3	4	5	6	7	8
0x1000	IF	ID	EX	ME	WB				
0x1004		IF	ID	EX	ME	WB			
0x1034			IF	ID	EX	ME	WB		
0x1038				IF	ID	EX	ME	WB	
0x103c					IF	ID	EX	ME	WB
Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: Answer the following questions about, or inspired by, ARM.

(*a*) [10 pts] MIPS lacks a counterpart to the ARM instruction shown below. *Hint: This has nothing to do with MIPS'* `movn`.

```
mov r1, #5   // Move the constant 5 to register r1.
```

☐ Explain how `r0` makes such a MIPS instruction unnecessary.

☐ Show how to perform the same operation using MIPS instruction(s).

(*b*) [5 pts] The ARM ISA states that the result of executing an instruction like `str r15, [r1]` is that either `PC+8` or `PC+12` is stored in memory, depending on the implementation. (Remember that ARM `r15` is an alias for `PC`.)

☐ What is the benefit of making the result of the store implementation dependent?

(*c*) [5 pts] Consider an ISA which stated that the number of branch delay slots could be either zero or one, depending on the implementation.

☐ As an ISA feature, how does this delay-slot implementation dependence compare in practicality to ARM's store `PC` implementation dependent behavior?

Problem 3: In RISC ISAs instructions fixed size, that is, all instructions are the same size. For example, in MIPS, all instructions are 32 bits. The character size in MIPS (and all ISAs mentioned in this test) is 8 bits.

(*a*) [5 pts] Describe a benefit of having fixed-size instructions. (This answer can be mentioned in the next answer's explanation.)

☐ Fixed-size instruction benefit for RISC.

(*b*) [5 pts] In the Itanium VLIW ISA all instructions are 41 bits. Why would 41-bit instructions be difficult or wasteful to implement in a RISC ISA, such as MIPS, but cause no difficulties in VLIW ISA implementations, including Itanium implementations.

☐ Forty-one bit RISC instructions difficult or wasteful because:

☐ Forty-one bit VLIW instructions not wasteful and make sense because:

Problem 4: Answer the following compiler questions.

(*a*) [10 pts] A company is considering removing a bypass connection in a design of an implementation. Analysis on good test programs shows that performance drops by 5% with the bypass removed (and no other changes).

☐ What can compiler writers do about the performance drop?

(*b*) [10 pts] Dead-code elimination is a commonly used compiler optimization, while profiling is used less often because it is a multi-step process.

☐ Using an example briefly describe dead-code elimination.

☐ Briefly describe the steps used in profiling.

☐ Which is more likely to result in disappointing results that surprise the programmer? Explain.

Name _____

Computer Architecture
EE 4720
Final Examination

7 May 2008, 10:00–12:00 CDT

Problem 1	_____	(10 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (10 pts) The tables below show the state of a dynamically scheduled system using method 3 during the execution of a code fragment.

- The code is preceded by many nop instructions.
- Instructions use either a 4-stage FP multiply unit (M1-M4) or a 2-stage FP add unit (A1, A2).
- Assume that there are unlimited RR, WF and execute (A,M) units.

(*a*) Show a program and pipeline execution diagram consistent with these tables.

☐ Show program and all stages used.

☐ On the physical register file show where physical registers are removed from the free list, using a `[`, and where they are put back in the free list, using a `]`.

☐ All destination registers can be determined exactly.

☐ The time for all stages can be determined exactly.

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	IF	ID													
		IF	ID												
			IF	ID											
				IF	ID										

ID Register Map

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f15 9			39												
f20 16		43		82	93										

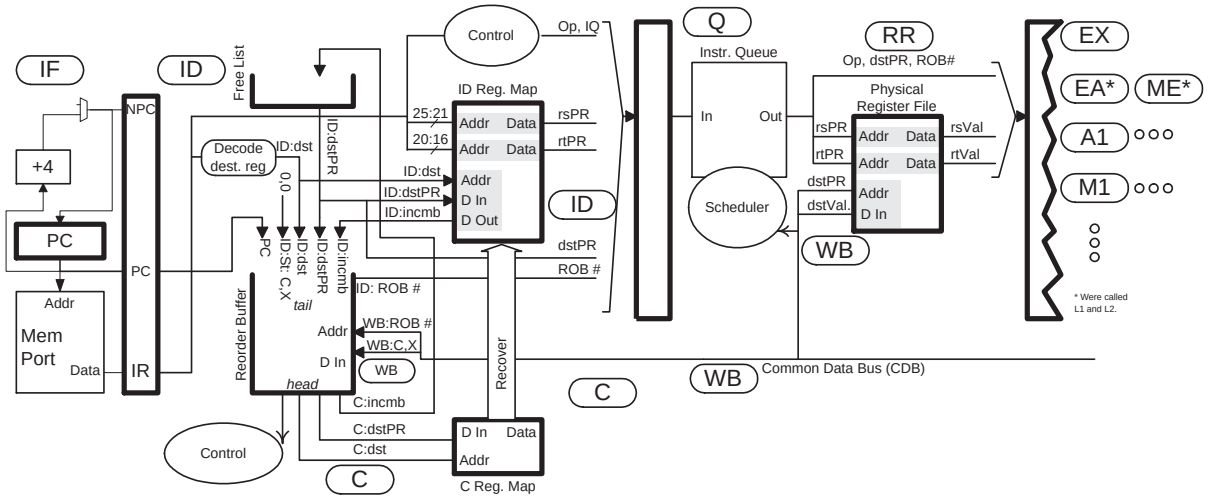
Commit Register Map

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f15 9												39			
f20 16								43					82	93	

Physical Register File

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	11														
16	22														
39											33				
43							44								
82									55						
93											66				
# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Problem 1, continued: The implementation diagram is provided below for references. Don't forget to answer the question below.



(*b*) In the code fragment on the previous page only two true dependencies are possible.

☐ Show them (by connecting the destination and source with an arrow).

☐ Briefly explain why other dependencies are not possible.

Problem 2: (30 pts) Answer the following branch predictor questions.

(*a*) The code below runs on two branch predictors, a bimodal predictor with a 2^{10} entry BHT and a local predictor with a 2^{10} entry BHT and a 9-outcome history.

The outcomes of branch B1 form a repeating pattern as shown below.

The outcome pattern for branch B2 can be constructed by tossing a fair coin, adding a `N N N` for heads or a `T T T` for tails, then repeating. More precisely, outcome $3i$ is a Bernoulli random variable with $p = .5$ and outcomes $3i + 1$ and $3i + 2$ match outcome $3i$, for $i = 0, 1, 2, \ldots$. For example, the following is a possible pattern of outcomes for B2: `N N N T T T T T T`. The following is **not possible** for B2: `N N N T T N T T T`, it's not possible because the number of consecutive N's or T's must be a multiple of 3.

```
LOOP:

B1:      N  N  N  N  T  T  T  N  N  N  N  T  T  T  N  N  N  N  T  T  T...

B2:       N  N  N  T  T  T  T  T  T  N  N  N  T  T  T  N  N  N  (See text.)

j LOOP
nop
```

For the questions below assume all tables are initially zero. All accuracies are after warmup.

☐ What is the accuracy of the bimodal predictor on branch B1?

☐ What is the accuracy of the local predictor on B1?

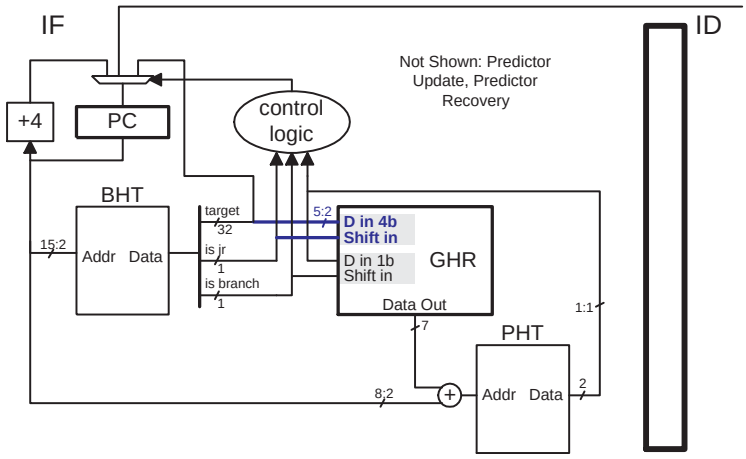
☐ What is the minimum local history size needed to predict B1 with 100% accuracy ignoring branch B2. Briefly explain.

☐ What is the approximate accuracy of the bimodal predictor on branch B2? Explain.

☐ What is the approximate accuracy of the local predictor on branch B2 ignoring B1? Explain.

☐ What is the approximate warmup time for the local predictor on B2? *Note: This problem was alot more difficult to get perfectly correct than originally intended.*

(*b*) The predictor to the right is from the solution to last semester's final exam (and this semester's Homework 4). It is similar to gshare except in how the GHR is updated. Like global and gshare, when a branch is predicted the predicted outcome is shifted into the GHR, but when a `jr` is predicted four bits of the target are shifted into the GHR.



The code fragment below is the one used to justify the predictor, except that now the code is in a four-iteration loop. Important assembler code is shown in the comments.

```
for ( iter=0; iter<4; iter++ ) {
  int c = getchar(); // Unpredictable
  switch (c) {                    // jr $t0  # Jump to correct case.
  case 'a': x = 3; j++; break;    // addi $t1, $0, 3;  j ENDSWITCH;  addi $t2, $t2, 1
  case 'b': x = 7; break;
    ...
  case 'z': x = 1; i++; break;
  }                               // ENDSWITCH:
  if ( x < 5 ) foo(); else bar();
}
```

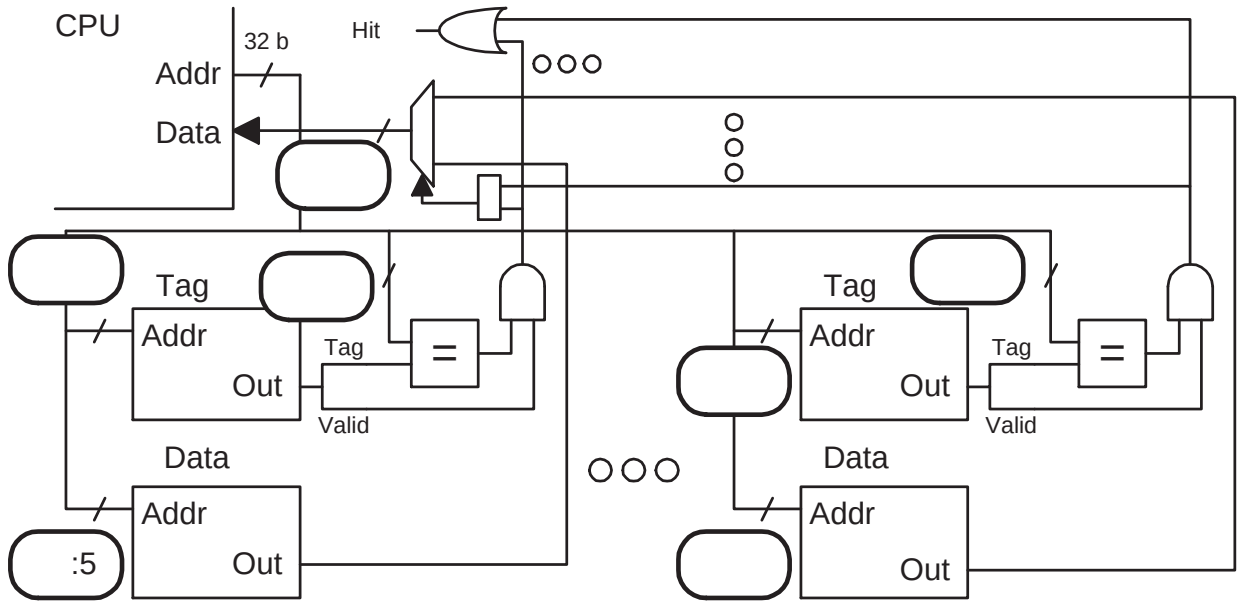
☐ Why might the prediction accuracy of the `for` loop branch be worse with the predictor above than an ordinary gshare?

☐ Considering the assembler code above, why might it make more sense to shift in the PC of the jump (`j`) instructions rather than the target of the `jr`? *Hint: This would only be useful when the case statements had branches.*

Problem 3: (20 pts) The diagram below is for a 32-MiB (2^{25}-character) 4-way set-associative cache with a line size of 4096 (2^{12}) characters, with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: | | | |

Problem 3, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
long *a = 0x20000000;    // sizeof(long) = 8 characters.
int i,j;
int ILIMIT = 1 << 10;    // = 2^10

for (j=0; j<2; j++)
  for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) The code below runs on a fully associative cache with 2^7-byte lines, **not the same as the previous cache**. Let h denote the hit ratio of the code below for a cache size of 8 MiB (2^{23} bytes). As always, assume the cache is empty before the code starts.

```
void p4(double *a, double *b) {
  // sizeof double = 8 characters.
  double sum = 0;
  int size = 1 << 8;

  for ( int d=0; d<size; d++ )
    for ( int col=0; col<size; col++ )
      sum +=  b[ col * size + d ]  *  a[ d * size + col ];
}
```

☐ What is the minimum cache size for which the hit ratio is h? Explain.

Problem 4: (15 pts) Answer each question below.

(*a*) Describe what's wrong with each execution below.

☐ What's wrong with this execution on our usual bypassed 5-stage MIPS implementation.

```
# Cycle        0  1  2  3  4  5
lw r1, 0(r2)   IF ID EX ME WB
add r3, r1, r4    IF ID EX ME WB
```

☐ What are the two problems below with this execution on our usual scalar statically scheduled MIPS implementation?

```
add.d f0, f1, f2  IF ID A1 A2 A3 A4 ME WF
```

☐ What's wrong with this execution on a 2-way superscalar dynamically scheduled machine?

```
# Cycle       0  1  2  3  4  5  6  7  8  9
add r1,r2,r3  IF ID Q  RR EX WB C
add r4,r1,r6  IF ID Q     RR EX WB C
add r7,r8,r9     IF ID Q  RR EX WB    C
add r8,r2,r1     IF ID Q     RR EX WB    C
# Cycle       0  1  2  3  4  5  6  7  8  9
```

(*b*) Alas, it is unlikely that there will soon be 16-way, statically scheduled, superscalar implementations that anyone would want to buy. Three reasons are started below, complete them.

☐ It would be too expensive because ...

☐ The clock frequency would be too low because ...

☐ A few programs might realize the full potential of the processor, but most won't because ...

Problem 5: (15 pts) Answer each question below.

(*a*) In ARM the PC is a general purpose register, `r15`. It could have been defined in ways consistent with ISA design principles, but it wasn't.

☐ Describe how the use of `r15` appears contrary to the usual goals of an ISA.

(*b*) The two code fragments below are supposed to do the same thing, the first is MIPS the second is SPARC. In both a branch is taken if a less-than comparison is true. The code would work correctly if the called subroutine returned immediately.

☐ Explain why the SPARC code may not execute as intended.

☐ Suggest a fix.

```
# MIPS Code
slt $s1, $s2, $s3       # Registers %s0-%s7 preserved.
jal some_subroutine
nop
bneq $s1, $0 SKIP1
...

! SPARC code
subcc %l2, %l3, %l1     ! Registers %l0-%l7 preserved.
call some_subroutine
nop
blt SKIP1
...
```

Hint: The following two questions are really asking about exceptions.

(*c*) A compiler writer is wondering whether dead-code elimination should remove the first assignment to x in the code fragment below (which sure looks dead, right?).

```
x = a / b;
x = a + c;
```

The guy down the hall wrote a program that included these two lines and that program would run differently if dead-code elimination removed the first assignment (the one with the division). The difference has nothing to do with timing or the size of the program.

☐ Why might the program have run differently?

☐ The compiler writer wants to DTRT (do the right thing), how does he or she find out what the right thing is?

(*d*) The code fragment below may have come from the code in the previous problem.

```
div.d f0, f2, f4
add.d f0, f2, f6
```

Designers of an implementation are considering whether to avoid WAW hazards like the one above by converting the `div.d` to a nop when the `add.d` reaches `ID`. This will work correctly under ordinary circumstances.

☐ Show a pipeline execution diagram illustrating the WAW hazard.

☐ Under what circumstances will this produce the wrong answer?

Problem 6: (10 pts) SPECcpu provides a separate set of training inputs to be used for feedback-directed optimization (FDO) techniques, such as profiling.

☐ Why is a separate training input set needed?

☐ Why might an honest and good tester want to substitute a different training set?

The run and reporting rules don't allow such a substitution.

☐ Why do you think the run and reporting rules don't allow a training set substitution when they allow so much flexibility on compilers, optimizations, and libraries?

21 Fall 2007

Name _____

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 8 November 2007, 10:40–11:30 CST

Problem 1	_____	(50 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

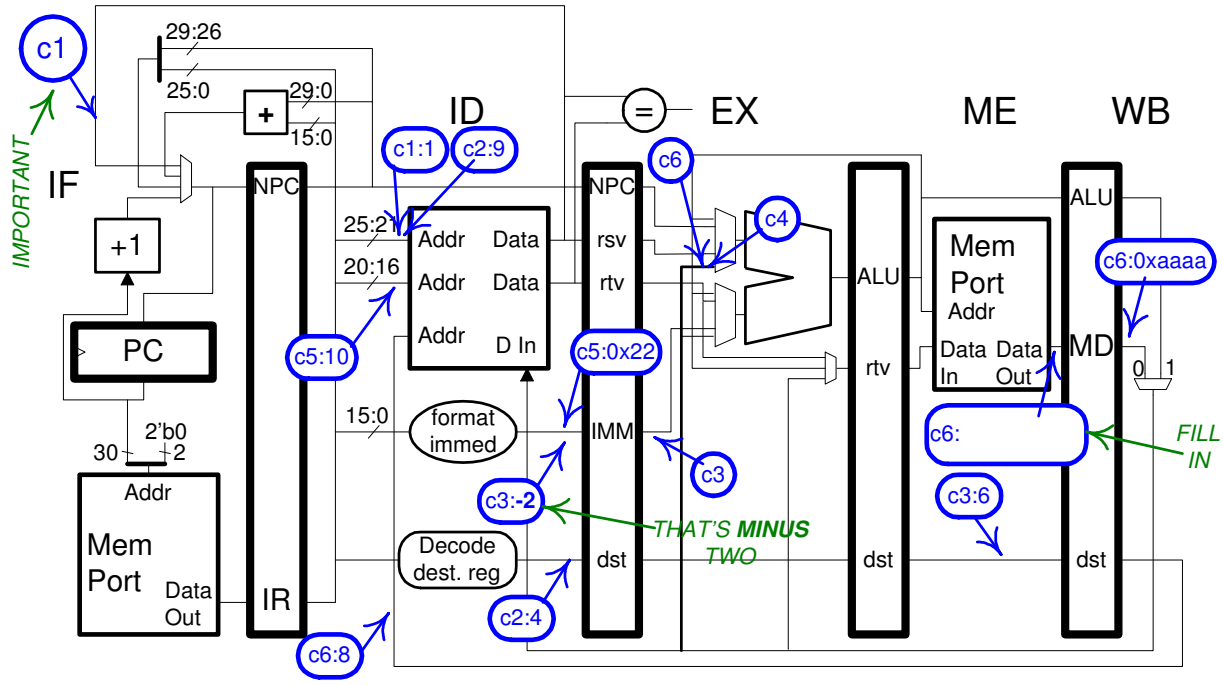
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that the fourth instruction has been provided. [50 pts]

☐ Finish a program consistent with these labels.

☐ All register numbers and immediate values can be determined.

☐ Be sure to fill the block marked *Fill In*.



Cycle:	0	1	2	3	4	5	6	7	8
	IF	ID	EX	ME	WB				
		IF	ID	EX	ME	WB			
			IF	ID	EX	ME	WB		
lb r3, 0xB(r1)				IF	ID	EX	ME	WB	
					IF	ID	EX	ME	WB
Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: The VAX `locc` instruction (from Homework 3) appears below, along with its encoding (taken from the Homework 3 solution).[20 pts]

```
 locc #65, r2, (r3)

Instruction Encoding:

 -opcode-    -- 1st operand ----     -- 2nd op -    -- 3rd op -
  locc        imm   PC*    immed      reg   r2       reg-d r3
              mode         value      mode           mode
  0x3a        0x8   0xf    0x41       0x5   0x2      0x6   0x3   <- Encoded value.
 7     0     7   4 3   0   7   0     7   4 3   0    7   4 3   0  <- Bit position.
```

(*a*) A MIPS implementation can easily retrieve its two source register values in one clock cycle.

☐ What about the VAX instruction formats makes one-cycle source register value retrieval more difficult in a VAX implementation (without lowering clock frequency)? Consider instructions with at most two source register operands.

(*b*) The constant 65 (`0x41`) is encoded in immediate mode, taking a total of two bytes, rather than literal mode, which would take only one byte if 65 were small enough for literal mode.

☐ Given the way VAX encodes operands one might expect the maximum literal size to be only four bits. Why?

☐ In fact, the maximum literal size is six bits. How is that accomplished? (If you don't know or remember the details then make up something reasonable.)

Problem 3: Under SPECcpu2006 base rules at most four compiler optimization switches can be used per language. [15 pts]

☐ What is the rationale for that restriction?

Suppose a tester would like to use five options:

```
cc -O4 --optimization-a --optimization-b --optimization-c --optimization-d
```

The tester modifies the compiler by combining options `a` and `b`, the modified compiler is made available as a product. Now compilation can be done consistent with the rules:

```
cc -O4 --optimization-ab --optimization-c --optimization-d
```

☐ Should that be considered cheating? Explain why or why not.

Problem 4: Answer each question below.[15 pts]

(*a*) In MIPS the branch instruction uses a 16-bit immediate to specify a displacement target, and the `jal` instruction uses a 26-bit immediate. Why does it make sense to choose a coding for `jal` that has a larger immediate?

(*b*) An ISA should be designed to support decades of implementations but invariably ISA designers are biased towards a first implementation at the expense of later ones. A branch delay slot (as present in MIPS and SPARC, for example) is a good example of such a phenomenon.

☐ Explain why the branch delay slot is a good example of a shortsighted ISA feature.

Name _____

Computer Architecture
EE 4720
Final Examination

11 December 2007, 15:00–17:00 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

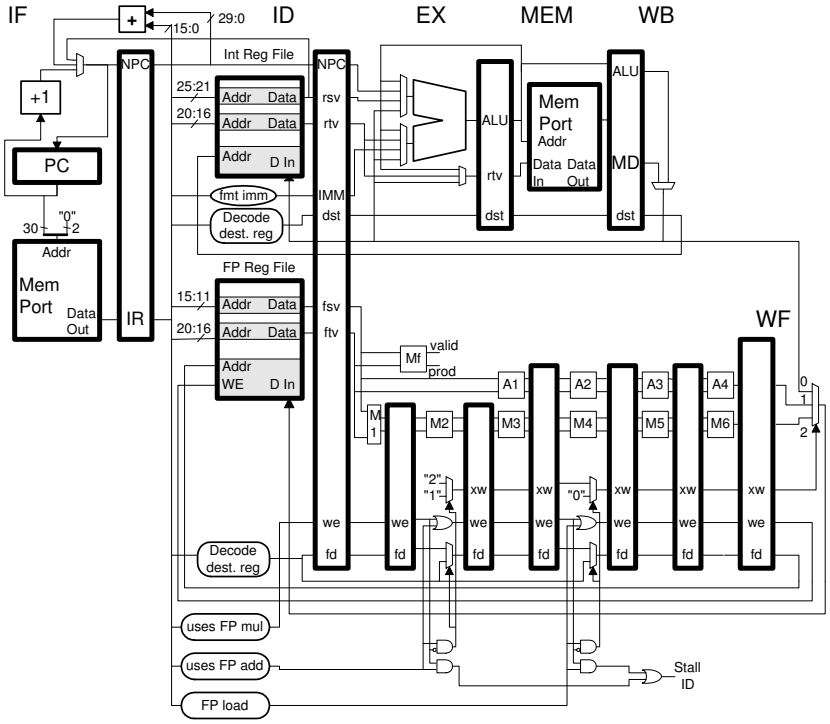
Good Luck!

Problem 1: (25 pts) Suppose it turns out that many `mul.d` instructions are executed with a 0 or 1 for one of the operands, in that case the product could be computed in less than the six stages used in the class implementation. The MIPS implementation below includes a new multiply fast unit, `Mf`, for such situations.

`Mf` has two inputs (unlabeled) and two outputs, `valid` and `prod` (they are not yet connected to anything). As with `A1` and `M1`, IEEE 754 doubles are expected at the inputs of `Mf`. If one of the inputs is 0 or 1 then the `valid` output will be `1` (otherwise it is `0`). If `valid` is `1` the `prod` output is the product of the two inputs; both outputs are available by the end of the cycle.

Call a multiply *fast* if one of its operands is 0 or 1.

USE NEXT PAGE FOR SOLUTION!



USE NEXT PAGE FOR SOLUTION!

(*a*) Add datapath and control logic for `Mf` meeting the requirements below:

☐ Add datapath so the `Mf` output will be written to the FP register file at the right time.

☐ In order of decreasing priority: the added datapath must be correct, must not increase critical paths, and should use as little new hardware as possible.

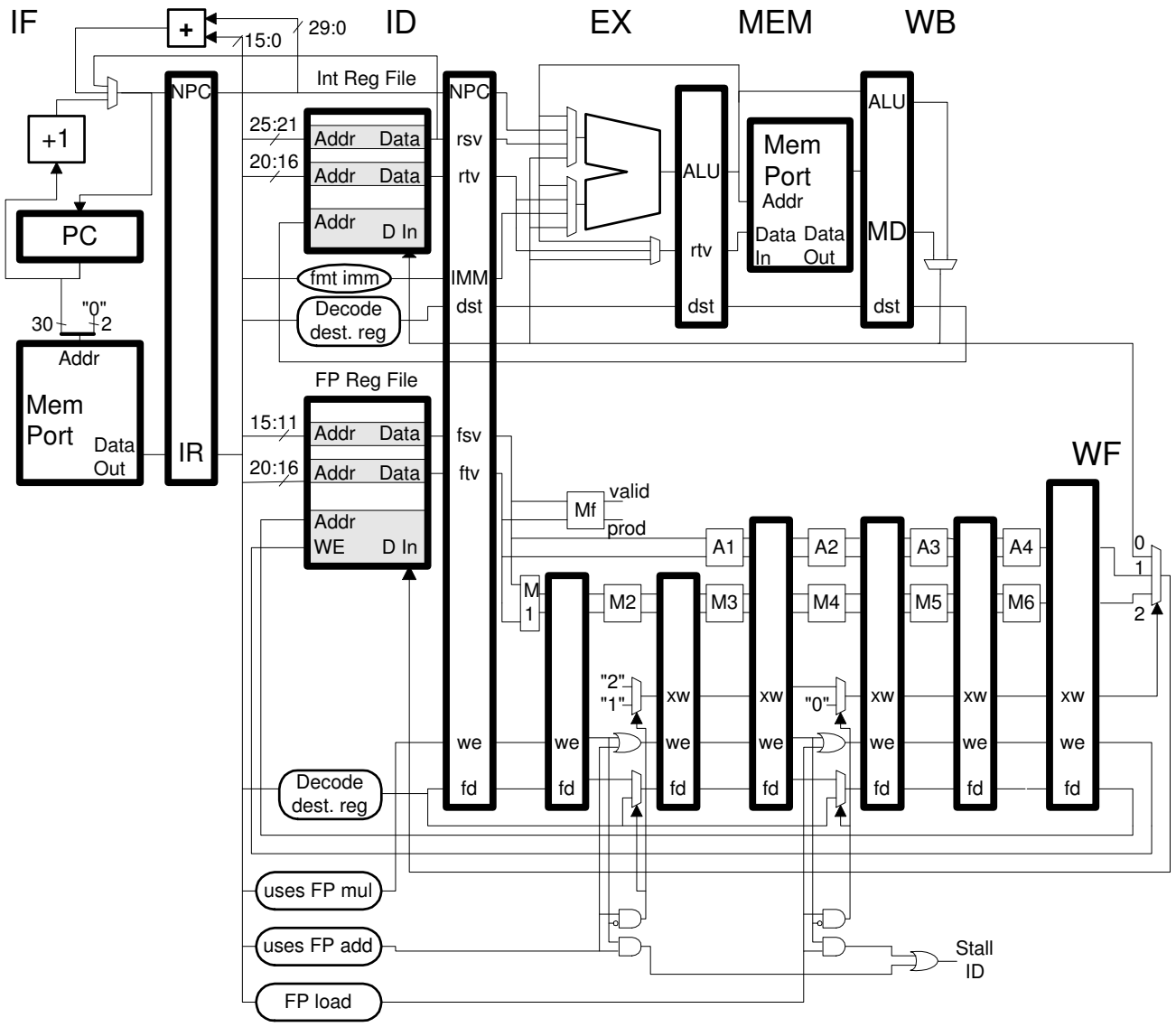
☐ The control logic must detect and handle the new `WF` structural hazard with preceding instructions that's possible when writing a fast product.

☐ If `Mf` is used then the `mul.d`'s usual `WF` slot should be available for other instructions. For example, suppose the second instruction after `mul.d` is an `add.d`. If the multiply is normal the `add.d` would stall, if the multiply is fast the `add.d` shouldn't stall (at least for the `WF` conflict with `mul.d`).

(*b*) Add bypass hardware and control logic so that the code below executes without a stall if the `mul.d` turns out to be fast. For this part assume that multiplexors at FP unit inputs won't add to critical path.

```
mul.d f2, f4, f6
add.d f8, f2, f10
```

☐ Bypass hardware for case above.



Problem 2: (25 pts) Answer the following predictor questions.

(*a*) The code below executes on three systems, one uses a bimodal predictor with a 2^{14}-entry BHT, one uses a local predictor with a 2^{14}-entry BHT and a 7-outcome local history, and one uses a global predictor with a 7-outcome global history. Consider the execution of the code below, all branches are shown.

```
BIGLOOP:
...
B1: beq r1, r2, SKIP1   T T T T N T T T T N T T T T N T T T T N ...
...
SKIP1:
...
B2: bne r3,r4 SKIP2      T T T N N N T T T N N N T T T N N N T T T N N N ...
...
SKIP2:
...
j BIGLOOP
nop
```

For partial credit show work, not just answer.

☐ Accuracy of bimodal predictor on B1 after warmup:

☐ Accuracy of bimodal predictor on B2 after warmup:

☐ Accuracy of local predictor on B2 after warmup:

☐ Warmup time of local predictor on B2:

☐ Minimum local history size for 100% accuracy of local predictor **on B2** (without ignoring B1) (show work):

☐ Accuracy of global predictor on B2 after warmup:

☐ Warmup time of global predictor on B2 (explain):

(*b*) Consider the execution of the code fragment below on a system using branch prediction. The value of `c` used in the switch statement is random, uniformly distributed over `a` to `z`, and outcomes are independent (like a 26-sided die). The branch implementing the `if ( x < 5 )` statement, which will be called the *if* branch, is taken 50% of the time.

```
int c = getchar(); // Unpredictable
switch (c) {
 case 'a': x = 3; j++; break;
 case 'b': x = 7; break;
 ...
 case 'z': x = 1; i++; break;
}
if ( x < 5 ) foo(); else bar();
```

As shown below, the `switch` construct is implemented using a dispatch table and a `jr`, and other jumps. The `switch` construct itself and the case blocks use no branches.

```
 # Part of code implementing switch construct.
 # Value in register $t1 has been computed using variable c.
 lw $t0, 0($t1)   # Load the address of the case statement corresponding to c.
 jr $t0           # Jump to case statement.
 nop
# Case 'a'
 addi $t5, $0, 3  # x = 3;
 j endswitch
 addi $t7, $t7, 1 # j++
...
```

The predictors covered in class would all achieve just a 50% prediction accuracy on the if branch.

Modify one of the predictors used in class so that it does better than 50% on the if branch. *Hint: Note that* `x` *is assigned a different constant in each case statement. A correct solution requires just a small modification of one of the predictors shown in class.*

☐ Briefly explain the idea behind your predictor.

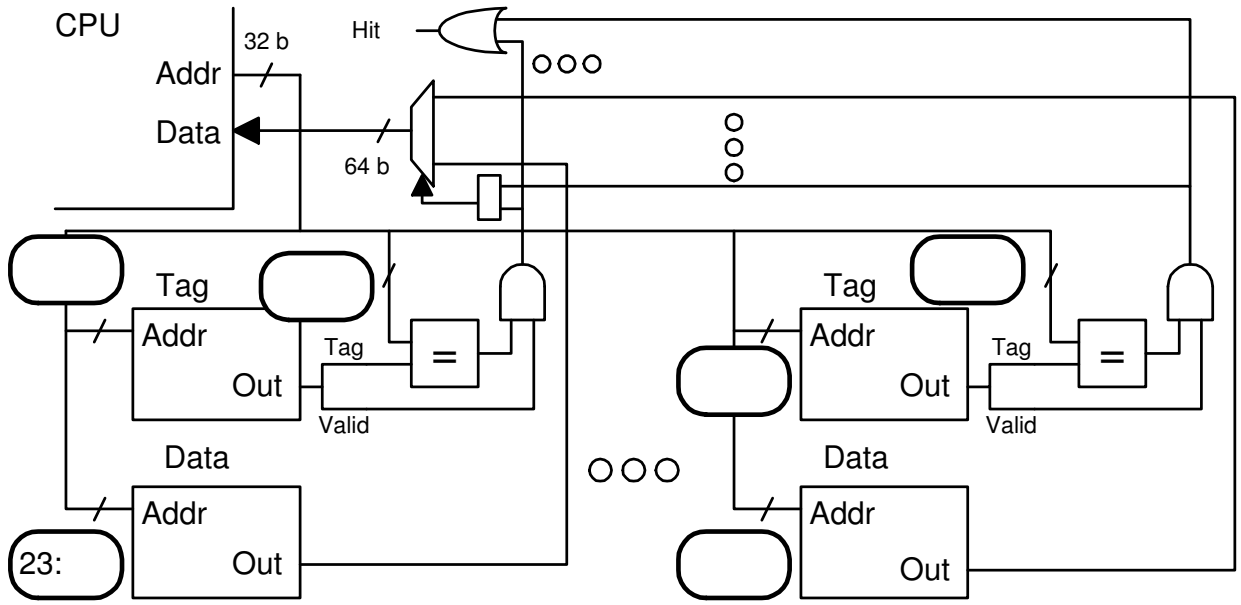
☐ Draw a diagram of the predictor.

☐ Show tables such as BHT and PHT.

Problem 3: (20 pts) The diagram below is for a 256-MiB (2^{28}-character) set-associative cache with a line size of 64 characters on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 3, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
long *a = 0x2000000;    // sizeof(long) = 8 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) Consider a 1 MiB (2^{20} byte) *direct mapped cache* with a line size of 256 characters (not the same as the one from parts a and b) for a system with a 32-bit address space. Suppose this cache has the following defect: a particular bit position in the tag comparison unit will match even if the tags differ. That is, if the bad bit position were 2 then tags `0x5` and `0x1` would match. Other cache hardware functions correctly. The cache is write through and write around.

☐ Complete the program below so that it finds the bad bit position in a small amount of time and assigns it to `badbit`.

☐ For maximum partial credit briefly describe your strategy.

- The cache is empty when the program starts.
- Assume that any address can be read or written.

```
char *a = 0x1000;
int bad_bit = -1;  // At end should be set to position of bad tag bit.
```

Problem 4: Answer each question below.

(*a*) (6 pts) A trap instruction is sort of like a procedure call (*e.g.*, `jal`) to the operating system.

☐ Describe a difference between a trap and `jal` in how the target address is specified.

☐ Describe another difference between a trap and a `jal`.

(*b*) (6 pts) A `log` (logarithm) instruction is to be added to an ISA. Group E wants to define the `log` instruction as producing the IEEE 754 double representation that is closest to the exact result. Group A would define `log` as producing any result within a certain number of bits of the exact result. Group A argues that the precision of an exact result is not needed and their approximate result is sufficient. *Group E agrees with this*, they want an exact result for other reasons. *Hint: Think about the reasons for separating ISA and implementation.*

☐ Why might group A want an approximate result?

☐ Why might group E want an exact result?

(*c*) (6 pts) Consider two scalar MIPS implementations, implementation A is similar to the one covered in class with the familiar stages `IF ID EX ME WB` while implementation B has stages `IF I1 I2 I3 I4 EX ME WB`. The two implementations run at the same clock frequency and are similar in other ways.

☐ Explain why implementation A does not need a branch predictor, or at best would only gain a small amount of performance.

☐ Explain why a branch predictor would help implementation B much more than implementation A.

☐ Consider the performance of implementation A and implementation B when branch prediction is perfect for both. Which (if any) is faster, and by how much? Explain, state any assumptions made.

(*d*) (6 pts) The code below executes on a two-way superscalar dynamically scheduled machine similar to the one presented in class. The `sub` instruction reads the value of `r1` from the register file in cycle 10, `xori` writes a value for `r1` in cycle 6 and `lw` writes a value for `r1` in cycle 9.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12
lw r1, 0(r2)     IF ID Q  RR EA          ME WB C
add r3, r9, r1   IF ID Q                 RR EX WB C
or  r6, r3, r8      IF ID Q                 RR EX WB C
xori r1, r4, 5      IF ID Q  RR EX WB               C
sub  r5, r1, r3        IF ID Q                 RR EX WB C
```

☐ Briefly explain why the code runs correctly despite the fact that `lw` writes *after* `xori`.

(*e*) (6 pts) Modern VLIW ISAs are designed with modern implementations in mind, unlike decades old RISC ISAs.

☐ Show the contents of a typical VLIW bundle.

☐ Provide an example of a VLIW feature that's designed to make implementation easier. Explain how it does so.

22 Spring 2007

Name _____

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 28 March 2007, 11:40–12:30 CDT

Problem 1	_____	(40 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

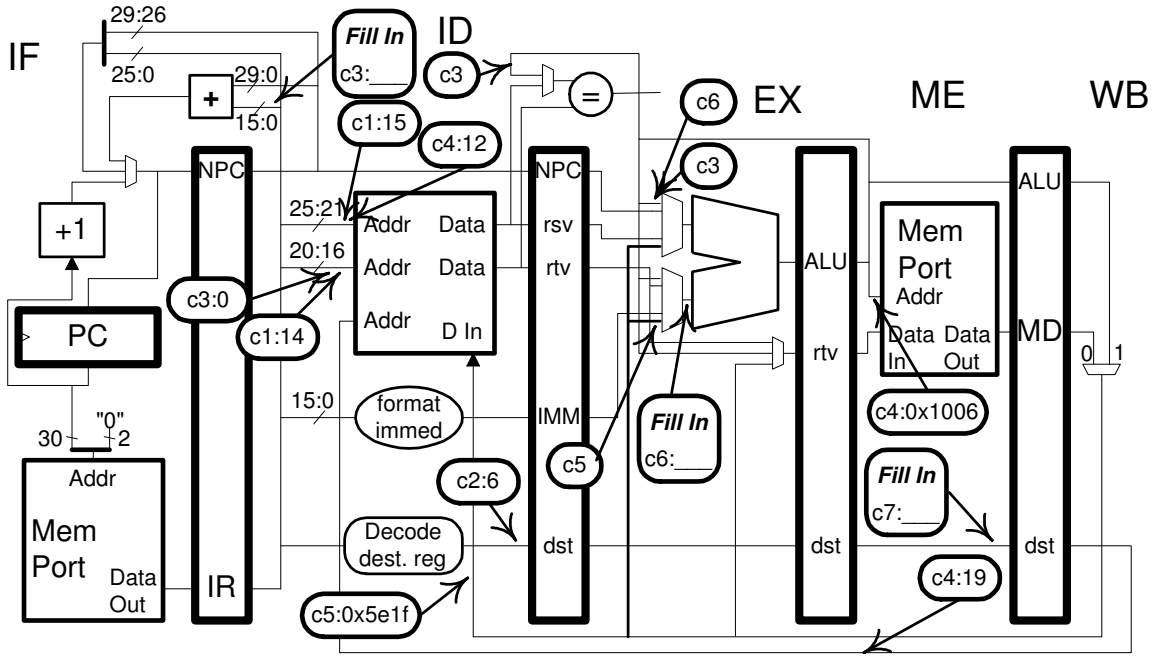
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c1:15 indicates that at cycle 1 the wire will hold a 15. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that the last instruction and the address of two instructions have been provided. [40 pts]

☐ Finish a program consistent with these labels.

☐ All register numbers and immediate values can be determined.

☐ Be sure to fill the three blocks marked *Fill In*.



	Cycle: 0	1	2	3	4	5	6	7	8
0x1000	IF	ID	EX	ME	WB				
		IF	ID	EX	ME	WB			
			IF	ID	EX	ME	WB		
				IF	ID	EX	ME	WB	
0x1030 sb r17, 3(r4)					IF	ID	EX	ME	WB
	Cycle: 0	1	2	3	4	5	6	7	8

Problem 2: Answer each question below.

(*a*) Show the encoding of the MIPS `xor` instruction below. In particular, show the name of each field in the encoded instruction, the fields' bit positions, and the fields' values. If you don't know the value of a field (there's one or two you're not expected to know) make up a value and label it "made up." *Hint: It is not cheating to look at the diagram for Problem 1.*

```
xor r5, r7, r10
```

☐ [10 pts] Encoding showing field names, bit positions, and values.

(*b*) Unlike some other benchmark suites, in SPECcpu the tester compiles the benchmarks (rather than having SPEC provide the benchmarks already compiled). [10 pts]

☐ How does this difference make SPECcpu valuable to those in the area of computer architecture?

☐ Why might this difference make SPECcpu less valuable to those looking for the fastest computer to run their favorite game?

(*c*) A company is considering adding a BCD data type to their new ISA. An analysis of a suite of Cobol programs, widely used by their customers, shows that the ISA's BCD data type would be extensively used. [10 pts]

☐ What more does the company need to know to make the decision?

Problem 3: Answer each question below.

(*a*) The MIPS code below loads a character from memory and places it in bit positions 15:8 of register `r3`. [10 pts]

```
lbu r1, 0(r2)   # Note: r2 can be any address.
sll r1, r1, 8
and r3, r3, r4  # r4 = 0xffff00ff
or  r3, r3, r1
```

☐ Explain how this code is similar to, and differs from, the operation performed by `lwl` and `lwr` (from Homework 1).

☐ **Given that MIPS already has `lwl` and `lwr`** is it worthwhile adding an instruction that performs the same operation as the code above? Explain, stating any necessary assumptions or made-up data.

(*b*) The first code fragment below, standard MIPS, uses a `slt` to perform a comparison for a branch. The second uses a proposed MIPS branch instruction that does the comparison itself and as a result the target is fetched one cycle sooner. The first execution is on *sImp*, an implementation of standard MIPS, the second execution is on *bImp*, an implementation of the proposed MIPS; bImp includes `blt` but is otherwise identical to sImp. [10 pts]

```
 # Standard MIPS  Cyc: 0  1  2  3  4  5  6  7
 slt r1, r2, r3        IF ID EX ME WB
 bne r1, r0,  TARG        IF ID EX ME WB
 nop                         IF ID EX ME WB
TARG: xor r4, r5, r6            IF ID EX ME WB

 # Proposed MIPS  Cyc: 0  1  2  3  4  5  6  7
 blt r2, r3, TARG      IF ID EX ME WB
 nop                      IF ID EX ME WB
TARG: xor r4, r5, r6         IF ID EX ME WB
```

☐ Why might the clock frequency of bImp be lower than sImp?

☐ How does knowing the percentage of branches in a program help determine if bImp can run the program faster than sImp (when compiled for the respective implementation)?

(*c*) All a compiler needs to know about the target (the implementation being compiled for) is its ISA. However, if the compiler also knows the implementation it can produce faster code. [10 pts]

☐ What's wrong with the following statement: *If the compiler also knows the target implementation it can make better register assignment decisions since it knows the exact number of registers available.*

☐ How can the compiler produce better code knowing operation latencies, such as the time needed for a `mul` instruction?

Name _____

Computer Architecture
EE 4720
Final Examination

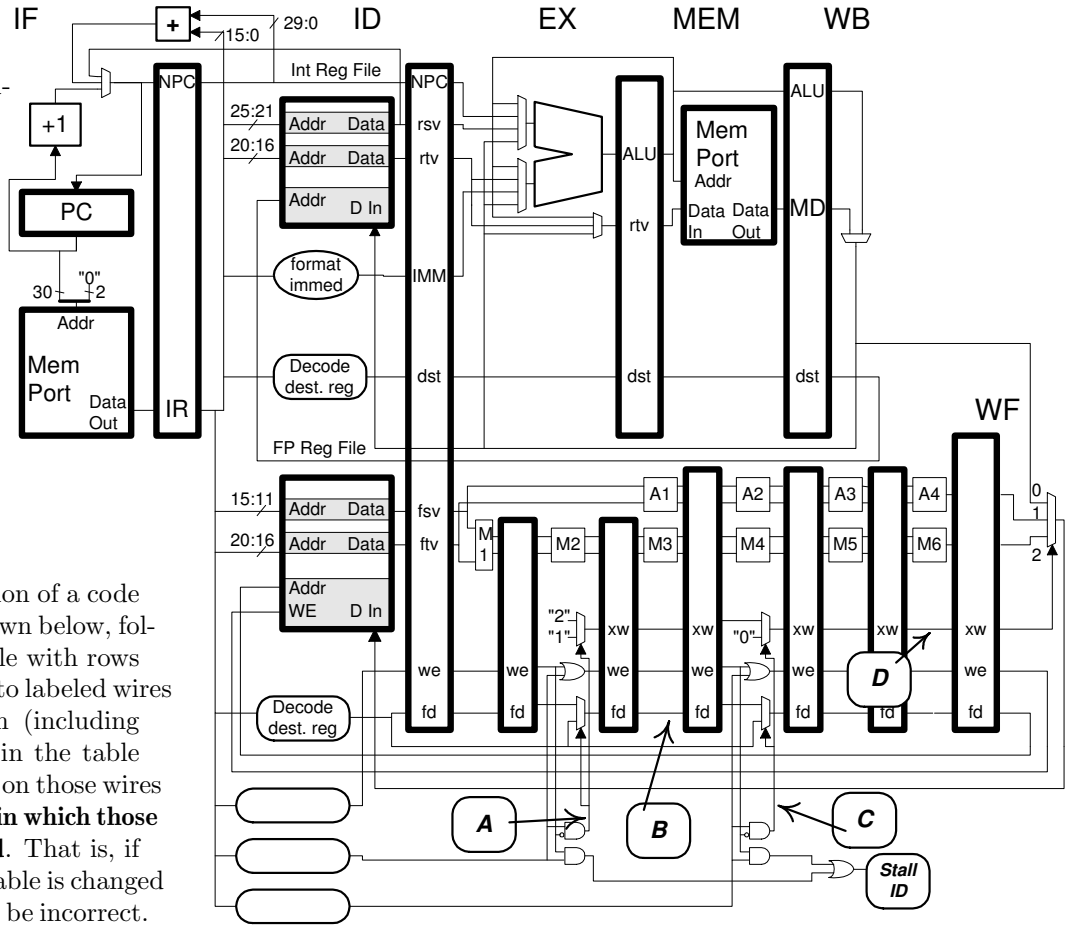
10 May 2007, 7:30–9:30 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (20 pts) The statically scheduled MIPS implementation illustrated to the right is taken from the class notes. To avoid making things too easy some descriptions were removed from logic blocks in the lower-left corner.



(*a*) The execution of a code fragment is shown below, followed by a table with rows corresponding to labeled wires in the diagram (including Stall ID). Fill in the table showing values on those wires **only for cycles in which those values are used**. That is, if a value in the table is changed execution must be incorrect.

☐ Complete the table, omitting unused values.

```
# Cycle                0    1    2    3    4    5    6    7    8    9    10
 mul.d f2, f12, f18    IF   ID   M1   M2   M3   M4   M5   M6   WF
 add.d f8, f10, f16         IF   ID   A1   A2   A3   A4   WF
 sub.d f6, f20, f14              IF   ID   ->   A1   A2   A3   A4   WF
 lwc1 f4, 0(r1)                       IF   ->   ID   ------->   EX   ME   WF
# Cycle                0    1    2    3    4    5    6    7    8    9    10
```

A:

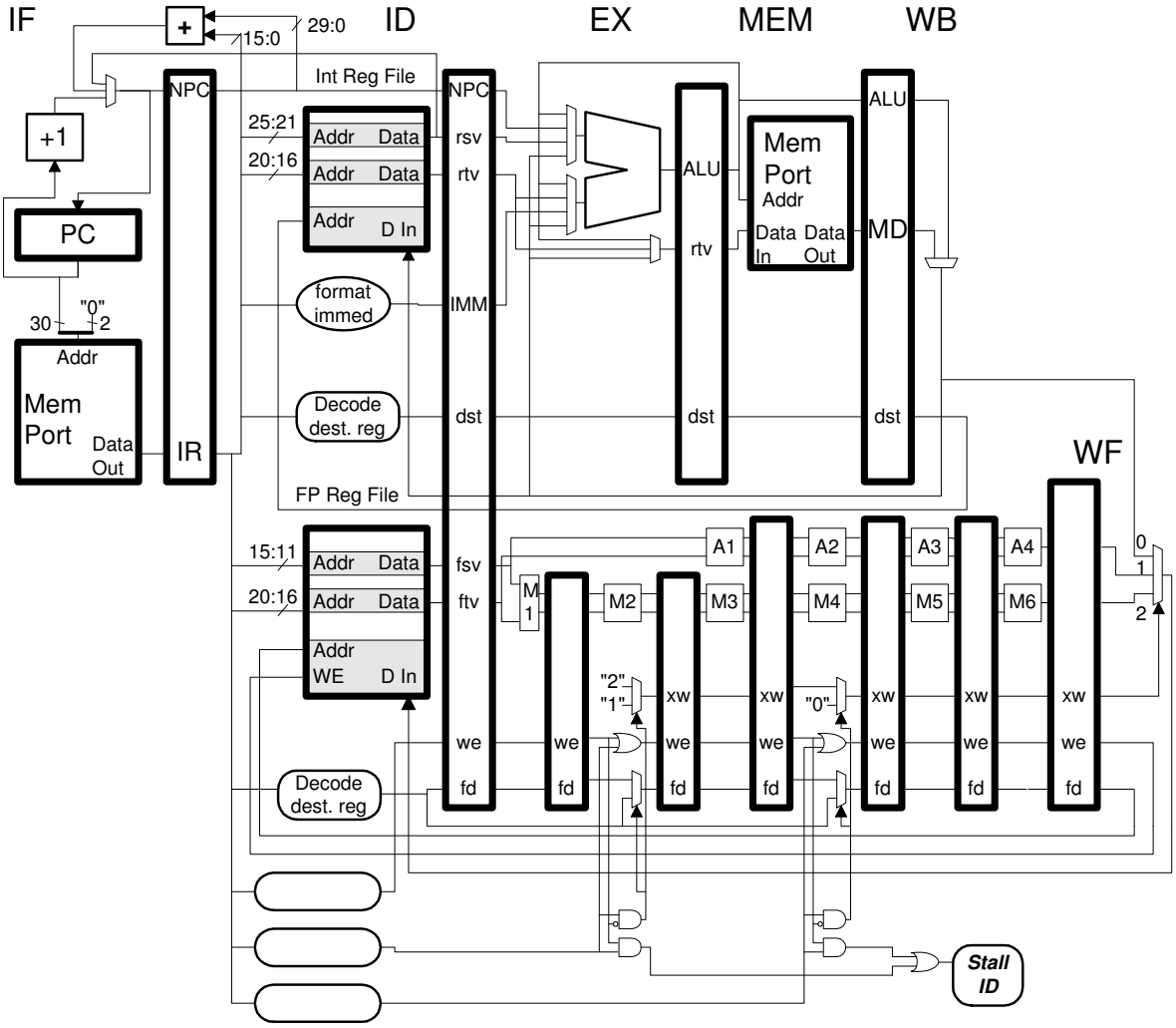
B:

C:

D:

Stall ID:

```
# Cycle                0    1    2    3    4    5    6    7    8    9    10
```



(*b*) Show the execution of the code below on the implementation assuming that all needed bypass are present. Don't forget to check for dependencies. (Instruction `mtc1` moves a value from an integer register to a floating-point register.)

☐ Pipeline diagram.

```
mtc1 f2, r7

add.s f3, f4, f2

add.s f6, f3, f8
```

(*c*) Add the bypass paths needed by the code above and show the control logic for the added paths. **Do not add unneeded bypass paths.** *Hint: Control logic should consist of two one-bit signals.*

☐ Bypass paths for code above.

☐ Control logic for added bypass paths.

Problem 2: (20 pts) Illustrated is the execution of some code on our dynamically scheduled MIPS implementation along with the contents of the ID register map, the commit register map, and the physical register file. The implementation itself is shown on the next page.

```
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
 mul.d f2, f4, f6   IF   ID   Q    RR   M1   M2   M3   M4   WF   C
 add.d f4, f2, f10       IF   ID   Q                   RR   A1   A2   A3   WF   C
 ldc1 f2,0(r1)                IF   ID   Q    EA   ME   WF                             C
 sub.d f2, f2, f8                  IF   ID   Q    RR   A1   A2   A3   WF                  C
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
ID Register Map
 F2:      12             9         71   99
 F4:      18                  51
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Commit Register Map
 F2:      12                                                       9                   71   99
 F4:      18                                                                      51
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Physical Register File
  9                      [                                   2.1                       ]
 12          2.0                                                  ]
 18          4.0                                                                 ]
 51                          [                                             4.1
 71                               [                   2.2                                  ]
 99                                    [                        2.3
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

(*a*) Answer the following

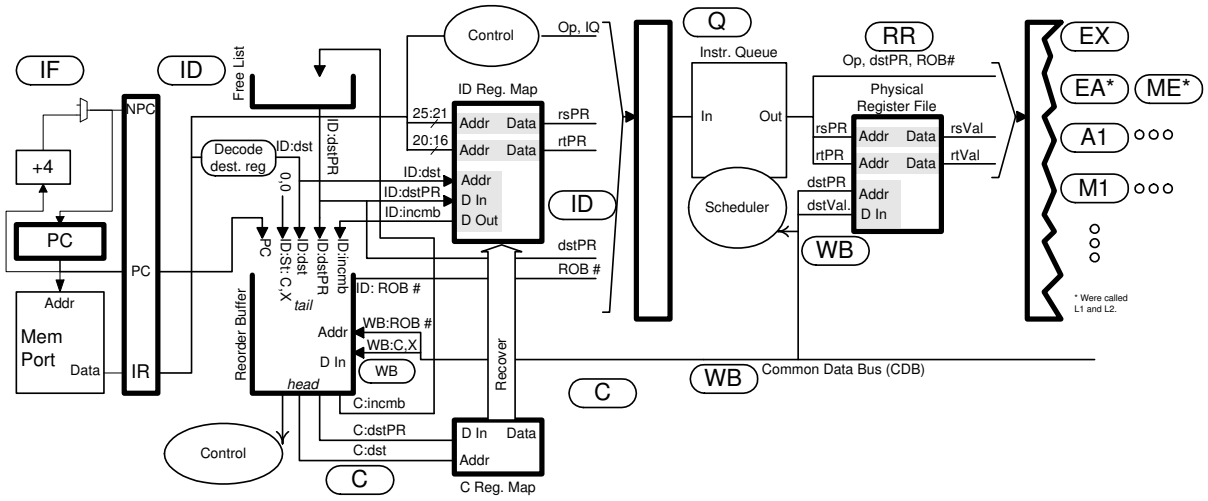
☐ Which physical register was allocated for `f4` in `add.d`?

☐ If one used the ID map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

☐ If one used the commit map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

☐ In cycle 11 where is the value of `f2` from `ldc1` located?

Problem 2, continued: Dynamically scheduled processor shown for reference.



Problem 2, continued:

```
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
 mul.d f2, f4, f6   IF   ID   Q    RR   M1   M2   M3   M4   WF   C
 add.d f4, f2, f10       IF   ID   Q                   RR   A1   A2   A3   WF   C
 ldc1 f2,0(r1)                IF   ID   Q    EA   ME   WF                           C
 sub.d f2, f2, f8                  IF   ID   Q    RR   A1   A2   A3   WF                C
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
ID Register Map
 F2:      12             9         71   99
 F4:      18                  51
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Commit Register Map
 F2:      12                                                       9                   71   99
 F4:      18                                                                      51
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Physical Register File
  9                    [                                   2.1                       ]
 12          2.0                                                  ]
 18          4.0                                                                 ]
 51                        [                                                 4.1
 71                             [                   2.2                                  ]
 99                                  [                              2.3
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

(*b*) Suppose that in cycle 11 the contents of physical register number 71 was somehow changed, perhaps due to a one-time problem. If the code executes as shown then the program runs correctly. But if a hardware interrupt happens (is taken) at the wrong time execution would be incorrect because of this change. Explain why, illustrate timing details on the diagram.

☐ Reason for incorrect execution.

☐ Timing details on diagram.

Problem 3: (20 pts) The MIPS code below runs on a system using a bimodal branch predictor of the indicated sizes. Branch outcomes are shown for each branch, the outcome patterns will continue to repeat.

```
BIGLOOP:
B1: 0x1000 beq r1, r2    T  T  T  T  T  N  T  N  N  T  T  T  T  T  T  N  T  N  N  T ...
... nonbranch insn.
...
B2: 0x1100 bne r3, r4     N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  ...
nop
j BIGLOOP
nop
```

(*a*) What is the accuracy after warmup of a bimodal branch predictor with a 2^{14}-entry BHT on branch B1?

☐ 2^{14}-entry BHT bimodal accuracy on B1.

(*b*) What is the accuracy after warmup of a bimodal branch predictor with a 2^{4}-entry BHT on branch B1?

☐ 2^{4}-entry BHT bimodal accuracy on B1.

(*c*) What is the smallest BHT size for which one can obtain the same accuracy on branch B1 as a 2^{14} entry table? Explain.

☐ Smallest size for 2^{14} entry accuracy.

☐ Reason.

(*d*) Normally the BHT in a MIPS implementation is indexed starting at bit position 2 (omitting the 2 least-significant digits) of the branch PC (address). For the following questions think about answers to the preceding parts but answer the question for ordinary programs, not the code sample above.

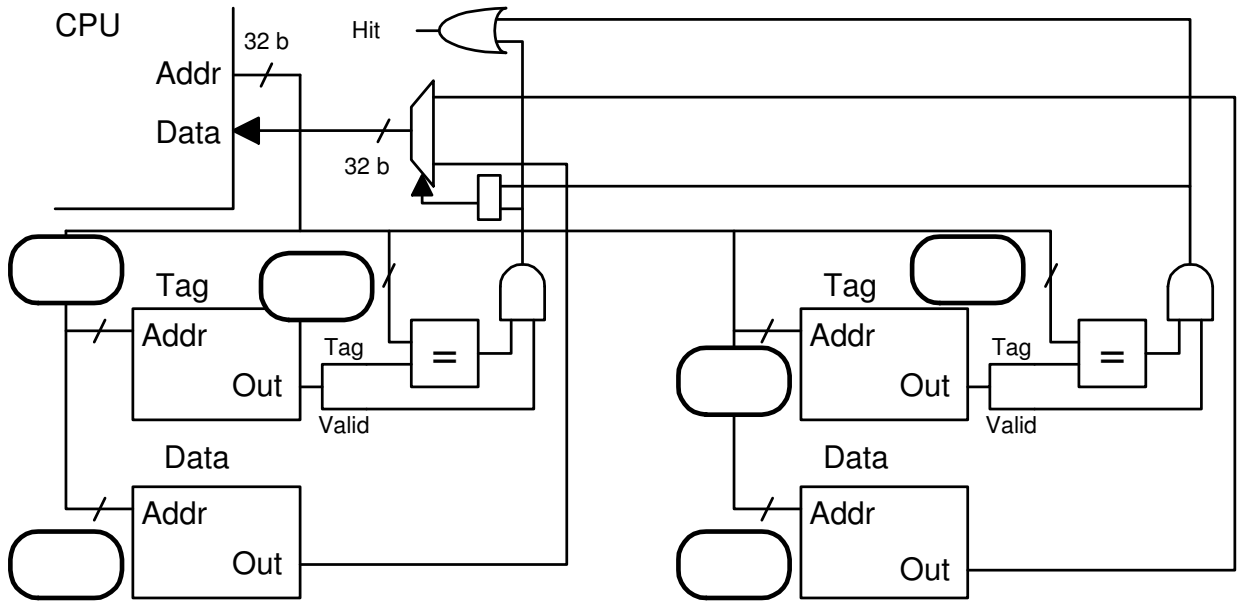
☐ Why might starting at position 3 or 4 be better?

☐ Why might starting at position 10 or 11 be worse?

Problem 4: (20 pts) The diagram below is for a 4-MiB (2^{22}-character) set-associative cache with a line size of 16 characters on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a 64-way set-associative cache with the same capacity and line size.

Address:

Problem 4, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

(*c*) The code below runs on a **direct mapped cache** with the same line size and capacity as the cache from the first part. Initially the cache is empty; consider only accesses to the arrays. Choose `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled in the minimum number of iterations (minimum `ILIMIT`). (Every access should be a miss.)

☐ `b`, `ILIMIT`, and `ISTRIDE`

☐ Briefly explain each choice.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.

char *b =                                              // FILL IN

int i;
int ILIMIT =                                           // FILL IN

int ISTRIDE =                                          // FILL IN

for(i=0; i<ILIMIT; i++)
  sum += a[ i * ISTRIDE ]  +  b[ i * ISTRIDE ];
```

Problem 5: Answer each question below.

(*a*) (5 pts) Consider trap instructions and instructions that raise exceptions.

☐ What are trap instructions typically used for?

☐ Sometimes when an instruction in a program raises an exception the program ultimately is allowed to continue. Give an example of such an exception, and what the handler might do.

(*b*) (5 pts) When a MIPS instruction raises an exception the type of exception is written to the cause register. SPARC V8 lacks an equivalent of a cause register, so what does it use as a substitute? Explain.

☐ SPARCs alternative to MIPS' cause register.

(*c*) (5 pts) An early critic might have said that the improvements realized by dynamically scheduled systems could be achieved on much less expensive statically scheduled systems by using better compilers. The particular compiler improvements would help statically scheduled systems but have no impact on dynamically scheduled ones. Consider two-way superscalar systems for the examples needed below.

☐ Explain what the compilers would have to do and why.

☐ Provide an example, showing code before and after optimization.

(*d*) (5 pts) What is it about loads that allow dynamically scheduled systems to outperform statically scheduled systems even with good compilers?

☐ Explain.

23 Fall 2006

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 27 October 2006, 12:40–13:30 CDT

Problem 1	_____	(50 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

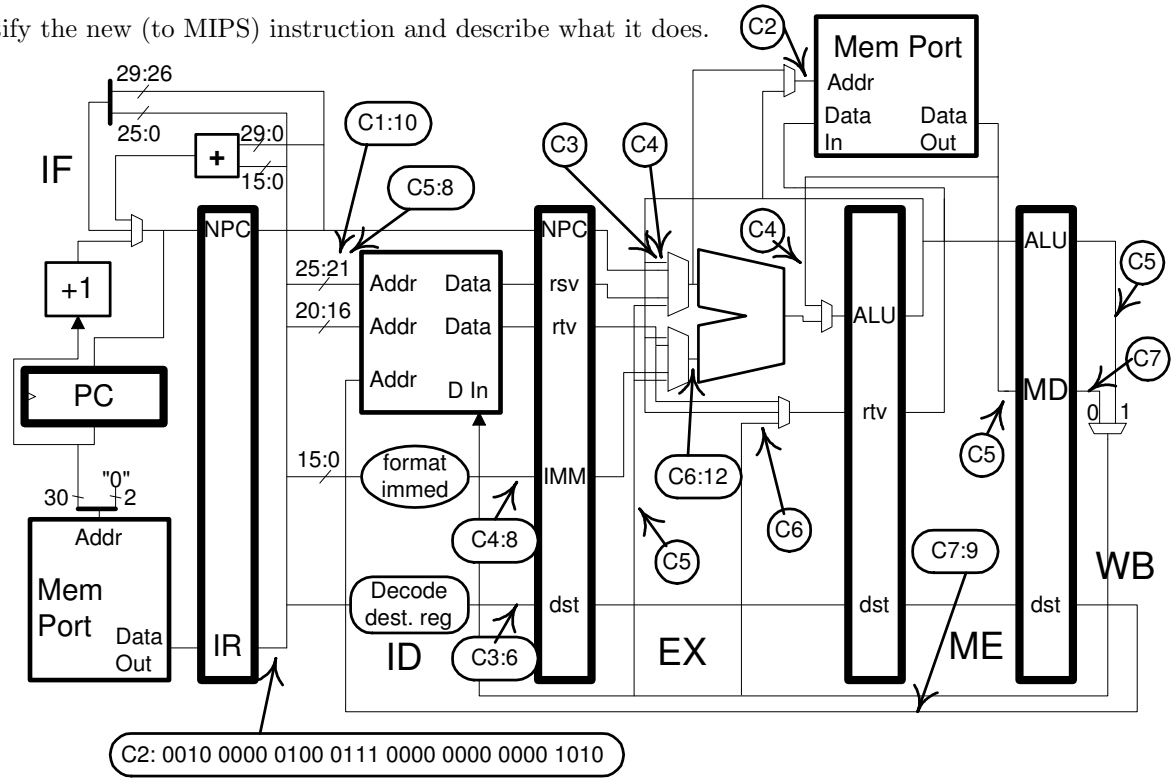
Good Luck!

Problem 1: In the MIPS implementation below the data memory port is in an unusual position that *avoids a stall* and allows the implementation of *a new (to MIPS but not CISC ISAs) instruction*. Some wires are labeled with cycle numbers and values that will then be present. For example, C1:10 indicates that at cycle 1 the wire will hold a 10. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. [50 pts]

☐ Write a program consistent with these labels.

☐ All register numbers and immediate values can be determined.

☐ Identify the new (to MIPS) instruction and describe what it does.



Cycle:	0	1	2	3	4	5	6	7	8
	IF	ID	EX	ME	WB				
		IF	ID	EX	ME	WB			
			IF	ID	EX	ME	WB		
				IF	ID	EX	ME	WB	
					IF	ID	EX	ME	WB
Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: Answer each question below.

(*a*) Both the SPARC `subcc` and MIPS `slt` instructions below determine if the contents of r1 (g1) is less than r2 (g2).

```
! SPARC
subcc g1, g2, g3

# MIPS
slt r3, r1, r2
```

☐ [5 pts] Show where each writes the comparison result, what is written, and how the result is used for a branch.

☐ [5 pts] Describe two ways in which the SPARC instruction is more powerful.

(*b*) One advantage of variable-length ISAs is that programs are shorter (take up less memory). For each code fragment below show how CISC instructions can reduce code size using the MIPS code below as the starting point of an example: Make up CISC instruction(s) for the code below, show how they might be encoded, and indicate how much smaller the code fragments are with the CISC instructions.

☐ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
lui r1, 0x1234
ori r1, r1, 0x5678
add r2, r2, r1
```

☐ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
jr r31
```

Problem 3: Answer each question below.

(*a*) The SPECcpu2006 benchmark contains two suites, CINT2006 and CFP2006.

☐ [8 pts] Why are there two suites? What would be the disadvantage of combining them into one suite?

(*b*) A company develops a new ISA and some implementations of it. It sells the implementations, but keeps the ISA secret.

☐ [8 pts] What's wrong with that?

(*c*) Operations on packed operand data types often use saturating arithmetic.

☐ [7 pts] What is it and why is it used? Explain using a typical application for packed-operand instructions.

(*d*) One reason to not use optimization is to make debugging (using a debugger) easier.

☐ [7 pts] Describe two ways optimization makes debugging more difficult. *Hint: Consider single-stepping through code and printing variable values.*

Name _____

Computer Architecture
EE 4720
Final Examination

14 December 2006, 17:30–19:30 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: In the MIPS implementation on the next page some wires are labeled with cycle numbers and corresponding values. For example, c3:1 indicates that at cycle 3 the pointed-to wire will hold a 1. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. The ALU label shows the arithmetic operation performed at the indicated cycle.(20 pts)

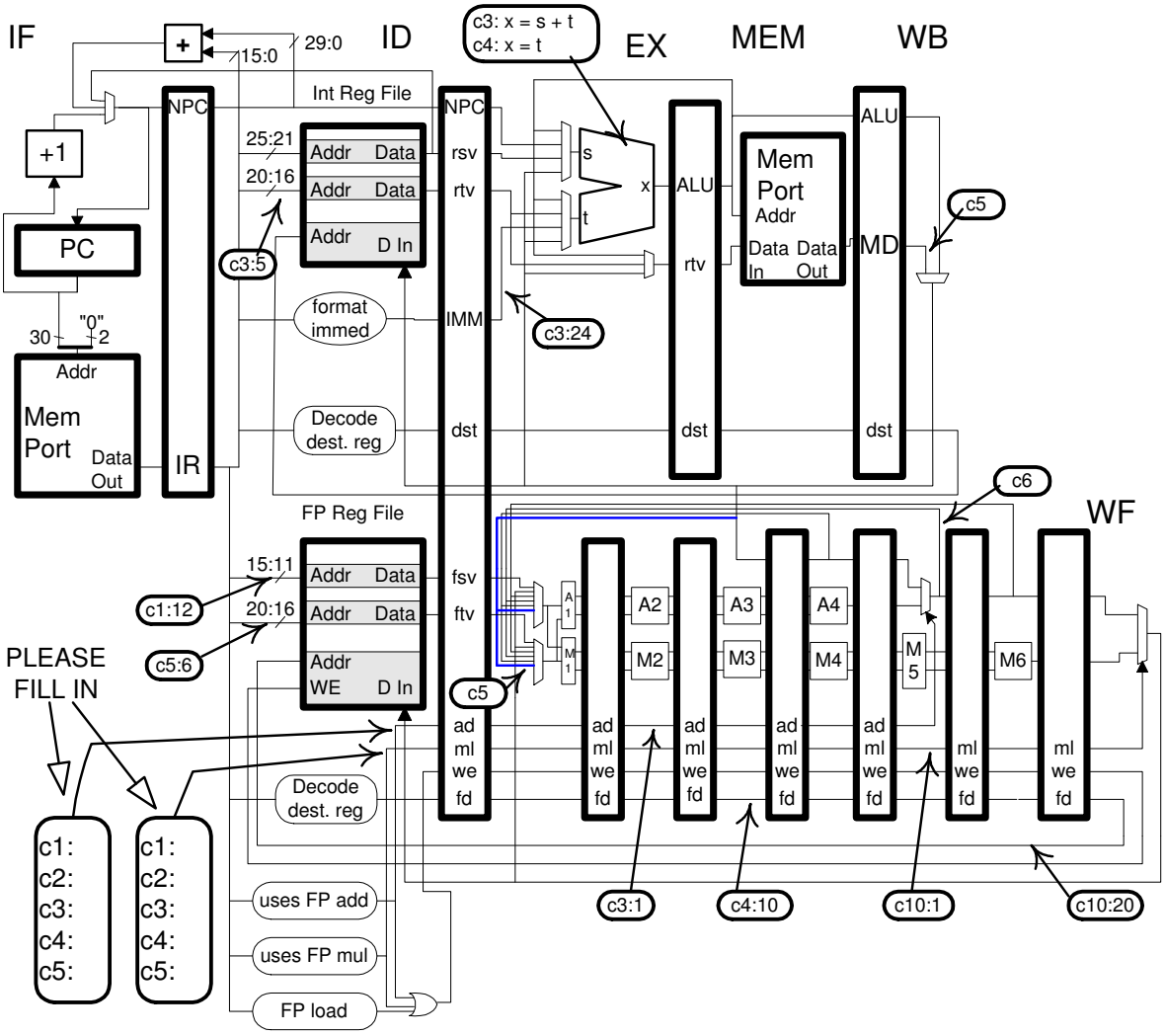
- There are no branches or other control-transfer instructions.
- There are no stalls.
- Every instruction writes the floating-point register file.
- Some instruction(s) read the integer register file.
- One instruction has only briefly been covered, make up a reasonable name for it if you don't remember it.

☐ Write a program consistent with these labels.

☐ Some registers can be determined exactly, others must be made up. **Use as many different register numbers as possible** while still being consistent with the labels.

☐ Fill in the block in the lower-left of the diagram.

Problem 1, continued:

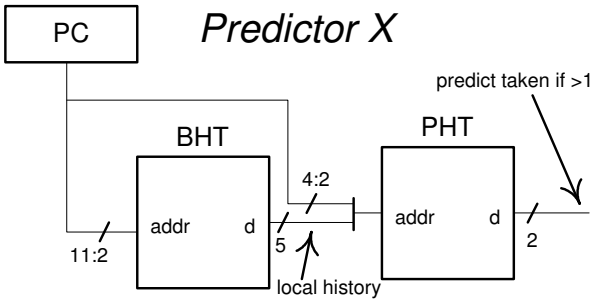



```
Cycle:  0   1   2   3   4   5   6   7   8   9   10  11  12
```

IF	ID	_1	_2	_3	_4	_5	_6	WF				
	IF	ID	_1	_2	_3	_4	_5	_6	WF			
		IF	ID	_1	_2	_3	_4	_5	_6	WF		
			IF	ID	_1	_2	_3	_4	_5	_6	WF	
				IF	ID	_1	_2	_3	_4	_5	_6	WF

```
Cycle:  0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: The code below runs on three systems which are identical except for the branch predictors. One system uses a bimodal predictor with a 1024-entry BHT, one uses a local history predictor with a 1024-entry BHT and a five-outcome history, and one uses Predictor X, illustrated below. (The PHT input is a concatenation of the local history and three branch PC bits.) The code below has two branches, B1 and B2, which execute in a repeating pattern as shown. There are no other branches in the code.(20 pts)



Note: In the original problem the input to Predictor X used an exclusive or rather than a concatenation.

```
LOOP:
 ..
0x1000: B1: bne r1,r2 SKIP1   N N N T T N N N T T N N N T T N N N T T ...
 ..
SKIP1:
 ..
0x1124: B2: bne r3,r4, SKIP2   N N N T T T N N N T T T N N N T T T ...
 ..
SKIP2
..
     j LOOP
```

☐ What is the accuracy of the bimodal predictor on branch B1 after warmup?

Note: In the original exam the questions below asked about B1 rather than B2.

☐ What is the accuracy of the local predictor on branch B2 after warmup? (Do not ignore branch B1 when answering this part.)

☐ What is the minimum local history size for the local predictor to achieve 100% accuracy on branch B2 (without ignoring B1)?

☐ What is the accuracy of Predictor X on branch B2?

☐ What is the minimum local history size for the Predictor X to achieve 100% accuracy on branch B2 (without ignoring B1)?

☐ Explain why predictor X has lower or higher accuracy than the local predictor on branch B2.

☐ As indicated above, B1 is at address `0x1000` and B2 is at address `0x1124`. How would different branch addresses affect the answers above?

Problem 3: Consider the execution of MIPS code below. The code follows a large number of `nop` instructions. As can be seen the system below is statically scheduled. In the questions below "relatively simple" means simple compared to dynamic scheduling.

(15 pts)

```
# PC             Cycle:  0  1  2  3  4  5  6  7
0x1ff0: lh r1, 0(r2)     IF ID EX ME WB
0x1ff4: lw r4, 8(r2)     IF ID -> EX ME WB
0x1ff8: addi r6, r6, 1   IF ID -> EX ME WB
0x1ffc: xor r8, r9, r10  IF ID -> EX ME WB
0x2000: sub r11, r8, r13    IF -> ID EX ME WB
0x2004: and r14, r11, r16   IF -> ID -> EX ME WB
# PC             Cycle:  0  1  2  3  4  5  6  7
```

☐ What is the minimum fetch/decode width of a system that could produce that execution? (The x in x-way superscalar)

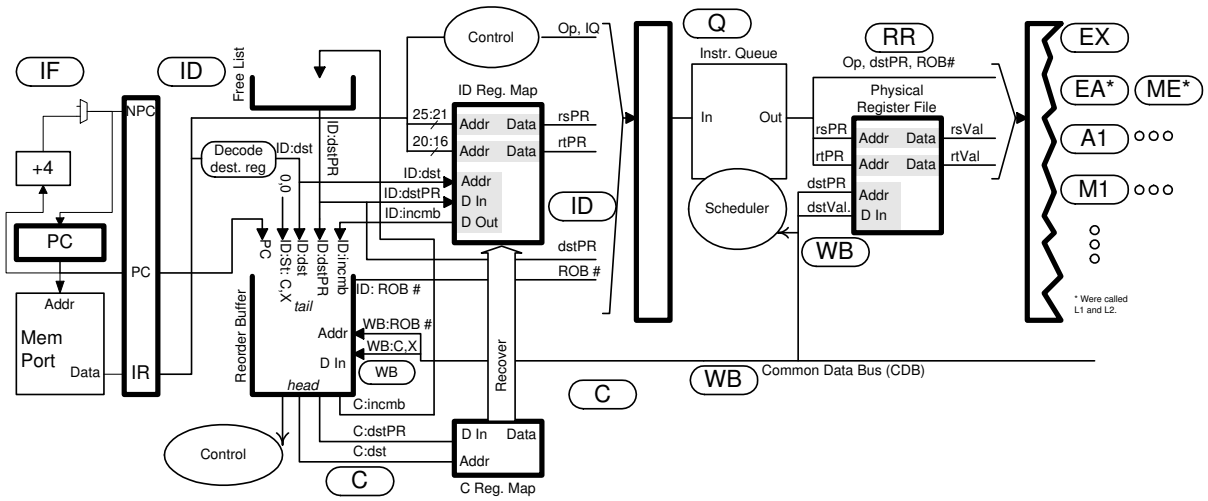
☐ Why is the fetch/decode width above a minimum and not an exact number?

☐ What caused the `and` to stall in cycle 4? Could the stall be avoided?

☐ What might have caused `lw` to stall? Suggest a relatively simple change to the implementation to avoid such stalls.

☐ What might have caused `addi` and `xor` to stall? *Note: The original question also asked for a "relatively simple solution."*

Problem 4: Consider the dynamically scheduled implementation below. (15 pts)



(*a*) Where is the logic for finding the (data) dependencies that requiring bypassing most likely to be?

☐ Indicate on diagram.

(*b*) Suppose due to a manufacturing error every entry in the ID register map is initialized to 12 after each reset, but the commit register map was properly initialized. Initial register values are not defined, so getting the wrong value for a register that was never written is not a problem here.

☐ Which code fragment below is more likely to encounter a problem? *Hint: It has something to do with the connection from the ID Register Map to the ROB.*

```
# Fragment A
 add r1, r2, r3
 add r2, r1, r5
 add r1, r6, r7
 add r2, r8, r9
 nop
 ..

# Fragment B
 add r0, r2, r3
 add r0, r1, r5
 nop
 ...
```

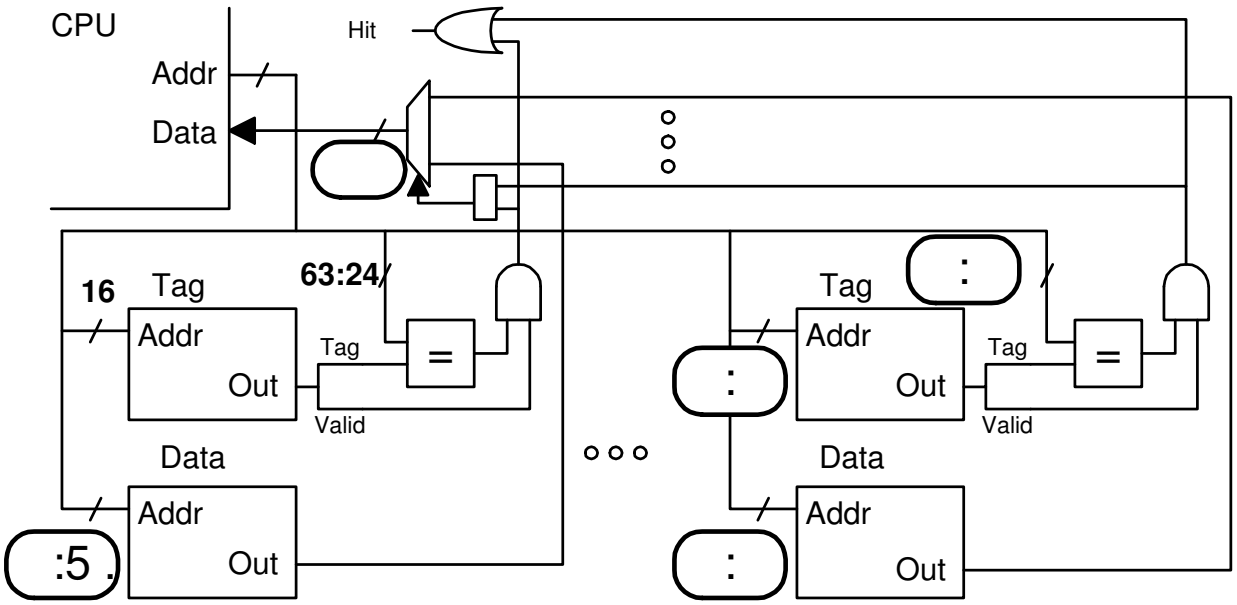
☐ Explain what goes wrong.

☐ Suppose millions of these defective implementations have already been manufactured. Suppose when turned on the processor starts executing code at address `0x1000`. What code could be put there to fix the problem? (It's not one of the fragments above because one of them only avoids it, later code could still trigger it.) *Hint: There's enough room for the answer below.*

Problem 5: The diagram below is for a 64-MiB (2^{26}-character) set-associative cache on a system with the usual 8-bit characters. (15 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 5, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 10;  // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i * 4 ];
```

(*c*) The code below also runs in the cache from part a. Find the minimum values of `JLIMIT` and `JSTRIDE` that will result in the code below having a 0% hit ratio.

☐ `JSTRIDE` and `JLIMIT`

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i, j;
int ILIMIT = 1 << 10;

int JLIMIT =                                          // FILL IN

int JSTRIDE =                                         // FILL IN

for(i=0; i<ILIMIT; i++)
 for(j=0; j<JLIMIT; j++)
  sum += a[i + j * JSTRIDE ];
```

Problem 6: Answer each question below.

(*a*) In MIPS, SPARC, and many other RISC ISAs memory accesses are aligned and so any instruction that uses an un-aligned address, such as `0x1001` for a MIPS word, will raise an exception usually resulting in the program exiting with a Bus Error.

Perhaps due to the stress of an upcoming code freeze for a product release, a mysterious programmer at Software Company X secretly hacked the operating system of their SPARC computers so that loads and stores to un-aligned addresses would complete correctly, as though un-aligned accesses were not forbidden. There would be no more bus errors. (5 pts)

☐ How might the mysterious programmer have done it?

☐ Should Software Company X reward or punish the mysterious programmer? Explain.

(*b*) The SRAM used to implement caches is costly but in many cache designs can provide 1- or 2-cycle hit latencies. If cost were not an issue, could one use SRAM for the entire memory system and get 1- or 2-cycle latencies on *all* memory accesses?(5 pts)

☐ Explain.

(*c*) Manufacturers have a great interest in having their processors score high in SPECcpu. Given this strong interest how can we be sure that benchmark selection and the run & reporting rules have not been chosen to favor a particular manufacturer? (This isn't kindergarten, so "because it's not allowed" is not a good answer.)(5 pts)

☐ Explain.

24 Spring 2006

Name _____

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 29 March 2006, 11:40–12:30 CST

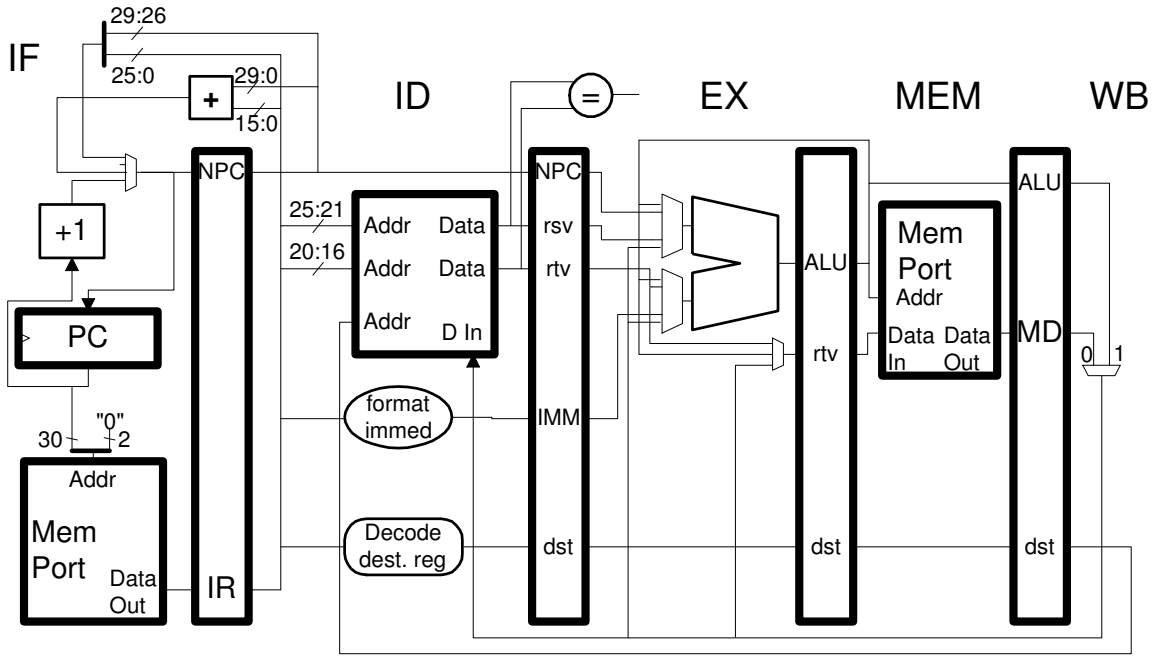
Problem 1	_____	(10 pts)
Problem 2	_____	(40 pts)
Problem 3	_____	(50 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The MIPS code below runs on the illustrated implementation. [10 pts]

☐ Show the execution of the code below on the illustrated pipeline.




```
# Cycle          0   1   2   3   4   5   6   7   8

lw r1, 0(r2)

add r1, r1, 7

sw r1, 0(r2)

# Cycle          0   1   2   3   4   5   6   7   8
```

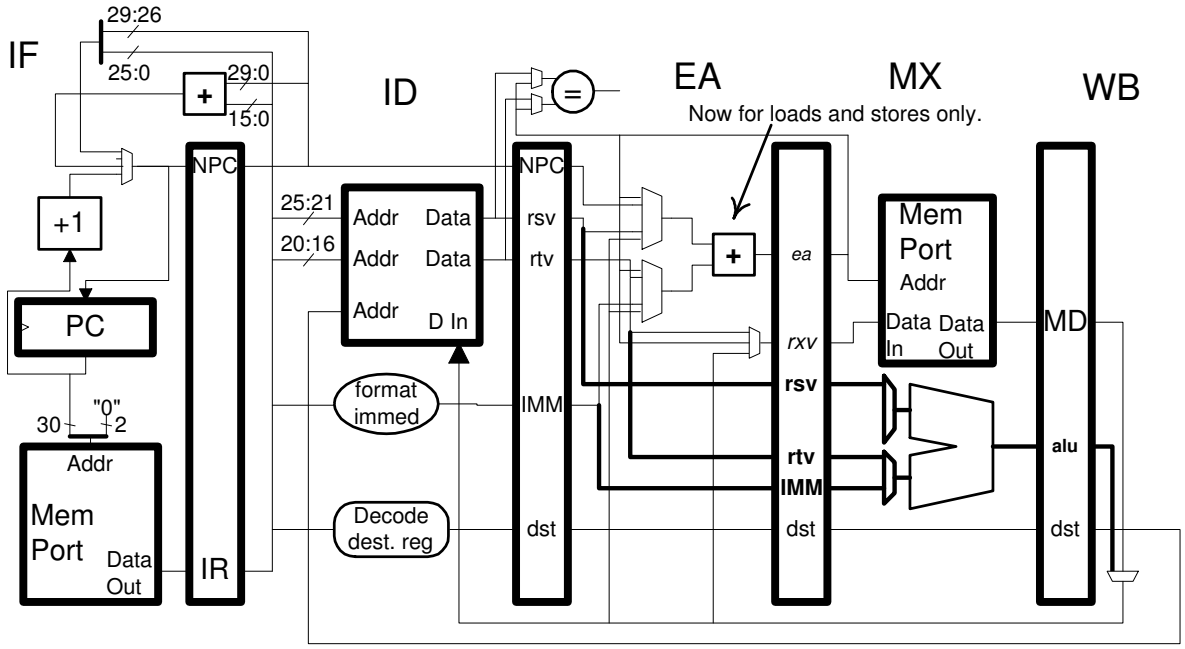
Note: To keep the test from getting too long the following parts were not included on the original exam. There should be at least one stall.

Think about the questions below for a few moments, then look at the problem on the next page.

☐ Is it possible to add bypass connections to eliminate the stall and gain performance?

☐ If bypasses won't work does that mean it's impossible to eliminate the stall?

Problem 2: The five-stage MIPS implementation below has an ALU in the MX (new name for MEM) stage (connections for that ALU are shown bold) and what was the ALU in the EA (new name for EX) stage is now just an adder and is to be used only for loads and stores.



(*a*) Add bypass connections needed so that the code below (same as last problem) executes as shown. Label those bypass connections: LAS-*c*, where *c* is the cycle number in which the bypass connection is used. Do not add unneeded bypass connections!

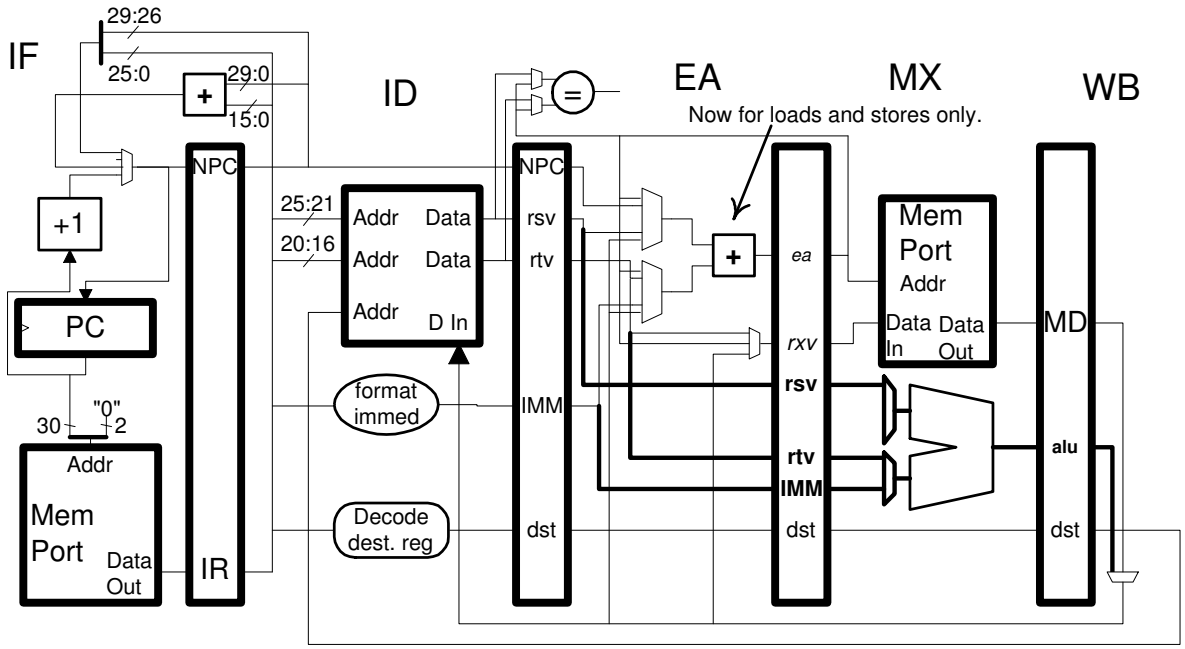
☐ [10 pts] Add bypass connections, label with LAS-*c*.

```
# Cycle          0  1  2  3  4  5  6
lw r1,0(r2)      IF ID EA MX WB
addi r1, r1, 7      IF ID EA MX WB
sw r1, 0(r2)           IF ID EA MX WB
# Cycle          0  1  2  3  4  5  6
```

(*b*) Several connections to the EA-stage adder are now unneeded (but were needed when it was an ALU). (Don't add new connections here.)

☐ [10 pts] Label unneeded EA-adder connections with an X at mux inputs.

Problem 2, continued: The implementation below is identical to the one on the previous page.



(*c*) For each code fragment below: Add reasonable bypass connections to eliminate stalls (if any). Show the execution and label the bypass connections that are used (new or existing) with a cycle number. If there is a stall circle Yes if an EX-stage ALU (in addition to the ME stage ALU) would remove the stall, if an EX-stage ALU wouldn't help circle No; if there is no stall don't circle anything. [20 pts]

☐ Add bypasses (if needed). Label bypasses used AS-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0   1   2   3   4   5   6   7   8
add r1, r2, r3   IF  ID  EA  MX  WB

sub r4, r1, r5
```

☐ Add bypasses (if needed). Label bypasses used AL-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0   1   2   3   4   5   6   7   8
add r1, r2, r3   IF  ID  EA  MX  WB

lw r4, 4(r1)
```

☐ Add bypasses (if needed). Label bypasses used LL-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0   1   2   3   4   5   6   7   8
lw r1, 12(r2)    IF  ID  EA  MX  WB

lw r3, 16(r1)
```

☐ Add bypasses (if needed). Label bypasses used AB-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0   1   2   3   4   5   6   7   8
addi r1, r1, 1   IF  ID  EA  MX  WB

beq r1,r2 TARG

nop
```

Problem 3: Answer each question below.

(*a*) A computer's scores on SPECint2000 are baseline, 2011; and result (peak) 2014. These baseline and result scores are close, who might be responsible and should they be proud or ashamed: [10 pts]

☐ How might the people who conducted the test be responsible for the close scores? Should they be proud or ashamed?

☐ How might the people who wrote the compiler be responsible for the close scores? Should they be proud or ashamed?

(*b*) On a SPARC system the handler for trap number 4 is located at address `0xa0001000` and is 100 instructions long. [10 pts]

☐ Show the Trap Base Register (TBR) value and Trap Table contents needed so that execution of a trap instruction will reach this handler. (Don't worry about returning.)

☐ Why isn't a user program allowed to call the handler directly, for example, using the ordinary call instruction `call 0xa0001000`. (The SPARC call has a 30-bit immediate field so the target address is not too far away.)

☐ What would happen if the user tried to execute the call instruction above?

Problem 3, continued:

(*c*) Packed integer and BCD data types are very superficially similar. [10 pts]

☐ Describe a situation in which a packed integer data type would be useful.

☐ Describe a situation in which BCD would be useful.

(*d*) Describe the contents of a typical bundle in a VLIW ISA. [10 pts]

☐ Bundle contents.

(*e*) Which is it more important to do a good job on, the ISA or the implementation? Explain. [10 pts]

☐ Good job on ISA or implementation? Explain.

Name _____

Computer Architecture
EE 4720
Final Examination

8 May 2006, 10:00–12:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

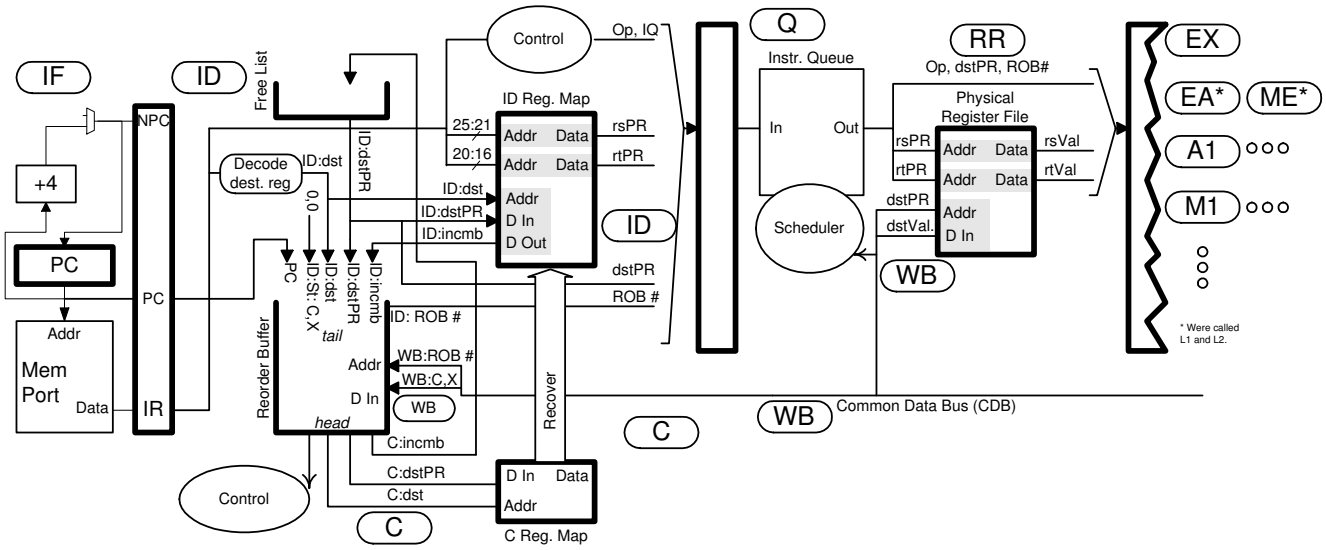
Alias _____

Good Luck!

Problem 1: The execution of code on a dynamically scheduled scalar MIPS implementation using Method 3 (the only one covered in class) is shown below. Beneath the diagram are signal labels, for example, `ID:dst`. (25 pts)

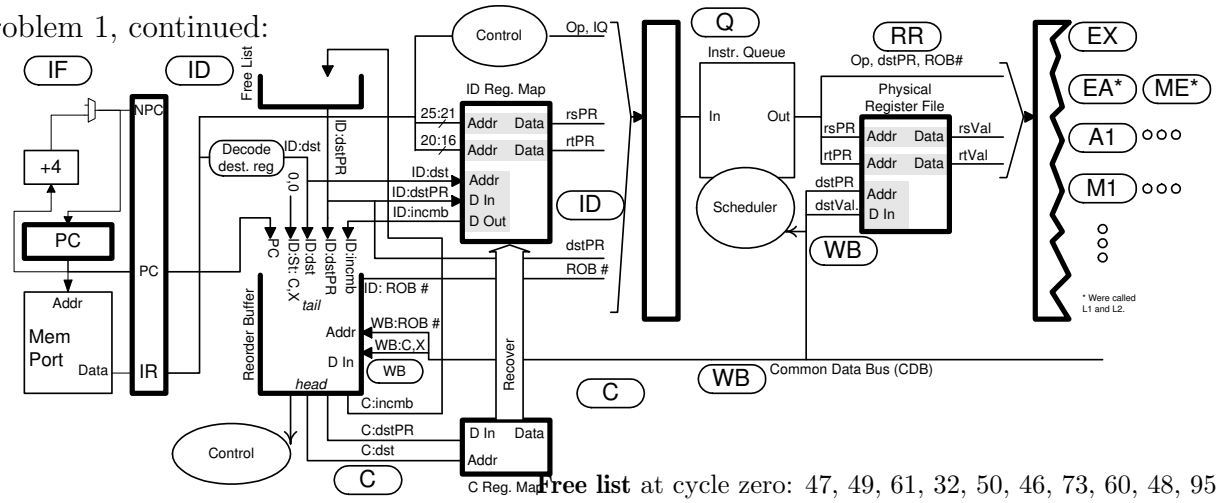
(*a*) On the next page show the values of the signals at the cycles they are used.

- Some values are shown, they are needed to determine other values.
- All values can be determined.
- Ignore the distinction between integer and FP registers.
- The free list contents at cycle zero is shown on the lower right-hand side of the figure on the next page.



Tables for answer on next page.

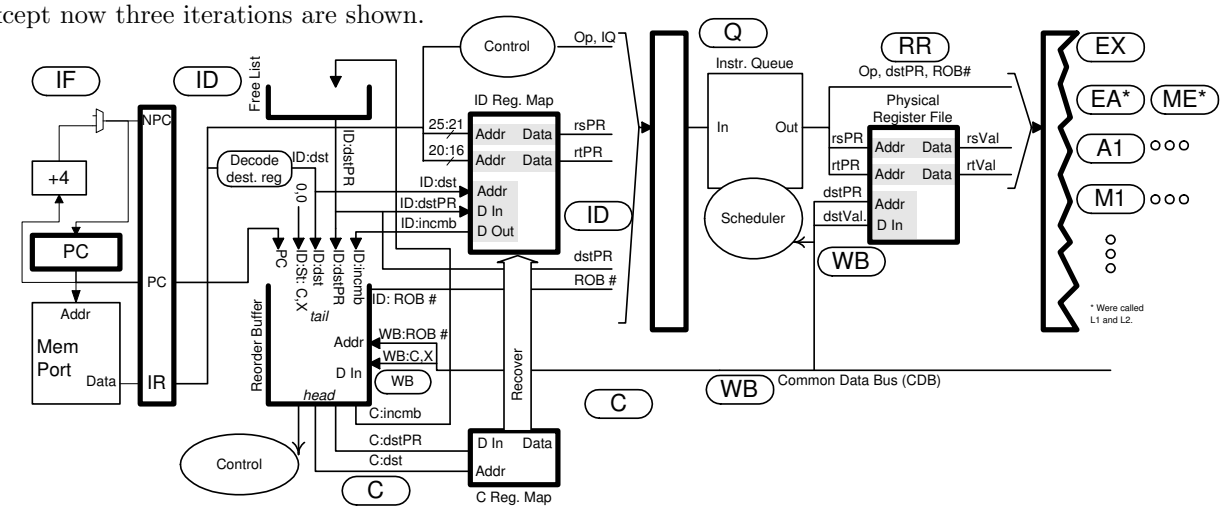
Problem 1, continued:



Free list at cycle zero: 47, 49, 61, 32, 50, 46, 73, 60, 48, 95

```
LOOP:              # First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
LOOP:              # Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:rsPR               56 30 30

ID:rtPR               63    54

ID:dst

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:dstPR

ID:incmb

RR:rsPR

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

RR:rtPR

WB:dstPR

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

C:incmb

C:dstPR

C:dst

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Problem 1, continued: The diagram below is for the same code on the same system as the previous part, except now three iterations are shown.




```
LOOP:            #  First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Third Iteration
 add.s  f2, f2, f4                    IF ID Q        RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                           IF ID Q  RR B  WB                C
 sub r5, r5, r6                             IF ID Q     RR EX WB             C
```

(*b*) What is the CPI for a large number of iterations? *Hint: It's not 1.*

☐ CPI?

(*c*) Suppose the ROB can hold 256 entries and there are an unlimited number of physical registers. Assuming a very large number of iterations, at about what cycle will fetch stall?

☐ Fetch will stall at cycle ≈

(*d*) Would the code above run faster on a four-way superscalar dynamically scheduled system with the same clock frequency and pipeline depth? Explain.

☐ Circle One: Faster Not Faster.

☐ Because

Problem 2: The code fragments below run on three systems which are identical except for the branch predictor: One uses a **bimodal** predictor with a 2^{14}-entry BHT, one uses a **local predictor** with an 8-outcome local history and a 2^{14}-entry BHT, and one uses a **global predictor** with an 8-outcome global history. (25 pts)

(*a*) Provide the information requested below.

- All accuracies are after warmup.
- For the warmup time show the approximate number of times the predicted branch needs to be executed before the predictor reaches its warmup accuracy. Assume the 2-bit counters start out at 0 or 3, whichever is worse.

```
BIG:addi r3, r0, 3
LP: bne r3, r0 LP  # Iterates four times.
    addi r3, r3, -1

    lw r1,0(r2)
C2: beq r1, 0 SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

☐ Bimodal: accuracy on `C2`:

☐ Bimodal: warmup time on `C2`:

☐ Local: accuracy of `C2`:

☐ Local: warmup time on `C2`:

☐ Local: smallest history size needed for 100% accuracy on `C2`:

☐ Global: accuracy on `C2`:

☐ Global: warmup time on `C2`:

☐ Global: smallest history size needed for 100% accuracy on `C2`:

Problem 2, continued:

(*b*) The code below is similar to the code on the previous page except the four-iteration loop has been replaced by two random branches. Each random branch will be taken with probability .5 and the outcome is independent of everything, including the other random branch.

```
BIG:lb r3, 0(r4)
    beq r3, r0  S2  # Random.
    lb r3, 1(r4)
S2: beq r3, r0  S3  # Random.
    addi r4, r4, 2
S3: nop

    lw r1,0(r2)
C2: beq r1, 0  SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

☐ Global: accuracy on `C2`:

☐ Explain using GHR values.

☐ Global: warmup time on `C2`:

☐ Explain using GHR values.

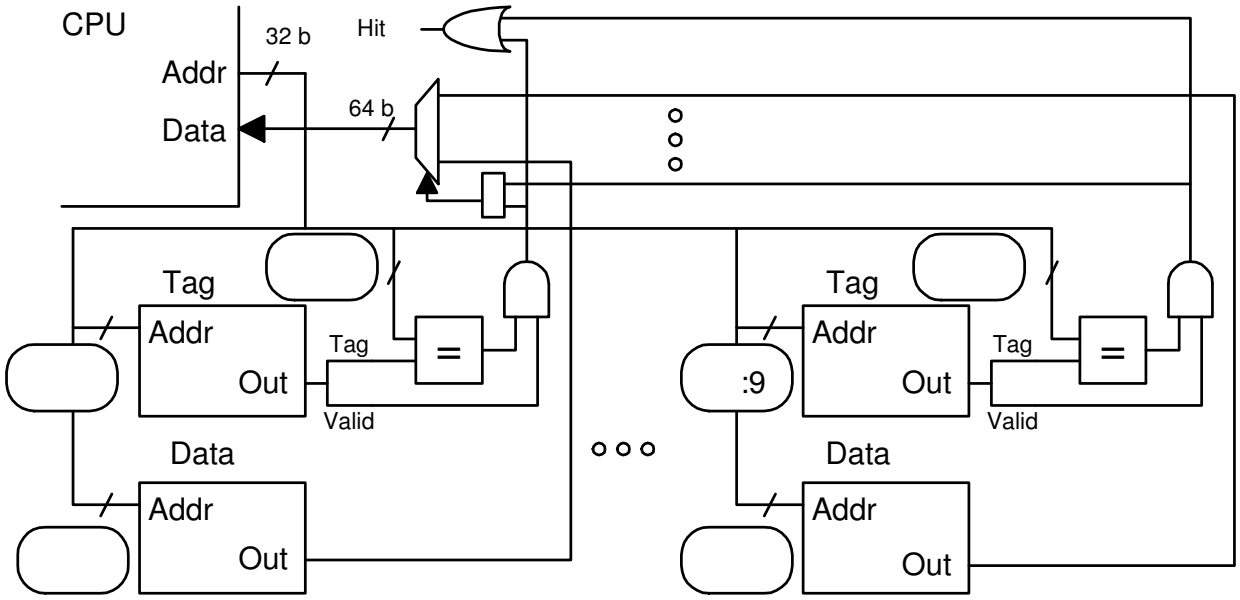
☐ Global: smallest history size needed for 100% accuracy on `C2`:

☐ Explain using GHR values.

Problem 3: The diagram below is for a 64-MiB (2^{26}-character) 16-way set-associative cache on a system with the usual 8-bit characters. (20 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 3, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;  // = 2^27

for(i=0; i<ILIMIT; i++) sum += a[ i ];
for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

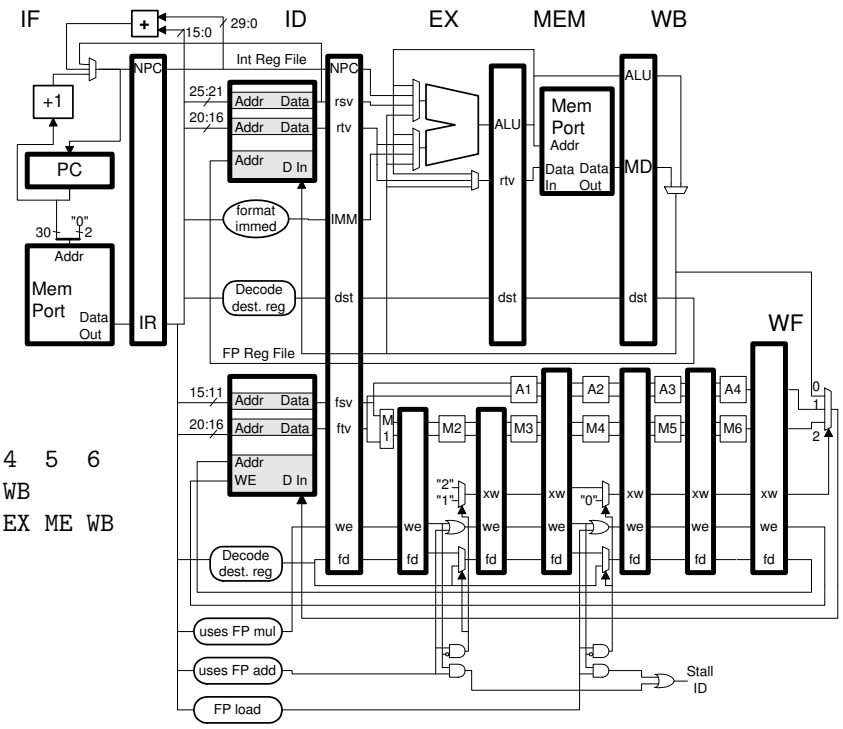
(*c*) The slightly different code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ Find the hit ratio. Very briefly explain.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;

for(i=0;        i<ILIMIT; i++) sum += a[ i ];
for(i=ILIMIT-1; i>=0;     i--) sum += a[ i ];
```

Problem 4: Answer each question below.

(*a*) The execution of some code fragments on the illustrated implementation appear below. Explain why each execution is impossible and show a corrected pipeline execution diagram. (7 pts)




```
# Cycle         0  1  2  3  4  5  6
lw r2, 0(r4)    IF ID EX ME WB
add r1, r2, r3     IF -> ID EX ME WB
```

☐ Fix.

☐ Was impossible because:

```
# Cycle         0  1  2  3  4  5  6  7
lw r2, 0(r4)    IF ID EX ME WB
add r1, r2, r3     IF ID -> EX ME WB
xor r5, r6, r7        IF ID -> EX ME WB
```

☐ Fix.

☐ Was impossible because:

```
add.d f0, f2, f4  IF ID A1 A2 A3 A4 ME WB
```

☐ Fix.

☐ Was impossible because:

(*b*) For each feature below indicate whether it is usually a feature of the ISA or the implementation, and explain why it's not a feature of the other. (7 pts)

Feature: *The opcode can be found in bits 31:26.*

☐ ISA or Implementation?

☐ Why not the other?

Feature: *The* `add` *in the code below will stall for one cycle.*

```
lw  r1, 0(r2)
add r3, r1, r4
```

☐ ISA or Implementation?

☐ Why not the other?

Feature: *Two consecutive delayed (as in MIPS) branches will yield unpredictable results.*

```
bneq r1, r2 DEST1
beq r3, r4 DEST2
addi r5, r6, r7
```

☐ ISA or Implementation?

☐ Why not the other?

Feature: *Integer addition overflow raises exception.*

☐ ISA or Implementation?

☐ Why not the other?

(*c*) Consider the use of a packed-operand 8-bit add (of the type described in class) to speed up the code fragments below. For each fragment indicate whether it's certainly feasible, feasible with certain assumptions, or not feasible. (6 pts)

```
extern char *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☐ Circle One: No. Yes! Yes, assuming ...

☐ Explain.

```
extern int *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☐ Circle One: No. Yes! Yes, assuming ...

☐ Explain.

```
extern char *a, *b, *c;
for(i=1; i<1024; i++) a[i] = a[i] + a[i-1];
```

☐ Circle One: No. Yes! Yes, assuming ...

☐ Explain.

(*d*) MIPS and other RISC ISAs typically have instructions using displacement addressing but lack register deferred addressing. (See the examples below.)

```
lw r1, 4(r2)   # Displacement addressing.
lw r1, (r2)    # Register deferred addressing.
```

Given that RISC and CISC ISAs have displacement addressing:

☐ (5 pts) Why is register deferred addressing helpful for CISC but not helpful for RISC?

(*e*) Dead-code elimination (DCE) is a common compiler optimization.

☐ (5 pts) What is it? Illustrate using an example.

25 Fall 2005

Name _____

Computer Architecture
EE 4720
Midterm Examination

Monday, 24 October 2005, 12:40–13:30 CDT

Problem 1 _____ (50 pts)

Problem 2 _____ (50 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: The partially completed routine on the next page is called with the address of a string which may contain characters that look like letters, for example, "`g0!1y`." The routine should replace those characters with the letters they look like; the example would be converted to "`golly`."

The string DMAP (see the next page) specifies the look-alikes. It specifies a letter (always lower case) followed by one or more look-alike characters followed by a comma (possibly followed by another group). For example, "`o0,l1!,`" indicates that the letter oh's look-alike is the digit zero and the letter el's look-alikes are the digit 1 and an exclamation point.

The routine on the next page is almost finished. The part at the end uses a look-up table to translate the string, and the part at the beginning has started setting up the look-up table. Write the code that finishes setting up the look-up table based on DMAP. [50 pts]

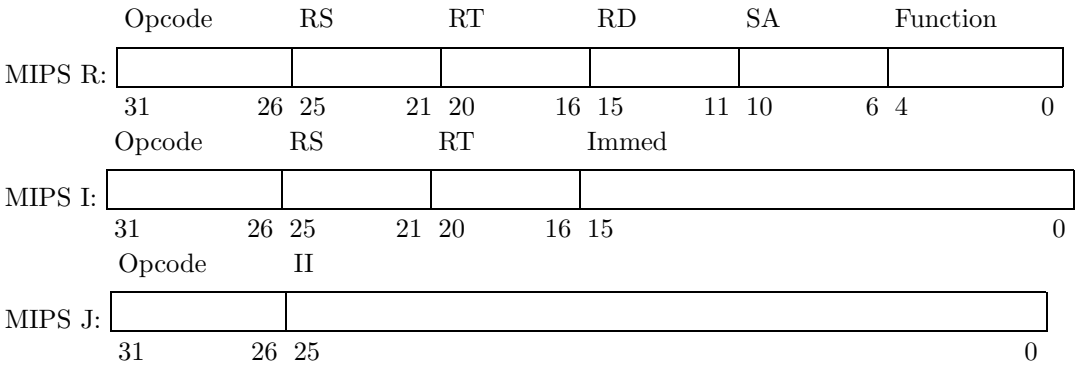
- ☐ Write the code that sets up the look-up table.
- ☐ The code must be reasonably efficient.
- ☐ The only synthetic instructions that can be used are `nop` and `la`.

Use the next page for the solution.

```
        ## Register Usage
        #
        # $a0: Procedure call argument. Address of string to translate.
        # There are no return values.

DMAP:   .asciiz "o0,l1!,c([,t+,s$,"  # Translate 0->o, 1->l, !->l, (->c, etc.

LUT:    .space 256

        la $t0, LUT
        addi $t1, $0, 255

        # First, initialize look-up table so every character is mapped to itself.
        #
LOOP0:  add $t3, $t0, $t1         # Compute address of LUT entry.
        sb $t1, 0($t3)            # Write default entry. (A->A, B->B, etc.)
        bne $t1, $0  LOOP0
        addi $t1, $t1, -1

        # Next, set up look-up table based on DMAP string. (Finish in solution.)
        #
        la $t0, DMAP
        la $t1, LUT
        addi $t5, $0, 44   # ','  Comma character separates groups.

        # Start solution here. (Can be done in 9 insn.)
```

```
        # Use the look-up table to translate the string.
        la $t1, LUT
LOOP3:  lb $t2, 0($a0)       # Load character of string.
        beq $t2, $0, EXIT2
        add $t2, $t2, $t1    # Compute address of look-up table entry.
        lb $t2, 0($t2)       # Load translated character.
        sb $t2, 0($a0)       # Write translated character to string.
        j LOOP3
        addi $a0, $a0, 1

EXIT2:  jr $ra
        nop
```

Problem 2: Answer each question below. The instruction format descriptions are provided for reference.



(*a*) A proposed new MIPS branch instruction compares a register value to a constant to determine if the branch should be taken. For example, the branch below is taken if the contents of `t1` is 123. [10 pts]

```
beqi $t1, 123,  TARGET
nop
```

☐ Why isn't it feasible to code such an instruction using any of the existing MIPS formats?

☐ How could a conditional control transfer instruction that compared a register to an immediate be coded using the MIPS formats? (The instruction would be different than `beqi`.)

☐ Show the instruction in assembly language.

(*b*) Show the results of addition for the 32-bit data types shown below. [10 pts]

☐ Unsigned Integer:

```
  0x0999
+ 0x0109
```

☐ BCD:

```
  0x0999
+ 0x0109
```

☐ Packed 4-bit integers with saturating arithmetic.:

```
  0x0999
+ 0x0109
```

(*c*) The MIPS ISA specifies that the sa (shift amount) field in the `add` instruction must be zero (an `add` instruction with a non-zero sa field value should cause an execution error). [10 pts]

☐ Describe a difficulty that might have arisen if the MIPS ISA had specified that implementations should ignore the sa field in an `add` instruction (and so the `sa` field could contain any value).

☐ Describe a difficulty that might have arisen if the MIPS ISA said nothing about the `sa` field in an add instruction.

☐ Describe a difficulty that might have arisen if the MIPS ISA did specify that sa must be zero but did not say what should happen if the field were non-zero.

(*d*) Consider a benchmark suite that's similar to SPECcpu in that the tester is responsible for compiling the programs but that unlike SPEC, specifies that the tester should not use any optimizations when compiling the benchmarks. [10 pts]

☐ Compared to SPECcpu base and peak (result) scores, how useful would such results be? Explain.

(*e*) Using a new compiler optimization a program's dynamic instruction count is cut in half. The values for CPI, IC, and ϕ are available for a run of the program compiled without the new optimization. Other than the instruction count, nothing has been measured for the program using the new optimization. How useful is the CPU performance equation for estimating the run time with the new optimization on the same system in this situation? Explain. [10 pts]

Name _____

Computer Architecture
EE 4720
Final Examination

13 December 2005, 12:30–14:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(17 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(33 pts)
Exam Total	_____	(100 pts)

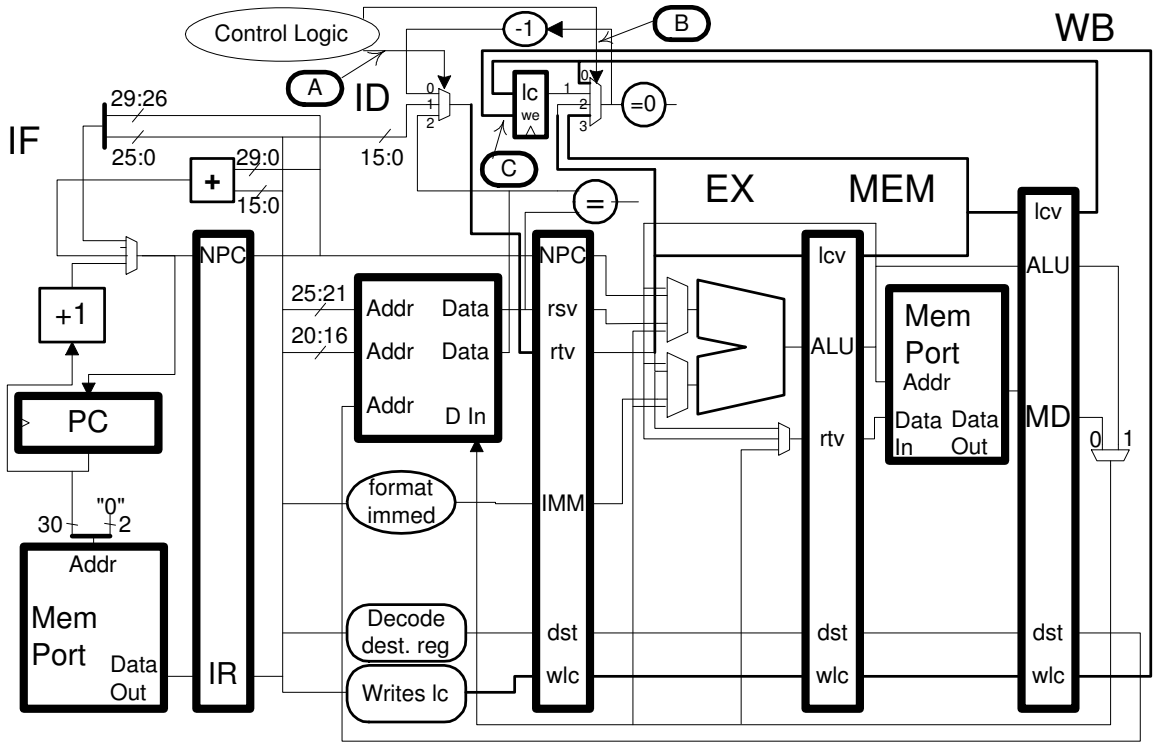
Alias _____

Good Luck!

Problem 1: The implementation of a version of MIPS with loop count instructions is shown below, these are the same instructions used in Homework 5. Instruction `mtlc rt` moves the contents of the `rt` register into a special loop count register, `lc`, and instruction `mtlci immed` moves a 16-bit immediate, `immed`, into `lc`. The loop-counted branch `bclz` is taken unless the `lc` register is zero, it also decrements `lc`. (In the diagram the upper input to the `lc` register is the data input and the lower input, `we`, is a write enable.)

Three wires in the implementation are labeled (A , B , and C). Show the values present on those wires for each cycle of the illustrated execution of the program in the space provided. Leave a value blank if the value has no effect, assume that instructions before and after the illustrated code are nops. Note that two iterations of the loop are shown. (15 pts)

```
 # Cycle            0    1    2    3    4    5    6    7    8
 mtlci 100          IF   ID   EX   ME   WB
LOOP:
 bclz LOOP               IF   ID   EX   ME   WB
 add r2, r2, r3               IF   ID   EX   ME   WB
# (2nd iteration)
 bclz LOOP                         IF   ID   EX   ME   WB
 add r2, r2, r3                         IF   ID   EX   ME   WB
 # Cycle            0    1    2    3    4    5    6    7    8
 A:                                                      <- ANSWER HERE

 B:                                                      <- AND HERE

 C:                                                      <- AND HERE TOO
 # Cycle            0    1    2    3    4    5    6    7    8
```



Problem 2: Complete the pipeline execution diagrams for the different MIPS implementations below. Pay attention to the type of implementations in each part, they're not all the same. Don't forget to **check for dependencies.**

☐ (5 pts) Scalar (not superscalar), statically scheduled, as illustrated in the previous problem.

```
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
add r1, r2, r3

lw r4, 6(r1)

xor r7, r4, r8
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

In all of the parts below the floating-point multiply unit has four stages and is fully pipelined, the stage labels are *M1-M4*. *The floating-point add unit is two stages and fully pipelined, the stage labels are* *A1 and A2*.

☐ (5 pts) Scalar, statically scheduled, full set of bypass paths.

```
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
mul.d f2, f4, f6

add.d f8, f2, f10

mul.d f2, f4, f14

sub.d f12, f2, f10
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

☐ (5 pts) Two-way superscalar, statically scheduled, full set of bypass paths. No alignment restriction on instruction fetches.

```
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
mul.d f2, f4, f6

add.d f8, f2, f10

mul.d f2, f4, f14

sub.d f12, f2, f10
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

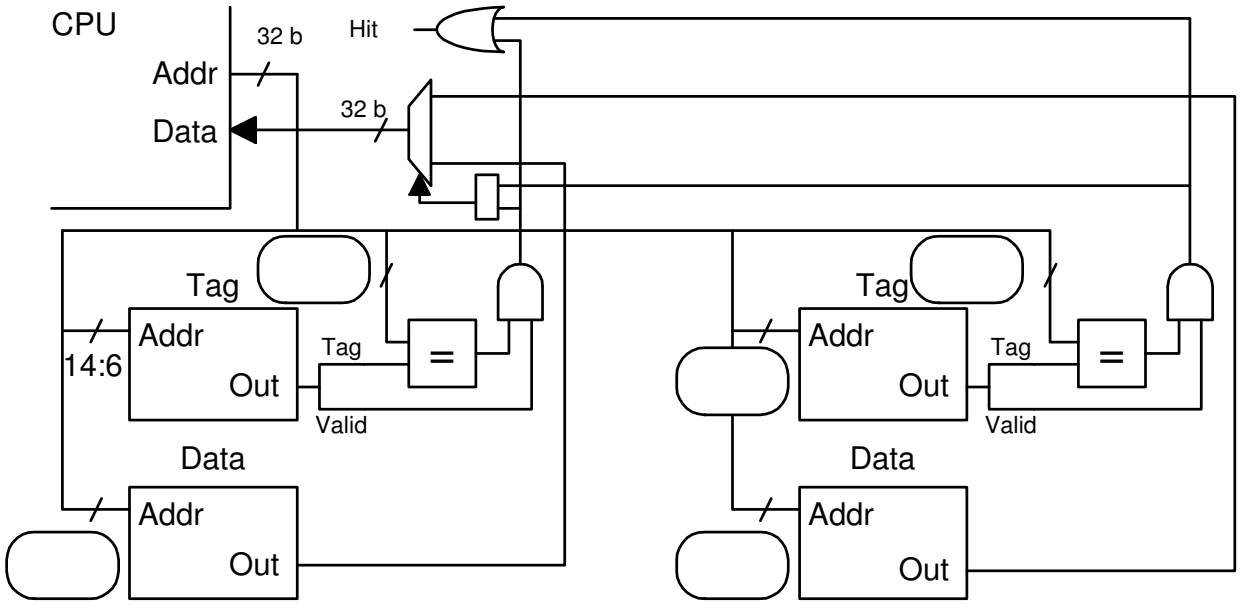
☐ (5 pts) Two-way superscalar, dynamically scheduled, with a full set of bypass paths. No alignment restriction on instruction fetches.

```
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
mul.d f2, f4, f6

add.d f8, f2, f10

mul.d f2, f4, f14

sub.d f12, f2, f10
# Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

Problem 3: The diagram below is for a cache on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: ☐ ☐ ☐

☐ Cache Capacity (Indicate Unit!!):

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address: ☐ ☐ ☐

Problem 3, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ (5 pts) What is the hit ratio running the code below?

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j;

for(j=0; j<3; j++)
  for(i=0; i<32; i++)
    sum += a[ i ];
```

(*c*) The slightly different code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ (5 pts) Set `ILIMIT` to the smallest value that will fill the cache.

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i;
int ILIMIT =                       <- ANSWER HERE

for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

Problem 4: Answer each question below.

(*a*) An n-way superscalar processor need not have n copies of everything that an ordinary scalar implementation has.

☐ (5 pts) Which of the following is it most important that an n-way superscalar processor **does** have n of (explain):

- Mem-stage memory ports.
- Write ports to the register file.
- Floating-point adders.

(*b*) In the dynamically scheduled MIPS implementation covered in class:(5 pts)

☐ When is a physical register allocated (removed from the free list and assigned to an instruction)?

☐ When is the physical register put back in the free list. Show an example that includes register numbers.

(*c*) Describe the contents of an Itanium (IA-64) bundle or a bundle in a typical VLIW ISA.

☐ (5 pts) Bundle Contents

Problem 5: Answer each question below.

(*a*) When a company wants to benchmark a new computer using SPECcpu it gets the benchmark suite from SPEC, compiles the code, runs the benchmarks, and proudly publishes the results. (The last step is optional.)

Below are two variations on this procedure, for each explain how the results might differ **from those obtained using the usual SPEC procedure**, and explain how useful the results would be to the typical user of SPECcpu results. (7 pts)

(1) SPEC gives a compiler (in addition to the usual material) to the company and allows the company to compile and run the suite following the usual SPECcpu rules (but using SPEC's compiler).

☐ Compared to actual SPEC rules and to (2), how might the results differ? Explain.

☐ Compared to actual SPEC rules and to (2), how useful would results be? Explain.

(2) The company gives the new computer and a compiler to SPEC and SPEC compiles and runs the benchmark suite using the usual SPECcpu rules.

☐ Compared to actual SPEC rules and to (1), how might the results differ? Explain.

☐ Compared to actual SPEC rules and to (1), how useful would results be? Explain.

(*b*) There is much less advantage in unrolling one of the loops below when the code is to run on a two-way statically scheduled superscalar MIPS implementation. *Note: The original problem did not mention the code was to run on a superscalar implementation.*(6 pts)

☐ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_A:
 lw r1, 0(r2)
 lw r1, 0(r1)
 add r3, r1, r3
 addi r2, r2, 4
 bneq r9, r0 LOOP_A
 addi r9, r9, -1
```

☐ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_B:
 lw r1, 0(r2)
 add r3, r3, r1
 lw r2, 4(r2)
 bneq r9, r0  LOOP_B
 addi r9, r9, -1
```

(*c*) Which are smaller, RISC programs or CISC programs. Provide some instruction examples to illustrate your answer. (5 pts)

☐ Example and explanation.

(*d*) The cost of implementing a rarely used instruction is deemed too high and so is omitted from an implementation. How can an operating system enable the implementation to run code that uses the rare instruction without modifying the code or examining it in advance. (5 pts)

(*e*) BCD data types were once popular ISA features. (5 pts)

☐ Were BCD data types absolutely necessary?

☐ What were the reasons for including BCD data types in an ISA?

☐ Why might a BCD data type be considered old fashioned and so not worthy of adding to an ISA now?

(*f*) MIPS-I is big endian but more recent MIPS versions can run in either big-endian or little-endian modes.

☐ (5 pts) Complete (possibly modifying) the MIPS program below so that it writes `v0` with a 0 if it is running on a system using big endian byte order or a 1 if running on a system using little-endian byte order.

```
# At finish: $v0: 0, if big endian;  1, if little endian.
lui $t1, 0x1122
ori $t1, $t1, 0x3344
sw $t1, 0($t2)
```

26 Spring 2005

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 1 April 2005, 11:40–12:30 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

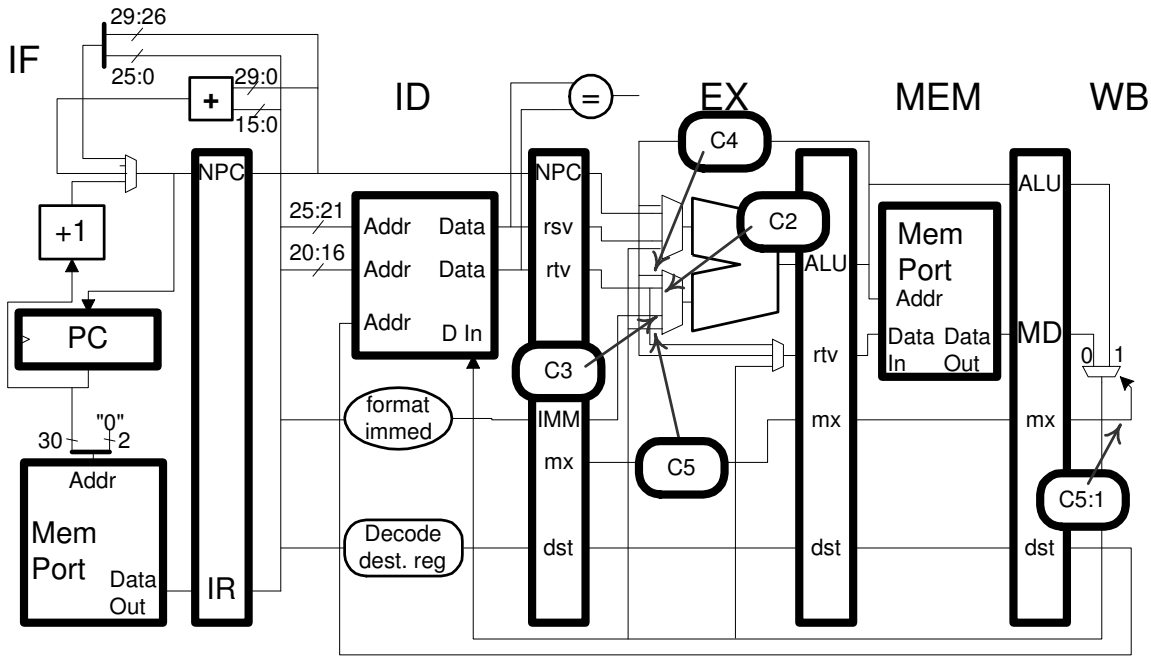
Problem 1: In the diagram below some wires are labeled with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly.

- There are no branches or other control-transfer instructions.

☐ [15 pts] Write a program consistent with these labels.

☐ All register numbers must be made up; **use as many different register numbers as possible** while still being consistent with the labels.

☐ [10 pts] Why would there be no solution to the problem if the C5:1 label in WB were changed to C5:0?



Cycle:	0	1	2	3	4	5	6	7
	IF	ID	EX	ME	WB			
		IF	ID	EX	ME	WB		
			IF	ID	EX	ME	WB	
				IF	ID	EX	ME	WB
Cycle:	0	1	2	3	4	5	6	7

Problem 2: In the diagram on the next page some wires are labeled with cycle numbers and corresponding values. For example, C1:8 indicates that at cycle 1 the pointed-to wire will hold an 8. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly.

- Unlike other problems of this type all registers are different and there are no dependencies.
- The code contains at least one floating-point add and one floating-point multiply.

☐ [10 pts] Write a program consistent with these labels; if a register number cannot be determined use rX or fX.

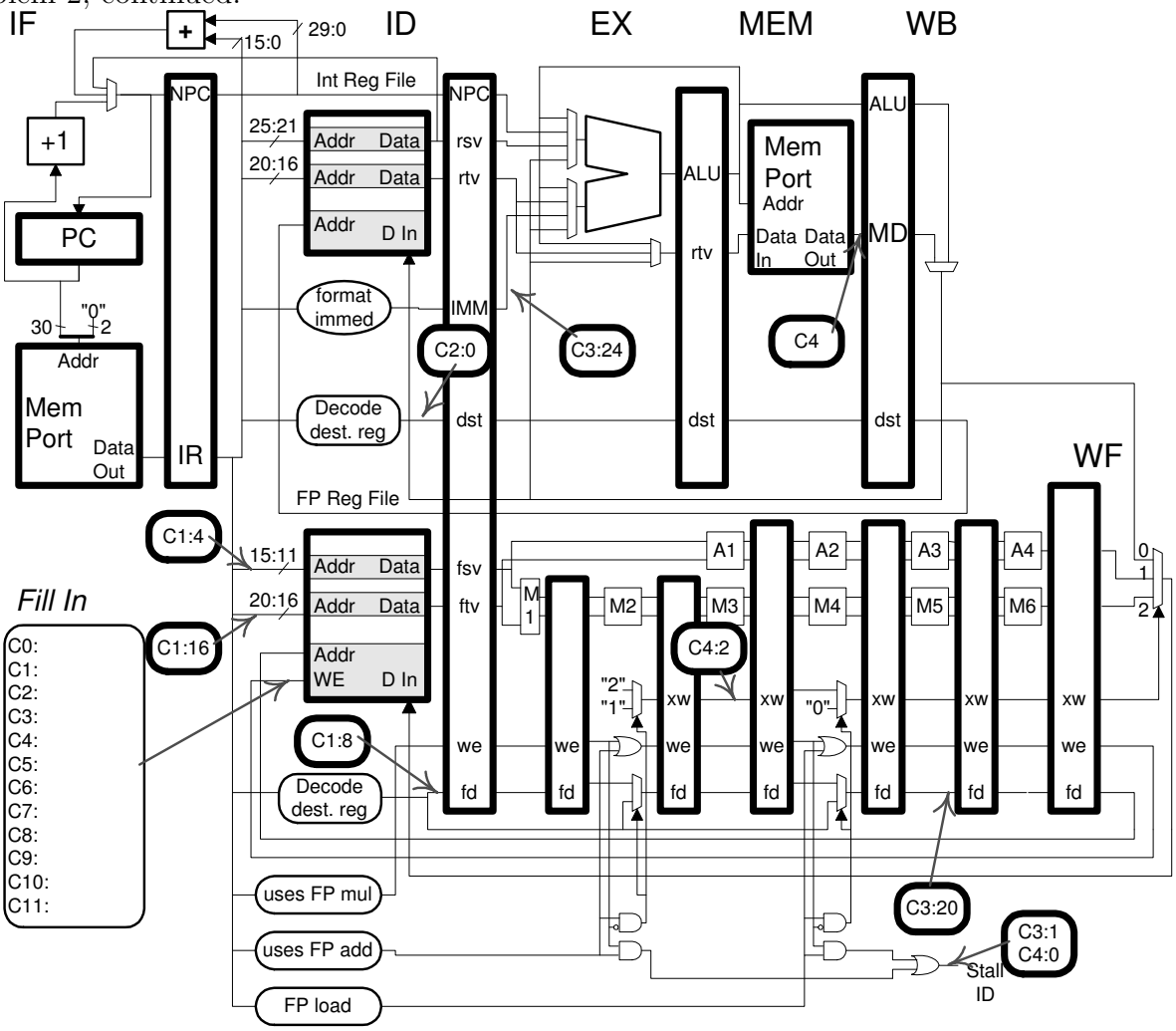
☐ [5 pts] Complete the pipeline execution diagram.

☐ [5 pts] Fill in the box on the lower-left of the diagram.

☐ [5 pts] Your answer should have a FP multiply instruction. Explain how you knew it was a multiply. (Don't answer "because I already found the add.")

☐ [5 pts] Your answer should have a FP add instruction. Explain how you knew it was an add. (Don't answer, "because I already found the multiply.")

Problem 2, continued:



Cycle:	0	1	2	3	4	5	6	7	8	9	10	11
	IF	ID										
		IF	ID									
			IF	ID								
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11

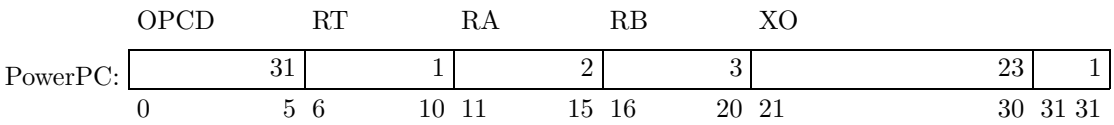
Be sure to complete all parts, including the justification for add and multiply.

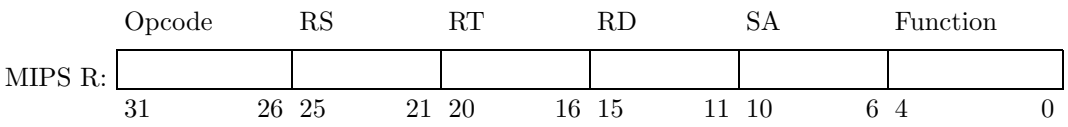
Problem 3: Answer each question below.

(*a*) The encoding of the PowerPC `lwzx r1, r2, r3` instruction is illustrated below, followed by the Type-R MIPS instruction format. [6 pts]

☐ If a field in the illustrated PowerPC instruction format has a counterpart in the MIPS R format draw an arrow between the fields. For example, an arrow should be drawn from OPCD to Opcode. Pay close attention to the register fields.

☐ If fields connected by an arrow are different sizes explain the significance of the difference. (What advantage or disadvantage would the format with the larger field have, if any.)





(*b*) Suppose that your favorite program is both in the SPEC CPU2000 suite and in the suite used by a popular personal computing magazine and that the input data used for testing the program are also identical. You are considering buying two computers, A, and B. The spec testing shows that your favorite program is faster on A but the magazine's test shows that it is faster on B. The exact same model of machine A is used for running the spec test and the magazine's test, likewise for B. [6 pts]

☐ Provide a possible reason for the discrepancy, the reason should help explain why SPEC is used by the CPU research and development community. *Note: In the original exam the text after discrepancy was offered as a hint.*

(*c*) SPEC members and associates are drawn from computer manufacturers, other industries, academia, and elsewhere. [6 pts]

☐ Would you trust SPEC benchmark results more (or less) if computer manufacturers were not allowed to join SPEC? Explain.

(*d*) In class we saw that the optimized π program (reproduced below) ran faster than the unoptimized version (good) but also had higher CPI than the unoptimized version (bad). *Note: the words "than the unoptimized version" was not included in the original exam.* [7 pts]

☐ Why was the CPI higher?

Hint: It has to do with something you learned early in grade school.

```
double i, sum = 0;
for(i=1; i<5000; )
  {
    sum = sum + 4.0 / i;
    i += 2;
    sum = sum - 4.0 / i;
    i += 2;
  }
```

Problem 4: Answer each question below.

(*a*) MMX and VIS are examples of ISA extensions that add packed-operand data types and instructions.[7 pts]

☐ How do packed-operand data types and instructions improve performance?

☐ Show a code example that can use such instructions and explain how they would be used.

(*b*) Name an advantage and a disadvantage that RISC's fixed-size instructions have over CISC's variable length instructions.[6 pts]

☐ Advantage:

☐ Disadvantage:

(*c*) A trap instruction is something like a subroutine call to the operating system. [7 pts]

☐ So why couldn't it just specify the address of the trap handler?

☐ What does the trap instruction specify (in SPARC but not MIPS) in place of an address and how does execution actually reach the trap handler.

Name _____

Computer Architecture
EE 4720
Final Examination

10 May 2005, 15:00–17:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (25 pts) The `madd.s fd, fr, fs, ft` instruction writes floating-point register `fd` with (`fr` × `fs`) + `ft`. The `fr` field is in bits 25:21, the other fields are in their usual places. The instruction is used in the code below:

```
# With ordinary instructions:
 mul.s f1, f2, f3
 add.s f1, f1, f4

# With a madd.s instruction:
 madd.s f1, f2, f3, f4
```

(*a*) Add datapath connections (including any connections to the register file) to the implementation on the next page (also shown below) to implement the `madd.s` instruction. The code below should execute as shown (pay attention to `f4`).

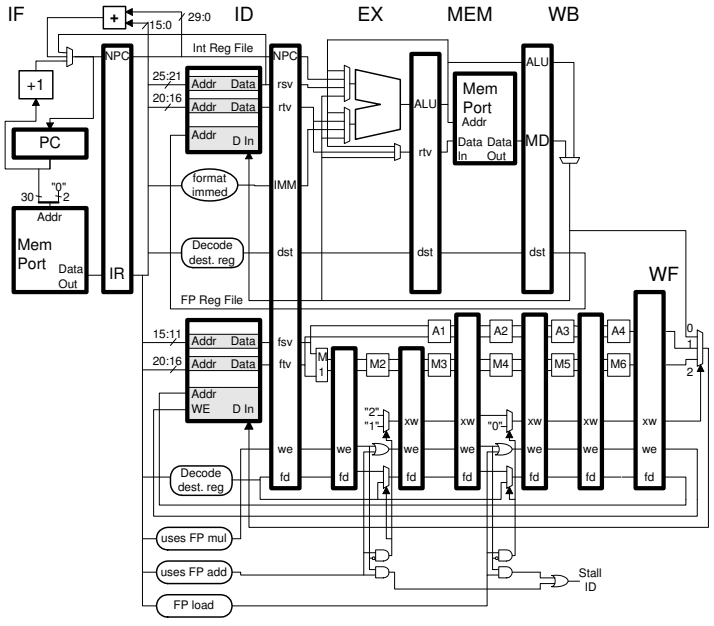
```
 madd.s f1, f2, f3, f4 IF ID M1 M2 M3 M4 M5 M6 A1 A2 A3 A4 WF
 lwc1 f4, 0(r2)           IF ID EX ME WF
 add.s  f5, f6, f7           IF ID A1 A2 A3 A4 WF
```

- Use the existing multiplier and adder.
- It should still be possible to execute ordinary floating point multiply and add instructions.

(*b*) Modify the logic so that `xw`, `we`, and `fd` work correctly for the `madd.s` instruction (and continue to work correctly for existing instructions).

(*c*) Without a `madd.s` instruction there is no possible structural hazard on `WF` in the implementation below for multiply because multiply is the longest latency instruction implemented. With `madd.s` that's no longer true, add control logic to detect this new structural hazard and generate the Stall ID signal.

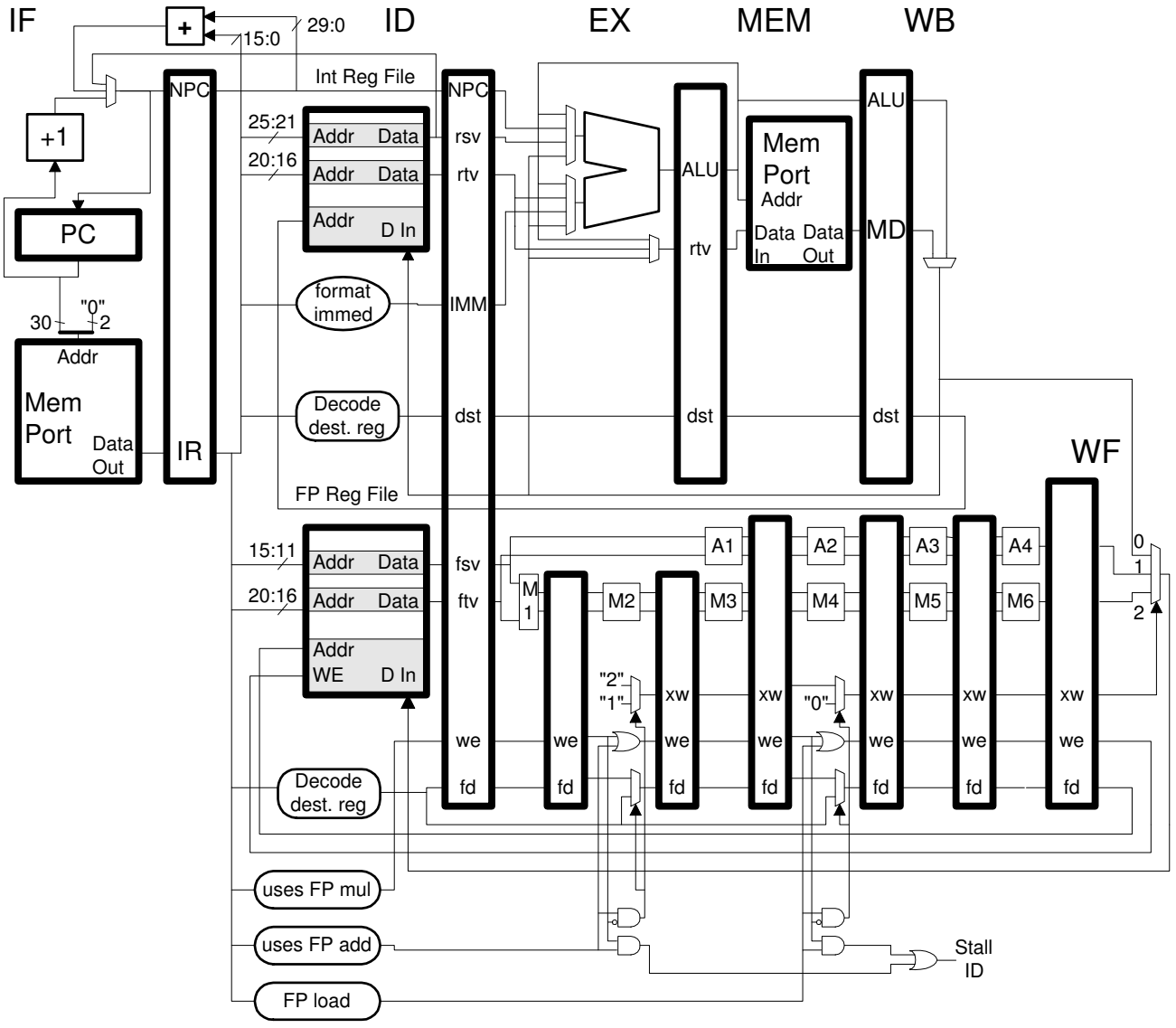
SOLVE PROBLEM ON THE NEXT PAGE.



SOLVE PROBLEM ON THE NEXT PAGE.

Problem 1, continued:

- ☐ Datapath for `madd.s`, don't break `add.s` or `mul.s`.
- ☐ Code on previous page must run as shown.
- ☐ Modify `xw`, `we`, and `fd` for `madd.s`, don't break other instructions.
- ☐ Detect and handle new structural hazard when `mul.s` in ID.



Problem 2: (15 pts) The execution of a MIPS program on a dynamically scheduled system using method 3 appears below. Complete the ID Register Map, Commit Register Map, and Physical Register File tables for registers `f2` and `f4`.

- The initial value of `f2` is 2.0, the mul writes 2.1, ldc1 writes 2.2, and sub.d writes 2.3. The initial value of `f4` is 4.0, the add writes 4.1. Make up physical register numbers as needed.

☐ As always, show table contents.

☐ Show where registers are removed from and placed back in the free list.

☐ Show initial values.

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mul.d f2, f4, f6	IF	ID	Q	RR	M1	M2	M3	M4	WF	C					
add.d f4, f2, f10		IF	ID	Q				RR	A1	A2	A3	WF	C		
ldc1 f2,0(r1)			IF	ID	Q	EA	ME	WF						C	
sub.d f2, f2, f8				IF	ID	Q	RR	A1	A2	A3	WF				C

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ID Register Map															

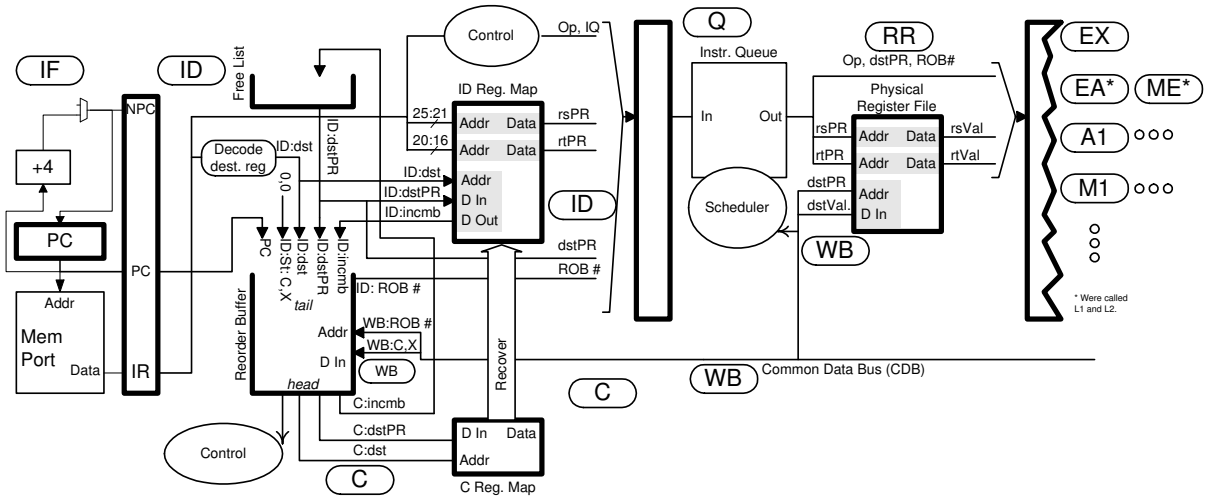
# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Commit Register Map															

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Physical Register File															

# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

HARDWARE SHOWN ON NEXT PAGE.

Problem 2, continued: Dynamically scheduled processor shown for reference.



Problem 3: (20 pts) The code below runs on three systems, one using a bimodal predictor (B) with a 2^{10}-entry BHT, a local predictor (L) with a 2^{10}-entry BHT and a 16-bit local history, and a global predictor (G) with a 16-bit GHR. The outcomes of B2 are shown, pay attention to where B1's outcomes would be located.

```
LOOP:
SHORT:
# B1 Iterates 10 times
B1: bne r3,r4 SHORT
addi r4, r4,1
...
B2: beq r1, r2 SKIP     N    N    T    N    T    N    N    T    N    T ...
...
SKIP:
j LOOP
nop
```

(*a*) Find the prediction accuracy of each predictor on each branch. Briefly explain.

☐ Accuracy predicting B1 on B:

☐ Accuracy predicting B1 on L:

☐ Accuracy predicting B1 on G:

☐ Accuracy predicting B2 on B:

☐ Accuracy predicting B2 on L:

☐ Accuracy predicting B2 on G:

(*b*) Determine the number of table entries used by branch B2 on each predictor:

☐ Entries for B2 on B:

☐ Entries for B2 on L:

☐ Entries for B2 on G:

(*c*) Suppose the history sizes were reduced.

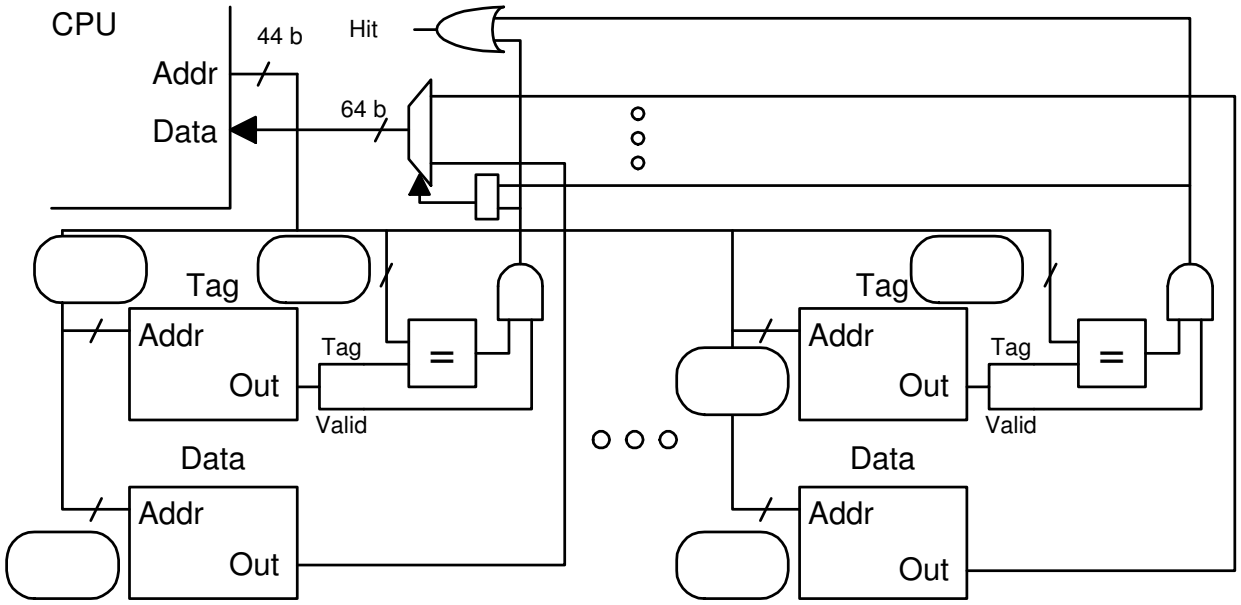
☐ What is the smallest local history size for which the accuracy on B2 will be unchanged (from the previous answer) when using the local predictor. Briefly explain.

☐ What is the smallest global history size for which the accuracy on B2 will be unchanged when using the global predictor. Briefly explain.

Problem 4: The diagram below is for a 256-kiB (2^{18} byte) 4-way set-associative cache with a line size of 16 characters on a system with 8-bit characters. (20 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram. *Pay attention to the width of the CPU address port.*



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 4, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☐ What is the hit ratio?

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j, ILIMIT = 0x1000000;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

The code in the problems below is to be run on a cache of unknown configuration, though it will be one of the types discussed in class (direct mapped or set-associative). In all cases the cache is empty when the program starts. Routine `get_miss_count()` *returns the number of cache misses at the time of the call (and does not cause any misses).*

(*c*) Complete the code so that `line_size` is assigned the correct line size based on the number of misses encountered and the nature of the code. Assume that all misses are due to access to `a`.

```
int ILIMIT = 0x10000;
char *a;
int miss_count_before = get_miss_count();
for(i=0; i<ILIMIT; i++) sum += a[i];
int miss_count_during = get_miss_count() - miss_count_before;

int line_size =                                          // FILL IN
```

(*d*) Complete the code so that `associativity` is aptly assigned. Assume that the cache is smaller than 256 MiB (2^{28})(but the exact size is not known), that the associativity is no larger than 64, and that there is a 64-bit address space.

```
int ISHIFT =                                             // FILL IN

int JSHIFT =                                             // FILL IN

char *a;  // Pointer to a really large array.
int miss_count_before = get_miss_count();

for(i=0; i<64; i++) sum += a[ i << ISHIFT ];

for(j=                          )                        // FILL IN
   sum += a[ j << JSHIFT ];

int miss_count_during = get_miss_count() - miss_count_before;

int associativity =                                      // FILL IN
```

Problem 5: Answer each question below.

(*a*) Suppose in a five-stage statically scheduled MIPS implementation (like the one in class) an instruction could raise an exception in the WB stage. Explain why that would make precise exceptions for that instruction impossible. Use a code example to explain what should happen for a precise exception and why its impossible (or very difficult) if the exception is raised in WB. (5 pts)

(*b*) Arrange the ISA families below in order by code (program) size, the one for which programs are smallest should be first. Starting at the second ISA in the arranged list, explain why code size is larger than the ISA above. (That is, provide three reasons why code size is larger.) (5 pts)

ISA families (in alphabetical order): CISC RISC Stack VLIW

☐ ISA families in code size order.

☐ Reasons for size differences.

(*c*) Deciding on a line size for a cache is a tough decision. (5 pts)

☐ Describe the behavior of programs that run better on caches with smaller line sizes.

☐ Describe the behavior of programs that run better on caches with larger line sizes.

(*d*) Answer the following question on cost.(5 pts)

☐ Name two parts of an n-way superscalar processor that cost about n times as much as a comparable scalar processor.

☐ Name two parts of an n-way superscalar processor that cost about n^2 times as much as a comparable scalar processor.

27 Fall 2004

Name _____

Computer Architecture
EE 4720
Midterm Examination 1.0.3

Friday, 22 October 2004, 10:40–11:30 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The routine below is to do a binary search on an array and return 1 if the item is found, 0 otherwise. Complete the MIPS program using the C++ code as a guide. [30 pts]

☐ The only allowable synthetic instruction is `nop`.

☐ Fill as many delay slots as possible. The order of the assembly code **does not** have to match the order of the C code.

```
#  unsigned int delta = size;  int pos = 0;
#  while( true ) {
#      delta = delta >> 1;
#      int next_pos = pos + delta;
#      int e = array[next_pos];
#      if( e == target ) return 1;
#      if( !delta ) return 0;
#      if( e < target ) pos = next_pos;
#    }
       # CALL VALUE: $a0,  array: Address of array. Each element is a WORD-SIZED int.
       # CALL VALUE: $a1,  size: Number of elements in array. A power of 2.
       # CALL VALUE: $a2,  target: Value to find.
       # RETURN VALUE: $v0,  1, if target in array; 0, if not in array.
       # Can modify registers $a0-a3, $t0-$t9
```

Problem 2: In MIPS and many other ISAs floating-point instructions read and write a FP register file while most other instructions read and write general-purpose (integer) registers. Suppose MIPS-I were modified so that FP instructions used the same registers as the integer instructions. The registers are still 32 bits and can be used in pairs by instructions operating on double-precision operands. The new version of MIPS is not compatible with MIPS-I, don't worry about that.

(*a*) For each MIPS-I instruction below indicate whether it will be retained in the new ISA, "KEEP", or will not be needed, "OMIT." Also add any new instructions that might be needed. If an instruction is kept but operates differently describe the difference. [15 pts]

☐ Indicate KEEP or OMIT.

☐ If omitted show an existing instruction that would be used in its place.

☐ If kept, explain how it would operate differently.

lwc1:

ldc1:

cvt.w.d:

cvt.d.w:

mfc1:

mtc1:

add.s: Keep, but operates on integer registers.

add.d:

c.lt.s:

c.le.d:

bc1t:

Hint: In `bc1t` *b is for branch, t is for true.*

(*b*) If your answer for `bc1t` above was "OMIT" describe below how the instruction can be kept (possibly modified). If your answer was "KEEP" explain how it can be omitted. This will affect other instructions, list them and explain how they are affected.

Problem 3: Using an interesting technique the program below returns the value of register X, where X is a number in register `$a0`. For example, if `$a0` held an 8 the program would return the value of `$8` (or `$t0` using the register name) in register `$v0`. Consider:

```
        jal copyreg
        addi $a0, $0, 8
        # At this point $v0 has contents of $t0.

        addi $v0, $t0, 0   # $t0 is $8, register number 8
        # At this point $v0 has contents of $t0.
```

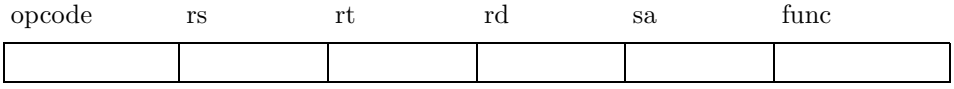
From the code above it looks like the `copyreg` routine is doing things the hard way. The one advantage is that it can copy a source register known only at run time, or that might change. The `addi` instruction always copies `$t0`. *Note: In the original exam the description and example above were not included.*

(*a*) Modify the program so that it takes a second call value, `a1`, which holds a second register number. The return value is put in that register. For example, if `a0` holds a 12 and `a1` holds a 3 then the routine copies the value in `r12` to `r3`. [17 pts]

☐ Show additions and changes. The added code must use a similar technique to the existing code.

Hint: Can be solved with two added instructions and a few minor modifications.

```
        # CALL VALUE:   $a0   A register number. (Except v0)


        # RETURN VALUE: $v0   The value in reg number X, where X is val in a0.

copyreg:
        la $v0, template    # Put address of label "template" into $v0.

        lw $v0, 0($v0)

        sll $a0, $a0, 21

        add $a0, $v0, $a0

        la $v0, mod

        sw $a0, 0($v0)

        jr $ra

mod:
        add $v0, $0, $0

template:
        add $v0, $0, $0
```



Format R: 31 26 25 21 20 16 15 11 10 6 5 0

Problem 3, continued:

(*b*) Suppose one were considering adding a new machine instruction, `copyreg`, that does the same thing as the routine above (before modification for part 1). Provide arguments for and against adding the new instruction. The arguments can include made-up data as long as it does not conflict with what has been given in this problem or in class. At least one of the arguments must refer to features of the code from part 1. [8 pts]

☐ Provide an argument in favor of adding the instruction.

☐ Provide an argument against adding the instruction. Do not simply reverse an assumption made above.

Problem 4: Answer each question below.

(*a*) A company is developing an implementation of an ISA. To guide their design they are using a **SUBSET** of the SPECcpu benchmark programs. [7 pts]

☐ Describe a situation in which that's a foolish thing to do and explain why it's foolish.

☐ Describe a situation in which that's a smart thing to do.

(*b*) Compiler optimizations are important. [8 pts]

☐ Describe a compiler optimization for which the compiler needs no knowledge of the ISA.

☐ Show an example.

☐ Describe a compiler optimization for which the compiler does need to know the target ISA.

(*c*) A goal in the design of MIPS and other RISC ISAs is to minimize the number of registers with specialized purposes. In MIPS `r31`, a.k.a. `$ra`, does have a specialized purpose. [7 pts]

☐ Why was it necessary to make an exception in this case?

(*d*) The target in MIPS branch instructions is specified by indicating the number of instructions to skip. [8 pts]

☐ Why couldn't such an approach be used in CISC ISAs?

☐ Specifying the number of instructions to skip saves two bits over what is needed for CISC targets. Why is saving two bits not as important in the design of CISC instructions?

Name _____

Computer Architecture
EE 4720
Final Examination

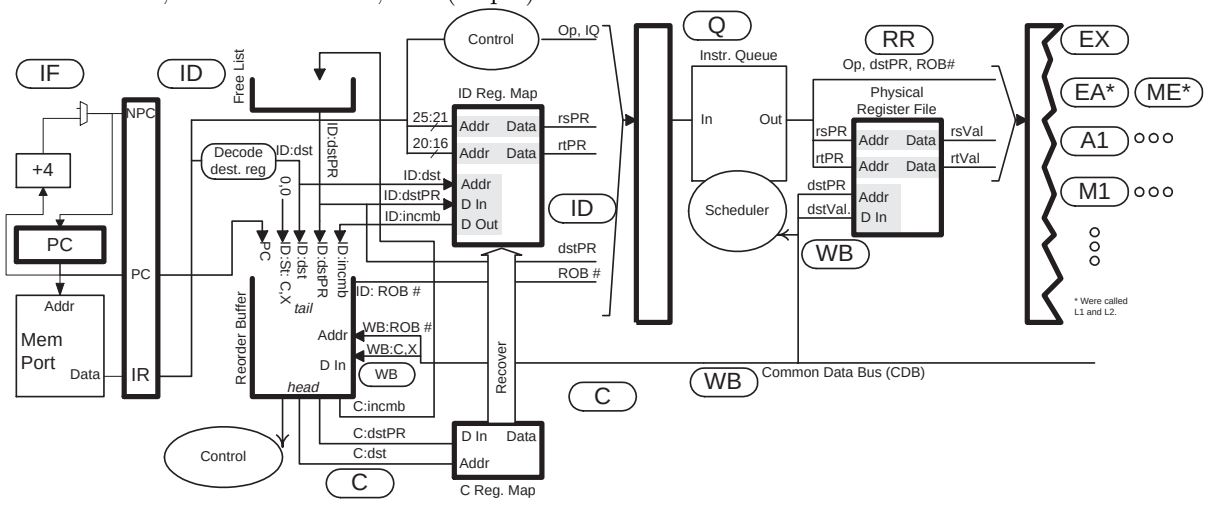
6 December 2004, 7:30–9:30 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The MIPS code below executes as shown on the illustrated dynamically scheduled scalar implementation. There are no exceptions or recoveries; the result (value to be written in `r2`) of the first instruction is 101, the second is 102, etc. (15 pts)



- ☐ Show where each instruction commits.
- ☐ The initial value of `r1` is 111 and the initial value of `r2` is 222. Fill in the tables to show these values.
- ☐ Complete the tables for the execution of the code. Take into account the results (see first paragraph).
- ☐ Show where registers are removed from and put back in the free list.

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
lw  r2, 0(r4)        IF  ID  Q   RR  EA  ME  WB
add r1, r2, r3           IF  ID  Q       RR  EX  WB
sw r1, 0(r4)             IF  ID  Q   RR  EA      ME  WB
xor r2, r5, r3               IF  ID  Q   RR  EX  WB
and r2, r2, r6                   IF  ID  Q       RR  EX  WB
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
# ID Map
```

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
# Commit Map
```

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
Physical Register File
```

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: An extended version of MIPS, called *MMMIPS*, includes memory-to-memory (MM) arithmetic instructions that can read the first source operand from memory and write a result to memory; the second source operand is always an immediate and the MM's are encoded in format I. Their mnemonics end with `.mm`, `.mr`, or `.rm` and they operate as described in the comments below. (25 pts)

```
lw     r1, 2(r3)        #  r1 = Mem[r3+2]  (Existing instruction, for your reference.)
add.mm (r4),(r5), 3     #  Mem[r4] = Mem[r5] + 3
sub.rm  r6, (r7), 3     #  r6 = Mem[r7] + 3
or.mr  (r8), r9,  3     #  Mem[r8] = r9 + 3
```

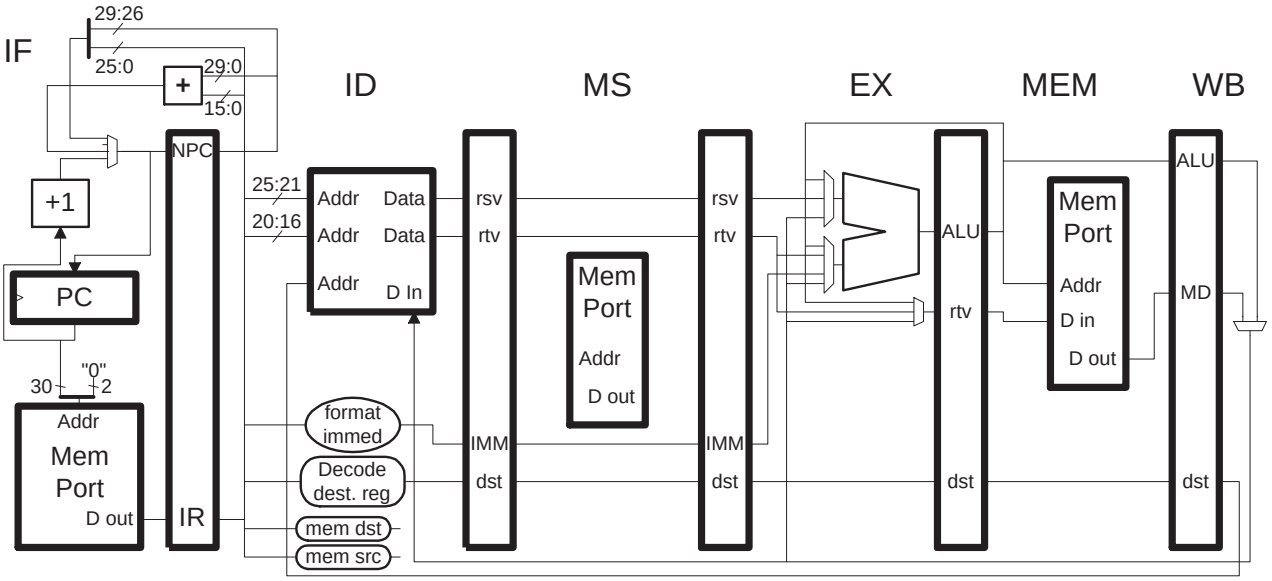
(*a*) Shown below is a partially completed implementation of MMMIPS. It includes a new stage, MS (the S is for source), that has a memory port for the source operand, shown unconnected. For use in a later part of this problem, boxes mem src and mem dst identify an instruction as having a memory source operand (output `1`) or a memory destination operand (output `1`) (an instruction can have both). An output of `0` means the respective operand is not from or to memory.

☐ Connect the MS-stage memory port for the MM instructions.

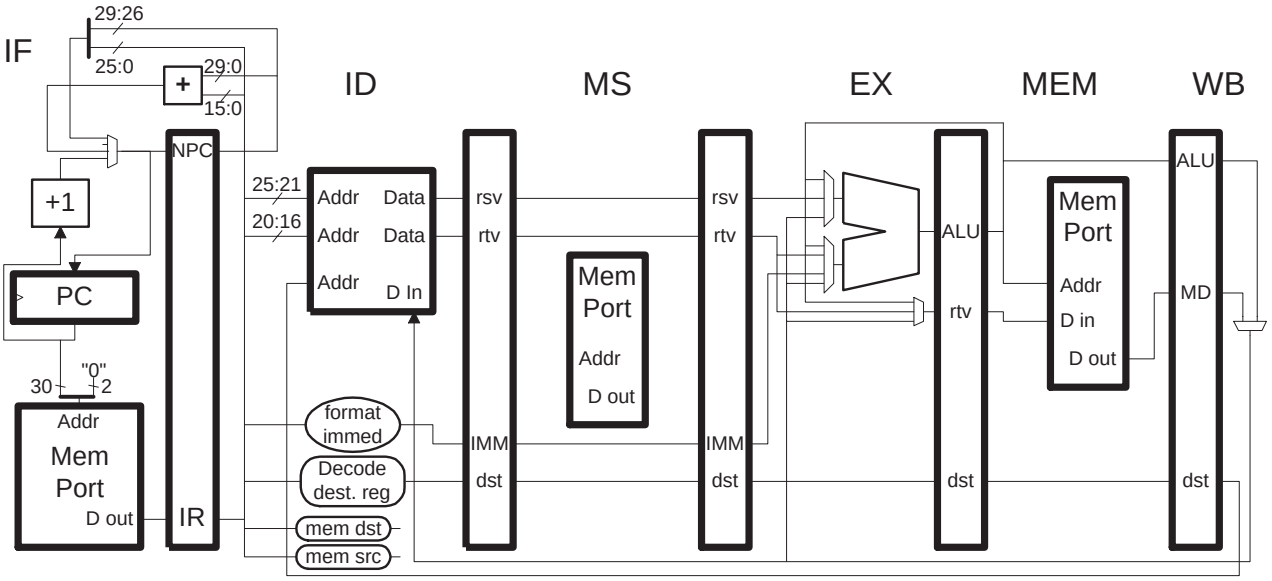
☐ Modify connections to the MEM-stage memory port for the MM instructions.

☐ Make sure existing MIPS load and store instructions continue to work correctly.

☐ For this part don't add bypassing or control logic.



For the problems below use either the diagram below or the one on the previous page.



(*b*) Add a bypass connection so that the code below can execute without a stall or show which existing bypass connection can be used.

☐ Add or identify the connection.

☐ Show which cycle the bypass connection is used.

```
# Cycle              0  1  2  3  4  5  6
add     r1, r2, r3   IF ID MS EX ME WB
sub.mr (r1), r4, 5      IF ID MS EX ME WB
```

(*c*) Can a bypass connection be added for the code below? If yes, should it be added? Explain.

☐ Show how and/or explain why not.

```
# Cycle              0  1  2  3  4  5  6
add    r1, r2, r3    IF ID MS EX ME WB
sub.rm r6, (r1), 5      IF ID MS EX ME WB
```

(*d*) Show the bypass connections (if any) and control logic so that the code below executes without a stall (and without causing other instructions to execute incorrectly, of course). The control logic should deliver a `BYPASS` signal to the EX stage at the right time, it should be `1` if a bypass of the type needed below is necessary. Do not connect it to anything.

☐ Show the control logic generating `BYPASS`.

☐ Add the bypass connection or show which existing one would be used.

```
# r1 = 0x1000, r5 = 0x1000
# Cycle              0  1  2  3  4  5  6
add.mr (r1), r2, 3   IF ID MS EX ME WB
sub.rm  r4, (r5), 6     IF ID MS EX ME WB
```

Problem 3: Answer each question below. Be sure to check each code fragment for dependencies. (25 pts)

(*a*) Show a pipeline execution diagram for the code fragment below running on the usual statically scheduled and bypassed scalar MIPS implementation. *Note: Bypassing and static scheduling not mentioned in the original exam.*

```
# Cycle         0    1    2    3    4    5   6   7   8

lw  r1, 0(r2)

add r3, r1, r4
```

☐ Solve this easy problem.

(*b*) The `sub.s` instruction below stalls in IF.

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11
add.s f4, f5, f6  IF ID A1 A2 A3 A4 WF
sub.s f7, f4, f8     IF -------> ID A1 A2 A3 A4 WF
add.s f9, f10, f11                IF ID A1 A2 A3 A4 WF
```

☐ What's wrong about the stall?

☐ Show correct execution.

Problem 3, continued:

(*c*) The code below executes on a dynamically scheduled machine of the type used in the first problem and in class. The `mul.s` stalls in ID.

```
# ROB initially empty.
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
add.s f4, f5, f6  IF ID Q  RR A1 A2 A3 A4 WF C
sub.s f7, f4, f8     IF ID Q           RR A1 A2 A3 A4 WF C
mul.s f9, f10, f11      IF ID -------> Q  RR M1 M2 M3 M4 M5 M6 WF C
```

☐ What's wrong about the stall?

☐ Show correct execution.

☐ If the "`ROB initially empty.`" comment was not there then the execution above would be possible, albeit misleading. How would the stall be possible?

(*d*) Show a pipeline execution diagram for the code below executing on a 2-way statically scheduled superscalar MIPS implementation. *Note: In the original exam "statically scheduled" was omitted so an answer for either type of system would be correct.*

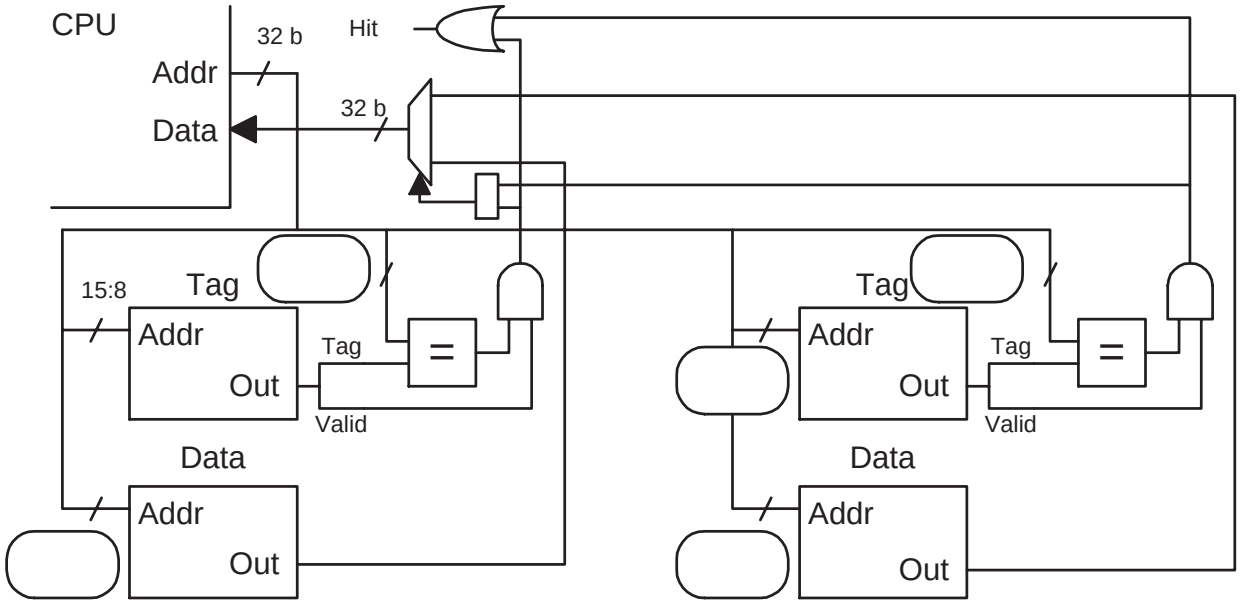
```
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11
0x1000:
 lw  r4, 0(r5)

 add r1, r2, r3

 lh  r6, 0(r4)

 xor r7, r1, r9

 sll r10, r11, 12

 srl r14, r15, 16

 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11
```

☐ Execution must be for decode logic of ordinary complexity. (See next item.)

☐ Execution must allow precise exceptions.

Problem 4: (10 pts) The diagram below is for a two-way set-associative cache on a system with 8-bit characters. Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: 

☐ Cache Capacity (Indicate Unit!!):

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for an eight-way cache with the same capacity and line size.

Address: 

Problem 5: Answer each question below.

(*a*) The register renaming technique used in the dynamically scheduled processor (what the register maps do) solves a problem encountered when instructions execute out of order.(5 pts)

☐ What is the problem and how does it solve it?

☐ Provide an example showing what would go wrong without renaming.

(*b*) When it comes to caches one line size does not fit all.(5 pts)

☐ When is it better to have larger line sizes? Explain how the larger lines help.

☐ When is it better to have smaller line sizes? Explain how larger lines would hurt.

(*c*) Why would the SPEC CPU benchmark suite be less useful if it contained one integer program and one floating-point program? (5 pts)

☐ Why not just one of each?

(*d*) How would you answer a critic who said that the SPEC CPU benchmarks were rigged to make rich and powerful company *I*'s products look good? (5 pts)

☐ They are not rigged because ...

(*e*) What is the difference between a hardware interrupt, an exception, and a trap (as defined in class)? (5 pts)

☐ A hw interrupt's unique feature.

☐ An exception's unique feature.

☐ A trap's unique feature.

28 Spring 2004

Name _____

Computer Architecture
EE 4720
Midterm Examination

Monday, 29 March 2004, 13:40–14:30 CST

Problem 1	_____	(40 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(35 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

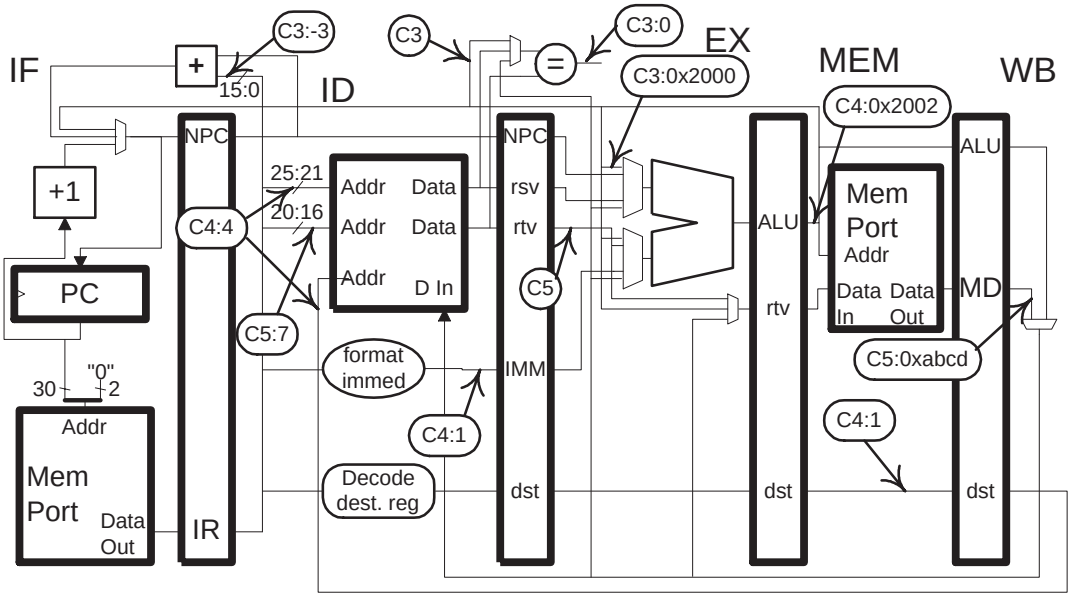
Problem 1: In the diagram below some wires are labeled with cycle numbers and values that will then be present. For example, C3:0 indicates that at cycle 3 the pointed-to wire will hold a 0. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. There are no stalls during the execution of the code. The first instruction (`or`) is shown (but don't forget to add the registers). [40 pts]

☐ Write a program consistent with these labels.

☐ Show the address of every instruction.

☐ Show every register number that can be determined and use `r10`, `r11`, etc. for other register numbers.

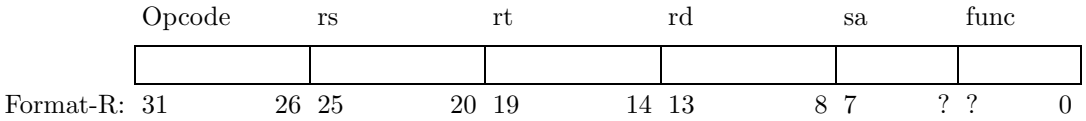
☐ Show the exact instruction (they can all be determined). For example, not just a load, but a load ____.

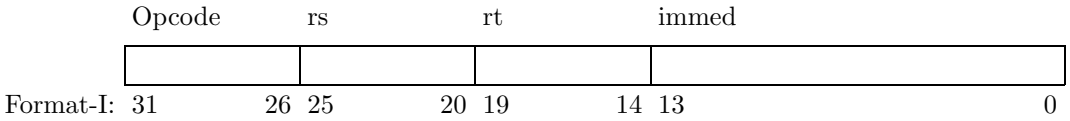


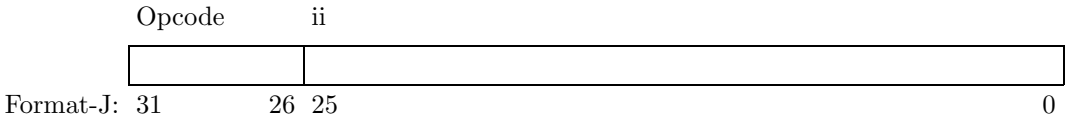
Insn Addr	Cycle: 0	1	2	3	4	5	6	7	8
0x1000: or	IF	ID	EX	ME	WB				
		IF	ID	EX	ME	WB			
			IF	ID	EX	ME	WB		
				IF	ID	EX	ME	WB	
					IF	ID	EX	ME	WB
Cycle:	0	1	2	3	4	5	6	7	8

Problem 2: Consider a new ISA, F-MIPS, similar to MIPS-I except that it has 64 rather than 32 general purpose registers. F-MIPS has R, I, and J instruction formats like MIPS-I but with modifications to handle the larger number of registers. A goal of F-MIPS is to have all of MIPS-I instructions.

Possible instruction formats are shown below. Some details of Format R are omitted and are the subject of the first question.







☐ [6 pts] Describe a problem with Format-R F-MIPS instructions using the Format R shown above.

☐ [7 pts] Fix the problem in Format R (show the changes in the illustration above). Explain the impact (this is important) on the coding of F-MIPS instructions.

☐ [6 pts] Explain an impact on typical F-MIPS Format-I instructions (compared to their MIPS-I counterparts) that would not apply to Format-R instructions.

☐ [6 pts] Why is something going to have to be done about `lui`? Describe a new version of `lui`, possibly using a new format, that would fix the problem. *Hint: That's load upper immediate.*

Problem 3: Answer each question below.

(*a*) A company has to choose between developing two new implementations of their ISA. Implementation *A* would have a peak (result) score of 2200 and a base (baseline) score of 2000 on the SPEC CINT2000 benchmarks. Implementation *B* would have a peak (result) score of 2150 and a base score of 2100 on the benchmarks.

☐ [0 pts] Which implementation should the company choose? *Hint: Either answer is correct.*

☐ [9 pts] Why? Your reason should say something about the difference between the peak and base scores and about the company's customers.

(*b*) Should a BCD data type be added to a modern general-purpose ISA?

☐ [8 pts] Explain why or why not, using the criteria discussed in class for adding data types to an ISA. (Discuss specific features of BCD, don't give an answer that could apply to any data type.)

(*c*) Re-write the SPARC code fragment below in MIPS-I. Use as few instructions as possible. [9 pts]

```
! Notes: r0 is the zero register;  destination is rightmost register;
!        be means branch if equal.  Use the same register names.

subcc %r1, %r2, %r0
add   %r3, %r4, %r5
be    TARG
xor   %r6, 10, %r6
```

(*d*) The code below includes a hypothetical MIPS predicated instruction. Re-write the code using real MIPS instructions. [9 pts]

```
     sub r3, r6, r7
(r1) add r2, r3, r4
     xor r5, r8, r7
```

Name _____

Computer Architecture

EE 4720

Final Examination

13 May 2004, 17:30–19:30 CDT

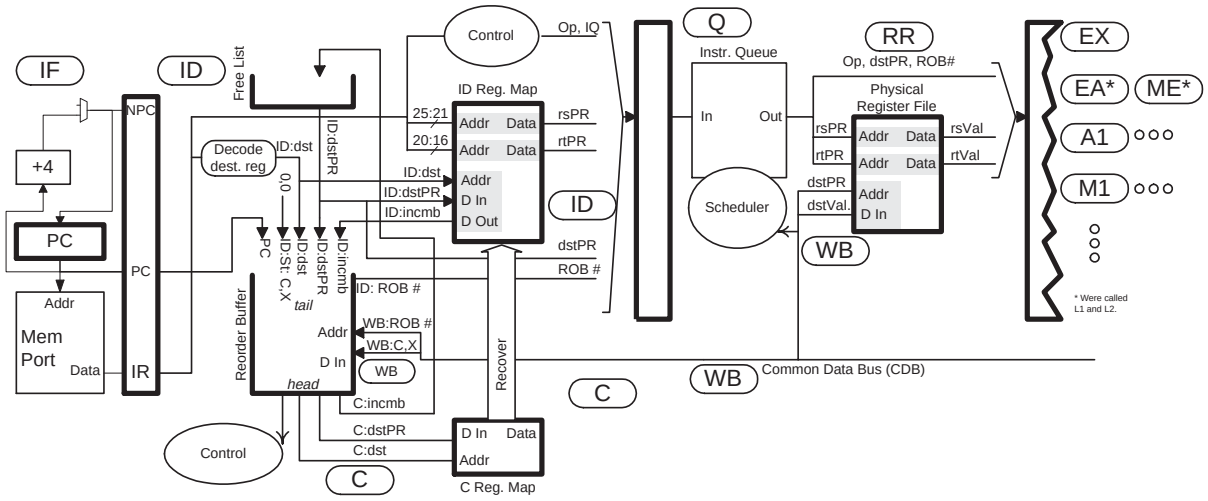
Problem 1	_____	(19 pts)
Problem 2	_____	(18 pts)
Problem 3	_____	(19 pts)
Problem 4	_____	(19 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The MIPS code below executes on the illustrated implementation. (19 pts)

- The implementation makes backup copies (not illustrated) of the ID map for each predicted branch instruction.
- The implementation is scalar, but there can be any number of writebacks per cycle.



The pipeline execution diagram on the next page shows the complete execution of the first instruction, a branch, and partial execution of the others. (Because of instructions before it, the branch enters `RR` after a wait of four cycles and commits after waiting another two.) The branch is mispredicted and some instructions will have to be squashed.

(*a*) Complete the pipeline execution diagram.

☐ Show where instructions are squashed.

☐ Show correct path instructions.

☐ Show each instruction until it is squashed or commits.

(*b*) Complete the tables:

☐ Physical register file. Assume the result of the load is 101, `xor` is 102, and `add` is 103. You can make up your own physical register numbers!

☐ Show where registers are removed from and put back in the free list.

☐ Complete the ID map. Don't forget to include the effect of branch misprediction recovery.

☐ Complete the commit map.

Problem 1, continued: Continued from previous page.

```
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16

 bneq r1, r2 SKIP    IF  ID  Q                   RR  B   WB          C

 lw r3, 0(r4)            IF  ID  Q   RR  EA              ME  WB

 xor r3, r8, r6              IF  ID  Q   RR  EX

SKIP:

 add r3, r3, r7                  IF  ID  Q   RR
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # ID Map
```

```
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # Commit Map
```

```
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 Physical Register file
```

```
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Problem 2: Answer each question about the illustrated MIPS implementation. (18 pts)

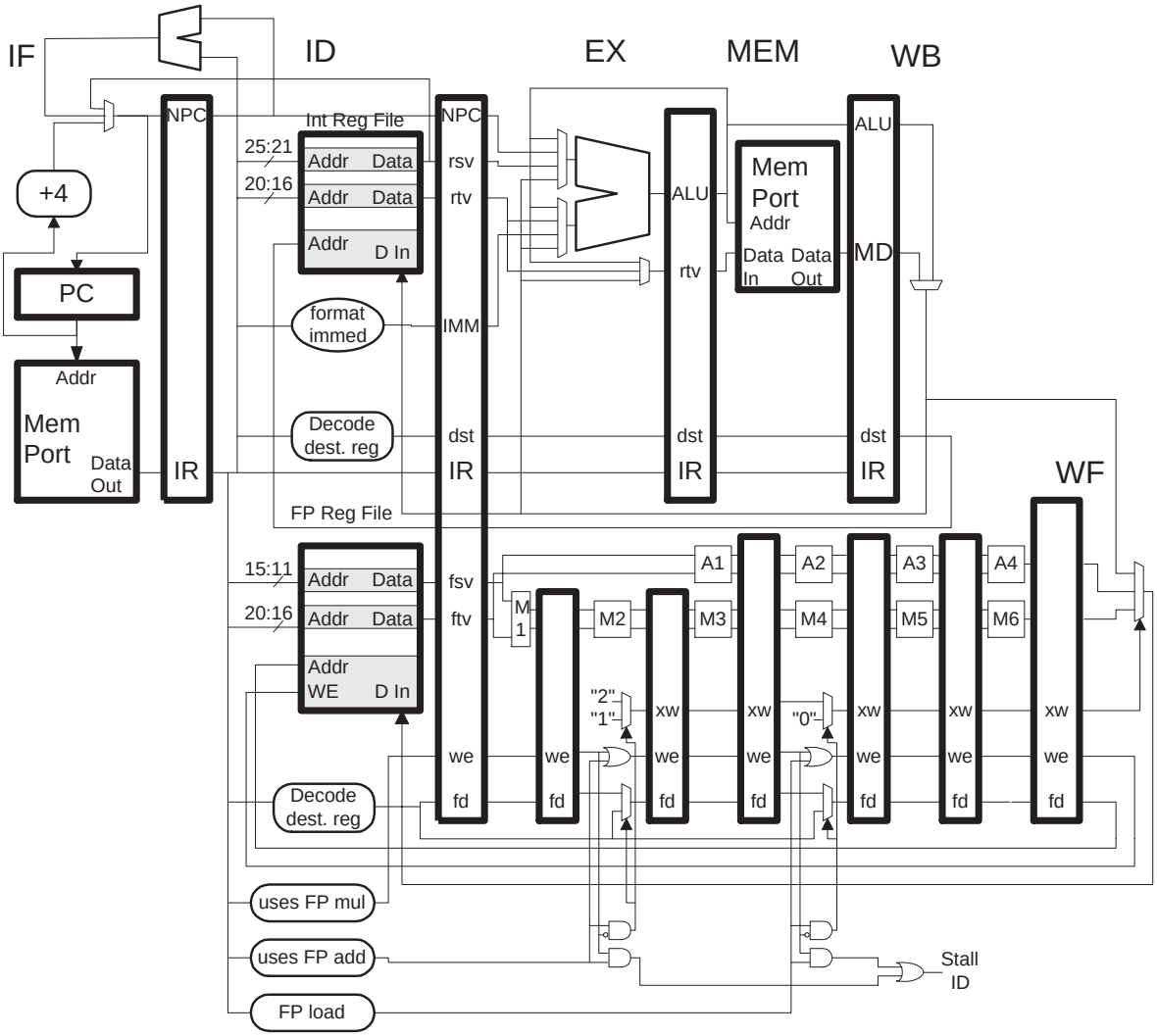
(*a*) Complete a pipeline execution diagram for the code below running on the illustrated implementation. Assume that any needed bypass connection is available and please check for dependencies.

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16

add.d f6, f2, f4

add.d f4, f6, f2
```

(*b*) In the diagram below add the bypass connection(s) needed for the code above. **Do not** add bypass connections that are not needed. (Don't worry if it's a tight squeeze in the diagram, but be legible.)

(*c*) Add control logic that generates the stall signal encountered in the code above.

(*d*) Add control logic for the bypass multiplexor added in a previous part.



Problem 3: The following questions refer to the assembly language code below. Branch outcomes are shown. (19 pts)

```
LOOP:
 0x1000:
B1: beq r1,r2 SKIP1     T  T  T  N  N  T  T  T  N  N  T  T  T  N  N
 nop                                                              !
 add r5, r6, r7                                                   A
SKIP1:                                                            !
 # **Ten** arithmetic instructions only until next branch, no other instructions.
 #                                                                !
B2: beq r3,r4 SKIP2      T  N  T  N  T  N  T  N  T  N  T  N  T  N  T  N
 nop
 add r8, r9, r10
SKIP2:
 add r11, r12, r13
 j LOOP
 nop
```

(*a*) Suppose the code above runs on a system using a bimodal branch predictor with a 256-entry branch history table. What would be the best prediction accuracy of branch `B1` and `B2` after warmup. *Hint: "Best" applies to* `B2` *but not* `B1`.

☐ Accuracy of `B1`.

☐ Accuracy of `B2`.

☐ Why is there a "best" accuracy?

(*b*) Suppose the code above runs on a system using a local predictor with a 256-entry BHT. What is the smallest local history size that would allow the two branches above to be predicted with 100% accuracy?

☐ Smallest size. Explain

(*c*) Suppose the code runs on a system using a gshare branch predictor with a 10-bit GHR. Show the contents of the GHR at time `A` (see the right-hand side of the diagram).

☐ Gshare GHR contents:

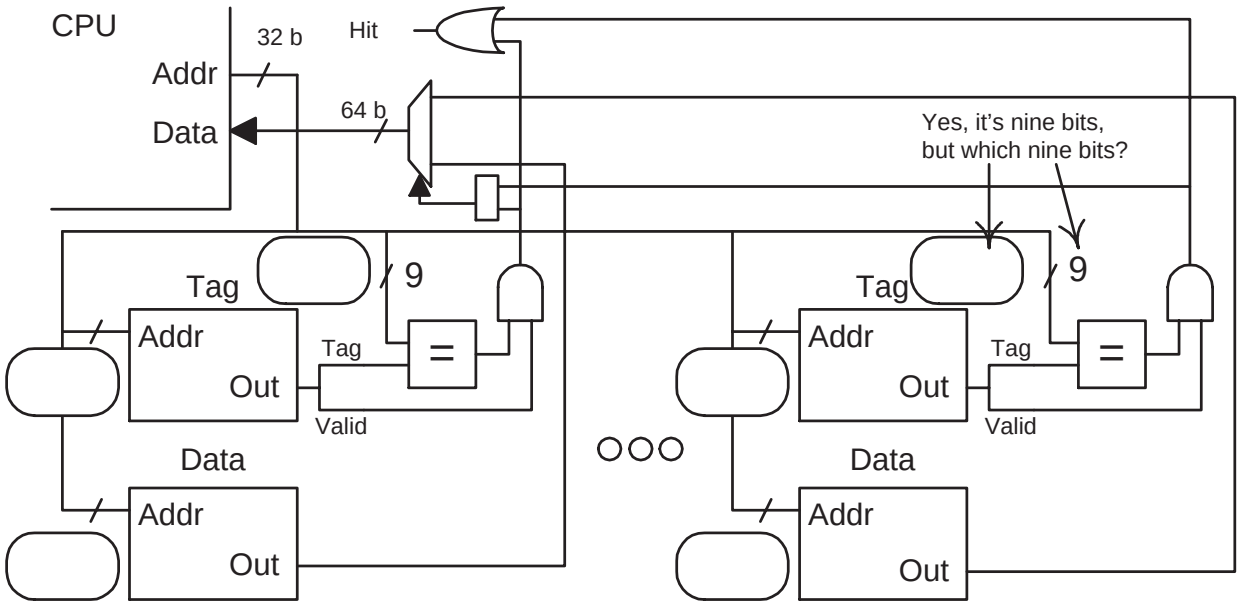
(*d*) Suppose the code runs on a PPC970 (or a POWER4) processor. Show the contents of what it uses for a GHR at time `A`. State any assumptions. *Note: This part based on a homework assigned Spring 2004, and would not be asked other semesters.*

☐ PPC970 GHR contents:

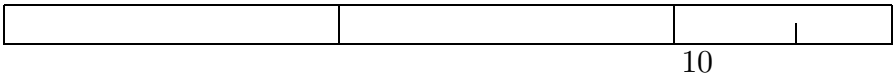
Problem 4: (19 pts) The diagram below is for a four-way set-associative cache on a system with 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

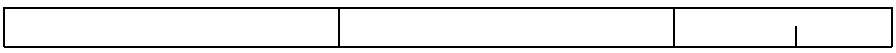
Address: 

☐ Cache Capacity (Indicate Unit!!):

☐ Memory Needed to Implement (Indicate Unit!!):

☐ Line Size (Indicate Unit!!):

☐ Show the bit categorization for a **direct mapped** cache with the same capacity and line size.

Address: 

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☐ What is the hit ratio for the program above?

(*c*) Choose values for the address of `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled with the minimum number of accesses. *Note: The original exam read "minimum number of misses," is easier to do.* The address of `b` must be greater than `a` and as small as possible.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i;


short int *b =


int ILIMIT =


int ISTRIDE =

  for(i=0; i < ILIMIT; i++)
    sum += a[ i * ISTRIDE ] + b[ i * ISTRIDE ];
```

Problem 5: Answer each question below.

(*a*) One way to improve the performance of an implementation is to redesign the processor so that it uses more stages. Increasing the number of stages beyond five will yield a big improvement in performance, but at some point adding stages will have little effect.(5 pts)

☐ Give a reason for this limit having to do with the pipeline latches.

☐ Give a reason for this limit having to do with program characteristics.

(*b*) An instruction does not raise precise exceptions (because the ISA says it does not have to). (5 pts)

☐ Name something its handler cannot do (but could do if the exception were precise).

(*c*) The code below uses an indirect load. Re-write it using MIPS instructions. (5 pts)

```
lw r1, @(r2)   # Load using indirect addressing.
```

(*d*) Every integer instruction that reads a GPR uses the `rs` and `rt` fields (or just one of them). What would be the disadvantage of a modified ISA in which some instructions use other fields to specify GPR source registers? (5 pts)

(*e*) Consider two options for the design of a load/store unit (LSU) for a dynamically scheduled system. The aggressive option would execute the code below quickly, while the conservative option would execute it more slowly. *Hint: the conservative option is how the LSU was described in class. Note: The original problem used* `r1` *for the address, which was not intended, but the answer with* `r1` *is similar.*

(5 pts)

```
div.d f2, f4, f6
sw r1, 0(r9)
sh r2, 0(r9)
sb r3, 0(r9)
lw r4, 0(r9)
```

☐ What is it about that code that requires special treatment?

☐ Describe how the aggressive option would execute the code.

☐ Give a brief argument for the conservative option, feel free to include made-up data (for the purposes of this test only!).

29 Fall 2003

Name _____

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 22 October 2003, 10:40–11:30 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(18 pts)
Problem 4	_____	(18 pts)
Problem 5	_____	(24 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The routine below is called with an unsigned integer in register `$a0` and the address of some allocated memory in register `$a1`. When it returns the memory at should `$a1` contain the hexadecimal representation of `$a0` as a null-terminated (C format) string. Complete the routine, follow the guidelines in the comments. (For **partial credit** write a routine that converts a string holding a hexadecimal number to an integer.) [25 pts]

```
################################################################################
# utoh: Convert unsigned integer to hexadecimal string.
#
        # $a0: Call Value: Unsigned integer to convert.
        # $a1: Call Value: Address of allocated memory.
        #      Write converted string to this address, assume there is enough.
        #      Sample strings: "1F3", "1", "0" written at $a1.

        # [ ] String should not have leading zeros. (Good: "123", Bad "00123".)
        # [ ] Fill as many delay slots as possible.
        # [ ] Registers $a0-$a3 and $t0-$t7 can be modified.

        # The ASCII value of '0' is 48,  the ASCII value of 'A' is 65

# Put solution here.
utoh:









        jr $ra
        nop
```

Problem 2: Code similar to the histogram program presented in class appears below. MIPS instruction formats are shown below for reference. [15 pts]

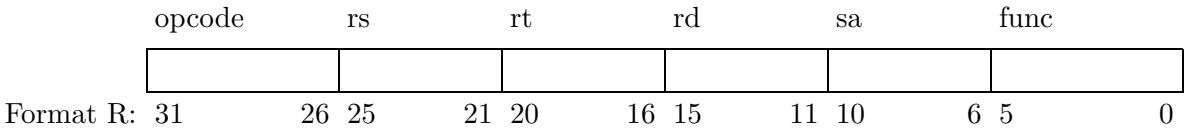
(*a*) Design three new MIPS instructions to reduce the size of the program fragment below

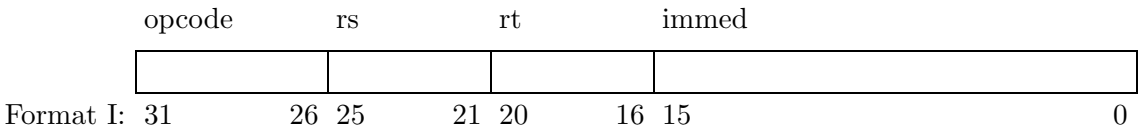
- Each new instruction should replace at least two related instructions in the program below. (Do not combine two *unrelated* instructions, such as `j` and `sw`.)

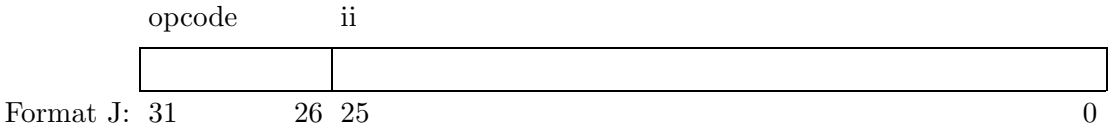
☐ Show the coding for the new instruction, making up the opcode and other field values as necessary. The coding should use one of MIPS' existing formats and should fit as naturally as possible.

- Do not worry whether the instruction is appropriate for a RISC ISA.

```
 addi $t7, $0, 26
LOOP:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
 beq $t1, $0, DONE
 addi $t1, $t1, -65
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
 addi $t4, $t4, 1
 j LOOP
 sw $t4, 0($t3)
```







(*b*) Show an instruction that can be created by combining several instructions from the program but that would be impossible to code. Explain why it would be impossible to code.

Problem 3: Answer each question below.

(*a*) In MIPS (and similar ISAs) there is a `lb`, `lbu` (load byte unsigned), and a `sb` but there is no `sbu` (store byte unsigned). Why not? [6 pts]

(*b*) Explain why the MIPS instruction below won't work: [6 pts]

```
add.d $f1, $f2, $f3
```

(*c*) Show the result of each add instruction below. The instructions execute on a machine with 32-bit registers that is capable of BCD, and packed integer, and ordinary integer arithmetic. The packed integer operations all use saturating unsigned arithmetic. [6 pts]

```
# r1 = 0x8080888
# r2 = 0x1020999


# Ordinary Integer Add
add r3, r1, r2            r3 =

# BCD Add
add.bcd r3, r1, r2        r3 =

# Packed Integer (4 bits per int.)
add.p4 r3, r1, r2         r3 =

# Packed Integer (8 bits per int.)
add.p8 r3, r1, r2         r3 =
```

Problem 4: Answer each question below.

(*a*) A company compiles and runs the SPECint2000 benchmarks on its new system, complying with all rules except one: it refuses to divulge the steps it used to compile the programs. Nevermind that it's against the rules, is it in the company's interest to keep this information secret? Explain. [6 pts]

(*b*) A program is compiled two ways, one for ISA A, implementation x, the other also for ISA A, but for implementation y. Explain what would be common to the two executables and what would be different. Provide an example. [6 pts]

(*c*) Explain the dead-code elimination (DCE) optimization using an example. [6 pts]

Problem 5: Answer each question below.

(*a*) CISC ISAs have variable length instructions. What advantages over RISC ISAs does that enable? Name at least two and briefly explain each advantage. [6 pts]

(*b*) RISC ISAs have fixed length instructions. What advantages over CISC ISAs does that enable? Name at least one and briefly explain each advantage. [6 pts]

(*c*) Why are programs written in a stack ISA small? Be brief but as specific as possible. [6 pts]

(*d*) Listed below are several behaviors or features. For each, explain whether it is usually an ISA feature, an implementation feature, or an ABI (application binary interface) feature. **Briefly explain why.** If it can fit into more than one category (for example, ISA or implementation) say so and explain why. [6 pts]

☐ An `add` instruction raises an exception on an overflow.

☐ The unused field in an instruction must be set to zero.

☐ Procedure call saves return address in a particular register.

☐ Multiply instruction takes four cycles.

Name _____

Computer Architecture
EE 4720
Final Examination

9 December 2003, 10:00–12:00 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: (20 pts) The execution of a MIPS code fragment on a dynamically scheduled machine is shown on the next page. The diagram shows the values on certain wires at certain cycles. For example, 4:65 means that at cycle 4 the labeled wire holds value 65. The physical register file table is completed, ID- and commit-map tables are blank.

- The FP add unit has 3 stages, the FP multiply unit has 5, and the EA and ME are used for loads and stores.
- All destination registers are floating point.
- WB and commit can be done in the same cycle (indicate with a WC).
- To keep things simple the result of every instruction is zero and there are no cache misses.

(*a*) The ID and commit register map tables are blank ...

☐ ... complete them (the ID and commit register map tables.)

☐ Show the correct architected register numbers, or for partial credit make them up. (Two are easy, the rest are interesting.)

☐ Show the initial values (just before cycle 0) in the ID and commit map tables.

(*b*) Complete the pipeline execution diagram.

☐ Be sure to show Q, RR, WB, C (or WC), and a possible functional unit.

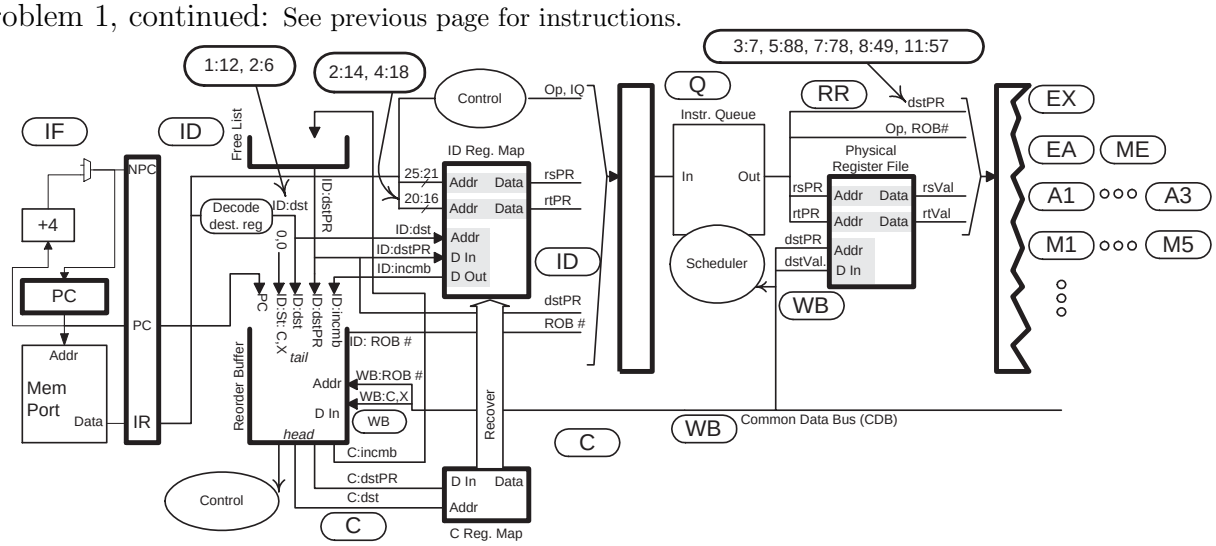
(*c*) Write a program consistent with these tables and labels.

☐ Choose consistent instructions.

☐ Choose consistent registers. If a register number cannot be determined, use a question mark.

- *Hint 1: In the physical register file table, put a "1" next to the first (earliest) register removed from the free list, put a "2" next to the second register removed from the free list, and so on. Similarly, put a "1" next to the first register put back in the free list, etc. To figure out which physical register belongs to which instruction destination (easy) use the fact that certain events occur in program order.*
- *Hint 2: To figure out which architected register an instruction is writing (interesting) remember what causes a register to be put back in the free list.*

Problem 1, continued: See previous page for instructions.



# Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	IF	ID															
		IF	ID														
			IF	ID													
				IF	ID												
					IF	ID											

# ID Map Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

# Commit Map Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

# Phys. Reg. File Cy	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7		[								0.				]			
33										]							
49			[										0.			]	
57						[										0.	
78					[							0.					
88				[					0.						]		
95													]				

# Phys. Reg. File Cy	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Problem 2: (20 pts) In all of the problems below please check the code samples carefully for dependencies. All implementations below are fully bypassed. Please check the code samples carefully for dependencies.

(*a*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 2.

☐ Show a pipeline execution diagram for the code.

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15

add.s f2, f4, f6     IF

add.s f8, f2, f12

add.s f14, f10, f16

# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

(*b*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☐ Show a pipeline execution diagram for the code.

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15

add.s f2, f4, f6     IF

add.s f8, f2, f12

add.s f14, f10, f16

# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

Problem 2, continued:

(*c*) The code below executes on a 2-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☐ Show a pipeline execution diagram for the code.

```
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15

LINE: # LINE = 0x1000
 add.s f2, f4, f6      IF

 add.s f8, f2, f12

 add.s f14, f10, f16

 and r1, r2, r3

 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

(*d*) In a correct solution to the problem above there should be at least one instruction for which a precise exception is impossible. If that describes your solution, show a pipeline execution diagram below in which all instructions could raise precise exceptions (even though they don't). It's also possible that in a correct solution to the problem above all instructions can raise precise exceptions. If so, show a pipeline execution diagram below in which some instructions cannot raise a precise exceptions. In the absence of exceptions all pipeline execution diagrams must show correct execution.

☐ Show the appropriate pipeline execution diagram, or show how the one above would be different.

☐ Identify those instructions for which precise exceptions are impossible (above or below) and explain why.

```
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
LINE: # LINE = 0x1000   (Either put solution here or show changes to previous solution.)

 add.s f2, f4, f6      IF

 add.s f8, f2, f12

 add.s f14, f10, f16

 and r1, r2, r3

 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

Problem 2, continued:

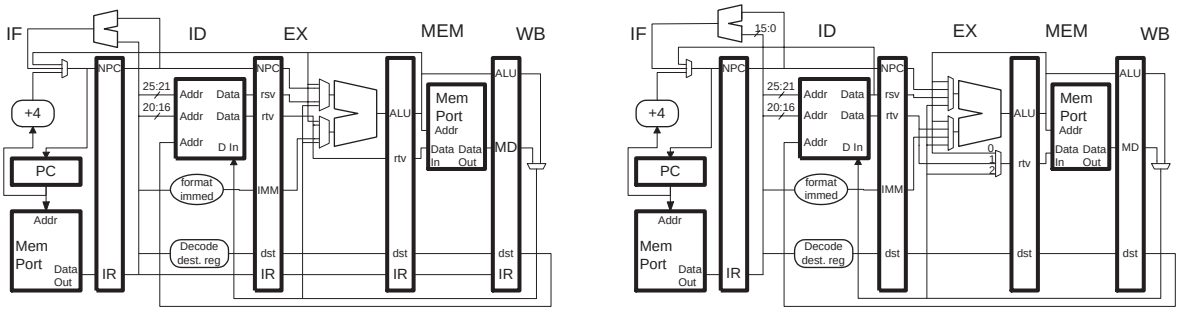
(*e*) The code below executes on a 2-way superscalar dynamically scheduled machine using method 3 (the only one covered this semester), the same one used in Problem 1. The FP add unit has a latency of 3 and an initiation interval of 1.

☐ Show a pipeline execution diagram. Don't forget the commit stage.

Assume an unlimited number of reorder buffer entries and physical registers.

```
 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
LINE: # LINE = 0x1000

 add.s f2, f4, f6      IF

 add.s f8, f2, f12

 add.s f14, f10, f16

 and r1, r2, r3

 # Cycle               0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

Problem 3: (20 pts) Two MIPS implementations are illustrated below, the right one has a multiplexor at the input to the `EX/MEM.rtv` pipeline latch, the left one does not.

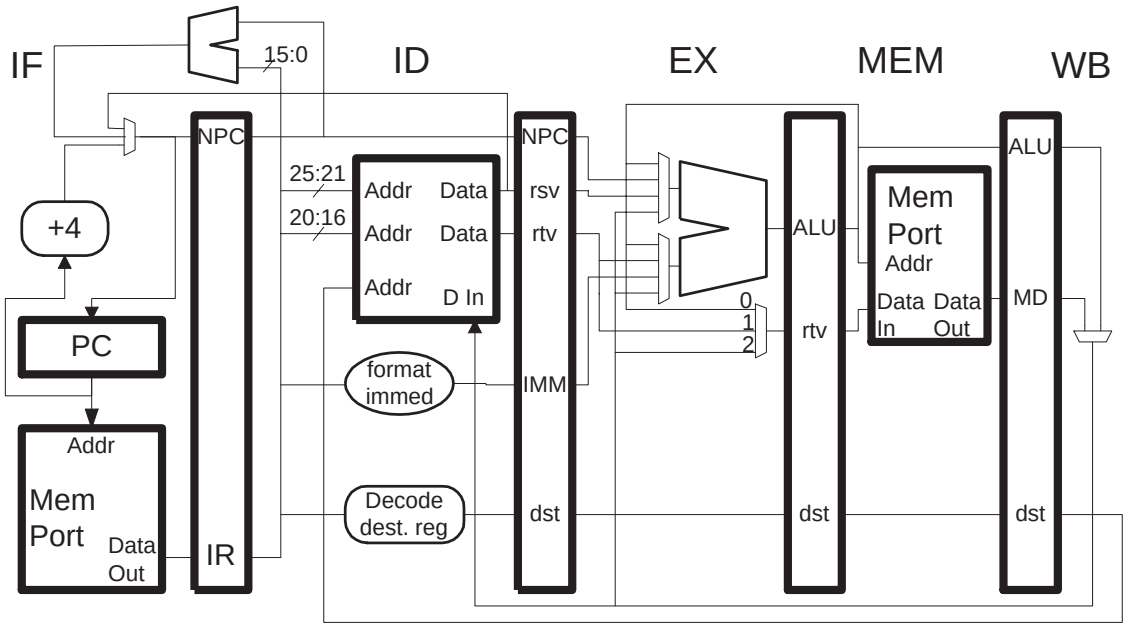


(*a*) Provide two code samples, one in which the multiplexor is useful and one in which it is not. Briefly explain.

(*b*) Suppose version 5.11 of a compiler was written for the implementation on the left and is in the hands of customers. Version 5.99 of the compiler also includes the right implementation and is being released soon. Which two compilation options would you have to use to take advantage of the changes made for the right implementation? (The exact names of the compiler options is unimportant, but it should be obvious what they do.) Briefly explain why each option is necessary and how it would affect the code.

Problem 3, continued:

(*c*) Design the control logic for the `rtv` multiplexor. Unlike Homework 4, **the logic must be in the EX stage**. Where appropriate, show which bits are being used, e.g., `12:5`.

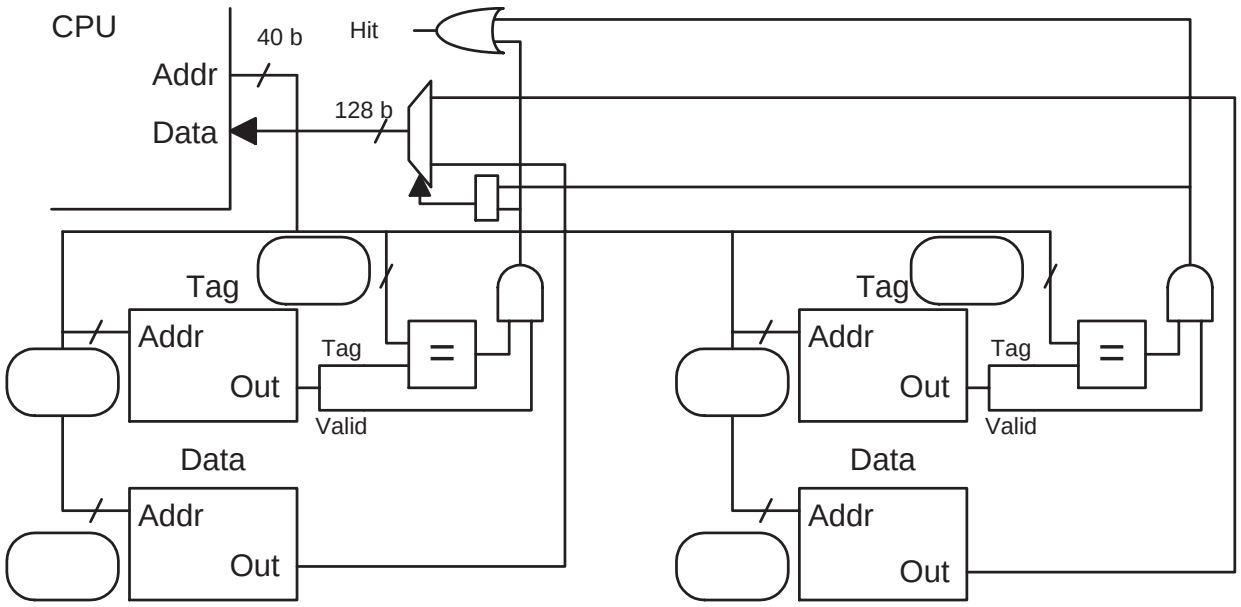


(*d*) There is a good reason why the control logic for the ALU input multiplexors should be in the ID stage that does not apply to the `rtv` multiplexor control logic. What is the reason, and why does it not apply to the `rtv` logic?

Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25} bytes) cache with 512-byte (2^9-byte) lines on a system with 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!):

☐ Show the bit categorization for a **fully associative** cache with the same capacity and line size. *Note: Emphasis not included in original exam.*

Address:

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☐ What is the hit ratio for the program above?

(*c*) Find the hit ratio for the code below running on the cache from the first part. Consider only accesses to the arrays and assume the cache starts out cold. State any assumptions made.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
char *b = 0x2000000;  // sizeof(char) = 1 character
char *c = 0x3000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ] + b[ i ] + c[ i ];
```

(*d*) Modify the addresses of `a`, `b`, and `c` to maximize hit ratio. Explain how the modified addresses improve hit ratio.

Problem 5: (20 pts) Answer each question below.

(*a*) The diagram below shows the branch outcome patterns for two branches.

```
# Loop contains only the branches shown.
BIGLOOP:

B1: 0x1000 beq $t1, $t2, SKIP1   N  N  N  N  T  T  N  N  N  N  T  T  N  N  N  N  T  T
...
B2: 0x1200 beq $v0, $v1, SKIP2    T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
...
    0x2010 j BIGLOOP
```

☐ How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table (BHT)?

☐ What is the minimum size of the BHT for which the accuracy in the previous part is possible? Explain.

☐ Why might it be pointless to perform branch prediction in the ID stage of the 1-way statically scheduled pipeline used in class?

(*b*) Answer the following questions about exception codes as defined for SPARC V8 and using the class terminology.

☐ What is an exception code number (or trap type)? *Note: In the original exam the question was shorter: "What is an exception code?"*

☐ How is it obtained for traps?

☐ How is it obtained for hardware interrupts?

☐ How is it obtained for exceptions?

☐ How is it used to start the handler?

(*c*) Consider two processors, one is a 6-way superscalar implementation of an ordinary ISA, say MIPS, the other is a 6-way implementation of a VLIW ISA, say Itanium (IA-64).

☐ Describe two features of Itanium (or some other VLIW ISA) that would allow it to execute faster than the superscalar implementation. Explain how the features allow faster execution.

☐ If the implementation for the VLIW ISA were faster why might the superscalar implementation still be better from a business perspective?

30 Spring 2003

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 21 March 2003, 13:40–14:30 CST

Problem 1	_____	(30 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

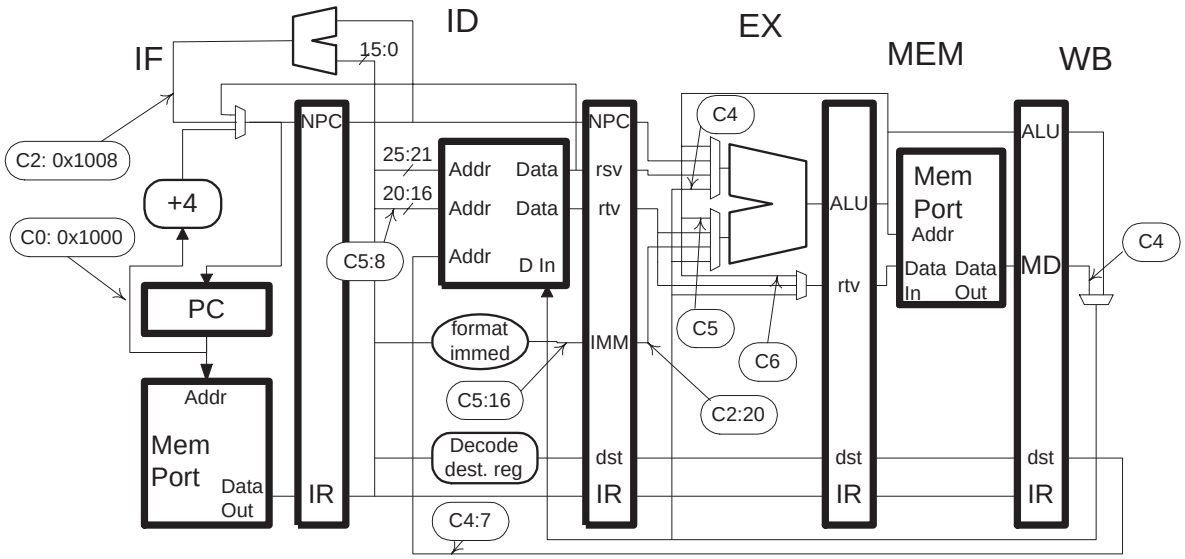
Problem 1: In the diagram below certain wires are labeled with cycle numbers and values that will then be present. For example, C5:8 indicates that at cycle 5 the pointed-to wire will hold an 8. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. There are no stalls during the execution of the code.[30 pts]

☐ Write a program consistent with these labels.

☐ The branch is taken. Use labels for branch targets and label the target line.

☐ Show the address of every instruction.

☐ Fill in every register number that can be determined and use `r10`, `r11`, etc. for other register numbers.




```
Cycle                  0  1  2  3  4  5  6  7  8

                       IF ID EX ME WB

                          IF ID EX ME WB

                             IF ID EX ME WB

                                IF ID EX ME WB

                                   IF ID EX ME WB

Cycle                  0  1  2  3  4  5  6  7  8
```

Problem 2: The CISC-A ISA is making its world premier in this exam!

- It has 32 integer registers (the MIPS names can be used) and the address size is 32 bits.
- Each operand in each instruction can use any appropriate addressing mode, including register, immediate, and the full range of memory addressing modes described in class.
- Instructions are variable size, the entire first byte is the opcode.

(*a*) Re-write the code below in CISC-A, making up appropriate instructions as needed. [10 pts]

☐ Take advantage of CISC-A's characteristics to reduce code size within the loop (primary goal) and outside the loop (secondary goal). (See the next problem.)

☐ Take advantage of: the available *addressing modes* (memory, register, and immediate), the *variable instruction size*, and *operand flexibility*.

☐ A correct solution includes at least three big-improvement-over-MIPS instructions.

☐ Identify non-MIPS addressing modes used.

```
 lui $t0, 0x1234
 ori $t0, $t0, 0x5678
 lw $t2, 0($s0)
 lw $t2, 0($t2)
 addi $t3, $t0, 0x1000
LOOP:
 lw $t1, 0($t0)
 addi $t0, $t0, 4
 bne $t0, $t3  LOOP
 add $t2, $t2, $t1
```

(*b*) Describe a possible coding for three of the CISC-A instruction used above. Don't choose three similar instructions. [10 pts]

☐ It should be obvious (to a computer engineer) that the full range of operands is available.

☐ The coding must follow the • bulleted • points above.

☐ Instruction size should be small, though not the absolute minimum size.

Problem 2, continued: Consider a second ISA, CISC-B, which differs from CISC-A in the following RISCy ways:

- Only load and store instructions can access memory.
- Each instruction has a fixed set of operand types, for example, an `addi` might be limited to one register destination, one register source and one immediate source.
- CISC-B still has variable-size instructions.

(*c*) Show how one such instruction might be coded. Try to choose a good example from the problem above. [5 pts]

☐ The coding should reflect and exploit CISC-B's operand restrictions.

(*d*) Show two program fragments: [5 pts]

☐ Show a program fragment that would be smaller in CISC-B than CISC-A. (No more than four instructions.)

☐ Show a program fragment that would be larger in CISC-B than CISC-A. (No more than four instructions.)

Problem 3: Answer each question below.

(*a*) Answer the following optimization questions. [10 pts]

☐ Explain the difference between front-end and back-end optimizations.

☐ Can typical front-end optimizations be performed by the back end? Explain using an example.

☐ Can typical back-end optimizations be performed by the front end? Explain using an example.

(*b*) It is important to computer manufacturers to have high SPEC benchmark ratings. For each technique of improving SPEC ratings, describe whether it will work, and if it won't describe why not. [10 pts]

☐ Try to have a favorable benchmark added to the suite by sending the name of the benchmark and a little something for their trouble (a bribe) to morally weak SPEC members. *In the exam as given the sentence started "Have a favorable . . .".*

☐ Modify the source code to the benchmarks, honestly improving performance on your system.

☐ Modify your compiler so that it honestly produces faster benchmark executables.

☐ Report results with certain benchmarks omitted.

(*c*) The typical ISA has one-, two-, four-, and eight-byte integers. [10 pts]

☐ Explain a possible benefit of having five-byte integers.

☐ Explain a difficulty with adding five-byte integers, taking in to account a certain feature (some would call it a restriction) of many RISC ISAs. *Hint: The name of the feature begins with an "a".*

(*d*) Provide the requested code examples and answer the questions. Two instructions suffice. [10 pts]

☐ Show code having a data dependency; circle the registers carrying the dependency.

☐ What is the name of the corresponding hazard?

☐ Show code having an anti dependency; circle the registers carrying the dependency.

☐ What is the name of the corresponding hazard?

Name _____

Computer Architecture

EE 4720

Final Examination

14 May 2003, 15:00–17:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The execution of a MIPS code fragment on a dynamically scheduled machine is shown in the tables and in the labels on the diagram, both on the next page. The tables show the contents of the ID Register Map, Commit Register Map, and the Physical Register File at each cycle. The diagram shows the values on certain wires at certain cycles. For example, 4:65 means that at cycle 4 the labeled wire holds value 65.

The following are functional unit segment labels: Load/store, L1 L2; floating-point add, A1 A2 A3 A4; floating-point multiply, M1 M2 M3 M4 M5 M6; integer, EX. The register maps handle both integer and floating-point registers.

(*a*) Write a program consistent with these tables and labels.(12 pts)

☐ Show a pipeline execution diagram, be sure to show where each instruction commits.

☐ Choose consistent instructions.

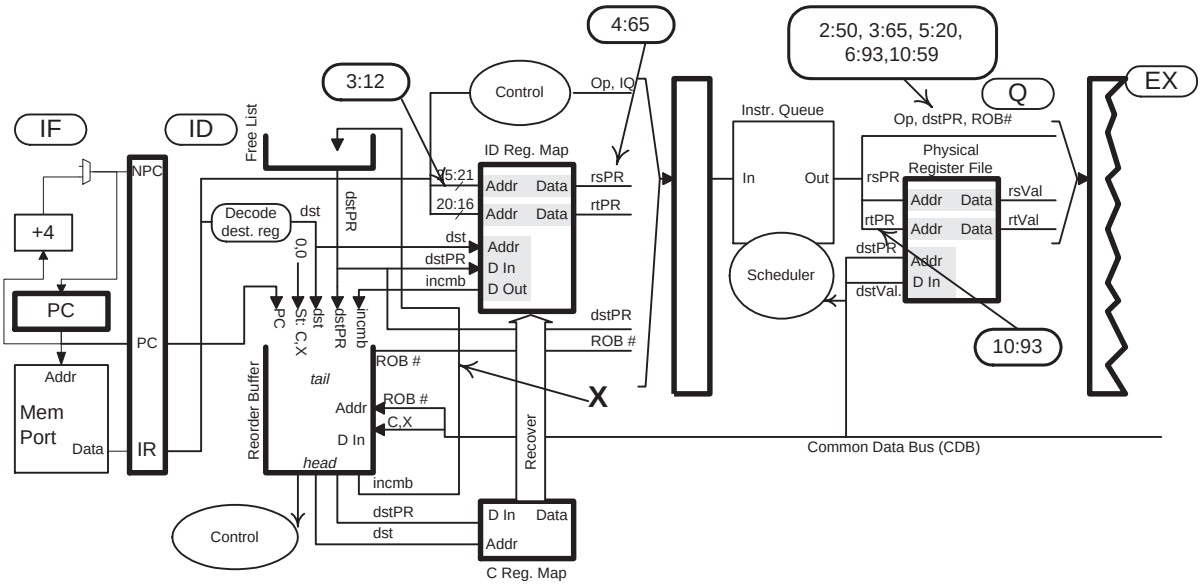
☐ Choose consistent registers. If a register number cannot be determined, use a question mark.

(*b*) Complete the tables on the next page as follows:(8 pts)

☐ Show where registers are added to, "]", and removed from, "[", the free list.

☐ Show the values on the line marked `X` in the illustration.

Problem 1, continued: See previous page for instructions.




```
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

                     IF ID

                        IF ID

                           IF ID

                              IF ID

                                 IF ID

                                    IF ID

# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

X (See Diagram)
# ID Map       Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                50    93    59
r5     3                   65
f10   10                         20
# Commit Map   Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                                  50          93          59
r5     3                                     65
f10   10                                                  20
# Phys. Reg. File  Cy 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
3              0
7              0.
10             0.
20                                           0.
50                                        0.
59                                                                  0.
65                                  100
93                                                       0.
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

Problem 2: The diagram below shows the branch outcome patterns for a branch. (15 pts)

```
BIGLOOP:

B1: 0x1000 beq $t1, $t2, SKIP1    N  N  N  T  T  N  N  N  T  T  N  N  N  T  T
...
B2: 0x1020 beq $v0, $v1, SKIP2
...
    0x2010 j BIGLOOP
```

☐ How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table?

☐ How accurately would branch B1 be predicted by a local history predictor with a 10-bit local history and a 2^{14}-entry branch history table?

☐ What is the minimum local history size needed to predict B1 with 100% accuracy (after warmup). No partial credit without an explanation.

☐ Find a pattern for branch B2 that will reduce the accuracy of the local predictor on branch B1. The branch history table (not to be confused with the pattern history table) remains 2^{14} entries and the history length remains 10 bits.

Problem 3: Answer the following load/store unit questions.

(*a*) Why don't store instructions write to the cache until they commit? (5 pts)

For the two problems below consider the four instructions (repeated) which are in the reorder buffer of a dynamically scheduled 1-way system. On a cache miss data will arrive in four cycles or more. The effective address for the second load is `0x1000`. (10 pts)

(*b*) Describe a scenario in which the data for address `0x1000` is not cached but the second load does not wait for its data.

☐ Show a pipeline execution diagram.

☐ Explain the involvement of the load/store queue and why the second load does not wait more than a cycle or two.

```
First:  lw $t1, 0($t2)

        addi $t0, $0, 4720

        sw 0($t1), $t0

Second: lw $t3, 0($t4)
        Eff. Addr is 0x1000
```

(*c*) Describe a scenario in which the data for address `0x1000` is cached but the second load waits for its data at least four cycles.

☐ Show a pipeline execution diagram.

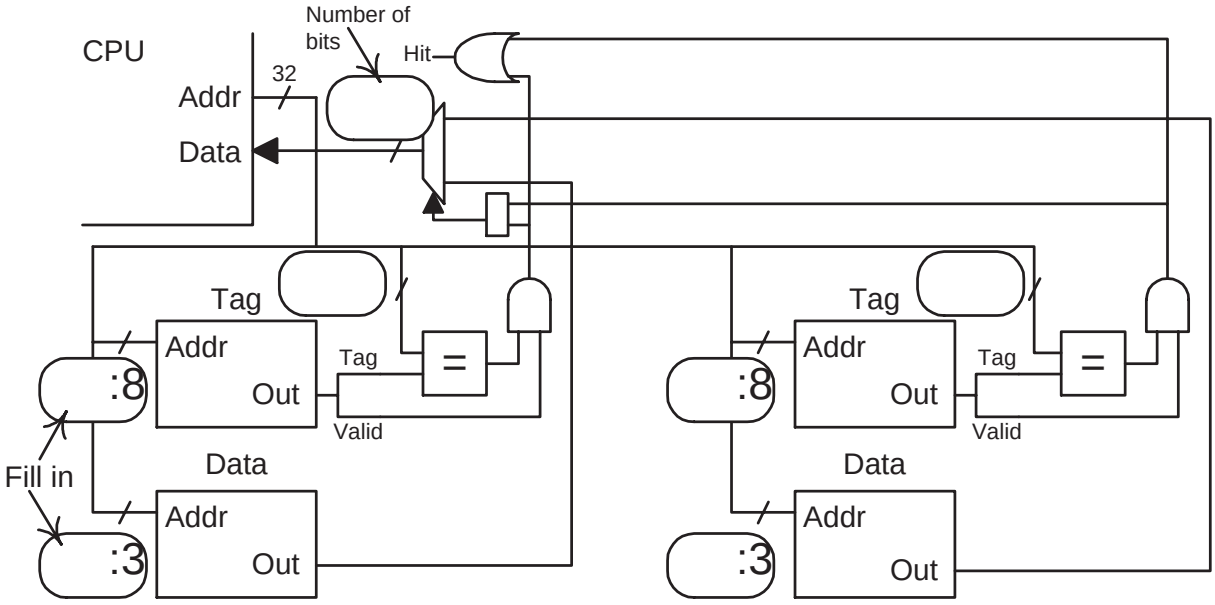
☐ Explain the involvement of the load/store queue and why the second load waits.

```
First:  lw $t1, 0($t2)

        addi $t0, $0, 4720

        sw 0($t1), $t0

Second: lw $t3, 0($t4)
        Eff. Addr is 0x1000
```

Problem 4: The diagram below is for a 2-MiB (2^{21} bytes) cache on a system with 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address: | | | |

☐ Associativity:

☐ Memory Needed to Implement (Indicate Unit!):

☐ Show the bit categorization for a four-way set-associative cache with the same capacity and line size.

Address: | | | |

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array. (7 pts)

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i * 2 ];
```

☐ What is the hit ratio for the program above?

☐ What is the minimum value of ILIMIT needed to fill the cache?

(*c*) The code below runs on the same cache as parts above. Initially the cache is empty; consider only accesses to the arrays. (6 pts)

☐ Choose values for `KLIMIT`, `MLIMIT`, `KSHIFT`, and `MSHIFT` so that array `matrix` is completely removed from the cache with the **minimum** number of accesses (to `b`). That is, each `j` iteration must begin with `matrix` reloaded. Don't forget that the cache is set associative.

```
int  *matrix = 0x1000000;    // sizeof(int) = 4 characters
char *b      = 0x2000000;
int sum, dummy, i, j, k, m;

int KLIMIT =                                    int MLIMIT =

int KSHIFT =                                    int MSHIFT =

for(j=0; j<3; j++)
 {
    for(i=0; i<1024; i++)
      sum += matrix[ i ];

    for(k=0; k<KLIMIT; k++)
      for(m=0; m<MLIMIT; m++)
        dummy += b[ ( k << KSHIFT ) + ( m << MSHIFT ) ];
 }
```

Problem 5: Answer each question below.

(*a*) Suppose the instructions below are being considered for the latest extension of the MIPS ISA. For each new instruction explain why it should be added or why it should not be added. Consider a *variety* of factors related to the ISA and implementation. (10 pts)

☐ A mask instruction. Based on analysis of benchmarks.

```
New Instruction
 mask $t1, $t2, 5

Equivalent Code Using Existing Instructions
 srl $t1, $t2, 5
 sll $t1, $t1, 5
```

☐ An integer negate instruction. Based on analysis of existing code. With this extension `sub` does not have to be used for negation!!!

```
New Instruction
 neg $t1, $t2

Equivalent Code Using Existing Instructions
 sub $t1, $0, $t2
```

☐ An indirect load. Useful based on most existing benchmarks.

```
New Instruction
 lwi $t1, 0($t2)

Equivalent Code Using Existing Instructions
 lw $t1, 0($t2)
 lw $t1, 0($t1)
```

☐ Added functionality for the `sllv` instruction. The existing `sllv` looks at the low 5 bits of the `rt` register, ignoring the other bits. It only shifts left. The improved instruction also shifts right if the `rt` register holds a negative value and left if it's positive.

```
Improved Instruction
  sllv $t1, $t2, $t3   # Shifts right if $t3 negative.

Equivalent Code Using Existing Instructions
  bltz $t3, SHIFTRIGHT
  nop
  sllv $t1, $t2, $t3
  j DONE
SHIFTRIGHT:
  sub $at, $0, $t3
  srlv $t1, $t2, $at
DONE:
```

(*b*) Answer the following questions about exceptions. (5 pts)

☐ Why are precise exceptions necessary for instructions like `lw` but optional for instructions like `div`?

☐ What can handlers for instructions that raise precise exceptions do that handlers for other instructions cannot? Explain how that capability is used for `lw`.

(*c*) Delayed branches are common in RISC ISAs. (5 pts)

☐ Explain why delayed branches were useful in early RISC processors, such as the 1-way statically scheduled MIPS implementation covered in class.

☐ Why are delayed branches of less benefit with more recent implementations?

(*d*) Why might a gshare branch predictor with a longer global history register (GHR) have a significantly longer warm up time? (5 pts)

(*e*) Why might a functional unit with an initiation interval of 4 and latency of 3 be less costly (as in dollars) than a functional unit with an initiation interval of 1 and a latency of 3?(5 pts)

☐ Include an illustration with your answer.

31 Fall 2002

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 25 October 2002, 10:40–11:30 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(13 pts)
Problem 3	_____	(13 pts)
Problem 4	_____	(44 pts)
Exam Total	_____	(100 pts)

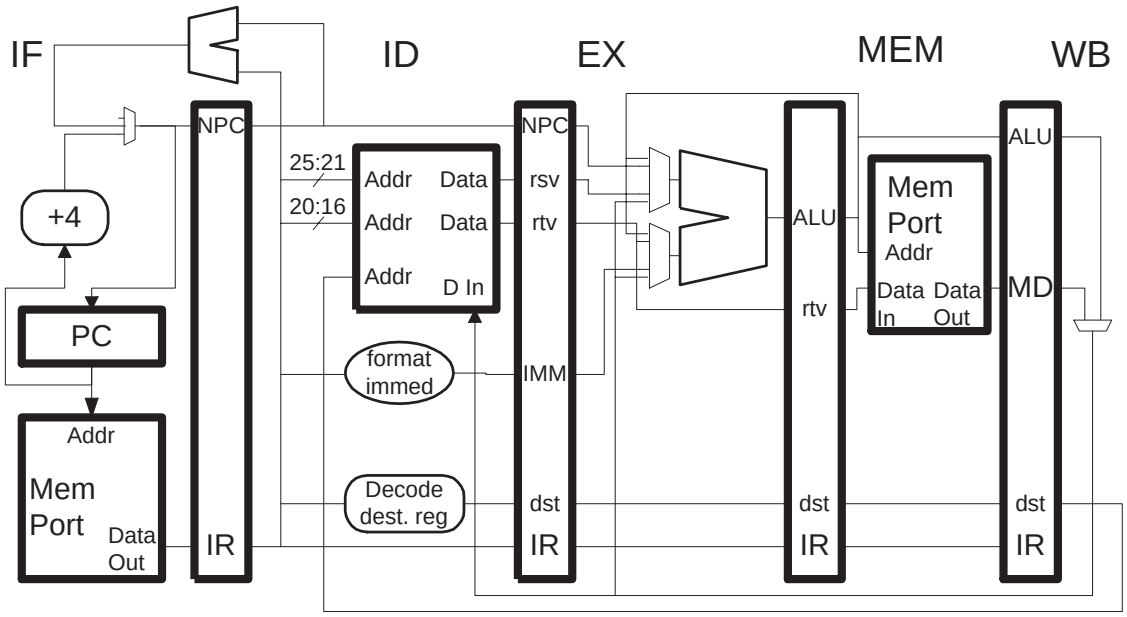
Alias _____

Good Luck!

Problem 1: A new MIPS branch instruction, `bieq rt,(rs) disp` (branch indirect equal) compares a register value to a memory location, if they are equal the branch is taken. The target is computed in the same way as other branches. In the code below, the contents of register `$s1` is compared to the contents of the memory location at address `$s2`. Like all MIPS control transfers, `bieq` has one delay slot. [30 pts]

(*a*) Modify the pipeline so that it can execute this new instruction. Show comparison units, multiplexors, and wires. **Do not** show control logic. For **partial credt** replace `bieq ...` with `bneq $s1,$s2, LOOP`.

- ☐ Use as much existing hardware as possible. **Do not** add a new memory port.
- ☐ The change should not reduce the clock frequency.
- ☐ Include the comparison unit for the branch condition.
- ☐ Add bypass paths so that the code below executes as shown.
- ☐ Label bypass paths with the cycle in which they are used. Include existing and any added bypass paths.
- ☐ Label the path carrying the branch target address with the cycle in which it is used.




```
LOOP:  # Assume bieq always taken.
 # Cycle               0    1    2    3    4    5    6    7    8    9   10
 srl  $s1, $s1, 1      IF   ID   EX   ME   WB

 addi $s2, $s2, 1           IF   ID   EX   ME   WB

 bieq $s1, ($s2), LOOP      IF   ID   EX   ME   WB

 sw   $s1, 100($s2)

 add  $0, $0, $0

 sub  $0, $0, $0
 or   $0, $0, $0
 xor  $0, $0, $0
```

Problem 1, continued:

(*b*) The pipeline execution diagram on the previous page only shows the first iteration.

☐ Continue the pipeline execution diagram to the beginning of the second iteration (when `srl` is fetched), *consistent with your solution to the first part.*

☐ Show which instruction is in each stage of the pipeline a cycle eight. Include squashed instructions, if any.

☐ Determine the CPI for a large number of iterations.

Problem 2: Convert the MIPS program below to SPARC, making use of condition codes. [13 pts]

- ☐ Make reasonable guesses about instruction names. Reasonable guesses will receive full credit.
- ☐ For the branch instructions, explain what the name stands for.
- ☐ Explain how each line of code uses the condition codes.
- ☐ Use the minimum number of registers.
- ☐ Do not rearrange instructions.

```
sub $s6, $s1, $s2
blt $s6, TARGET1
and $s3, $s6, $s4
beq $s3, $0, TARGET2
nop
beq $s6, $0 TARGET3
add $s5, $s6, $s3
```

Problem 3: Sometimes it's necessary to jump to a location specified in the program. [13 pts]

(*a*) SPARC V8 includes an instruction that can jump anywhere in the address space using an address specified in the instruction, for example, `call 0xabcd1234`.

☐ How does the `call` instruction, which is 32 bits, manage to specify a 32-bit jump target?

☐ Switching now to MIPS, show the minimum number of instructions needed to jump to address `0xabcd1234`. (If `jr` is used the instructions must put the address in a register. The address cannot be loaded from memory.)

(*b*) Without introducing a new format, add a new instruction to MIPS that would allow a jump to an arbitrary address using two instructions. The address must be specified in the instructions themselves.

☐ Use the instruction in an example.

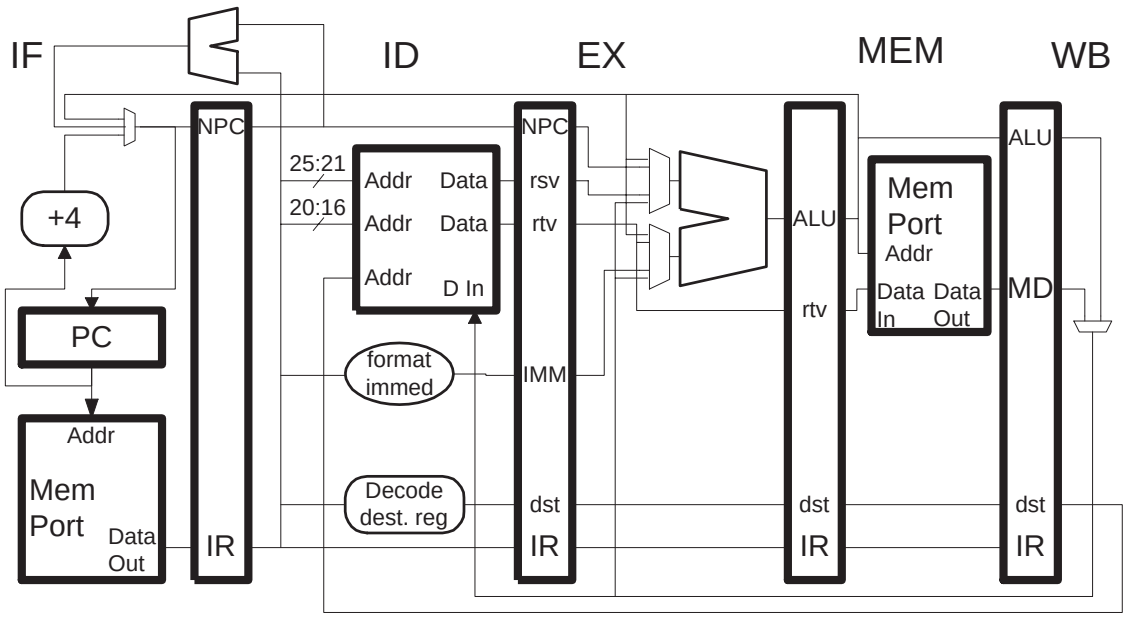
☐ Show the coding for the new instruction.

Problem 4: Answer each question below.

(*a*) Certain RISC ISA features are intended to facilitate pipelined implementations. [12 pts]

☐ Using the diagram below, briefly explain how RISC implementations benefit from fixed-length instructions and how things would be different with variable-length instructions.

☐ Using the diagram below, briefly explain how RISC implementations benefit from load/store instructions (restricting memory access to load instructions) and how relaxing the restriction would complicate things.



(*b*) Before there was MMX there was BCD. [12 pts]

☐ Why are binary coded decimal (BCD) and packed-integer data types superficially similar? What is an important difference between them?

☐ Describe a computation in which packed integer data types yield a performance advantage over ordinary integers. Show a brief advantage of the computation.

☐ Why were BCD data types added to some ISAs?

(*c*) Describe two compiler optimizations that reduce instruction count. [8 pts]

(*d*) The BAPCO benchmarks used by PC magazine and others evaluates a computer by timing the execution of a suite of benchmarks. The suite of benchmarks is drawn from common applications, such as Adobe Photoshop. The applications are in executable form, source code is not used.

Like BAPCO, SPEC CPU2000 consists of a suite of common applications, unlike BAPCO source code is used. Assume that the BAPCO and SPEC benchmarks are both well chosen and aimed at roughly the same user community. [12 pts]

☐ How is source code used in preparing the SPEC CPU 2000 benchmark results?

☐ Consider two processors, A and B. Suppose BAPCO ranks A faster but SPEC ranks B faster. How might the use of source code account for this difference?

☐ Taking in to account the differences due to the use of source code, who should make buying decisions based on BAPCO results? Who should make buying decisions on SPEC results?

Name _____

Computer Architecture
EE 4720
Final Examination

12 December 2002, 10:00–12:00 CST

Problem 1	_____	(10 pts)
Problem 2	_____	(17 pts)
Problem 3	_____	(11 pts)
Problem 4	_____	(17 pts)
Problem 5	_____	(45 pts)
Exam Total	_____	(100 pts)

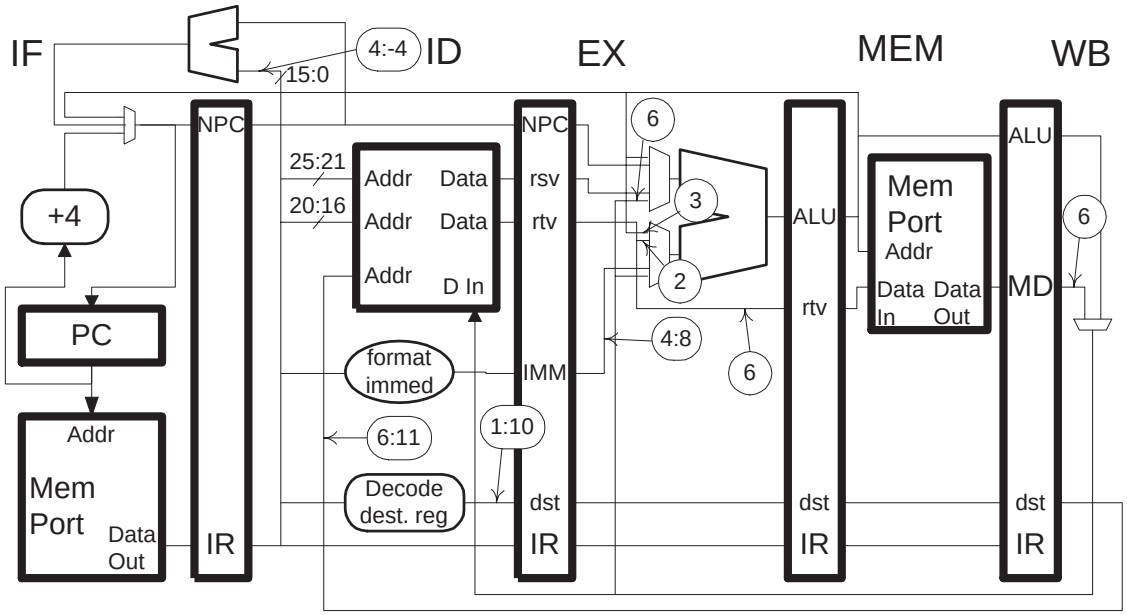
Alias _____

Good Luck!

Problem 1: In the diagram below certain wires are labeled with cycle numbers and, in some cases, values that will then be present, for example, 2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions, use the PED to help solve the problem. (10 pts)

☐ Write a program consistent with these labels.

☐ Use labels for branch targets (if any) and label the target line.




```
Cycle                0  1  2  3  4  5  6  7  8

                     IF ID EX ME WB

                        IF ID EX ME WB

                           IF ID EX ME WB

                              IF ID EX ME WB

                                 IF ID EX ME WB

Cycle                0  1  2  3  4  5  6  7  8
```

Problem 2: The PED below shows execution on a defective 1-way dynamically scheduled machine using Method 3. A diagram appears on the next page. The circuitry that's supposed to restore the ID map after a branch misprediction is not working, the ID map is left unchanged. Everything else works correctly, in particular the free list is correctly restored to the exact state it was right after the `lw`. (17 pts)

(*a*) For this incorrect execution:

☐ Show where each instruction commits.

☐ Complete the ID map. (See the phys. reg. file for the free list.)

☐ Complete the commit map, include the initial state.

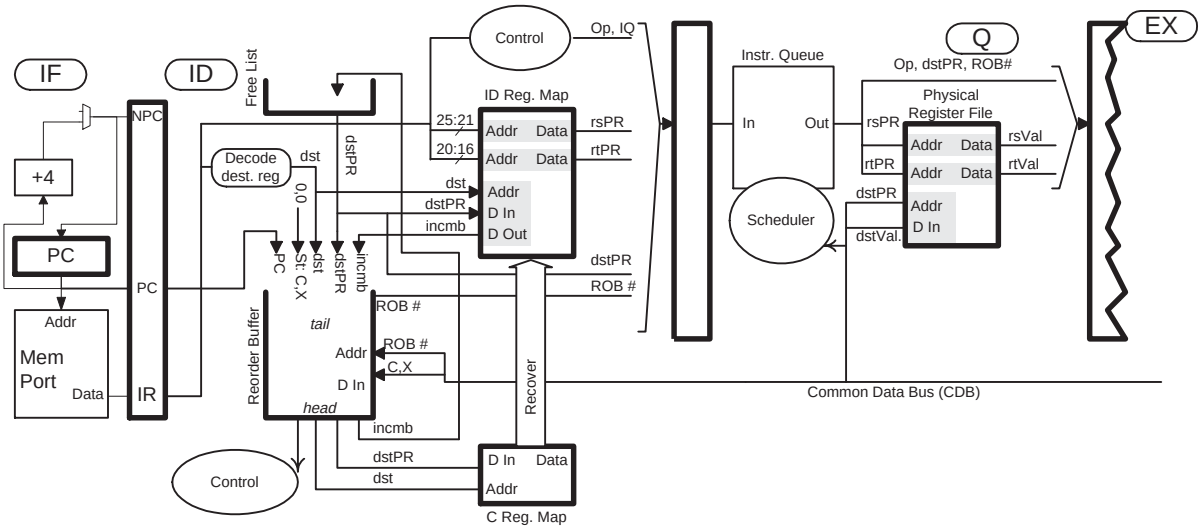
☐ Complete the physical register file.

☐ **In addition to other information** the physical register file should include a `[` in the cycle a register is removed from the free list and a `]` in the cycle it is returned to the free list.

☐ Circle incorrect entries (due to the defect) in the tables below.

```
 # lw loads a 0x200, lb loads a 0x202
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 bne $s1, $s2  SKIP  IF ID            Q  B  WB
 lw  $t1, 16($t1)       IF ID       Q  L1       L2 WB
 addi $t4, $t4, 3          IF ID Q  EX WB    x (squash)
 ori $t5, $t4, 0x30           IF ID Q  EX WB x (squash)
 nop ... (lots of nops)
SKIP:
 lb   $t1, 16($t1)                                IF ID Q  L1       L2 WB
 addi $t4, $t4, 3                                    IF ID Q  EX WB
 ori  $t5, $t4, 0xc0                                    IF ID Q  EX WB
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
ID Map
 t1   3
 t4   7
 t5  10
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Commit Map
 t1
 t4
 t5
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Physical Reg File.  All values in hex.  Free List: 18, 20, 23, 30, 37
 3  100
 7  101
 10 102
 18 103
 20 104
 23 105
 30 106
 37 107
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

Problem 2, continued: The illustration below is similar to one used in class; it is provided for reference.



(*b*) Suppose 1000 instructions later an instruction finds `0x4720` in register `t4` despite the fact that the code above wrote something else and no instruction wrote `t4` after that. (If the free list is used improperly the question might apply to `t5`, not `t4`. See the diagram for hints on proper use of the free list.)

Note: On the original exam register `t5` *was used instead of* `t4`. *Register* `t5` *can not change unexpectedly, though it still has an incorrect value.*

☐ How could that have happened? Be specific in how it's due to the defect.

(*c*) How could the defect have gone undetected?!? (That's not the question.)

☐ Suppose the `lw` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Remember, the only defect is with recovering the ID map on a branch misprediction. *Note: Original exam omitted "on a branch misprediction."*

☐ Now suppose the `lb` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Once again, the only defect is with recovering the ID map on a branch misprediction.

Problem 3: The code below runs on three systems: one using a bimodal (one-level) predictor, I; one using a two-level local history predictor, L; and one using a two-level gshare predictor, G. The global history register is 16 bits and the local history is 16 bits. The branch history tables have 256 entries. Assume that each branch below has its own BHT entry. The branch outcomes for B2 and B3 are provided for your solving convenience. Note that B3 has the same pattern as B2 but at a different phase, the phase difference is important. (11 pts)

```
BIGLOOP: LOOP:
 B1: bne $t1, $0, SKIP  # Random, independent, taken 25% of time.
 ...
 SKIP:
 B2: bne $t2, $t5, SKIP2 # Pattern: N N N N N N T T N N N N N N T T N N N N N N T T ....
 ...
 SKIP2:
 B3: bne $t7, $t8, SKIP3 # Pattern:  N N N N N T T N N N N N N T T N N N N N N T T N ....
 ...
 SKIP3:
 B4: bne $t9, $t10, LOOP # Iterates 50 times.
...
 j BIGLOOP
 nop
```

(*a*) Determine the prediction accuracy for each branch and each predictor. The accuracies should be after warmup. **Do not** compute the number of entries. Approximate the accuracy of B1 predictions.

☐ B1 on I: Accuracy:

☐ B1 on L: Accuracy:

☐ B1 on G: Accuracy:

☐ B2 on I: Accuracy:

☐ B2 on L: Accuracy:

☐ B2 on G: Accuracy:

☐ B3 on I: Accuracy:

☐ B3 on L: Accuracy:

☐ B3 on G: Accuracy:

☐ B4 on I: Accuracy:

☐ B4 on L: Accuracy:

☐ B4 on G: Accuracy:

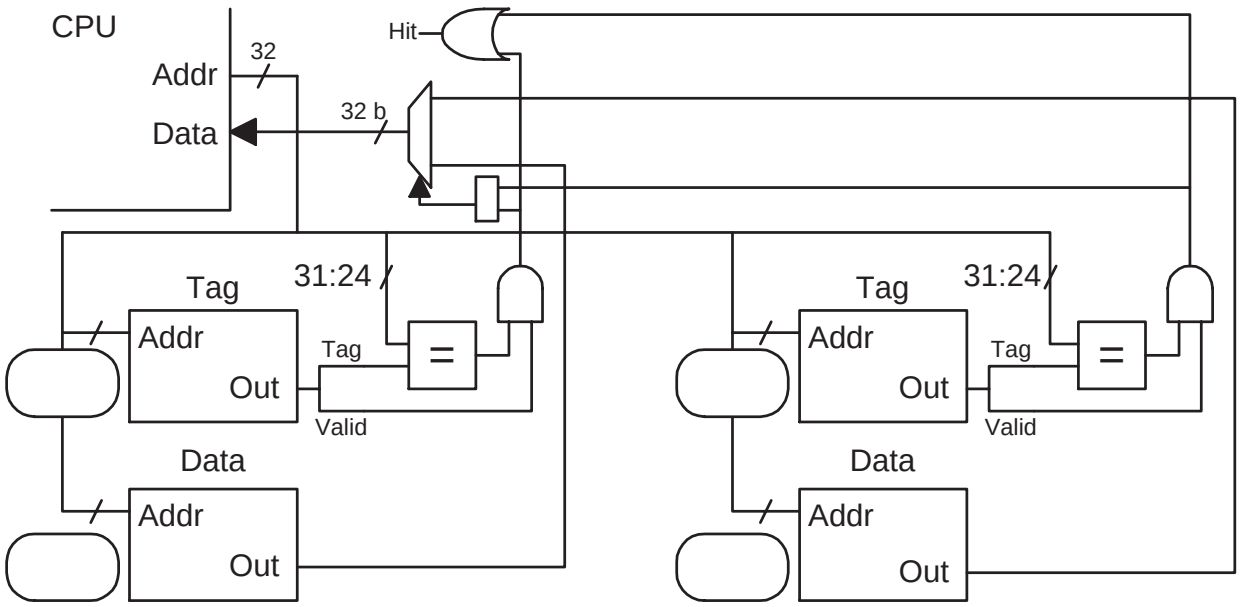
(*b*) A gshare predictor using a 15-bit GHR (instead of the 16-bit GHR used above) should give the same prediction accuracy for branch B2. How small can the GHR be made without changing the prediction accuracy of B2? Assume there are no collisions. *Note: The original question asked about B3.*

(*c*) For the local predictor, how small can the local history be made without affecting the prediction accuracy of B2 and B3?

Problem 4: The diagram below is for a cache with 256-character lines on a system with 8-bit characters. (17 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Cache Capacity (Indicate Unit!):

☐ Memory Needed to Implement (Indicate Unit!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array.

```
int *a = 0x100000;  // sizeof(int) = 4 characters
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ];
```

☐ What is the hit ratio for the program above?

(*c*) In the problem below, only consider accesses to the arrays.

```
int *a = 0x10000;    // sizeof(int) = 4 characters
int *b = 0x20000;
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ] + b[ i ];
```

☐ What is the minimum size of a direct mapped cache for which the program above will have a 100% hit ratio on the second j iteration? Explain.

☐ What is the minimum size of a two-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

☐ What is the minimum size of a three-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

Problem 5: Answer each question below.

(*a*) In the PED below for a statically scheduled 2-way superscalar machine the `xor` instruction stalls in IF even though there is an empty space in ID. (5 pts)

```
add s1, s2, s3    IF ID EX ME WB
or  s4, s1, s5    IF ID -> EX ME WB
xor s6, s7, s8       IF -> ID EX ME WB
and s9, s10, s11     IF -> ID EX ME WB
```

☐ Why?

☐ Would it be difficult or would it be impossible to modify the implementation (but keeping it statically scheduled) so that such an instruction could move to ID? Explain.

(*b*) In a dynamically scheduled system why should store instructions wait until they are ready to commit before storing the data? (6 pts)

(*c*) The same code fragment executes on 1-way, statically scheduled systems with the specified FP add functional unit(s). For each show a pipeline execution diagram. Don't forget to consider *all* structural hazards and check carefully for dependencies. All adders take a total of four cycles to compute a result (latency is 3). (5 pts)

☐ One adder, `A`, initiation interval: 1.

```
add.d f0, f2, f4

add.d f6, f0, f8

add.d f10, f0, f12
```

☐ One adder, `A`, initiation interval: 2.

```
add.d f0, f2, f4

add.d f6, f0, f8

add.d f10, f0, f12
```

☐ One adder, `A`, initiation interval: 4.

```
add.d f0, f2, f4

add.d f6, f0, f8

add.d f10, f0, f12
```

☐ Two adders, `A` and `B`, each has initiation interval: 4.

```
add.d f0, f2, f4

add.d f6, f0, f8

add.d f10, f0, f12
```

(*d*) One problem in Homework 5 was to analyze the execution of an unoptimized program to compute π. Many people got the question below wrong, here's your chance to get it right! (5 pts)

☐ Should computer engineers analyze the performance of processors running unoptimized code? Explain.

(*e*) Consider a new data type, binary coded trianary. This 32-bit data type consists of 16 radix-3 digits, each coded with two bits, in the same way a BCD data type would consist of 8 radix-10 digits. (8 pts)

☐ Why might 3- and 9-fingered aliens find this data type useful?

☐ Explain how the data type would be useful in accessing arrays in which the element size was a power of three.

In a meeting next week you plan to argue that this data type should be included in the next revision of the ISA, one of several proposed new data types. Only one will be chosen. Three-fingered aliens are **not** expected to make up a large portion of your customer base.

☐ How will you argue that the data type should be included?

☐ What will you do in the next week to prepare your argument?

(*f*) In Homework 6 the performance of a linear search of an array and linked list were compared. On the dynamically scheduled systems the search was much faster on the array. Now consider the array program on a statically scheduled system. (8 pts)

```
\scoreable Would the array program perform as well?  Consider
the effect of memory latency and line size.\par

\solution{The array program would still perform better than the linked
list program, but with longer memory latency or shorter line size the
array program on the dynamically scheduled system would run
substantially faster.  }
```

☐ Explain. *Hint: the solution above was intentionally included.*

(*g*) Predicated instructions can be used to avoid branches. (8 pts)

☐ Why do predicated instructions have maximum benefit over branches that skip over one instruction (compared to predicated instructions used to avoid branches that skip over more than one instruction)?

☐ Suppose the compiler is considering whether to replace a branch that skips n instructions with predicated instructions. How could the compiler use an estimate of prediction accuracy, f_{correct}, and resolution time, t_{resolve}, as well as implementation details to make its decision?

32 Spring 2002

Name _____

Computer Architecture
EE 4720
Midterm Examination

Friday, 22 March 2002, 13:40–14:30 CST

Problem 1	_____	(35 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

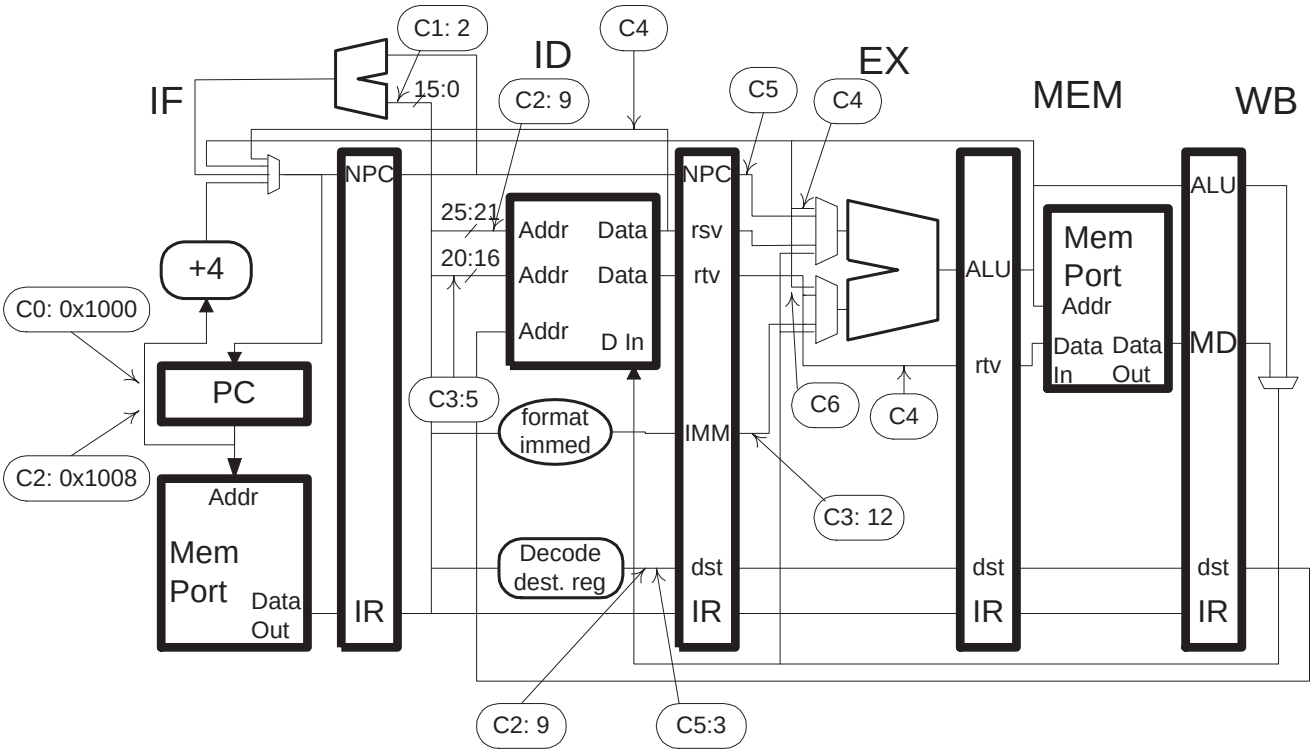
Alias _____

Good Luck!

Problem 1: In the diagram below certain wires are labeled with cycle numbers and values that will then be present, for example, C2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions. *Note: The diagram in the original exam had an error that resulted in contradictory information about the third instruction (fetched in cycle 2). The error was a* C3:5 *pointing to the input to the* `ID/EX.dst` *latch; that now points to the rt address input to the register file.* [35 pts]

☐ Write a program consistent with these labels.

☐ Use labels for branch targets (if any) and label the target line.




```
# Cycle          0    1    2    3    4    5    6    7    8    9
                 IF   ID   EX   ME   WB
                      IF   ID   EX   ME   WB
                           IF   ID   EX   ME   WB
                                IF   ID   EX   ME   WB
                                     IF   ID   EX   ME   WB
                                          IF   ID   EX   ME   WB
```

# Cycle	0	1	2	3	4	5	6	7	8	9

Problem 2: Consider the CISCy instruction below: [25 pts]

(*a*)

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

☐ What makes it CISCy? (CISCy means characteristic of CISC ISAs.)

☐ Ignoring instruction size, why would it be difficult to implement such an instruction on the kind of five-stage pipeline used in class for MIPS?

Problem 2, continued:

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

(*b*) How could the single-issue (not superscalar) pipeline used in class be modified so that instructions like the one above could be executed with a potential for a CPI of 1 without impacting clock frequency? (Instructions like the one above have one destination and two sources, the destination and first source can either use a register value or memory value specified with displacement addressing, the second source can be either a register value or an immediate.) Feel free to throw hardware at the problem, but do not assume that the hardware is any faster than what we've been using.

☐ Show a sketch of the pipeline, briefly explaining what should be done in each stage.

☐ Show a pipeline execution diagram for the following code on your new pipeline, assuming no dependencies.

```
add 0x1234(r1), 8(r2), r3

or r4, 7(r5), r6

and 0x1234(r7), r8, r9
```

☐ In the part above, why was it necessary to *assume* no dependencies?

Problem 3: Answer each question below.

(*a*) What'll it be? One FP adder with an initiation interval of 2 and a latency of 3 (four cycles of computation) or two FP adders each with an initiation interval of 4 and a latency of 3? [10 pts]

☐ What is the maximum number of FP adds per second with each alternative on a 1 GHz system?

☐ Show a code fragment in which one of the alternatives is slightly faster. For the alternative with two adders, use label "A" for one adder and "B" for the other.

☐ Ignoring the cost of the adders themselves, which alternative would be more costly and why?

(*b*) SPEC benchmark numbers are provided in "base" and "peak" forms. [10 pts]

☐ How are they different?

☐ When should an intelligent (passed EE 4720) computer buyer use the base numbers, and when should the buyer use the peak numbers?

(*c*) Stack ISAs had burrowed their way in to our past and percolated to the fore in the past decade. [10 pts]

☐ Why are programs compiled for stack ISAs smaller than the same programs compiled for other ISAs? (Assume all compilers are of high quality.)

☐ Why would superscalar implementations of stack ISAs be less efficient (further from the ideal CPI) than superscalar implementations of RISC and some other "conventional" ISAs? Ignore instruction fetch problems. Consider the simple stack programs presented in class.

(*d*) At last superscalar implementations and VLIW ISAs will wage their epic [tm] battle now at the dawn of the twenty-first century.[10 pts]

☐ What distinguishes a superscalar implementation from a nonsuperscalar implementation? (VLIW isn't part of this question).

☐ Explain two ways in which VLIW machines overcome problems encountered in superscalar implementations of conventional (say RISC) ISAs?

Name _____

Computer Architecture
EE 4720
Final Examination

18 May 2002, 12:30–14:30 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(37 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(23 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

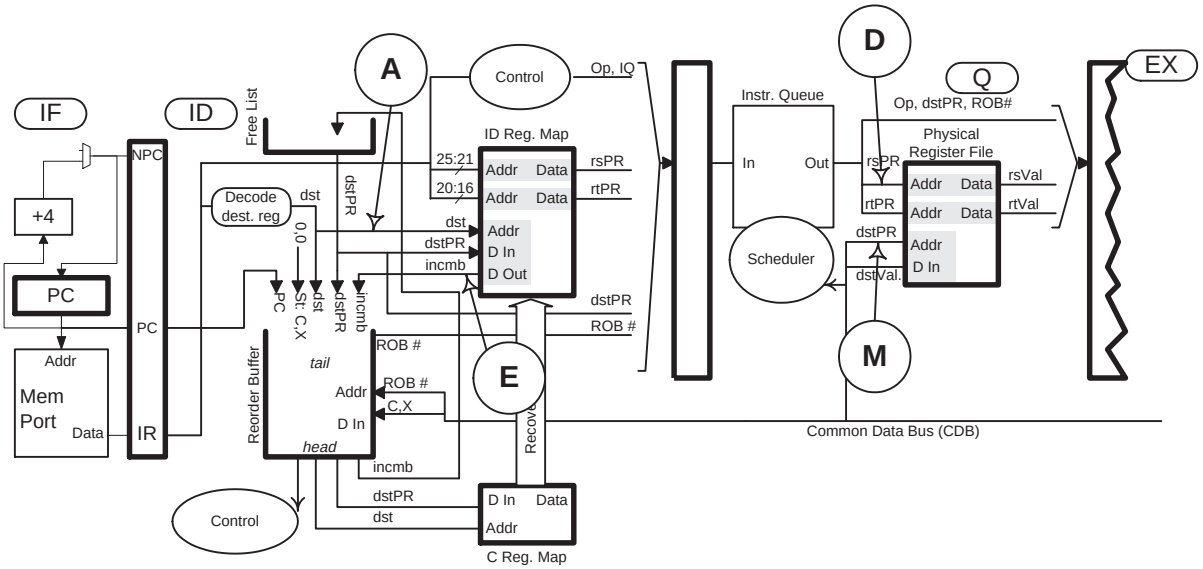
Problem 1: The (hopefully familiar) code fragment on the next page executes as shown on the dynamically scheduled system illustrated below.

(*a*) For each entry in the Physical Register File table on the next page place a "[" at the cycle(s) at which a physical register is allocated (removed from the free list) and place a "]" at the cycle(s) at which it is placed back in the free list. (This was part of the solution to Homework 5.) (10 pts)

(*b*) The diagram below includes circled letters, these letters appear in a Signals Values Table on the next page.

☐ (10 pts) Fill in the table showing the signals that will appear on the labeled wires.

- Use register names (`f0`, `$1`, etc.) for architected registers (but not for physical registers!).
- Use a question mark for a physical register number that cannot be determined from the tables.
- Since the machine is four-way superscalar each wire holds four values. Do not show values for the instructions that are not in the table, such as the two instructions following `sub`.
- Assume that values have been correctly computed, even if they depend upon preceding instructions in the same group. (This affects row E in the table.)



```
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 ldc1 f0, 0($1)   IF ID Q  L1 L2 WC
                           IF ID Q  L1 L2 WB              C
                                    IF ID Q  L1 L2 WB             C
                                             IF ID Q  L1 L2 WB
 mul.d f0, f0, f2 IF ID Q        M1 M2 M3 M4 M5 M6 WC
                           IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                    IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                             IF ID Q        M1 M2 M3 M4 M5
 sdc1 0($1), f0   IF ID Q  L1                      L2 WC
                           IF ID Q  L1                      L2 WC
                                    IF ID Q  L1                      L2 WC
 addi $1, $1, 8   IF ID Q  EX WB                     C
                           IF ID Q  EX WB                   C
                                    IF ID Q  EX WB                   C
 bne $2, $0 LOOP     IF ID Q  B  WB                   C
                             IF ID Q  B  WB                 C
                                      IF ID Q     B  WB               C
 sub $2, $1, $3      IF ID Q  EX WB                   C
                             IF ID Q  EX WB                 C
                                      IF ID Q  EX WB                 C
 # ID Map         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99              97,96    94,93    91,90
 $1 98              95       92       89
 # In cycle one first 97 is assigned to f0, then 96 (replacing 97).  The
 # same sort of replacement occurs in cycles 4 and 7.
 # Commit Map     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99                          97               96    94 93    91 90
 $1 98                                              95       92       89
 # Physical Reg File    2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 99               1.0
 98               0x1000
 97                            10
 96                                              11
 95                         0x1008
 94                                     20
 93                                                       2.2
 92                                  0x1010
 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17

 # Signal Values  0    1    2    3    4    5    6    7    8    9    10   11   12

 A



 D



 E



 M

 # Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12
```

Problem 2: The `add` instruction below is a predicated version of a MIPS instruction (for this exam). If the contents of register `$s5` (for the example) is non-zero the `add` instruction executes normally. If it's zero then it's as though the `add` never executed. Any GPR can be used to hold the predicate.

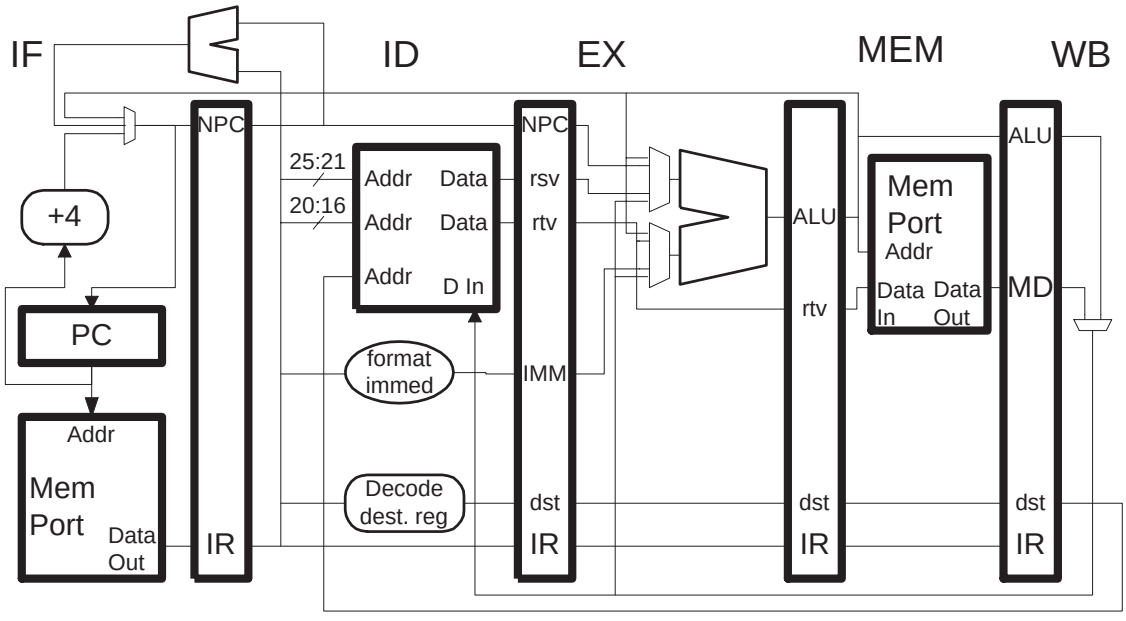
```
 # Cycle                          0  1  2  3  4  5  6
           xor $v1, $a0, $a1      IF ID EX ME WB
           lw $s5, 0($t2)            IF ID EX ME WB
     ($s5) add $s1, $s2, $s3            IF ID EX ME WB
```

(*a*) Suppose there are predicated versions of other two-source, one-destination instructions. How might they be coded in MIPS? The coding should be chosen to ease implementation. (5 pts)

(*b*) Modify the pipeline below to implement predication; it should run the code above correctly and without stalls (as illustrated). The added hardware must detect whether the predicate is true or false and take appropriate action. It should include bypassing, for example, to handle the dependency carried by `$s5` or from `xor` if the predicate were `$v1`.

Hint: A correct solution uses multiplexors, an $\boxed{is\ P}$ *(box that recognizes a predicated instruction), assorted gates, and some modifications to existing components.*

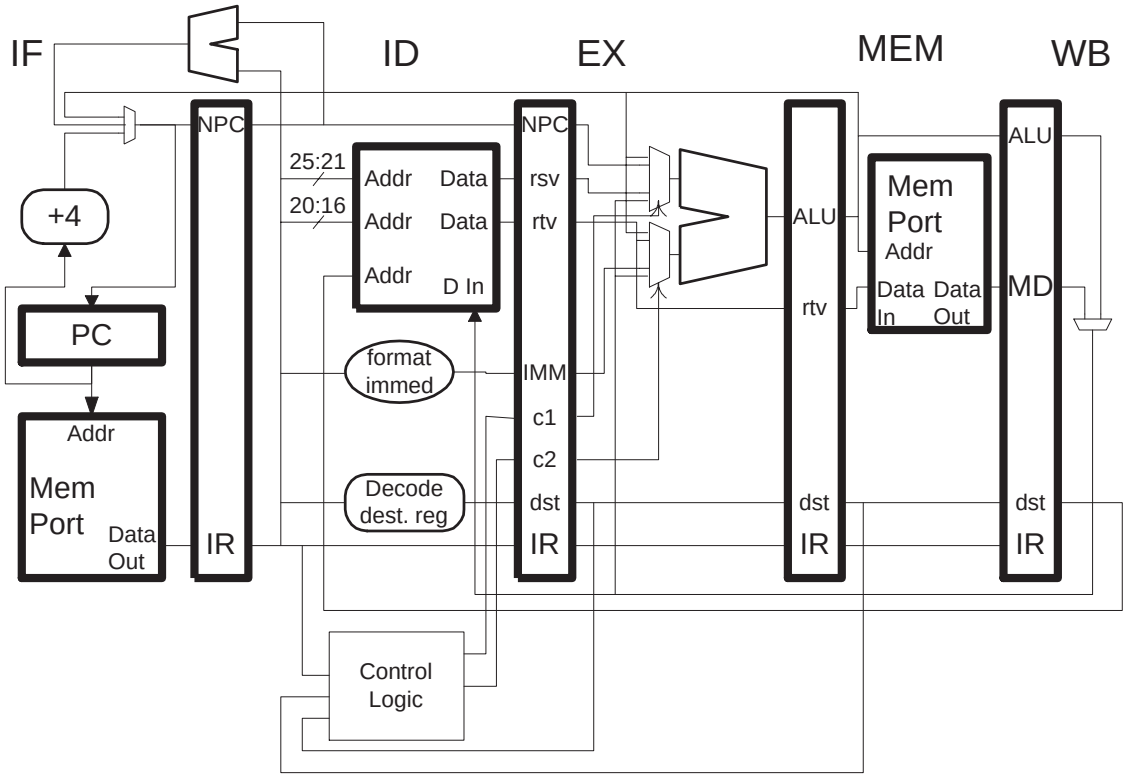
(9 pts)



Problem 2, continued:

(*c*) Assume predicates were implemented as requested in the previous part. Explain why bypassing of the dependency carried by `$s1` in the program below won't work for the hardware below. Explain how the problem might be fixed. (8 pts)

```
# Cycle                      0  1  2  3  4  5  6
          xor $s5, $a0, $a1   IF ID EX ME WB
          lw $s5, 0($t2)         IF ID EX ME WB
     ($s5) add $s1, $s2, $s3        IF ID EX ME WB
          or  $s4, $s1, $v0            IF ID EX ME WB
```



Problem 2, continued: Consider such predicated instructions in a dynamically scheduled implementation using method 3. The implementation without predicate prediction (the next two parts) should realize the benefit of predicated instructions: avoiding the need for branches.

(*d*) How might the ID Register Map be updated for predicated instructions? (For partial credit: Why is this a good question?)(5 pts)

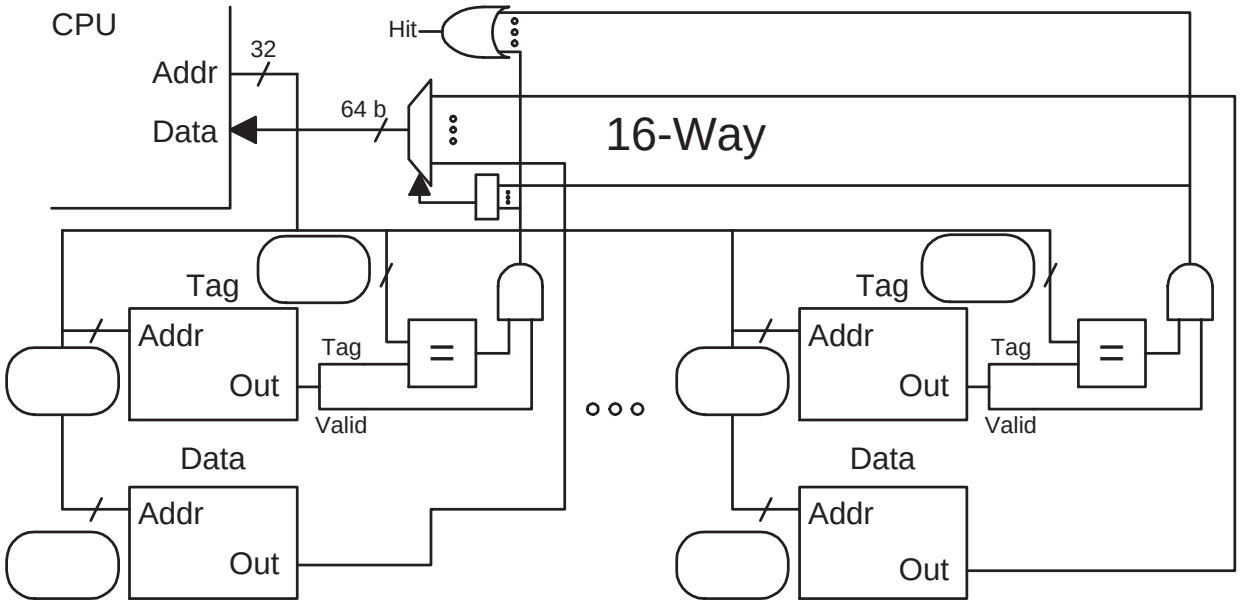
(*e*) Based on your answer above, how is execution affected if the predicate turns out to be false? (5 pts)

(*f*) Consider a dynamically scheduled system using predicate prediction. What should be done when a predicate is mispredicted? Under what circumstances (properties of program being run) would it be faster than a system without predicate prediction? Under what circumstances would it be slower? (5 pts)

Problem 3: The diagram below is for an 2 MiB (2^{21} byte) 16-way set-associative cache, with a line size of 128 characters, for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Memory Needed to Implement (Indicate Unit!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
half *h = 0x100000;  // sizeof(half) = 2 characters
int sum;
int i,j;
int ISTRIDE = 1;
int ILIMIT = 1024;

for(j=0; j<4; j++)
  for(i=0; i<ILIMIT; i++)
    sum += h[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

☐ What is the hit ratio for the program above?

☐ Find the *minimum* value of `ISTRIDE` needed to maximize the miss ratio. (That is, *minimize* the hit ratio, make things really slow.) Don't forget that the cache is set associative. Do not modify `ILIMIT`.

Problem 4: Answer each question below.

(*a*) In the pipeline execution diagram below the multiply is squashed to avoid the WAW hazard. How does this make a precise exception impossible? (5 pts)

```
mul.d  f0, f2, f4  IF ID M1x
add.d  f0, f6, f8     IF ID A1 A2 A3 A4 WB
```

(*b*) How do processes share memory in a virtual memory system? Provide an example in which two processes share address `0x12000` but address `0x34000`, used by each process, is not shared. The addresses must be used in the example. (5 pts)

(*c*) Show how the branch history table and pattern history tables are connected in a local history branch predictor. Show how the table is indexed and how its contents are used to make a prediction. (5 pts)(For partial credit show a one-level [bimodal] predictor.)

(*d*) One way to improve performance is to divide the pipeline into more stages, as in the Pentium 4 compared to the Pentium III. This does not reduce the amount of time it takes to compute things, such as sums, though. Given that: (8 pts)

☐ How are dependencies potential performance limiters when dividing pipeline stages?

☐ How does the Pentium 4 Fast ALU get around that?

☐ Are dependencies still a problem or can we now use zillion-stage pipelines (at least as far as dependencies are concerned)? Explain.

☐ Why is branch prediction accuracy more important when there are more stages?

33 Fall 2001

Name _____

Computer Architecture
EE 4720
Midterm Examination

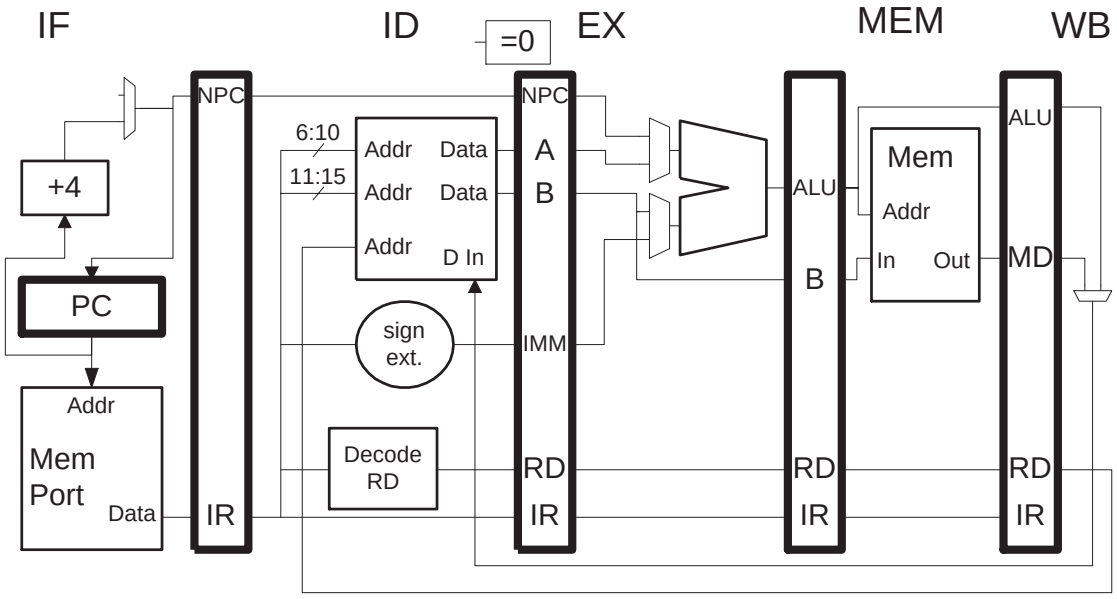
Friday, 26 October 2001, 13:40–14:30 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(60 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The DLX implementation below lacks bypass paths and, worse than that, lacks the hardware needed for control-transfer instructions.



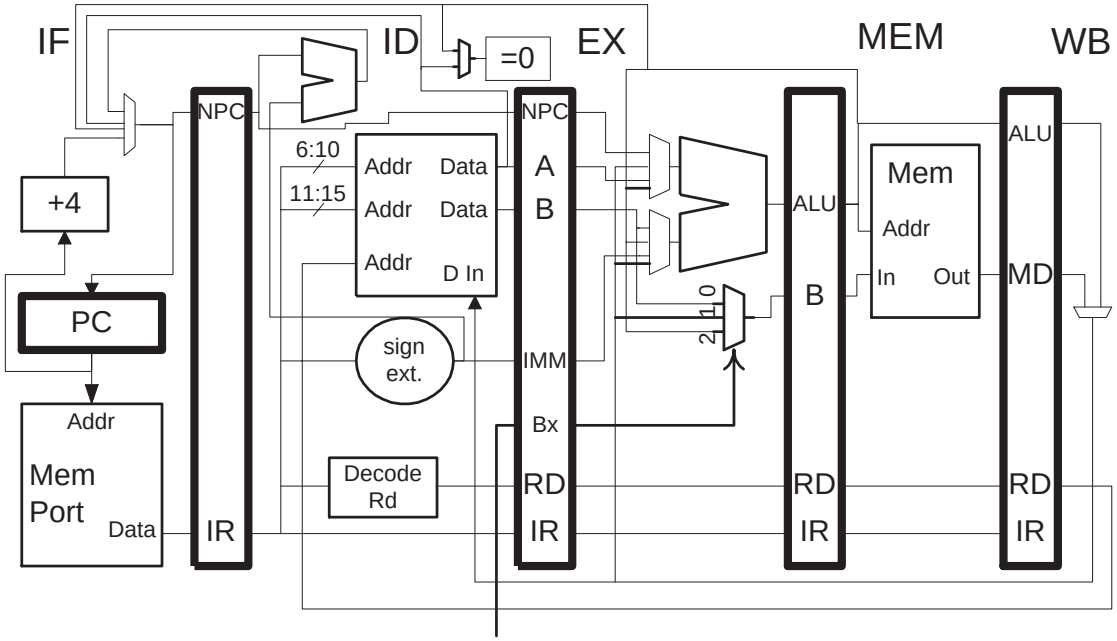
☐ [5 pts] Add exactly the hardware needed so that the control-transfer instructions execute *as shown* below. Include a connection to the $\boxed{=0}$ box used in determining whether a branch is taken.

☐ [5 pts] Add exactly those bypass paths necessary so that the code below executes *as shown*. Check the code carefully for dependencies, including all those related to the `jalr` instruction.

☐ [5 pts] Show the cycles in which each added wire will be used.

```
! Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 ori r1, r1, #15    IF ID EX ME WB
 bnez r1, SKIP         IF ID EX ME WB
 add r0, r0, r0           IF IDx
 xor r0, r0, r0              IFx
SKIP:
 sub r20, r20, r21              IF ID EX ME WB
 jalr r20                          IF ID EX ME WB
 xor r0, r0, r0                       IFx
 ...
 add r15, r31, r0                        IF ID EX ME WB
 or  r16, r16, r15                          IF ID EX ME WB
! Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
```

Problem 2: The DLX implementation below includes bypass paths into the `EX/MEM.B` register.



☐ [5 pts] Write a program that uses all three paths into the `EX/MEM.B` register. *Hint: It's easy.*

☐ [10 pts] Design the control logic for the multiplexor feeding the `EX/MEM.B` register. The control logic should be in the ID stage and feed into the `ID/EX.Bx` pipeline latch provided in the diagram above.

Problem 3: Registers `r1` and `r2` each contain a signed integer, call them i and j. Register `r10` contains an address, call the address A. Let $p = i \times j$.

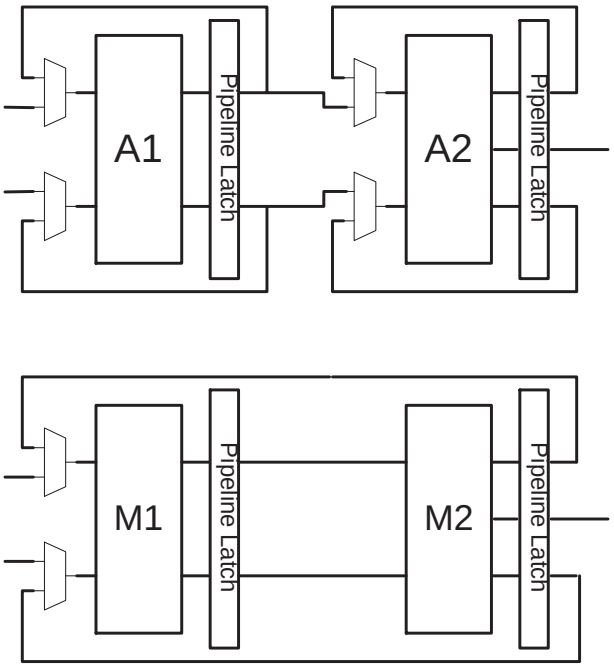
☐ [10 pts] Starting at address A write p in three formats: integer, double-precision floating-point, and single-precision floating-point. Maintain as much precision as possible.

Hint: A reasonable solution would use `movitof` *and* `cvtXtoY` *instructions.*

```
! Initially: r1 and r2 each contain an integer, i and j.
!            r10 contains an address.
!
!            At Mem[r10]    write i * j (integer format).
!            At Mem[r10+?]  write i * j (double-precision FP)
!            At Mem[r10+??] write i * j (single-precision FP)
```

Problem 4: Answer each question below.

(*a*) In the two pipelined functional units below an instruction must pass through each segment twice. The `A`'s are for FP addition and `M`'s are for FP multiplication.



☐ [10 pts] Complete the pipeline execution diagram below for a system using these functional units. Don't overlook the dependency through `f6`.

```
addd f0, f2, f4

addd f6, f8, f10

muld f12, f6, f14

muld f16, f18, f20
```

(*b*) The code below, which of course is not DLX, uses memory-indirect and autoincrement addressing.

```
lw r1, @(r2)  ! Memory-indirect load.
lw r4, (r5)+  ! Autoincrement
lh r6, (r7)+  ! Autoincrement
```

☐ [10 pts] Rewrite the code in DLX.

(*c*) The three types of interrupts discussed in class are traps, hardware interrupts, and exceptions.

☐ [6 pts] For each one explain how the exception code is determined.

☐ [6 pts] For each one explain where control returns after the handler completes.

(*d*) An ISA has two implementations, A and B; each implementation has a well-written compiler.

☐ [5 pts] Would the code compiled by A's compiler run on implementation B? Briefly explain.

☐ [5 pts] A program is compiled using A's compiler and B's compiler. How might the compiled code differ? Provide a reason for the difference.

(*e*) You have become the owner of a large American computer company, congratulations.

☐ [6 pts] How can you (legally) influence the decision-making process so that SPECs next benchmark suite does not unfairly put your company's products at a disadvantage? (Note: Bribery is illegal in the U.S.)

(*f*) DLX does not have delayed branches, but many other RISC ISAs do.

☐ [6 pts] What is a delayed branch and how does it help?

(*g*)

☐ [6 pts] How and why is the CPI affected in an implementation re-designed for a higher clock frequency?

Name _____

Computer Architecture
EE 4720
Final Examination

11 December 2001, 7:30–9:30 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The code fragment below executes as shown on a single-issue dynamically scheduled system using a physical register file (Method 3). Initially register `f0` contains a 0, `f2` contains a 0.2, and `f4` contains a 0.4. The value computed by each instruction is shown near the right margin. (20 pts)

- [ ] Show where each instruction commits.
- [ ] Complete the ID- and commit-stage register map tables.
- [ ] Complete the physical register file table.
- [ ] Be sure to show the initial values for `f0`, `f2`, and `f4` in the register maps and the register file.
- [ ] In the physical register file use a [to indicate when a register is removed from the free list and use a] to indicate when it's put back.

```
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
addd f0, f0, f2    IF ID Q                          A1 A2 WB                         (.2)
addd f4, f2, f4       IF ID Q  A1 A2 WB                                             (.6)
addd f2, f0, f8          IF ID Q                             A1 A2 WB               (1.)
addd f0, f4, f8             IF ID Q  A1 A2 WB                                      (1.4)
addd f4, f2, f8                IF ID Q                                A1 A2 WB     (1.8)
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22

ID Register Map:   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
f0
f2
f4

Commit Reg Map:    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
f0
f2
f4

Physical Reg File: 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
0    0
1    .2
2    .4
3
4
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
5
6
7
8
9
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
```

Problem 2: The code fragment below executes on three different systems, I, L, and G, each using a different branch predictor:

I: One-level predictor, 2^{16}-entry BHT. L: Sixteen-bit (two-level) local history. G: Sixteen-bit (two-level) gshare.

In the spaces provided below show the accuracy of, and the number of entries used by, the indicated branch on the indicated system. The accuracy and number of entries should be after warmup. The number of entries used for the gshare scheme can be approximate. If there is more than one table, show the number of entries in each table separately.(20 pts)

☐ B1 on I: Accuracy: Entries:

☐ B2 on I: Accuracy: Entries:

☐ B3 on I: Accuracy: Entries:

☐ B1 on L: Accuracy: Entries:

☐ B2 on L: Accuracy: Entries:

☐ B3 on L: Accuracy: Entries:

☐ B1 on G: Accuracy: Entries:

☐ B2 on G: Accuracy: Entries:

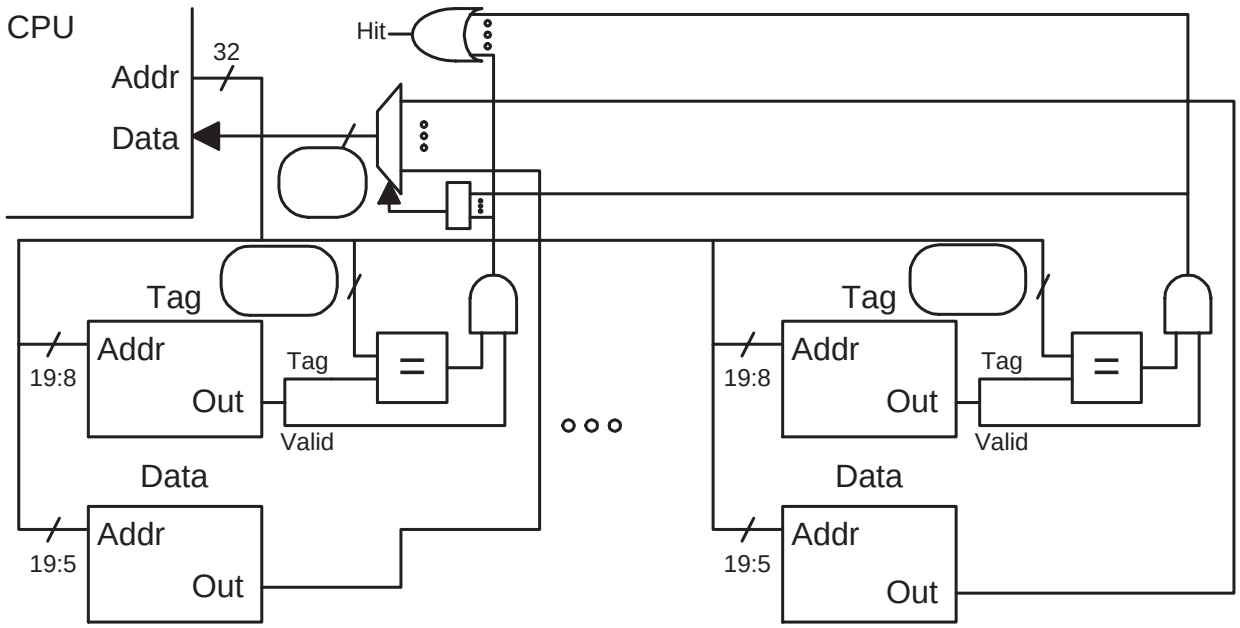
☐ B3 on G: Accuracy: Entries:

```
BIGLOOP:

LOOP2: ! Iterates 100 times
 ! Random, 0 or 1
    lw r1, 0(r2)
    addi r2, r2, #4
B1: bnez r1, SKIP1
    add r10, r10, r11
SKIP1:
    sub r3, r2, r12
B2: bnez r3, LOOP2

    addi r1, r0, #3
LOOP3:
    subi r1, r1, #1
B3: bnez r1, LOOP3

j BIGLOOP
```

Problem 3: The diagram below is for an 8 MiB (2^{23} byte) set-associative cache for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☐ Fill in the blanks in the diagram.



☐ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

Address:

☐ Associativity:

☐ Line Size (Indicate Unit!):

☐ Memory Needed to Implement (Indicate Unit!):

☐ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Address:

Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
double *d = 0x100000;  // sizeof(double) = 8 characters
double sum;
int i;
int ISTRIDE = 1;
int ILIMIT = 64;

for(i=0; i<ILIMIT; i++)
  sum += d[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

☐ What is the hit ratio for the program above?

☐ Modify `ISTRIDE` and `ILIMIT` so that the cache, which remember is set-associative, is completely filled *with the minimum number of accesses*.

Problem 4: Answer each question below.

(*a*) The contents of the load/store queue in a dynamically scheduled system is shown below for $t = 4720$. At this particular time the cache is empty, but with a load/store queue full of instructions it won't be empty for long. The instruction in entry `L0` is the oldest.

The EA field indicates the effective address, a `#1` in this field indicates that the effective address cannot be computed until the instruction in ROB entry #1 (not shown) completes. It completes at $t = 7700$.

The Data field indicates the data that will be written by the store instruction in the entry.

The cache is nonblocking and the load/store queue can process an unlimited number of instructions per cycle.

☐ (5 pts) For each load instruction at $t = 4721$ if it's complete show what data it has loaded otherwise indicate the first thing it's waiting for.

```
    Type  EA      Data   Data loaded or reason for waiting.

L8: load  #1

L7: load  0x1000

L6: store 0x2000  66

L5: store 0x1000  55

L4: load  0x1000

L3: store #1      33

L2: store 0x1000  22

L1: load  0x2000

L0: load  0x1000
```

(*b*) The exception recovery mechanism in the R10000 does not need a commit map, the one in Method 3 does.

☐ (5 pts) But the R10000 takes longer to recover. Why does it take longer and how much longer does it take?

(*c*) In some schemes for recovering from branch mispredictions on dynamically scheduled systems recovery can start at completion rather than waiting for commitment.

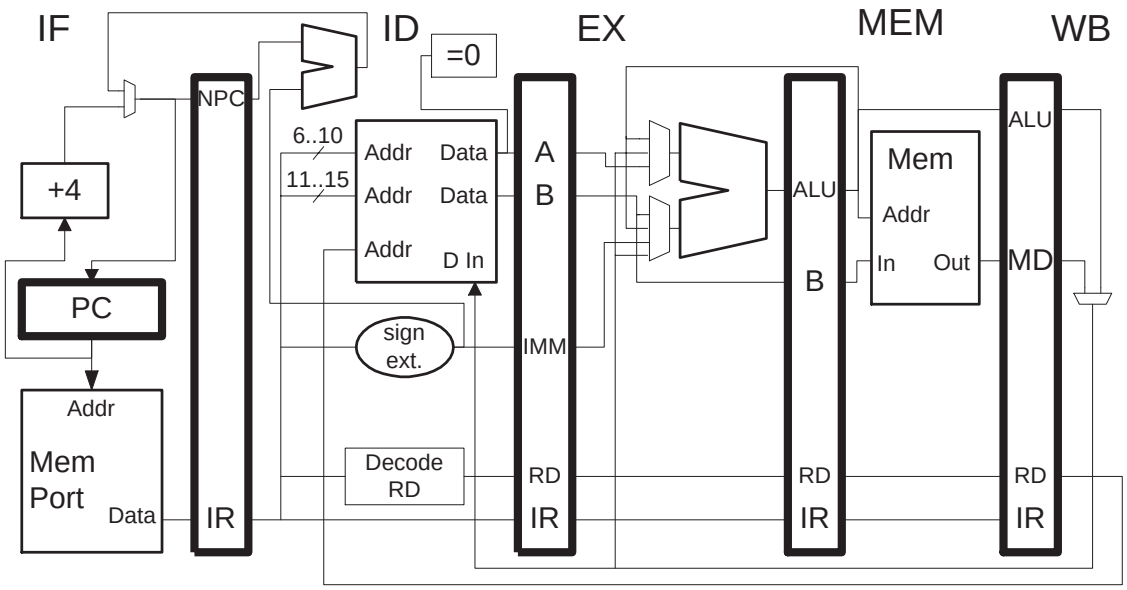
☐ (5 pts) What additional hardware is needed for this? How is it used?

(*d*)

☐ (5 pts) Show how bits are categorized in a real address and a physical address in a system with a 64-bit virtual address space, a 40-bit real address space and 2^{14}-character pages.

(*e*) Connections for the `jr` instruction are omitted from the pipeline below.

☐ (5 pts) Add them.



☐ (5 pts) Show a pipeline execution diagram for the code below consistent with the modified hardware above.

```
 slli r10, r1, 2
 addi r10, r10, r2
 lw   r10, 0(r10)
 jr   r10

TARG: !Target of jr.
 add r20, r21, r22
```

(*f*) Here are the IA-64 instructions used in the solution to Homework 3: `ld1` (load byte), `cmp.eq` (compare equal), `cmp.le` (compare less than or equal), `cmp.gt` (compare greater than), `add` (add of course), `st1` (store byte), and `br` (branch).

☐ (5 pts) Convert the DLX code below to IA-64, be sure to use predicated instructions.

☐ Show the stops.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```

☐ (5 pts) Besides needing fewer instructions, how does predication speed execution? (Not necessarily in IA-64 implementations.) Modify the DLX program (without changing the branch and jump) so that predication would be less useful.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```

34 Spring 2001

Name _____

Computer Architecture
EE 4720
Midterm Examination

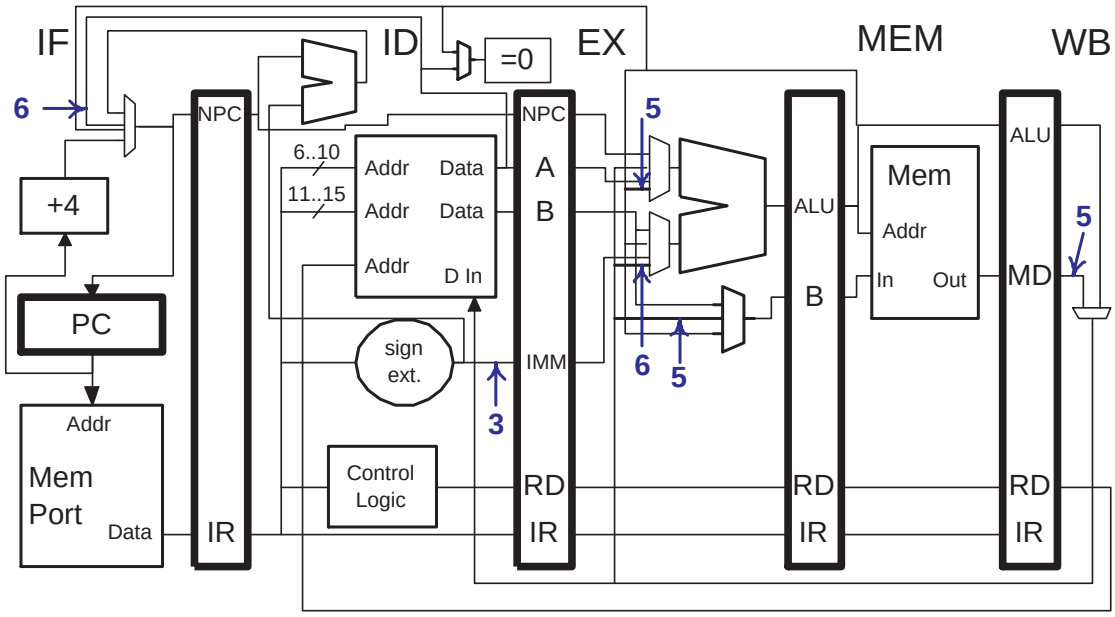
Wednesday, 21 March 2001, 13:40–14:30 CST

Problem 1	_____	(35 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(40 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: In the DLX implementation below six paths are marked with a number, a cycle in which the path will be used. Write a DLX program that uses those six paths at the indicated cycles. Some marked paths are *bypass* paths and some are not; the bypass paths can be used **only** at the cycles indicated. A pipeline execution diagram is provided for your convenience.



☐ [15 pts] Choose the register operands so the bypass paths are used **only** at the cycles indicated.

☐ [15 pts] Choose the instructions so that the marked paths will be used as indicated. There need be only one control transfer instruction.

☐ [5 pts] One instruction will be squashed. Show where by crossing out the segment labels (`ID`, etc.) in the diagram.

```
! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
            IF  ID  EX  MEM WB
                IF  ID  EX  MEM WB
                    IF  ID  EX  MEM WB
                        IF  ID  EX  MEM WB
                            IF  ID  EX  MEM WB
                                IF  ID  EX  MEM WB
                                    IF  ID  EX  MEM WB
                                        IF  ID  EX  MEM WB
                                            IF  ID  EX  MEM WB
! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: The DLX code below runs on a dynamically scheduled system that uses reorder buffer entries to name destination registers (Method 1). The floating-point adder has **five stages**, `A1` through `A5`, and is fully pipelined. The common data bus (CDB) can handle an unlimited number of writebacks per cycle, but other parts of the implementation are ordinary. The cache is non-blocking. The `cvtftoi` instruction uses the FP adder; the `movfptoi` instruction uses the integer (`EX`) functional unit. The pipeline is fully bypassed.

(*a*) For this part no instructions raise exceptions.

☐ [7 pts] Show a pipeline execution diagram for the code. Reorder buffer and reservation station numbers **do not** have to be shown.

☐ [5 pts] Indicate where each instruction commits.

Check the code carefully for dependencies! Register `r1` *is part of a long chain of dependencies. Pay attention to the register equality and inequality comment.*

```
!  r4 = r2,  r6 != r2,  r6 != r1
!Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16 17 18 19
addf f0, f1, f2

cvtftoi f0, f0

movfptoi r1, f0

subd f0, f4, f6

sd 0(r1), f10

sd 0(r2), f0

ld f12, 0(r4)

ld f14, 0(r6)

!Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16 17 18 19
```

(*b*) Suppose an arithmetic exception is discovered for the `subd` instruction when it is in the first FP adder stage, `A1`, in an execution of the program above.

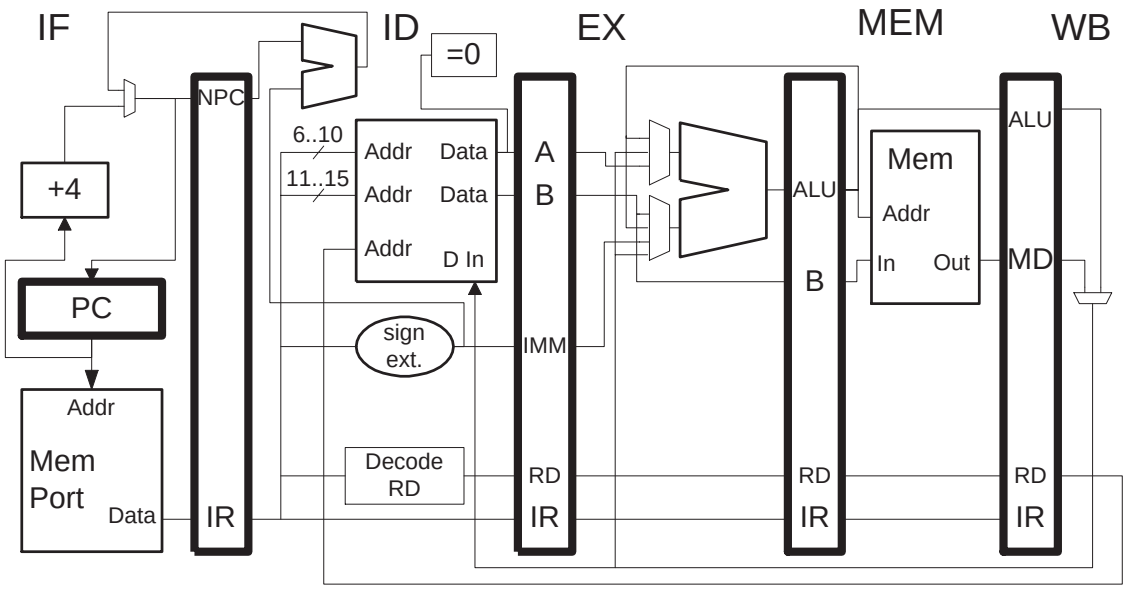
☐ [6 pts] At what cycle is the reorder buffer flushed and the handler fetched?

☐ [7 pts] Show the contents of the reorder buffer when the status of the `subd` instruction is set to exception. (For partial credit show the contents of the reorder buffer halfway through execution, be sure to indicate the cycle.)

Problem 3: Answer each question below.

(*a*)

☐ [8 pts] Design the control logic for the WB-stage multiplexor. The control logic itself must be in the ID stage. Show the connection to the mux and number the mux inputs. *Hint: This is an easy problem.*



(*b*) In the pipeline execution diagram below `multd` is delayed because of a true dependency with the `addd` instruction.

☐ [8 pts] Why can't any realistic implementation do things this way?

```
addd  f0, f2, f4   IF ID A1 A2 A3 A4 WB
multd f6, f0, f8               IF ID M1 M2 M3 M4 M5 M6 WB
subd  f10, f12, f14               IF ID A1 A2 A3 A4 WB
```

(*c*) Unlike DLX, many ISAs, such as SPARC, use a condition code register for integer branch conditions.

☐ [4 pts] In what way is DLX's method more flexible?

☐ [4 pts] In what way is a condition code register more flexible?

(*d*) An ISA may have variable-width instructions, fixed-width instructions, and bundled instructions.

☐ [4 pts] How do the different alternatives affect displacement branches?

[4 pts] Name an ISA category (type) that uses each instruction format:

☐ Variable-Width Instructions:

☐ Fixed-Width Instructions:

☐ Bundled Instructions:

(*e*) Packed-operand data types and instructions are *de rigueur* for any *au courant fin-de-siècle* ISA. (Are a must-have for any up-to-date end-of-the-century ISA.)

☐ [8 pts] What are packed-operand data types and instructions? Show a short program that would benefit from these. The program can be in a high-level language or even pseudo code. Do not show the packed-operand instructions, just explain what they would do.

Name _____

Computer Architecture
EE 4720
Final Examination

11 May 2001, 15:00–17:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The *reference count* of a value stored in a register is the number of times the value is referenced (read) by instructions before being overwritten. In the example below the reference count of the value written by the `add` is zero because it is overwritten before being read. The reference count of the value written by the `lw` is two.

```
add r1, r2, r3   ! A first value for r1 is defined (written).
lw  r1, 0(r4)    ! A second value for r1 is written, the first one is never used.
sub r5, r5, r1   ! The second value is referenced (read).
xor r6, r1, r7   ! The second one is referenced again.
or  r1, r0, r0   ! A third value for r1 is defined.
```

Three new instructions are to be added to DLX to obtain reference count information. The instructions refer to two registers, `rc.z` and `rc.nz`. Register `rc.z` holds the number of values written to `r1` with a zero reference count and `rc.nz` holds the number of values written to `r1` with a non-zero reference count. The counts are updated on either the first reference or when a new value is defined, whichever is earlier.

Instruction `rc.reset` sets both of these registers to zero. Instruction `movstoi RD, rc.z` moves the contents of `rc.z` to a general-purpose register (shown as `RD`), `movstoi RD, rc.nz` moves the contents of `rc.nz` to a general-purpose register.

The instructions are used in the code below.

```
rc.reset           ! rc.z -> 0,  rc.nz -> 0
add r1, r2, r3
lw  r1, 0(r4)      ! rc.z -> 1
sub r5, r5, r1     ! rc.nz -> 1
xor r6, r1, r7
and r1, r0, r0
or  r1, r0, r0     ! rc.z -> 2
slli r1, r1, #1    ! rc.nz -> 2
movstoi r8, rc.z   ! r8 -> 2
movstoi r9, rc.nz  ! r9 -> 2
add r10, r8, r9    ! Total number of writes to r1.
```

The new instructions should work for `r1` and no other register. (25 pts)

(*a*) For each new instruction below cross out the instruction type that could not reasonably be used to code it. Circle the type that would best be used to code it.

☐ `rc.reset` Possible types: Type R, Type I, Type J.

☐ `movstoi RD,rc.z` Possible types: Type R, Type I, Type J.

(*b*) As the alert test taker has noticed, there is already a `movstoi` instruction, used to move a special-purpose register (such as the processor status word, not covered much in class) to a general-purpose register.

☐ The new `movstoi RD,rc.z` instruction can be coded as a new instruction (with its own opcode) or as a new use of `movstoi`. That would depend on how the existing `movstoi` is defined. Explain that. *Hint: Suppose there was only one special register in DLX.*

Problem continued on next page.

(*c*) Modify the pipeline below to implement the new instructions. The code on the previous page should execute properly.

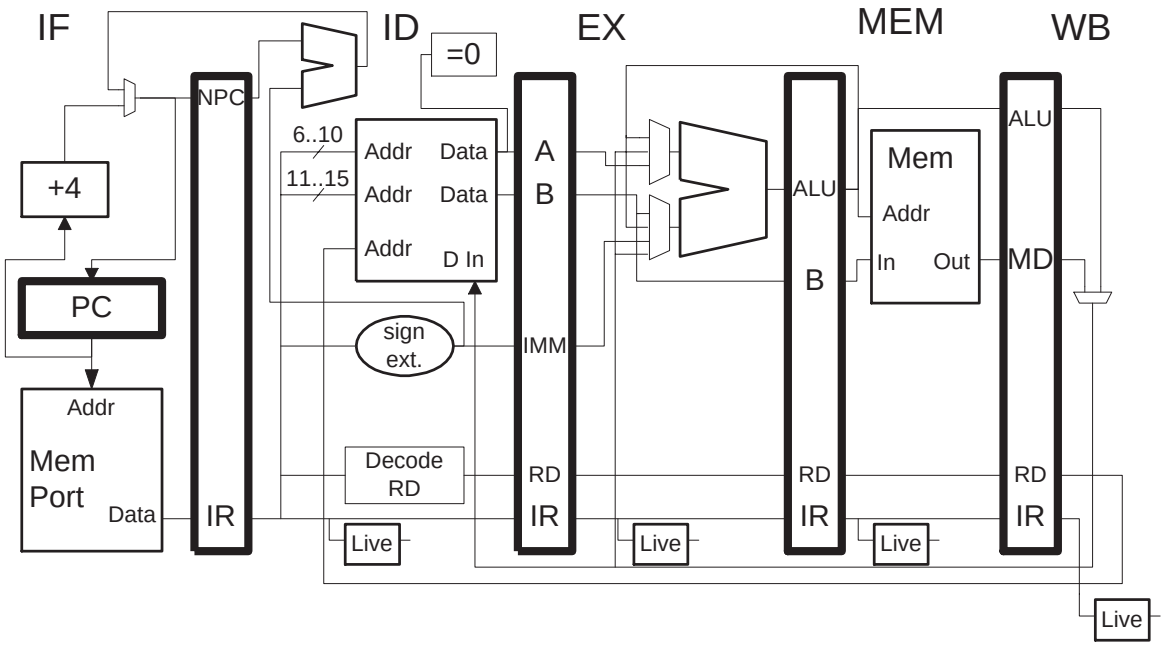
Assume that all Type-R instructions have two source operands and all Type-I instructions have one source operand. Use [is Type R] and [is Type I] to recognize these instruction types.

Assume that Decode RD in the diagram can recognize the new instructions

☐ The counts should not include squashed instructions.

☐ Show control logic for all multiplexor and registers that you add. Use symbols like [=rc.reset] to recognize instructions, **show the inputs to these boxes**.

☐ Don't forget the modifications for `movstoi RD, rc.z` and `movstoi RD, rc.nz`.



Problem 2:

(*a*) The code below executes on a dynamically scheduled single-issue (not superscalar) machine using Method 1, register names are reorder buffer entry numbers. The multiply unit is six stages and is fully pipelined (latency 5, initiation interval 1). The add unit has a latency of 3 and an initiation interval of 2. The CDB can handle an unlimited number of writebacks per cycle. Floating point exceptions are **not** precise. (15 pts)

☐ Show a pipeline execution diagram.

☐ Show where instructions commit.

☐ Show changes to the reorder buffer, register map and register file at each cycle. Register f0 initially contains zero, f2 initially contains 20.0; f4, 40.0; etc. Use line numbers for reorder buffer entry numbers.

```
L1:multd  f0, f2, f4

L2:addd   f0, f0, f6

L3:addd   f2, f2, f8

L4:addd   f10, f12, f14
```

☐ The pipeline execution diagram above is the same whether or not exceptions are precise. Why?

(*b*) The code below executes on a two-way superscalar statically scheduled DLX implementation with perfect branch and jump target prediction. (10 pts)

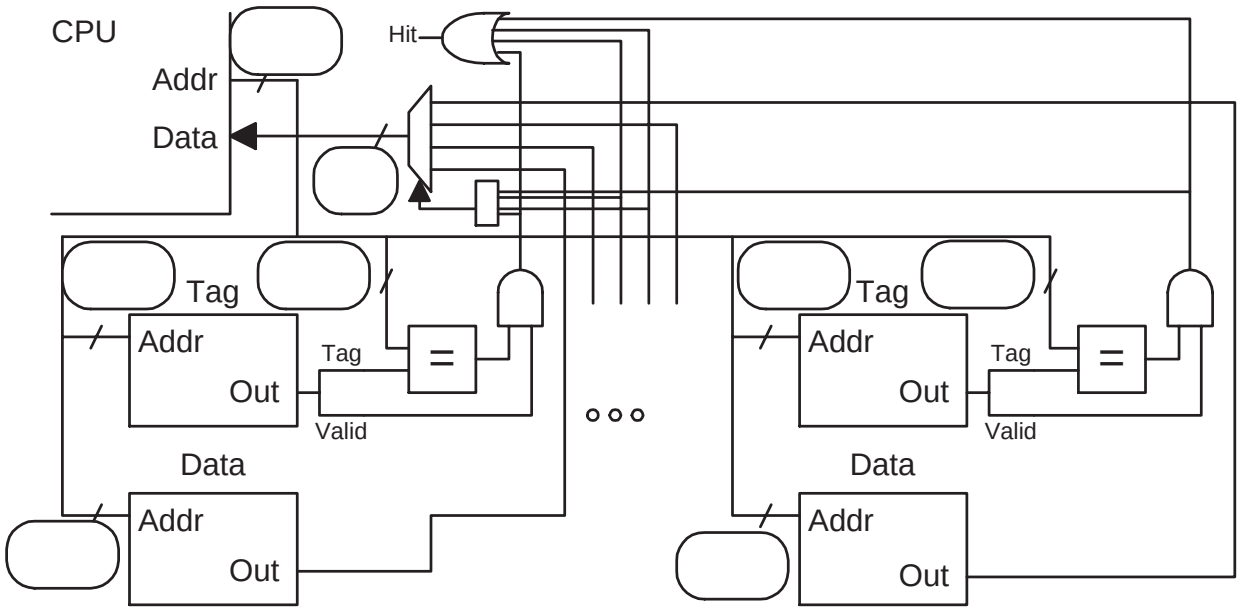
☐ Show a pipeline execution diagram for the worst-case execution of the code below for two iterations. (The worst case is achieved by proper choice of the labels, `LOOP` and `THERE`.)

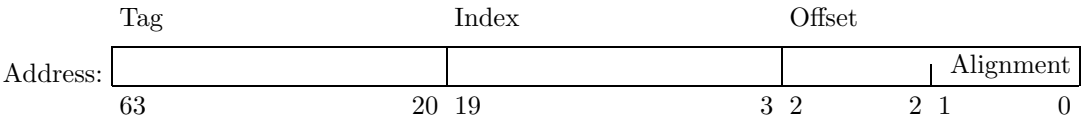
☐ Explain why it's worst case.

☐ What is the CPI of the for a large number of iterations?

```
 addi r1, r0, #2000
LOOP:
 subi r1, r1, #1
 j THERE
HERE: ! Immediately follows the instruction above.
 bnez r1, LOOP

THERE:
 add r2, r2, r3
 j HERE
```

Problem 3: The cache for a system implementing an ISA with 10-bit characters is described by the following incomplete schematic diagram and address bit categorization.





(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☐ Fill in the blanks in the diagram.

☐ In the diagram above the processor's data out port is not shown. Given what is shown is the cache probably write back or write through? Argue your answer in terms of what is missing or what is present.

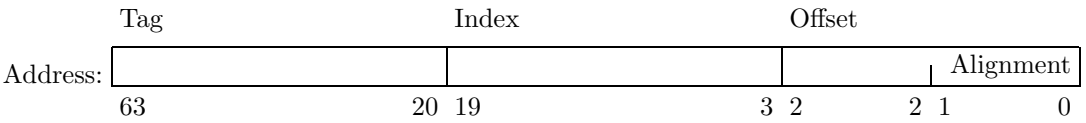
☐ What is the associativity of this cache? (Look carefully.)

☐ What is the capacity of the cache? Specify units!

☐ How much memory does it take to implement the cache? Specify units!

Problem 3, continued: Continue to use the cache from the previous part.

(*b*) When the program below starts execution no data is cached. Consider only memory accesses needed for the array access. The address bits are repeated for convenience. (15 pts)




```
extern char *a;    // & a[ 0 ] = 0x1000000;
int i, j, k, t;
int ISTRIDE = 1024;
for(k=0; k<2; k++) for(i=0; i<256; i++) for(j=0; j<32; j++)
  t += a[ ISTRIDE * i + j ];
```

☐ What is the hit ratio?

☐ What is the smallest line size that would maximize the hit ratio?

☐ What is the smallest *reasonable* line size that would maximize hit ratio?

☐ What is the smallest power-of-two value of ISTRIDE for which a two-way set-associative cache is necessary to avoid conflict misses?

☐ If ISTRIDE were larger than the answer to the previous question but not a power of two, would there be lots of conflict misses? Explain.

Problem 4: Answer each question below.

(*a*) One-level predictors are usually outperformed by two-level predictors. (5 pts)

☐ Write a code fragment containing a branch that would be predicted well by a two-level predictor such as gshare but poorly by a one-level predictor.

(*b*) Virtual memory allows two processes on the same processor to use the same address as though they were on different systems, that is, one process never loads what the other stores. (5 pts)

☐ Why doesn't one process loading from, say, address `0x1000` get data previously stored by another process at address `0x1000`?

(*c*) A processor implementing a 64-bit ISA has the following integer add latencies: 8- and 16-bit integers, 0 cycles; 32-bit integers, 1 cycle; 64-bit integers, 2 cycles; 128-bit integers, 3 cycles. (5 pts)

☐ Why are these latencies not appropriate for a 64-bit ISA? In what important way would the processor suffer?

(*d*) Precise exceptions are optional for floating-point instructions but required for integer instructions. (5 pts)

☐ Why is this so?

☐ Illustrate your answer with an example showing an integer instruction that raises an exception.

☐ Explain what goes wrong if the exception is not precise.

(*e*) What are stops and lookaheads? (5 pts)

☐ How are they used?

☐ What benefit to they have to implementations?

☐ Explain the similarities and differences between stops and lookaheads.

35 Fall 2000

Name _____

Computer Architecture
EE 4720
Midterm Examination, Part I

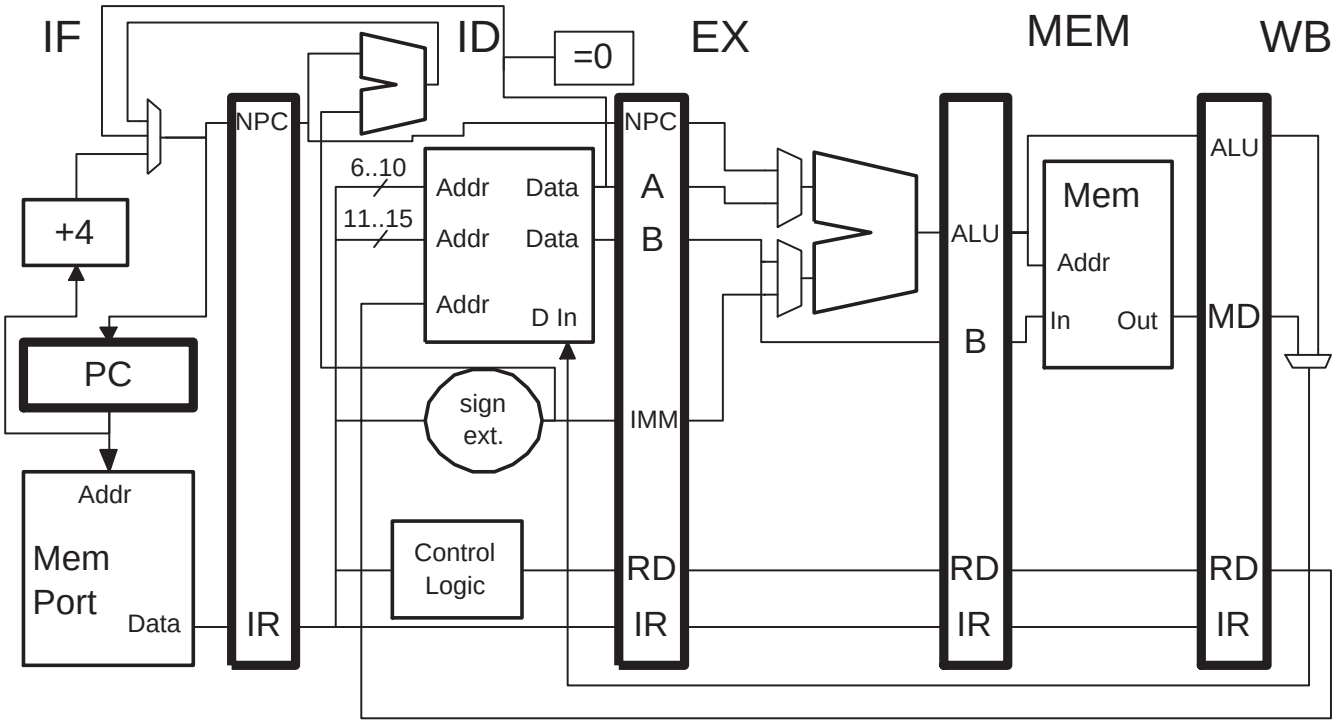
Monday, 16 October 2000, 12:40–13:30 CDT

Problem 1	_____	(17 pts) Mon.
Problem 2	_____	(17 pts) Mon.
Problem 3	_____	(16 pts) Mon.
Problem 4	_____	(13 pts) Wed.
Problem 5	_____	(17 pts) Wed.
Problem 6	_____	(20 pts) Wed.
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The program below executes on the pipeline below as illustrated in the pipeline execution diagram below. Bypass paths do not appear in the illustration (below).

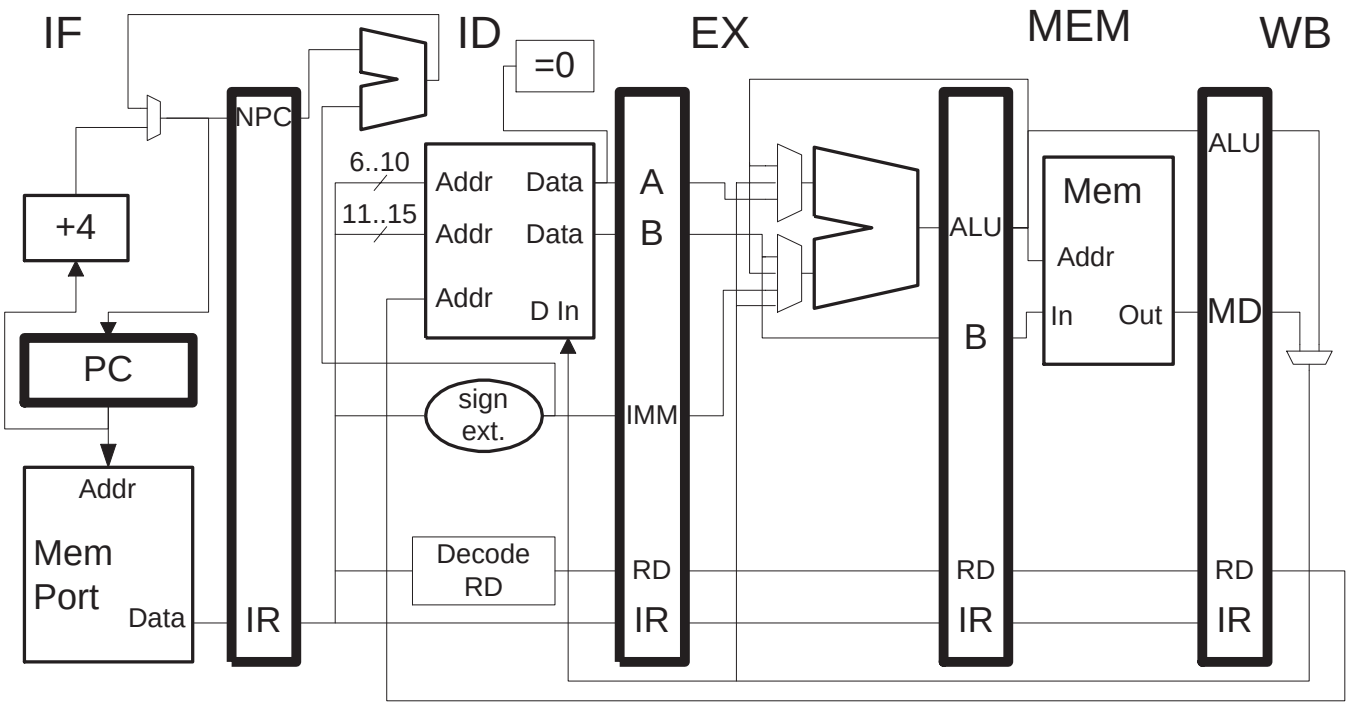



```
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11
andi r8, r8, #31  IF ID EX ME WB
add  r10, r9, r8     IF ID EX ME WB
bnez r8, LINEX          IF ID EX ME WB
jalr r10                   IF ID EX ME WB
xor                           IFx
...
subi r31, r31, #8                IF ID EX ME WB
sw   0(r10), r31                    IF ID EX ME WB
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11
```

☐ [10 pts] Add *exactly* those bypass paths that are needed so that the code (above) executes as shown. Credit will be deducted for unneeded bypasses. *Please, please, please check the code carefully for dependencies.*

☐ [7 pts] Next to each bypass path indicate the cycle(s) in which it will be used.

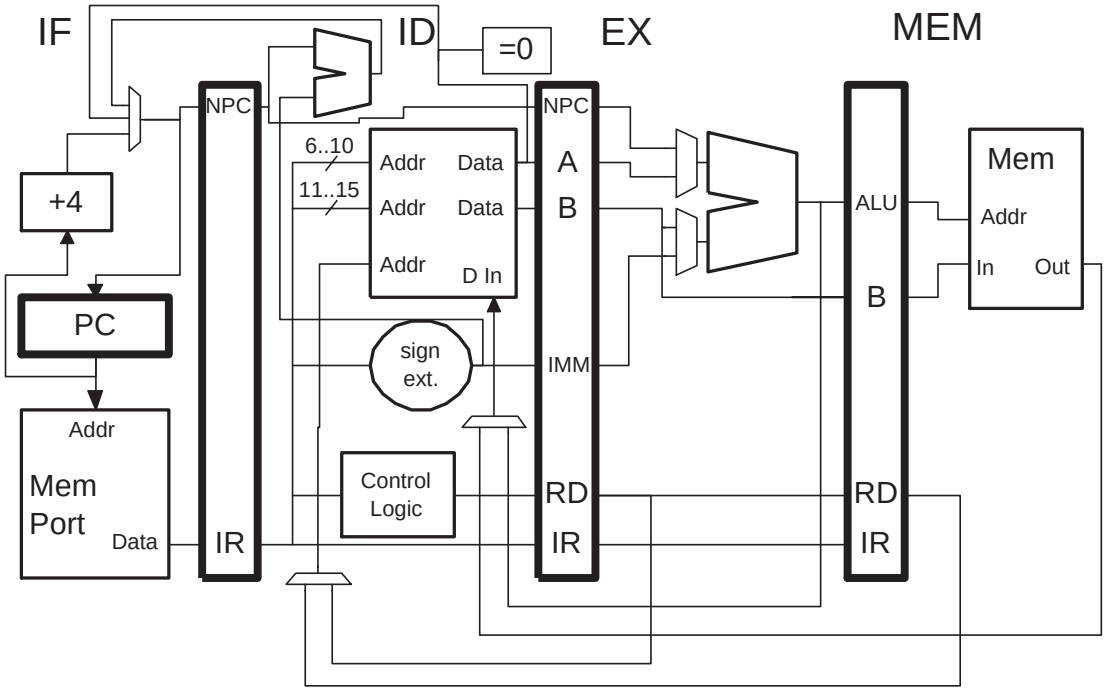
Problem 2: As described in class, postincrement instruction `lw r1,(r2+)` loads the value at memory address `r2` into register `r1` and stores `r2+4` in `r2`. Postincrement stores are similar. The pipeline below is to be modified so that it can execute postincrement loads and stores for bytes, half words, and words. A logic block size can be used; its input is the `opcode` and `func` fields; the output is 0 for a postincrement with a byte-size load or store value, 1 for a postincrement with a half-word value, 2 for a postincrement with a word-size value, and 3 if the instruction is not a postincrement load or store.



☐ [10 pts] Show the datapath changes needed so that the pipeline can execute postincrement store instructions. Include the control logic for obtaining the amount to increment the address by but do not include any other control logic.

☐ [7 pts] Show the additional datapath changes needed so that the pipeline can execute postincrement loads. These changes must not add structural hazards and the load must execute without stalling the pipeline (assuming favorable dependencies).

Problem 3: The four-stage (no WB) pipeline below includes an *Express Writeback* feature, eliminating the need for bypass connections. Instructions proceed through the pipeline slightly differently than the DLX pipeline presented in class. **Do not** add or assume the presence of bypass connections.



☐ [6 pts] Show a pipeline execution diagram for the code below on this pipeline. (Don't forget to look for dependencies.) State any assumptions about how the register file operates.

```
add r1, r2, r3

lw  r4, 8(r1)

sw  12(r5), r4
```

One disadvantage of *Express Writeback* is that it introduces a structural hazard.

☐ [6 pts] Show a program that encounters the hazard and a pipeline execution diagram showing how the hazard can be avoided by stalling. (*Hint: the program can be just two instructions.*)

☐ [4 pts] Explain how *Express Writeback* affects the critical path.

Part II on Wednesday at 12:40 precisely.

Name _____

Computer Architecture
EE 4720
Midterm Examination, Part II

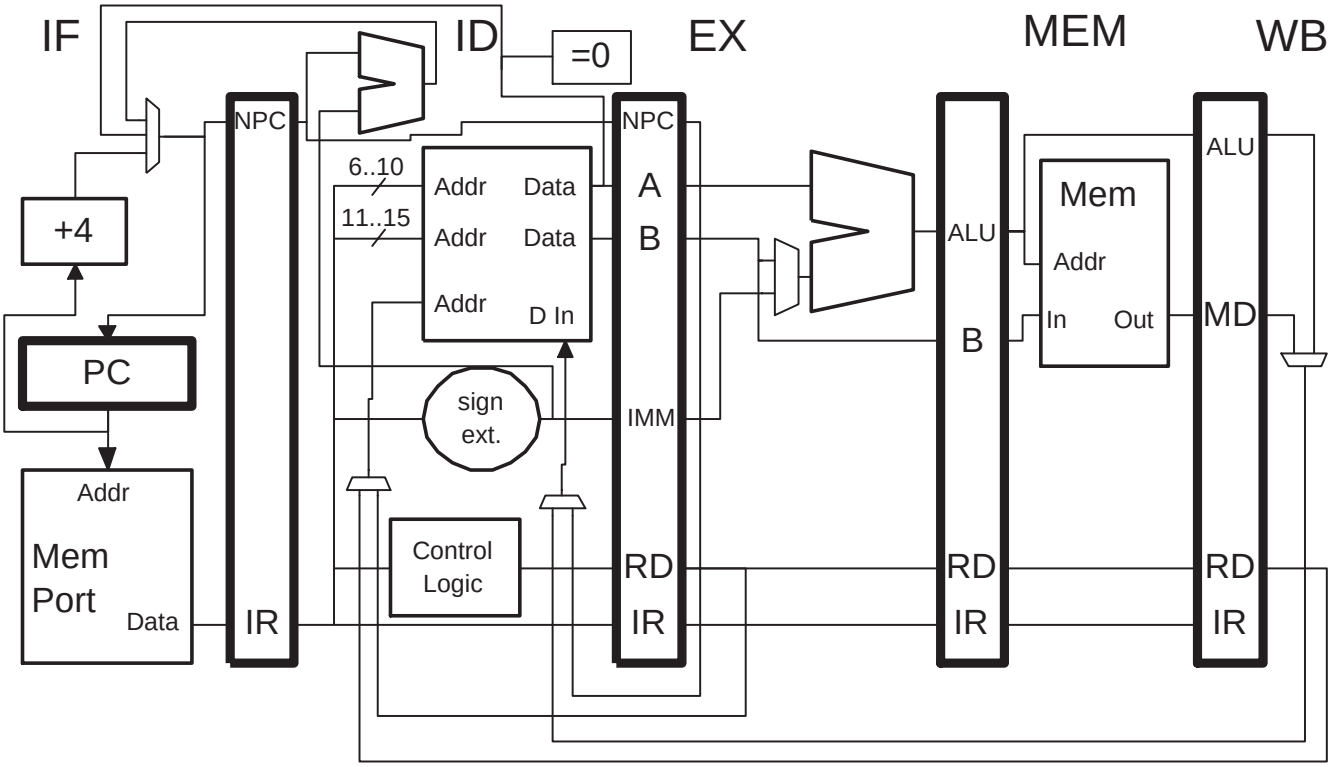
Wednesday, 18 October 2000, 12:40–13:30 CDT

Problem 1	(17 pts)	Mon.
Problem 2	(17 pts)	Mon.
Problem 3	(16 pts)	Mon.
Problem 4	(13 pts)	Wed.
Problem 5	(17 pts)	Wed.
Problem 6	(20 pts)	Wed.
Exam Total	(100 pts)	

Alias _____

Good Luck!

Problem 4: The diagram below includes naïve hardware for implementing the `jal` and `jalr` instructions. With this hardware precise exceptions are impossible.



☐ [6 pts] Explain why precise exceptions are impossible here.

☐ [7 pts] Show a program including a `jalr` that encounters an exception and which does not run correctly (after the exception handler returns) because of the way `jalr` is implemented. Briefly explain why it does not run correctly. (For partial credit answer the question using an instruction other than `jalr`.)

Problem 5: For some, 16 bits is just not enough. Consider a new DLX instruction, `mbi` (move big immediate), which moves a large immediate into register `r1`. (Register `r1` is always used, a different register cannot be specified.) The following code uses the new instruction:

```
mbi  0x12345     ! Move 0x12345 into register r1.
add  r2, r1, r2  ! Use 0x12345 in an add.
```

☐ [5 pts] Describe how the instruction could be coded using an *existing* DLX instruction type to get the biggest immediate possible. Specify the size of the immediate.

Ignoring the previous part, suppose one wanted the immediate to be 30 bits.

☐ [4 pts] Why are 30-bit immediates impossible using an existing instruction type?

☐ [4 pts] Describe how a 30-bit immediate `mbi` could be coded using a *new* DLX instruction type. (See the next subpart before answering.)

Whether it is possible to add a 30-bit immediate instruction and maintain compatibility depends on certain details of DLX which during lectures were usually made up on the spot.

☐ [4 pts] What kind of details were those? Make them up so that the new instruction is compatible.

Problem 6: Answer each question below.

(*a*) Loads and stores in DLX are aligned.

☐ [6 pts] What does that mean? Provide examples of aligned and unaligned accesses.

(*b*) As described in class, an ISA implementing a time data type might have instructions to determine the number of days between two times. (Say between 18 October 2000 and 15 December 2000.)

☐ [7 pts] Give two reasons why an ISA should *not* include such a time data type.

(*c*) RISC and CISC are sometimes seen as two competing architectural styles.

☐ [7 pts] Name two features that distinguish RISC processors from CISC processors.

Name _____

Computer Architecture
EE 4720
Final Examination

8 December 2000, 12:30–14:30 CST

Modified

Problem 1	_____	(20 pts)
Problem 2	_____	(14 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(17 pts)
Problem 5	_____	(29 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: New DLX instruction `jalr.safe` is like a `jalr` instruction except that it automatically returns after a certain number of instructions in the called routine have been executed. Suppose `r1` contains `0x1000` and `r2` contains `23` when `jalr.safe r1, r2` is executed. Execution would jump to address `0x1000` and `PC+4` would be saved in `r31`. Nothing special happens if a `jr r31` (a return) is executed within 23 instructions. If after 23 instructions a return is not executed control will automatically return to the instruction following `jalr.safe`. When `jalr.safe` is used the called procedure cannot call another procedure.

(*a*)

☐ (5 pts) Show how `jalr.safe` might best be coded.

(*b*) Modify the pipeline on the next page so that it executes the `jalr.safe` instruction.

- The output of the Live box is 1 if the corresponding stage contains a non-squashed instruction that will advance to the next stage in the next cycle. (The instruction is neither squashed nor stalled.)
- =Ret can be used for detecting return instructions, =jalr.safe for `jalr.safe` instructions, etc.

Continued on next page.

Problem 1, continued:

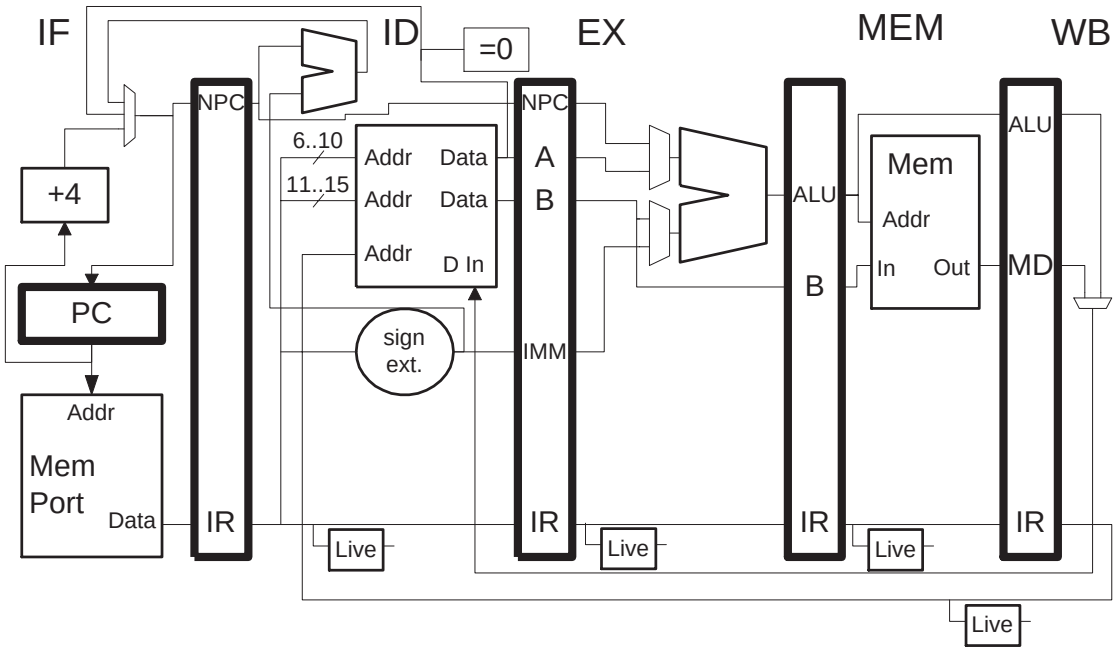
☐ (7 pts) Show the hardware needed to properly save the automatic return address and count. (For the automatic return address do not use the contents of `r31`, instead add a register for this address.)

☐ (2 pts) Base the return on the number of instructions that will complete, squashed instructions should not be counted.

☐ (2 pts) Make sure `jalr.safe` can be squashed before it sets a return address.

☐ (2 pts) Generate a signal named `Auto Return`, it shall be true when there is to be an automatic return and false other times.

☐ (2 pts) Explain what the controller must do when `Auto Return` is asserted.



Problem 2: The programs below run on statically and dynamically scheduled systems. All systems are single issue (not superscalar), have perfect branch prediction, have an unlimited number of functional units, and use non-blocking caches. The programs run for a large number of iterations, and the first `lw` in **every** iteration will miss the cache. On a cache miss data arrives 10 cycles after `MEM` or `L2` is entered. The line size is 1024 characters.

```
! Program 1                    ! Program 2
LOOP:                          LOOP:
 lw   r1, 0(r2)                 lw   r1, 0(r2)
 addi r2, r2, #1024             lw   r2, 4(r2)
 add  r3, r3, r1                add  r3, r3, r1
 bneq r1, LOOP                  bneq r1, LOOP
```

(*a*) Suppose the programs were are run on a statically scheduled machine and loop unrolling was being considered. *Note: The following important point was not included in the 2000 final exam.* The statically scheduled machine treats load misses like floating-point operations: it allows them to complete out of order if there are no name or data dependencies with following instructions.

☐ (1 pts) For a statically scheduled system applying loop unrolling to Program 1 would improve performance: (*a*) ☐ by a large amount; (*b*) ☐ by a small amount; (*c*) ☐ not at all; (*d*) ☐ none of the above.

☐ (1 pts) For a statically scheduled system applying loop unrolling to Program 2 would improve performance: (*a*) ☐ by a large amount; (*b*) ☐ by a small amount; (*c*) ☐ not at all; (*d*) ☐ I do not wish to reveal my intent.

☐ (5 pts) Explain the two answers above. In particular explain, if appropriate, why loop unrolling is more effective on one program than the other.

(*b*) Suppose the programs were to be run as is (not unrolled).

☐ (1 pts) Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 1: (*a*) ☐ much faster; (*b*) ☐ slightly faster; (*c*) ☐ about the same speed; (*d*) ☐ dimple.

☐ (1 pts) Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 2: (*a*) ☐ much faster; (*b*) ☐ slightly faster; (*c*) ☐ about the same speed; (*d*) ☐ I assume that if my answer below is correct points will not be deducted for this choice.

☐ (5 pts) Explain the two answers above. In particular explain, if appropriate, why the dynamically scheduled machine is more effective on one program than the other.

Problem 3: The code below executes on a dynamically scheduled system with branch prediction but without branch target prediction. The branch is predicted taken but it ultimately is not taken. The processor is single-issue (not superscalar) but, conveniently, has an unlimited number of functional units and can handle any number of write-backs per cycle. At most one instruction per cycle can be committed.

(*a*) The code below executes on such a machine in which the register map is **not** backed up when branches are decoded. Registers are intentionally omitted from the last three instructions, assume those instructions are not data-dependent on the loads.

☐ (6 pts) Complete the pipeline execution diagram for this system until all instructions complete.

☐ (2 pts) Show instruction commitment and squashing.

- Don't forget to check for dependencies!

```
! Branch predicted taken, but branch is NOT taken.
! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
 lw r3, 0(r4)    IF   ID   L1   L2   RS   RS   RS   RS   RS   L2   WB
 lw r1, 0(r2)         IF   ID   L1   L2   RS   L2   WB

 bneq r1, TARGET

 xor  r5, r6, r7

 sgt  r8, r9, r10
 ...
! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
TARGET:
 add r11, r12, r13

 sub r14, r15, r16

 and r17, r18, r19

! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
```

Problem 3, continued: (*b*) The code below executes on a version of the machine in which the register map is backed up (checkpointed) when branches are decoded.

☐ (6 pts) Complete the pipeline execution diagram for this system until all instructions complete.

- Show instruction commitment and squashing.

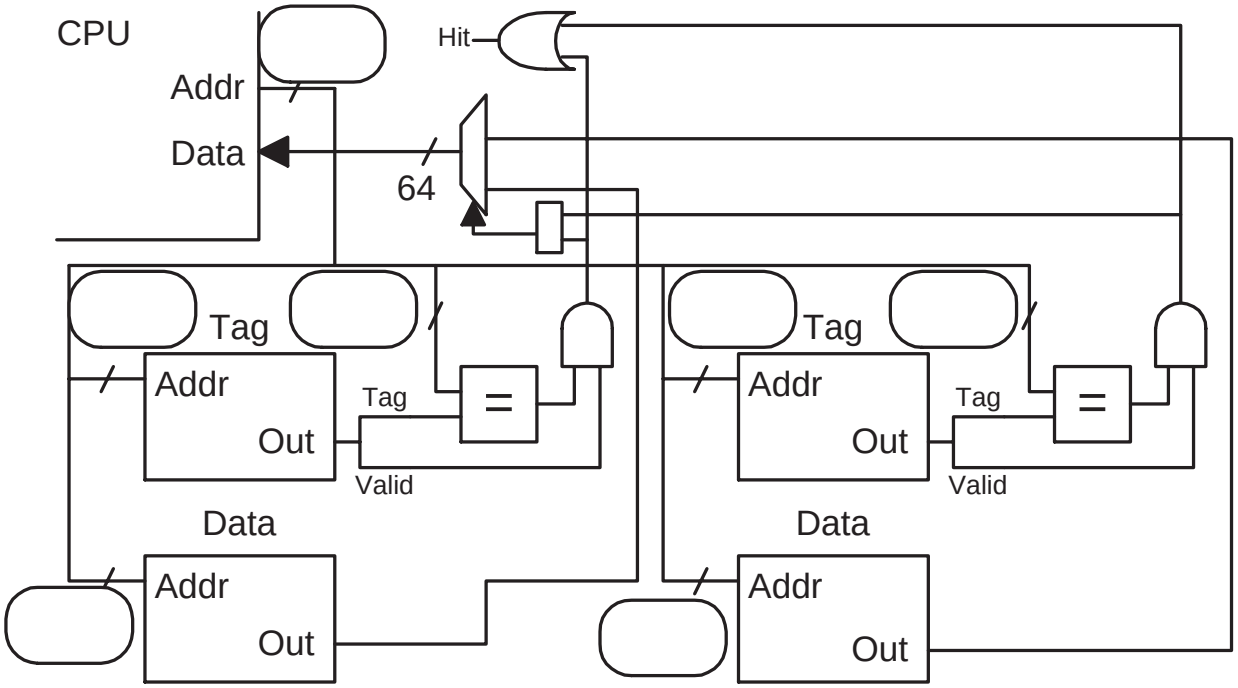
```
! Branch predicted taken, but branch is NOT taken.
! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
 lw r3, 0(r4)    IF   ID   L1   L2   RS   RS   RS   RS   RS   L2   WB
 lw r1, 0(r2)         IF   ID   L1   L2   RS   L2   WB

 bneq r1, TARGET

 xor r5, r6, r7

 sgt r8, r9, r10
 ...
! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
TARGET:
 add

 sub

 and

! Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16
```

(*c*) Suppose the processor from the previous part has the following defect: it does not back up or restore the register map, but it executes instructions as though it did. Suppose execution up until the code fragment above is okay.

☐ (6 pts) Add registers to the last three instructions in the previous part so that the `xor` instruction executes correctly but the `sgt` (set greater than) executes incorrectly.

Problem 4: A system has a 1-megabyte (2^{20} byte) two-way set-associative cache with 256-character blocks and a 50-bit address space addressing **16**-bit characters.

☐ (7 pts) Fill in the rounded boxes in the diagram below so that it describes this cache.



☐ (5 pts) Show the smallest set of addresses that cannot all be in this cache at the same time.

☐ (5 pts) What would be the associativity of a fully associative cache with the same capacity and block size as this one?

Problem 5: Answer each question below.

(*a*) The DLX program below runs on a system using a one-level branch predictor with a 2^{16}-entry BHT, each BHT entry is a two-bit saturating counter. The loop iterates many times.

Please do not confuse `andi` with `addi`.

```
LOOP:
 addi r1, r1, #1
 andi r2, r1, #1
 bneq r2, SKIP
 nop
 nop
 nop
 nop
SKIP:
 sub  r3, r1, r4
 bneq r3, LOOP
```

☐ (4 pts) What are the best-case and worst-case prediction accuracies for the first branch. Briefly explain.

☐ (3 pts) What is the smallest BHT size for which there will not be a collision between the two branches?

(*b*) Consider a dynamically scheduled four-way superscalar processor with a common data bus (CDB) that can handle two write-backs per cycle.

☐ (3 pts) Compare its speed to that of an ordinary dynamically scheduled two-way superscalar processor. Justify your answer.

☐ (3 pts) Compare its speed to that of an ordinary dynamically scheduled four-way superscalar processor. Justify your answer.

(*c*)

☐ (4 pts) Why is branch target prediction potentially more useful for DLX `jalr` instructions than it is for `bneq` and `beqz` instructions?

(*d*)

☐ (4 pts) What is a predicated instruction? Show how predicated instructions can be used in the code below. (Exact syntax is not important.)

```
 beqz r1, SKIP
 add  r2, r2, r3
SKIP:
 or   r4, r4, r5
```

(*e*) Consider a statically scheduled DLX implementation in which the floating-point add functional unit is two stages and the floating-point multiply functional unit is four stages, and both are fully pipelined. An exception can occur in any stage of the FP functional units.

☐ (4 pts) Show how the code below would execute to ensure precise exceptions.

```
muld  f0, f2, f4

addd  f2, f8, f10

or    r1, r2, r3
```

☐ (4 pts) Suppose that floating-point instructions did not have precise exceptions. Show how a test instruction could be used to ensure that an exception in `muld` was precise. Illustrate with a pipeline execution diagram.

36 Spring 2000

Name _____

Computer Architecture
EE 4720
Midterm Examination

22 March 2000, 13:40–14:30 CST

Problem 1 _____ (35 pts)

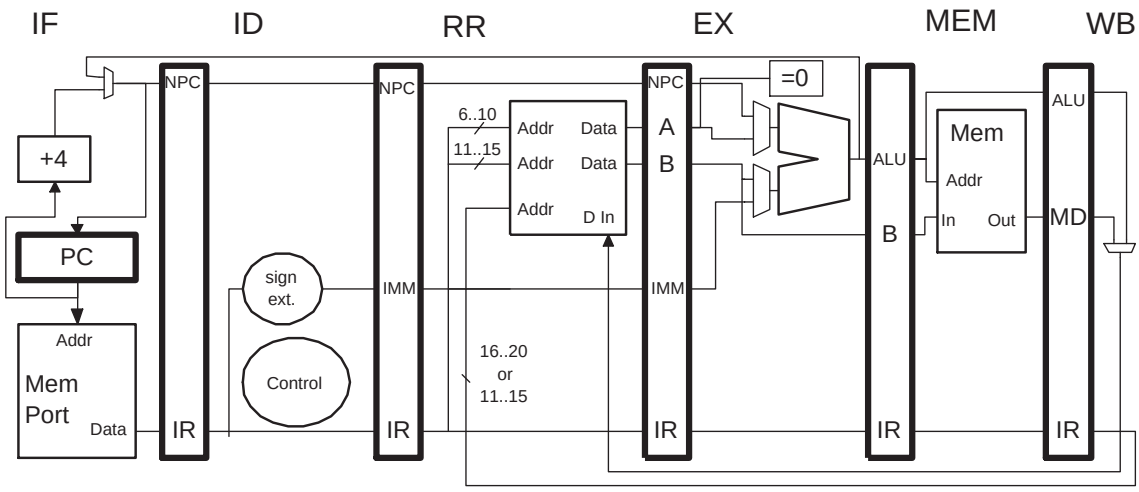
Problem 2 _____ (20 pts)

Problem 3 _____ (45 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: The DLX implementation below has six stages. (The work done by ID is now done by ID and RR.)



(*a*) The execution of some code on this pipeline is shown below. Add *exactly* the bypass paths needed so that the code executes as illustrated. Next to each bypass path indicate the cycle(s) in which it will be used. (Do not add bypass paths that won't be used in the execution of the code below.) (10 pts)

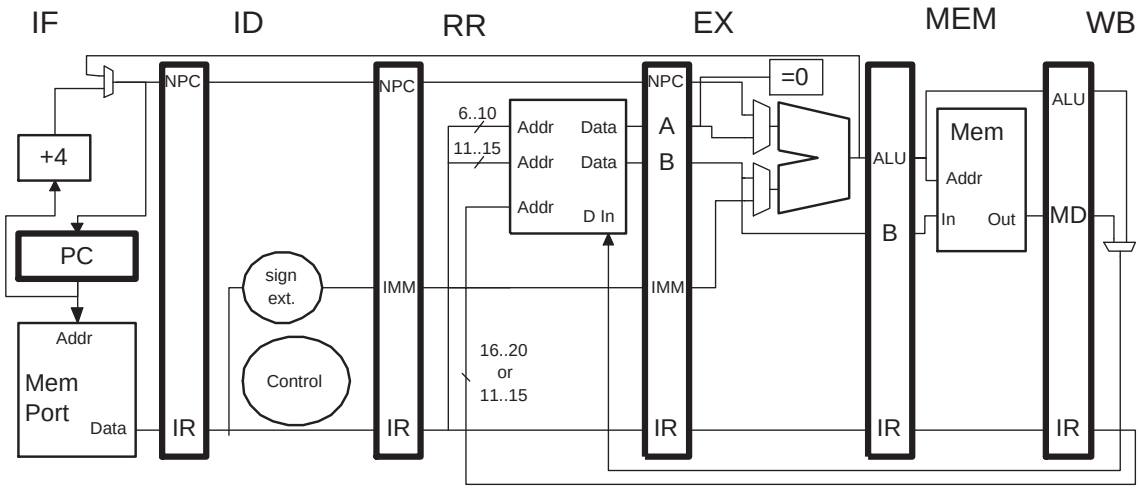
```
! Cycle          0  1  2  3  4  5  6  7  8
add  r1, r2, r3  IF ID RR EX ME WB
lw   r5, 10(r1)     IF ID RR EX ME WB
xor  r4, r1, r2        IF ID RR EX ME WB
and  r6, r5, r4           IF ID RR EX ME WB
```

(*b*) Show the execution of the code below on this pipeline until `bneq` reaches IF a second time. The branch is taken. Be sure to base the CTI behavior on the hardware shown above. Show where instructions are squashed. (10 pts)

```
LOOP:
 ! Branch Taken.
 bneq r1, SKIP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
SKIP:
 j LOOP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
```

Problem 1, continued: The figure and code below are identical to the ones on the previous page.

(*c*) Add branch and jump hardware so that the code executes as quickly as possible. Additional adders can be used, the fewer the better. Branches and jumps can be handled separately. The register file cannot be moved or duplicated and cannot be read before the RR stage. (Jumps and branches both use displacement addressing. In branches the displacement is in bits 16 to 31 and in jumps the displacement is in bits 6 to 31. Do not show control hardware.) (10 pts)



(*d*) Show how the code below executes on the modified pipeline. As before, show execution until `bneq` enters IF a second time. (5 pts)

```
LOOP:
 ! Branch Taken.
 bneq r1, SKIP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
SKIP:
 j LOOP
 add  r2, r3, r4
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
```

Problem 2: The code below executes on a dynamically scheduled processor that uses reorder buffer entry numbers to name registers. There are 128 reorder buffer entries, the next free entry is #1.

(*a*) Show when each instruction commits and show the contents of the register map, register file and reorder buffers using the space provided. Show only the instruction line numbers (`A`, `B`, `C`, `D`) in the reorder buffer. Indicate cycle numbers above the reorder buffer. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`A:addd f0, f2, f0`	IF	ID	A0	A1	A2	A3	WB										
`B:addd f6, f0, f6`		IF	ID	RS	RS	RS	A0	A1	A2	A3	WB						
`C:addd f0, f0, f6`			IF	ID	RS	RS	RS	RS	RS	RS	A0	A1	A2	A3	WB		
`D:addd f6, f2, f8`				IF	ID	A0	A1	A2	A3	WB							

Register Map

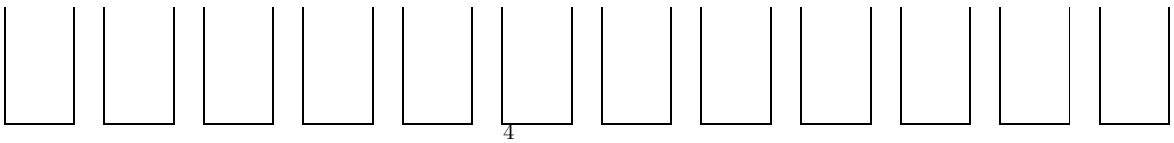
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
`f0`	10																
`f2`	20																
`f6`	60																
`f8`	80																

Register File

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
`f0`	10																
`f2`	20																
`f6`	60																
`f8`	80																

Cycle:

Reorder buffer:



Problem 2, continued: (*b*) The code below is identical to the code above, however the second add instruction raises an exception in cycle 9 (indicated by a star). Complete the pipeline execution diagram and other tables for this situation for the instructions shown. (Do not show the trap handler.) Show the contents of the reorder buffers after each change. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A:addd f0, f2, f0	IF	ID	A0	A1	A2	A3	WB										
B:addd f6, f0, f6		IF	ID	RS	RS	RS	A0	A1	A2	*A3*							
C:addd f0, f0, f6			IF	ID	RS	RS	RS	RS	RS	RS							
D:addd f6, f2, f8				IF	ID	A0	A1	A2	A3	WB							

Register Map

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
f0	10																
f2	20																
f6	60																
f8	80																

Register File

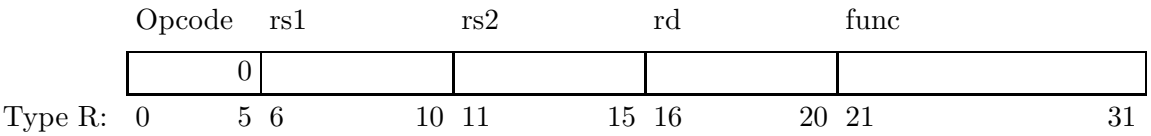
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
f0	10																
f2	20																
f6	60																
f8	80																

Cycle:

Reorder buffer:

Problem 2: Answer each question below.

(*a*) Why do DLX Type-R instructions have a `func` field? (7 pts)

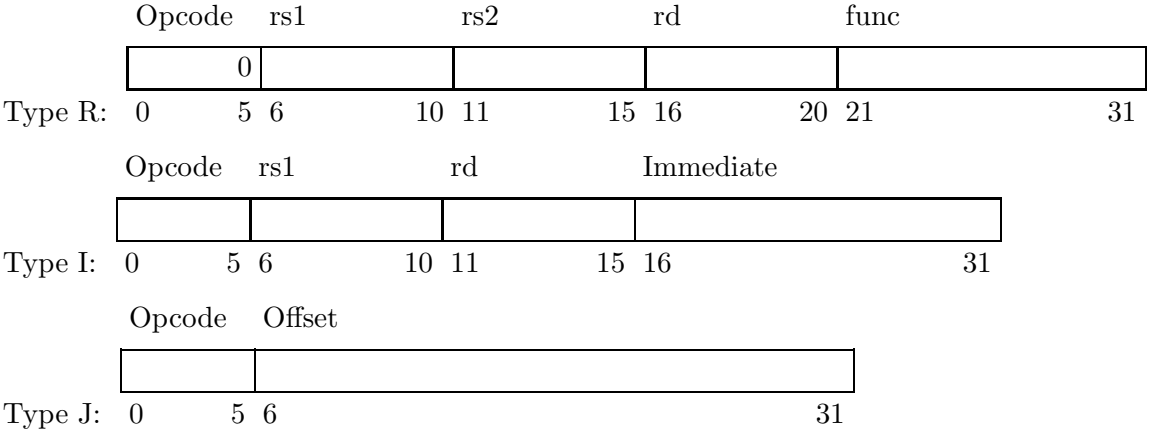


(*b*) Would there be any advantage in pipelining the integer ALU used in the statically scheduled DLX implementation? Explain. (7 pts)

(*c*) Why do ISAs include one-byte integers? (What are the advantages of using them when running on typical implementations.) (7 pts)

(*d*) The DLX implementations covered in class start reading registers before the instruction is decoded. Why is this possible? Modify the instruction codings so that it is no longer possible. That is, with the modified codings decoding would have to be performed *before* register read (as in problem 1). (8 pts)

The DLX codings are given below for reference and can be used to explain your answer.



(*e*) Ignoring floating-point instructions, how can precise exceptions be implemented on the statically scheduled (Chapter 3) DLX implementation? Illustrate your answer with an example in which exceptions occur out of order but are handled in order. Show the code and a pipeline execution diagram, indicate where the exceptions are occurring. (9 pts)

(*f*) What is an advantage of a stack ISA over a RISC ISA? What is a disadvantage of a stack ISA over a RISC ISA? (7 pts)

Name _____

Computer Architecture

EE 4720

Final Examination

8 May 2000, 10:00–12:00 CDT

Problem 1	_____	(20 pts)
Problem 2	_____	(10 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(21 pts)
Problem 5	_____	(39 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

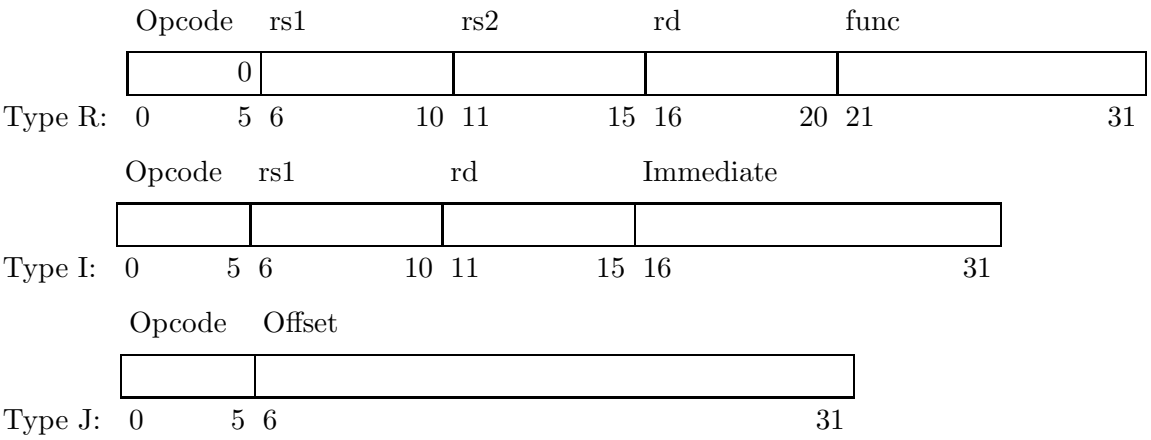
Problem 1: An extended DLX ISA, Triple-DLX [tm], includes new three-operand integer ALU instructions. (Assume that the integer ALU can perform integer multiply.) Some sample instructions appear below:

```
add_add r1, r2, r3, r4 ! r1 = r2 + r3 + r4
add_mul r5, r6, r7, r8 ! r5 = ( r6 + r7 ) * r8
mul_add r5, r6, r7, r8 ! r5 = ( r6 * r7 ) + r8
```

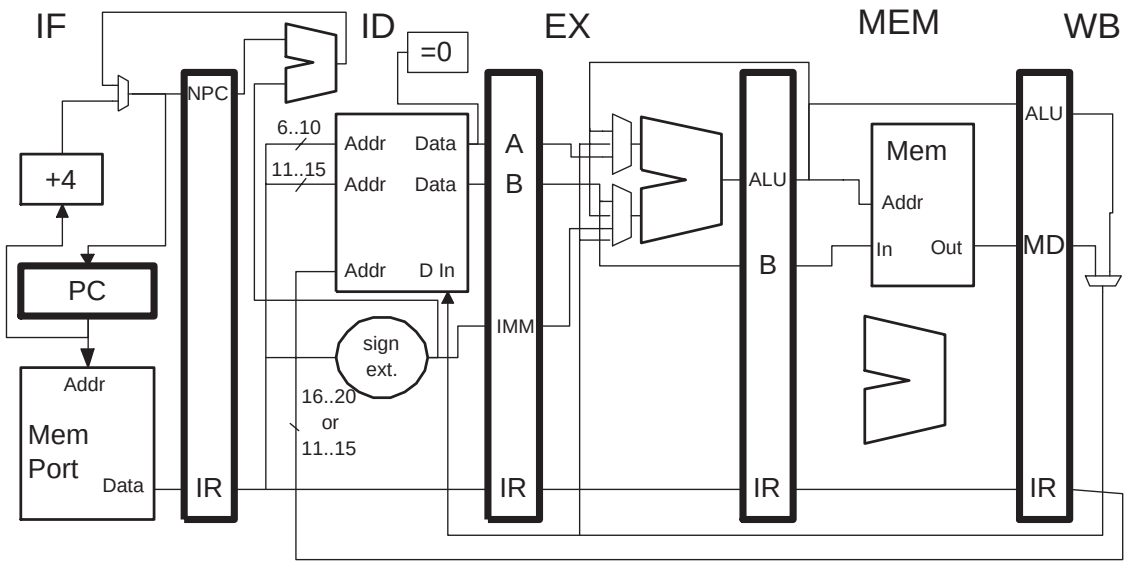
(*a*) (8 pts)A new instruction type, Type-T, will be used for these instructions. Show how the new Type-T instructions can be coded. *The coding should be chosen to ease implementation and should allow for at least 64 three-operand instructions.* Assume that there are six free opcode field values and seven free func-field values available for your use.

- Explain how to distinguish Type-T instructions from Type-R, Type-I, and Type-J instructions.
- How many Type-T instructions can be provided using your coding?

The DLX codings are given below for reference and can be used to explain your answer.



(*b*) Modify the pipeline below so that it can execute the three-operand instructions. A second ALU has been placed in the MEM stage; it should be used to help implement the instructions. The register file is among the parts that need to be modified. (6 pts)



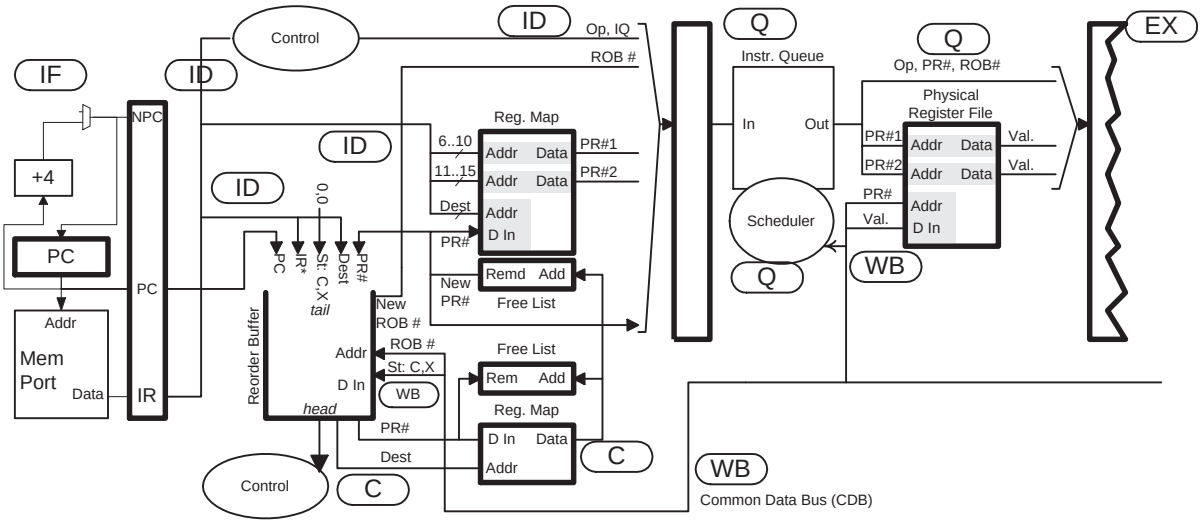
(*c*) Show a pipeline execution diagram for the code below assuming that all needed bypass paths are available. The code should execute as fast as possible. **Add** any needed bypass paths to the diagram above. Do not add any bypass paths that are not needed by the code below. Label each bypass path (added or already present) with the cycle in which it is used. Please be sure not to miss any true dependencies. (6 pts)

```
add_add r1, r2, r3, r4

add_sub r5, r1, r2, r3

sub_sub r6, r1, r5, r6

sub_add r7, r1, r5, r6
```

Problem 2: The code appearing on the next page executes on a dynamically scheduled machine using physical registers to name destination registers.

Complete the tables, showing only changes. Show where instructions commit. Show only entries associated with registers `f0` and `f6` in the physical register file.

At cycle zero, register `f0` contains a 5, `f2` contains a 20, `f4` contains a 40, and `f6` contains a 60. None of the instructions raise exceptions. The diagram below is provided for convenience; it is the same one used in class. (10 pts)



Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
add f0,f2,f4	IF	ID	Q								A0	A1	WB					
add f6,f0,f4		IF	ID	Q									A0	A1	WB			
add f0,f4,f4			IF	ID	Q					A0	A1	WB						
add f6,f0,f4				IF	ID	Q								A0	A1	WB		
sub f0,f2,f4					IF	ID	Q	A0	A1	WB								

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Arch. Reg.	ID Register Map
f0	17
f6	6

	ID Free List
	12 7 8 4 3

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

	Commit Register Map
f0	17
f6	6

	Commit Free List
	12 7 8 4 3

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Phys Reg.	Physical Register File

Problem 3: Provide a pipeline execution diagram for the code below running on a dynamically scheduled *two-way superscalar* machine using reorder buffer entry numbers to name destination operands. Be sure to show when instructions complete.

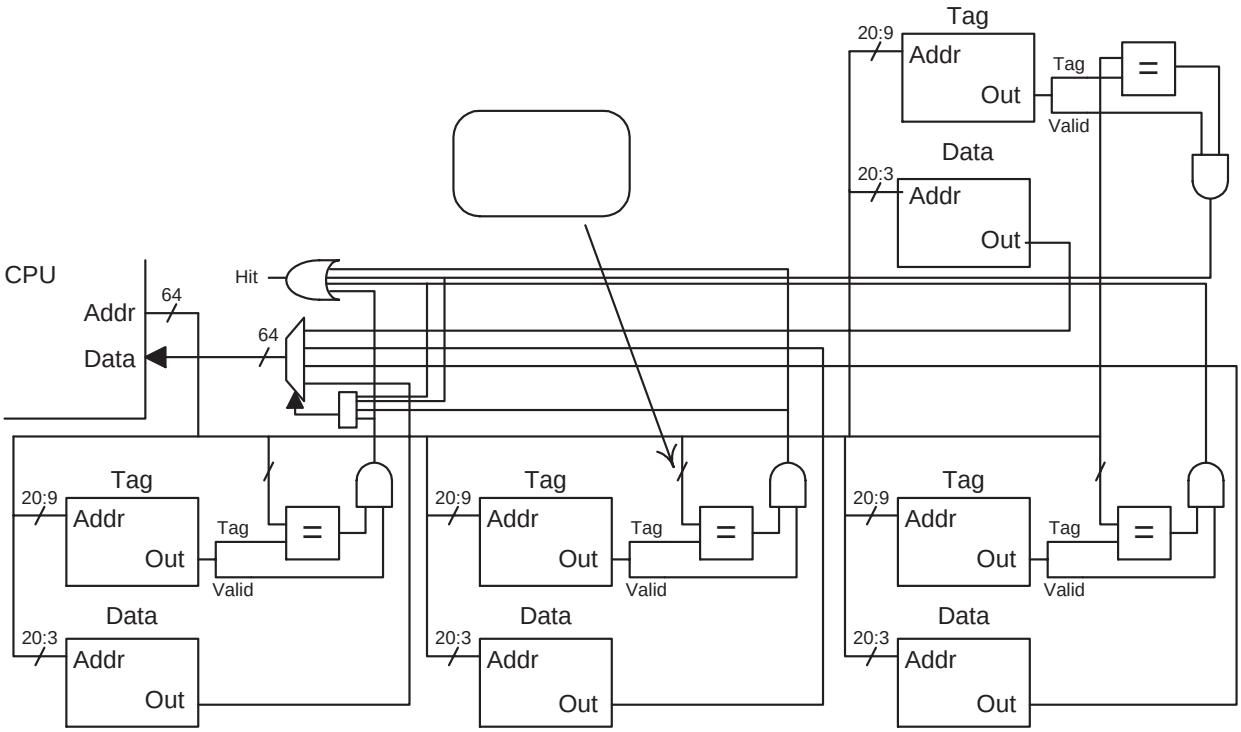
There are more than enough reservation stations and reorder buffer entries. Do not show reservation station numbers in the diagram. There are two load-store units and the *cache is nonblocking*.

The first `lw` misses the cache, the data arrives 6 cycles after it first enters the L1 stage; the second `lw` also misses the cache, its data arrives 4 cycles after it first enters the L1 stage. (10 pts)

Carefully check the code for dependencies before working on a solution.

```
LINE: LINE = 0x1000


 lw r3, 0(r1)

 lw r5, 0(r2)

 sw 0(r4), r5

 lw r7, 0(r9)

 sw 0(r3), r1

 lw r8, 0(r10)
```

Problem 4: A system uses the following cache:



(*a*) Determine the value of the following parameters for the cache illustrated above. Be sure to specify units (bits, bytes, etc.). Answers can be in the form of mathematical expressions. (8 pts)

Associativity:

Number of Sets:

Address Space Size:

Block (Line) Size:

Cache Capacity:

Amount of Memory to Implement Cache:

Character Size:

Tag Bits: (Can show on diagram.)

(*b*) Write a program that will fill the cache using the minimum number of accesses. (Ignore instruction accesses.) (8 pts)

(*c*) Determine the parameters for a direct-mapped cache with the same block size and the same capacity designed for the same system. (5 pts)

Associativity:

Number of sets:

Address Space Size:

Block (Line) Size: (Same, don't answer.)

Cache Capacity: (Same, don't answer.)

Character Size:

Tag Bits:

Problem 5: Answer each question below.

(*a*) The program below runs on a system using a gselect branch predictor. What is the minimum global history size needed so that there is a good chance that the last branch will be correctly predicted at the last iteration (after warmup)? Assume that there are no collisions. Explain your answer. (7 pts)

```
 addi r1, r0, #10
 add  r5, r0, r0
LOOP:
 lw r2, 0(r3)
 add r5, r5, r2
 subi r1, r1, #1
 addi r3, r3, #4
 bneq r1, LOOP
```

(*b*) What is the advantage of backing up (checkpointing) the register map when a branch is encountered in a system using branch prediction and dynamic scheduling? Explain how execution would be different if the register map were not backed up. (7 pts)

(*c*) Why are three-way superscalar machines difficult to build but three-instruction-bundle VLIW ISAs are common? (5 pts)

(*d*) An ISA is almost like DLX except, oops, the `lbu` (load byte unsigned) instruction was omitted, and so the program below won't run on an implementation of this ISA. Modify the program so that it will run, and run as though an `lbu` instruction was used. (In other words, replace `lbu r1, 1(r2)` by instructions that do the same thing.) (5 pts)

```
lbu r1, 1(r2)
```

(*e*) The table below shows virtual and physical addresses in use in a virtual memory system having a 32-bit address space. What is the largest possible page size this system can have? Using this page size (or some other one, but show what page size is being used) show the possible contents of a two-level page table storing this virtual-to-physical mapping. State any assumptions made. (For partial credit solve the problem for a one-level page table.) (10 pts)

Virtual	Physical
0xfea34b62	0x74b4b62
0xfeb92b90	0x14b2b90
0xeaa31f16	0x77b1f16

(*f*) Describe two ways that loop unrolling improves performance. (5 pts)

37 Fall 1999

Name _____

Computer Architecture
EE 4720
Midterm Examination

20 October 1999, 10:40–11:30 CDT

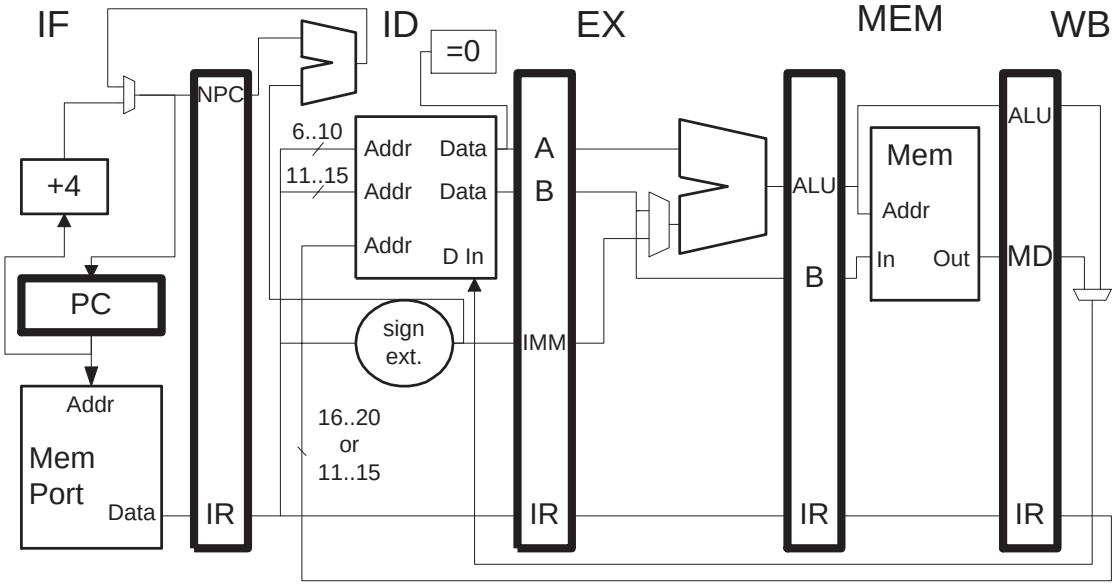
Problem 1	_____	(33 pts)
Problem 2	_____	(33 pts)
Problem 3	_____	(34 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1:

(*a*) Add *exactly* the bypass paths that are needed so the code below executes as shown in the pipeline execution diagram. (**Don't** add any bypass paths that are not needed by the code.) (15 pts)



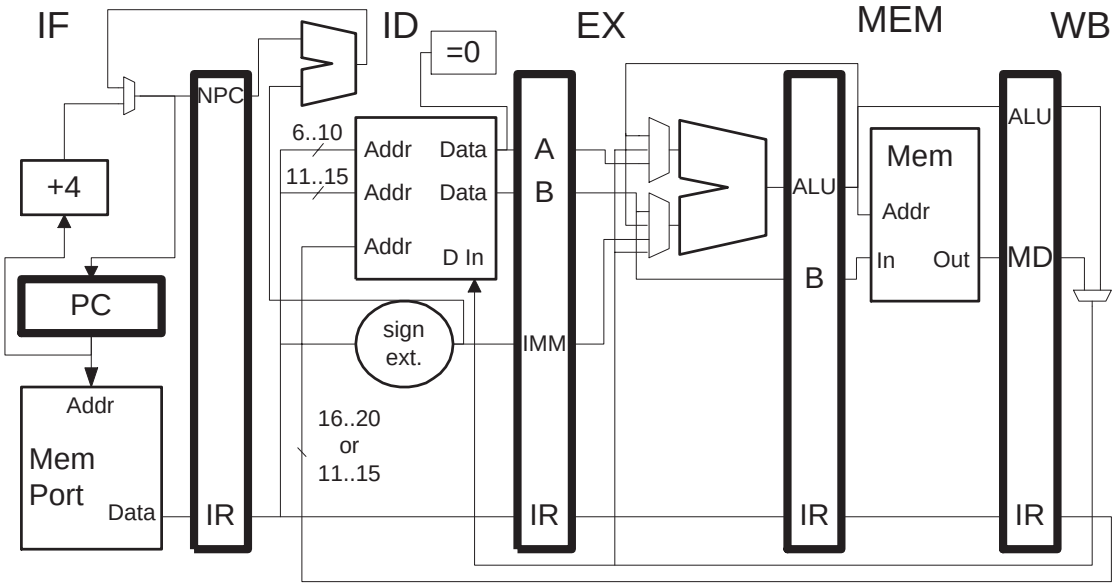

```
! r1 = 0x1010  r2 = 0x2020, r3 = 0x3030, r4 = 0x4040, r5 = 0x5050
! Cycle            0   1   2   3   4   5   6   7
LINE1:  LINE1 = 0x100
 lw   r1, 16(r2)   IF  ID  EX  MEM WB  IF  ID  EX ...
 addi r2, r2, #4       IF  ID  EX  MEM WB  IF  ID ...
 sw   20(r2), r1           IF  ID  EX  MEM WB  IF ...
 beqz r2, LINE1                IF  ID  EX  MEM WB ...
 and  r3, r4, r5               IFx
```

(*b*) Show the values of the registers listed below in **cycle 4** of the execution above using the added bypass paths. Assume that the load instruction retrieves a `0x1234` from memory. Instructions are squashed by replacing them with an `or r0, r0, r0`. (18 pts)

IF.PC	ID/EX.A	EX/MEM.ALU	MEM/WB.ALU
IF/ID.NPC	ID/EX.B	EX/MEM.B	MEM/WB.MD
IF/ID.IR	ID/EX.IMM	EX/MEM.IR	MEM/WB.IR
	ID/EX.IR		

Problem 2:

(*a*) Show a pipeline execution diagram for the code below on the implementation shown until `lw` is fetched a second time. The first branch is not taken but the last one is. The only bypass paths available are the ones shown. (18 pts)




```
LOOP:
 lw   r1, 0(r2)
 andi r3, r1, #4
 beqz r3, LINE1
 add  r4, r4, r1
 j LINE2
LINE1:
 add  r5, r5, r1
LINE2:
 sw   4(r2), r1
 addi r2, r2, #8
 bneq r2, LOOP
 xor  r5, r6, r7
```

(*b*) Rewrite the code below (which is the same as the code on the previous page) using one-cycle delayed branches and predicated instructions and schedule the code so that it executes as quickly as possible. (Do not unroll the loop.) Assume that bypass paths are provided for the predicated instructions. It should be possible to remove all stalls, but if any remain point them out (for partial credit). (15 pts)

```
LOOP:
 lw   r1, 0(r2)
 andi r3, r1, #4
 beqz r3, LINE1
 add  r4, r4, r1
 j LINE2
LINE1:
 add  r5, r5, r1
LINE2:
 sw   4(r2), r1
 addi r2, r2, #8
 bneq r2, LOOP
 xor  r5, r6, r7
```

Problem 3: Answer each question below.

(*a*) Why do DLX branches (and branches in many other ISAs) use displacement addressing? Why don't branches use indirect addressing (destination address in a register) instead of displacement addressing? (8 pts)

(*b*) The code below executes on an implementation that uses a reservation register to detect WB structural hazards. At cycle zero the reservation register contains all zeros. Show the state of the reservation register at the end of each cycle below. Indicate which (if any) bit positions are tested in each cycle. (9 pts)

```
! Cycle             0  1  2  3  4  5  6  7  8  9  10
multf f0, f1, f2    IF ID M0 M1 M2 M3 M4 M5 WB
addf  f3, f4, f5       IF ID A0 A1 A2 A3 WB
subf  f6, f7, f8          IF ID -> A0 A1 A2 A3 WB
gtf   f9, f10, f11           IF -> ID A0 A1 A2 A3 WB
nop
nop
...
```

(*c*) Many packed operand instructions perform saturating arithmetic. What is saturating arithmetic? Provide an example. (8 pts)

(*d*) In homework 3, a special return address register (`ERA`) was used to hold exception return addresses. The jump and link instructions, `jal` and `jalr`, use `r31` for the return address; is this an option for exceptions? Explain. (9 pts)

Name _____

Computer Architecture

EE 4720

Final Examination

Primary: 6 December 1999, 10:00–12:00 CST
Alternate: 7 December 1999, 15:00–17:00 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The code in the table on the next page executes on a dynamically scheduled machine using reorder buffer entry numbers to rename registers. The implementation performs branch and branch target prediction.

There are an unlimited number of reorder buffer entries, reservation stations, and functional units. Integer instructions use the `EX` functional unit, branches use the `B` unit, and loads and stores use the load/store unit, consisting of segments `L1` and `L2`.

When the code in the table starts to execute all register values are available, as shown in the tables on the next page.

The `lw` instruction will suffer a miss; it will finish `L2` five cycles after entering `L2`, loading a `100`.

The `bneq` instruction is predicted not taken but is, in fact, taken.

(*a*) For this subproblem the register file is not backed up when branches are encountered. Using the tables provided on the next page show a pipeline execution diagram for the code and the changes to the register map and register file at the end of each cycle. Do not show reservation station numbers or reorder buffer entries in the pipeline execution diagram itself. The entry number for the next available reorder buffer entry is 1. Show when instructions commit or when they are squashed. (15 pts)

(*b*) Explain how execution would be different if the register file were backed up when branches are encountered. A second diagram is not necessary, just show where execution differs and how it differs. (5 pts)

There is one more part.

Problem 1, continued:

Pipeline Execution Diagram																
Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw r4, 0(r5)`	IF																
`add r1, r2, r3`																	
`or r2, r1, r3`																	
`bneq r4, SKIP`																	
`sub r1, r2, r5`																	
`SKIP:`																	
`add r2, r1, r2`																	

Register Map																
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
`r1`	10																
`r2`	20																
`r3`	30																
`r4`	40																
`r5`	50																

Register File																
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
`r1`	10																
`r2`	20																
`r3`	30																
`r4`	40																
`r5`	50																

Problem 1, continued:

(*c*) The DLX code in the table below executes on the dynamically scheduled machine described above. The machine uses a load/store queue and a nonblocking (lockup free) cache as described in class. For this part the cache miss latency is lower: If an instruction in `L2` encounters a cache miss it returns to its reservation station for three cycles then returns to `L2`.

Show the execution of the code in the table below. Show when instructions commit but **do not** show reservation station or reorder buffer entry numbers.

The instructions at lines `Line1` and `Line3` miss the cache. The contents of each register is different. (5 pts)

Pipeline Execution Diagram															
Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`Line1: sw 0(r1), r2`	\F														
`Line2: lw r3, 0(r2)`															
`Line3: lw r1, 0(r4)`															
`Line4: sw 0(r1), r2`															
`Line5: lw r5, 0(r2)`															

Don't forget part (b)!!!!

This problem consists of four parts. For full credit do parts (a), (b), and (c) only. For **reduced** *credit do parts (a) and (d). If you have time but lack confidence do all parts, the grade will be* $\max\{a+b+c, a+d\}$.

Problem 2: The code below is run on systems having a 32-bit address space, 1-byte characters, and using 256-kibibyte (2^{18}-byte) caches as described below. Before the code is run the cache is cold (empty). Only consider accesses to the array, `a`.

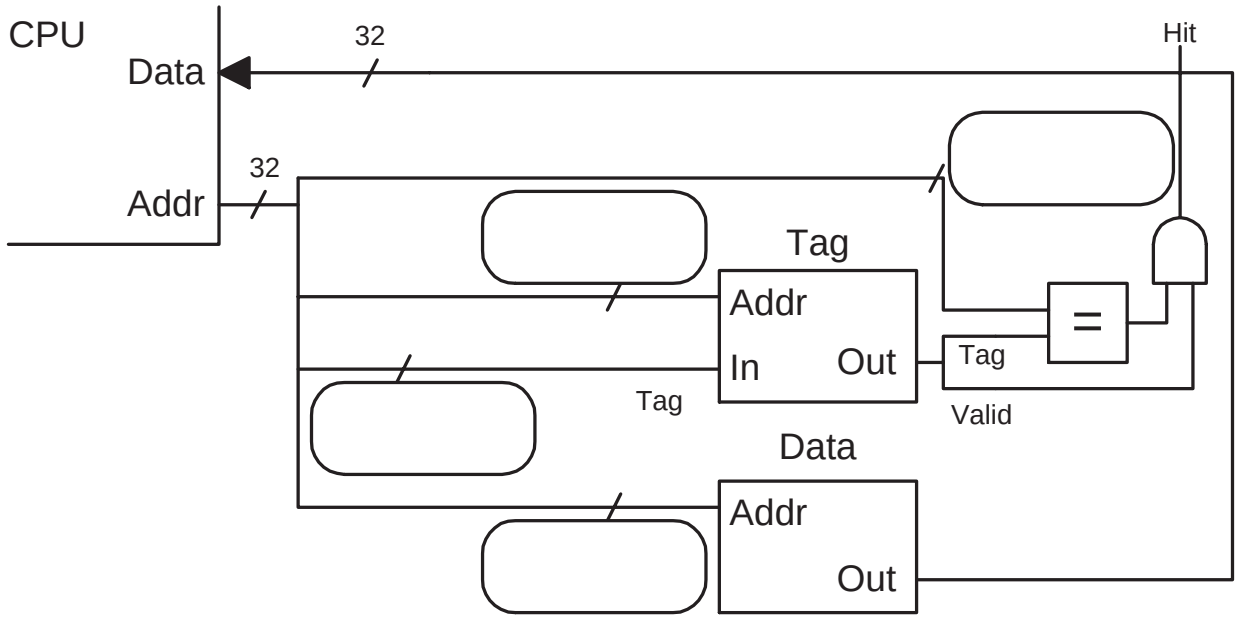
```
// sizeof(int) = 4 characters
int *a = 0x1000000;  // Storage allocated elsewhere.
for(x=0; x<4; x++)
  for(i=0; i<512; i++)
    for(j=0; j<8; j++)
      sum += a[ i * 1024 + j ];
```

(*a*) Find the hit ratio encountered executing the code above on a direct-mapped 2^{18}-byte cache with a line size of 8 characters. How much of the cache is filled? (9 pts)

(*b*) Suppose the associativity of the cache could be increased while fixing the cache capacity at 2^{18}-byte and the line size at 8. What would the hit ratio be if the associativity were 2? What is the smallest associativity needed to achieve the maximum hit ratio on the code above? Is that associativity practical? Explain. (8 pts)

Problem 2, continued:

(*c*) Suppose **any** address bits could be used to form the index (set number, address used in the tag store) and any line size could be used. (Lines must still be contiguous.) In the diagram below show which address bits should be used (using the rounded boxes) to maximize the hit ratio of the code above. The capacity of the cache must still be 2^{18}-byte. (8 pts)



(*d*) **Optional:** *Only solve this part if you cannot solve parts (b) and (c). Total credit will be lower.*

This part is unrelated to the previous parts. Show how $2^{34} \times 32\,\mathrm{b}$ memory devices should be connected to implement a 40-bit address space on a system with four-bit (one-nibble) characters and a bus width of 64 bits. Include the alignment network. (8 pts)

Problem 3: The code below is compiled and run on two machines, one using a one-level branch predictor and the other using a two-level gselect branch predictor. Both predictors use a 4096-entry BHT, the gselect predictor uses a 3-branch global history. Each entry holds a 2-bit saturating counter.

```
for(i=0; i<100000; i++)
{
Line1:  if( a == 1 ) aa++;
Line2:  if( b == 1 ) bb++;
Line3:  if( C == 1 ) cc++;
Line4:  if( i & 0x2 ) { x++; }  /*  N N T T N N T T N N T T N N T T ... */
Line5:  if( i & 0x4 ) { y++; }  /*  N N N N T T T T N N N N T T T T ... */
Line6:  if( i & 0x2 ) { z++; }
}
```

(*a*) The ISA has a 32-bit address space, all instructions are 32 bits (four characters) and must be aligned. How is the address for the BHT in the gselect predictor obtained? Be sure to specify bit positions. (9 pts)

(*b*) Assume that exactly one branch instruction is generated for each `if` statement and that the compiler does no optimizing. What is the prediction accuracy for each of the last three `if` statements using the one-level predictor after a large number of iterations? (8 pts)

(*c*) As above, assume that exactly one branch instruction is generated for each `if` statement and that the compiler does not do any optimizing. What is the prediction accuracy for each of the last three `if` statements using the gshare predictor after a large number of iterations? *Hint: The solution to this part does not require tedious computation or the construction of lengthy tables.* (8 pts)

Problem 4: Answer each question below.

(*a*) Synthetic instruction `clr 12(r2)` writes a zero to the memory location at address `12 + r2`. How could it be added to DLX? (5 pts)

(*b*) What would be the disadvantage of a RISC ISA that had 1048576 (2^{20}) integer (general-purpose) registers? (5 pts)

(*c*) Why would it be inappropriate to add a memory indirect load instruction to DLX. (For example, `lw r1,@(r2)`.) Justify your answer for the statically scheduled DLX implementation. Weigh the complexity of changes needed to the implementation against expected benefit over a software-only solution. Be brief. (5 pts)

(*d*) The code below runs on a dynamically scheduled 4-way superscalar machine with perfect branch target prediction and a cache that never misses. There are an unlimited number of reorder buffer entries, reservation stations, and functional units. This machine is not 100% perfect: its fetch mechanism is the type described in class. What is the minimum and maximum CPI executing the code below for a large number of iterations? Explain the conditions under which minimum and maximum CPI are encountered. (5 pts)

```
LOOP:
 lb   r1, 0(r2)
 addi r2, r2, #1
 bneq r1, LOOP
```

(*e*) What is the difference between a write-through cache and a write-back cache? Which one needs a dirty bit and why? (5 pts)

38 Spring 1999

Name _____

Computer Architecture
EE 4720
Midterm Examination

12 March 1999, 10:40–11:30 CST

Problem 1 _____ (25 pts)

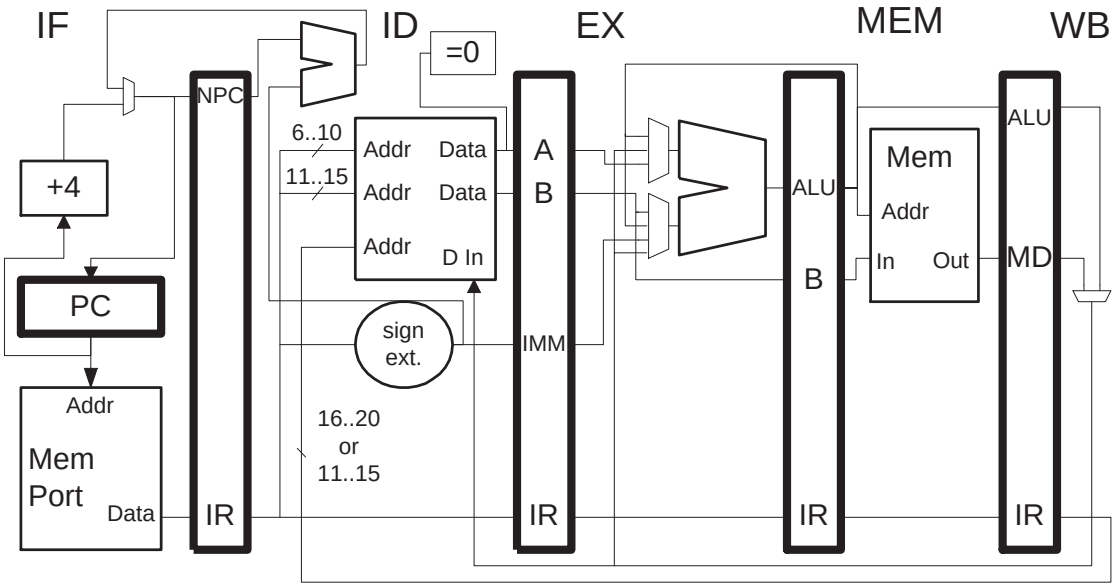
Problem 2 _____ (30 pts)

Problem 3 _____ (45 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: Design control logic to generate the control signal for the multiplexor at the lower input to the ALU. The control logic should be located in the ID stage and should generate a two-bit integer for the multiplexor. The integer specifies which multiplexor input to use, they are numbered from zero starting at the top. (Input 0 connects to `ID/EX.B`, 1 connects to `EX/MEM.ALU`, etc.) The logic can use units that test for equality of their two inputs, `=`, and units that test for instruction formats, `= Type I`, `= Type R`, and `= Type J`, and can use the usual logic gates. Base the setting on instruction type, rather than the exact opcode. Show how the control signal is connected to the multiplexor. (25 pts)



Problem 2: Consider the code below.

```
LOOP:
 lw   r1, 0(r2)
 addi r3, r1, #12
 sw   4(r2), r3
 add  r5, r5, r1
 addi r2, r2, #8
 slt  r6, r2, r7
 bneq r6, LOOP
 xor  r8, r9, r10
```

(*a*) Show a pipeline execution diagram for execution up to the second time `lw` enters instruction fetch. Use the pipeline from problem 1. As with homework 3, a bypass path is unavailable if it's not shown. What is the CPI for a large number of iterations? (10 pts)

(*b*) Unroll the loop so that two iterations of the code above is performed by one iteration of the unrolled loop. (Assume the number of iterations in the original loop is a multiple of two.) Schedule the unrolled loop to minimize stalls. (10 pts)

(*c*) What is the CPI of the unrolled and scheduled loop found above? What conclusions about performance improvement can and cannot be made by comparing the CPI of the original and unrolled loop? What is the performance improvement? (Give a number for performance improvement, don't just say "it's good.") (10 pts)

Problem 3: Answer each question below.

(*a*) Show an example of DLX code that encounters a WAW hazard on the Chapter-3 implementation of DLX (in which the multiply floating-point functional unit has an initiation interval of 1 and a latency of 6 and the add floating-point functional unit has an initiation interval of 1 and a latency of 3) but which does not encounter a WAW hazard on a DLX implementation which is identical except the FP add latency is 1 and the FP multiply latency is 4. The code should not encounter a structural hazard on either implementation. (12 pts)

(*b*) Why are there separate `lh` (load half) and `lhu` (load half unsigned) instructions, a `sh` (store half) instruction but **no** `shu` (store half unsigned) instruction? (11 pts)

(*c*) The code below is for a stack architecture. Rewrite the code below in DLX using as few instructions as possible. Assume that `ADDRA` is in `r10`, `ADDRB` is in `r11`, `ADDRX` is in `r12`, and `ADDRY` is in `r13`. The data at the addresses are double-precision floating-point values. (11 pts)

```
push ADDRX
push ADDRX
mult
push ADDRA
push ADDRB
add
push ADDRX
mult
push ADDRA
push ADDRB
mult
add
add
pop ADDRY
```

(*d*) Explain two ways in which precise exceptions can be made optional for floating-point instructions. That is, the programmer may choose to have precise exceptions where they are needed or may choose to not have precise exceptions were performance is most important. Explain what the programmer would have to do for each of the two ways. (11 pts)

Name _____

Computer Architecture
EE 4720
Final Examination

5 May 1999, 7:30–9:30 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: With the following extension to the DLX ISA a program can get a count of how many times a particular address has been read or written by load and store instructions. The count is kept in a *load/store counter (LSC)* which is incremented whenever a load or store instruction uses a particular effective address. Two new instructions are added, `setlsc` (type I) and `getlsc` (type R). Instruction `setlsc r1,r2+#3` sets the effective address to `r1` and initializes the counter to `r2+#3`. Instruction `getlsc r3` copies the LSC to `r3`.

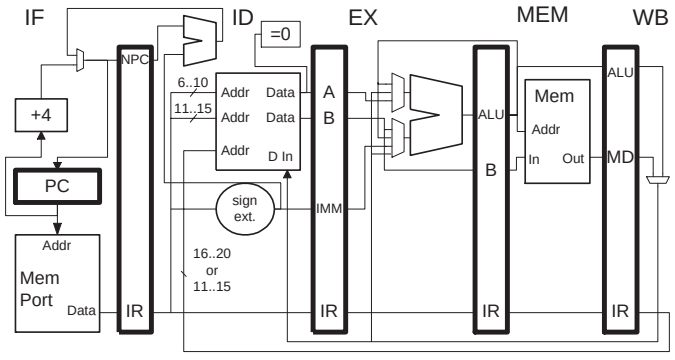
For example, consider the code below. The first instruction, `setlsc`, initializes the count to zero and sets the address to watch to the value of `r1`. When `lw` executes the count will be incremented because the addresses match. If `r1=r10-8` then the count will be incremented again when `sw` executes, otherwise the count will remain at one. The `getlsc` instruction will load `r2` with a two, if `r1=r10-8`, or a one, otherwise.

```
setlsc r1, r0+#0
lw r5, 0(r1)
sw 8(r10), r11
getlsc r2
add r3, r3, r2
add r4, r4, r2
```

(*a*) Show the changes necessary to add the two instructions to the pipeline below. (The illustration and program are repeated on the next page for convenience.)(20 pts)

- Include the hardware, including control, needed to set, increment, and read the LSC.
- The code above (and any other valid DLX program) must execute correctly and with as few stall cycles as possible.
- The solution can use basic gates, multiplexors, instruction recognizers (`=getlsc`, `=setlsc`, `=load/store`, etc.), equality testers (`=`), incrementers, and similar logic.
- Equality testers produce their output in $\frac{1}{2}$ cycle, instruction recognizers produce an output in much less than one cycle.

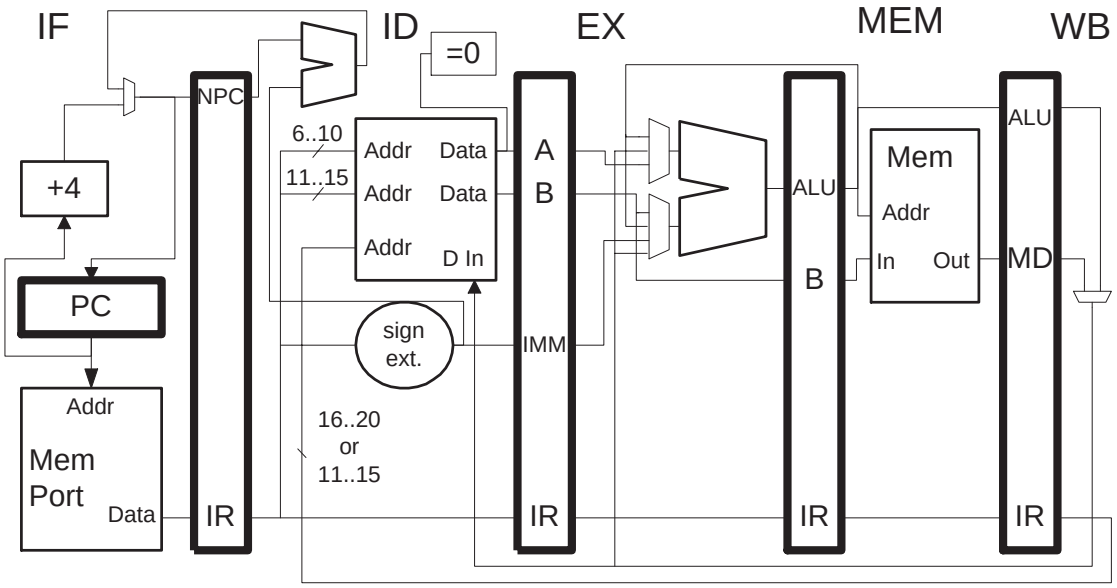
(*b*) If the solution above works correctly when instructions raise exceptions you've solved this part too. Otherwise, explain how an exception could result in incorrect execution and how it might be fixed. (Of course, the pipeline handled exceptions correctly before the changes for the first part.) (5 pts)



Use the next page for the solution.

Problem 1, continued:

```
setlsc r1, r0+#0
lw r5, 0(r1)
sw 8(r10), r11
getlsc r2
add r3, r3, r2
add r4, r4, r2
```



Problem 2: Provide pipeline execution diagrams for the code and machines indicated below. All machines have the following features in common:

- Dynamically scheduled processor using register renaming with a reorder buffer.
- Branch and branch target prediction (but no branch folding).
- Speculative execution past predicted branches with the reorder buffer used for misprediction recovery.
- Functional unit inputs connect to CDB (and reservation stations).
- Functional units and reservation stations are as listed below. The quantity of functional units is given for both an ordinary single-issue machine and a 4-way superscalar machine.

Quantity		Functional	Abbr.	Latency	Initiation	Reservation
Single	4-Way	Unit			Interval	Station Nums
1	1	Load/Store	L	1	1	0-1
1	4	Integer	EX	0	1	2-3,13-15
1	2	F.P. Add	A	1	1	4-6
1	1	F.P. Mul.	M	7	2	7-8
1	1	Branch	BR	0	1	9-10
1	1	F.P. Divide	DIV	22	23	11-12

(*a*) Show the execution of the code below on a single-issue (not superscalar) machine as described above.

The branch is predicted taken and its target is correctly predicted. However, the branch is in fact not taken. Show execution until the last instruction completes. Show when each instruction commits; indicate canceled instructions with an X. (Note: `ltf f0,f3` sets the floating-point condition register to `f0<f3`; `bfpt SKIP` branches to `SKIP` if the floating-point condition is true. The `ltf` instruction uses the FP add unit. (9 pts)

```
 addf  f0, f1, f2
 ltf   f0, f3
 bfpt  SKIP
 lf    f0, 0(r1)
SKIP:
 addf  f4, f0, f5
 addi  r1, r1, #4
```

(*b*) Show the execution of the code below on a single-issue (not superscalar) machine as described above. The branch is correctly predicted not taken. Show execution until the last instruction completes. Show when each instruction commits. Show the state of the reorder buffer when `addf` reaches writeback. (8 pts)

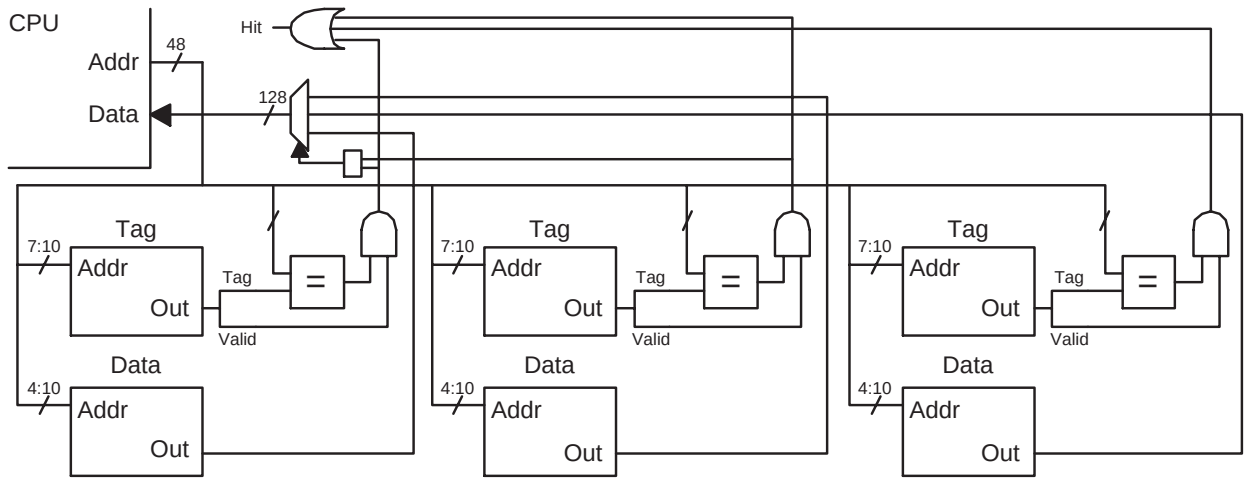
```
 multf f0, f1, f2

 ltf   f0, f3

 bfpt  SKIP

 lf    f0, 0(r1)

SKIP:

 addf  f4, f0, f5

 addi  r1, r1, #4
```

(*c*) Show the execution of the code below on a 4-way superscalar processor as described above. Instructions are fetched in aligned blocks of four. The branch is correctly predicted taken. All targets are correctly predicted. Show execution until the last instruction completes. Show when each instruction commits. (8 pts)

```
LINE0: ! LINE0 = 0x100c
 beqz r1, LINE1
 addf f0, f0, f1
 j LINE2
LINE1:
 addf f0, f0, f2
 add  r2, r2, r3
 and  r2, r2, r4
LINE2:
 addf  f0, f0, f3
 addf  f5, f5, f0
 addf  f6, f6, f0
 addf  f7, f7, f0
 addf  f8, f8, f0
 and   r5, r6, r7
 or    r8, r9, r10
 xor   r11, r12, r13
 sub   r14, r15, r16
```

Problem 3:

(*a*) A system is equipped with the cache shown below.



Determine a value for each of the following. **Be sure to specify units (bits, bytes, etc.)** (10 pts)

Character Size (Size of item at a single address.):

Associativity:

Block Size:

Number of Sets:

Cache Capacity (Amount of data that can be stored.):

Memory Needed to Implement Cache:

The problems on this page are for the cache described below, **not** *the cache from the previous page.*

Consider a system with a 64-bit address space ($a = 64$) which addresses the usual 8-bit (1-byte) characters ($c = 8$) and is equipped with a 2-way set-associative cache with 64-byte lines and 256 sets. The cache uses LRU replacement.

(*b*) Initially the cache is empty. What is the hit ratio for accesses to the array in the code below. Ignore all other memory accesses. (7 pts)

```
extern char *the_array;
// Note: sizeof(char) = 1 byte.
// &the_array[0] = 0x10000
for(i=0; i<16; i++) a += the_array[ i * 8 ];
```

(*c*) Choose values for `i_limit` and `stride` so that the program below completely fills the cache using the minimum number of accesses. Assume the cache is initially empty and only include accesses to the array. (8 pts)

```
extern char *the_array;
// Note: sizeof(char) = 1 byte.
// &the_array[0] = 0x10000

int i_limit =

int stride  =

for(i=0; i<i_limit; i++) a += the_array[ i * stride ];
```

Problem 4: Answer each question below.

(*a*) Show how the DLX instructions below are encoded. That is, write the instructions as numbers. Make up numbers for opcode and func fields, but use actual values for other fields. *Hint: If you can't remember field sizes look at the illustration for problem 1.* (5 pts)

```
 beqz r7, SKIP
 add r1, r2, r3
 xori r4, r5, #6
SKIP:
```

(*b*) Describe an advantage of VLIW over superscalar processors. Describe a disadvantage of VLIW over superscalar processors. (Use the VLIW ISA described in class.) (5 pts)

(*c*) What is a translation lookaside buffer (TLB) and what does it do? (5 pts)

(*d*) Show how an address can be constructed for the branch history table in an (m, n) two-level correlating branch predictor. (Two ways were presented in class, show either one.) (5 pts)

(*e*) Re-write the program below using DLX instructions. Additional registers may be used. (5 pts)

```
lw  r1,@(r2)          ! Memory indirect addressing.
add r3, r4, (r5)      ! Register indirect addressing.
ld  f0, (0xfedcba01) ! Direct addressing.
```

39 Spring 1998

Name _____

Computer Architecture
EE 4720
Midterm Examination

17 March 1998, 19:00–21:00 CST

Modified

Problem 1 _____ (42 pts)

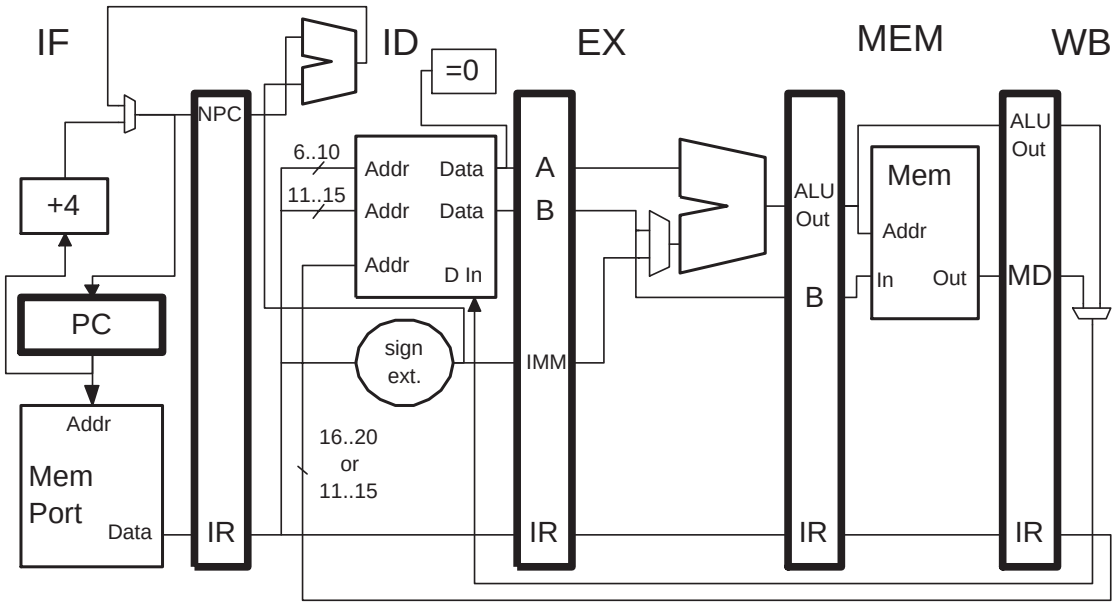
Problem 2 _____ (32 pts)

Problem 3 _____ (26 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: The DLX implementation below uses ID-stage address calculation (as illustrated) and bypassing (which is not illustrated) including the branch condition bypassing developed in homework 3. The branches do not include delay slots.




```
! Initial values: r1=2028, r2=0xf11, r3=5, r4=5004, r5=-1000, LOOP=0x3000, MEM[0xf11]=0x97
LOOP:  ! Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
 lb r1, 0(r2)

 add r2, r2, r3

 lw r4, 3(r2)

 sub r5, r1, r4

 bneq r5, LOOP

 addi r2, r2, #1
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

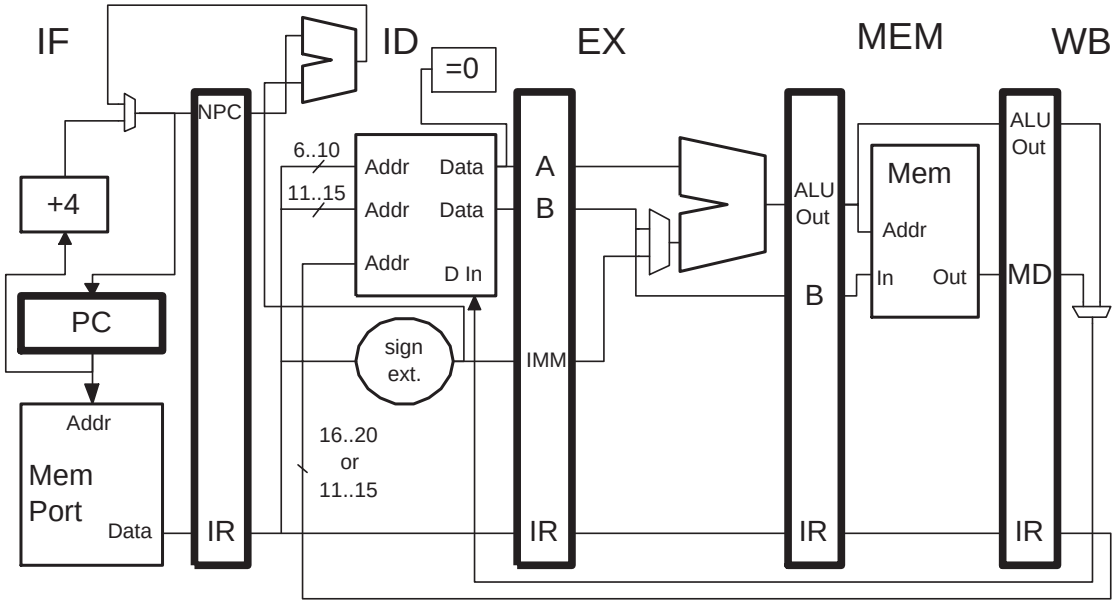
(*a*) Show a pipeline execution diagram for two iterations of the code on the DLX implementation. *Don't forget, bypass paths are present but not shown.* The space to the right of the program or a separate sheet may be used. A description of some mnemonics appears on the next page. (7 pts)

(*b*) On the figure above show exactly those bypass paths needed to execute the code. (7 pts)

(*c*) What is the CPI while executing the loop (assuming a large number of iterations)? (7 pts)

(*d*) Show the contents (values, not functions of register names) of the pipeline latches at the middle of the first cycle that `lb` is in WB. Include PC and NPC. For IR contents, just show the mnemonic; if an IR contains a nulled instruction show the original mnemonic and "(nulled)." Show the values on the diagram above, write the value within the stage to which it applies with an arrow pointing to the latch or register. (7 pts)

(*e*) Show a pipeline execution diagram of two iterations of the loop below on the DLX implementation illustrated (it's the same as the earlier one). As before, bypass paths are provided but not shown, including hw3 bypass paths. The target of the `beqz r2, SKIP2` instruction is really just the next instruction. Assume that no special hardware optimizations have been made. (7 pts)




```
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20

 addi r1,r0,#0x200
SKIP1:
 andi r2, r1, #1

 beqz r2,SKIP2
SKIP2:
 subi r1, r1, #1

 bneq r1,SKIP1

 xor  r1, r1, r1
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

(*f*) Compute the CPI of the execution of the loop above for a large number of iterations. (7 pts)

For Reference:

Mnemonic	Description
`addi`	Add immediate
`andi`	Logical "and" immediate
`beqz r1, TARGET`	Branch if `r1` equals zero.
`bneq r1, TARGET`	Branch if `r1` not zero.
`lbu`	Load byte unsigned.
`lb`	Load byte.
`lw`	Load word.

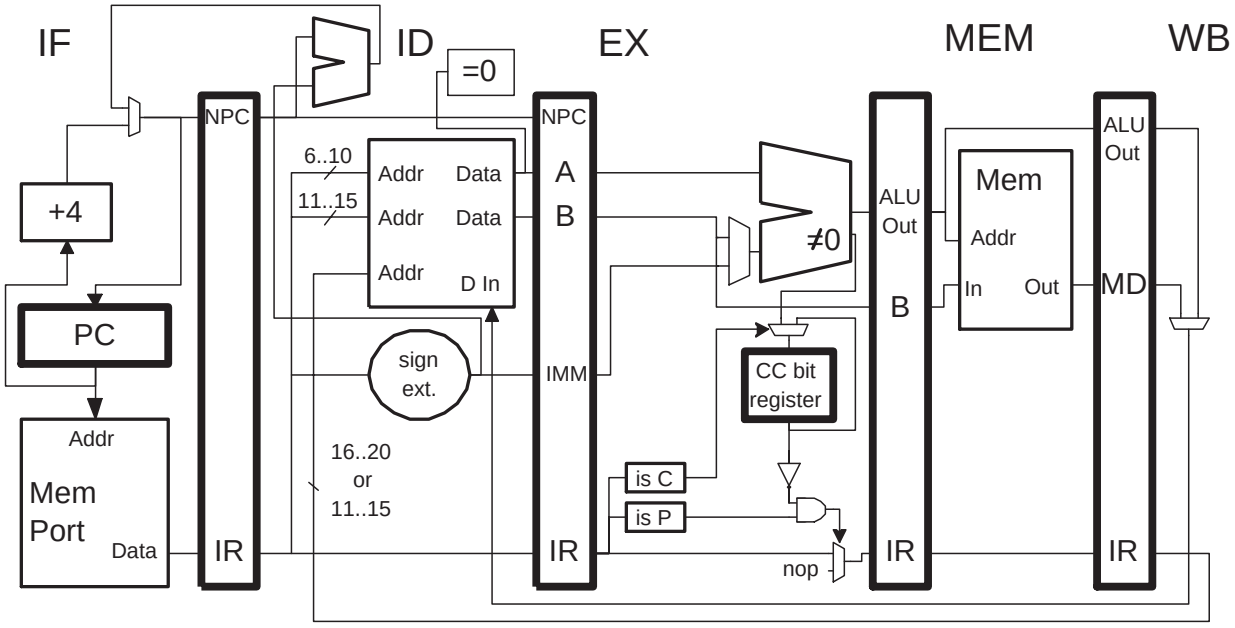
Problem 2: Consider an ISA, DLXPC, which is like DLX except it uses a condition code bit and predicated execution. The condition code bit is set by the execution of new integer arithmetic instructions, to zero if the result is zero and to one if the result is nonzero. The ISA also includes predicated arithmetic instructions which complete only if the condition code bit (set by the most recent condition-code setting instruction) is 1, otherwise they have no effect. The mnemonic for a predicated instruction has a "p" appended, *e.g.*, `add_p`, and the mnemonic for an instruction that sets the condition code has a "`c`" appended, *e.g.*, `add_c`. An instruction can be both predicated and set the condition code; for example, `add_pc` only executes if the condition code (set by a previous instruction) is 1, and sets the condition code itself.

(*a*) Using these instructions rewrite the code below so that it uses fewer instructions and registers. Assume that the value in register `r1` is not used after `beqz`.(6 pts)

```
 sub r1, r2, r3
 beqz r1, SKIP
 add r4, r5, r6
SKIP:
 addi r4, r4, #1
```

(*b*) The pipeline below implements DLXPC—almost: it will not execute predicated, condition-code setting instructions (such as `xor_pc`) correctly if an exception occurs at a certain time. Show code and a pipeline execution diagram that illustrates the problem. Be sure to point out where the exception occurs and what goes wrong. *Hint: the problem would not occur if the execution of some other instructions could be stopped in the cycle that an exception was detected.* (9 pts)

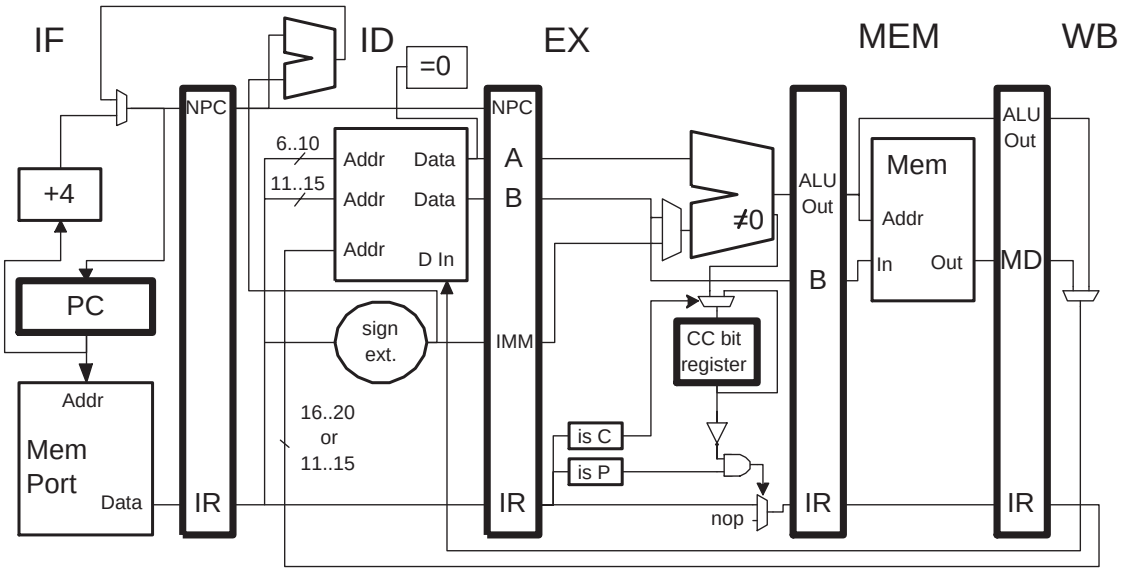
In the figure below the ALU has a second output, which is 1 if the main output is not equal to zero, 0, otherwise. The output of is C is 1 if the instruction is a type that sets the condition code (*e.g.*, `add_c`); the output of is P is 1 if the instruction is predicated, zero otherwise.



(*c*) Modify the pipeline to fix the problem. (9 pts)

(*d*) Suppose the ISA also includes a new branch instruction, mnemonic `b_p`, that tests the condition bit (instead of a register) and branches if it's true. To implement the branch instruction an ID-stage `take_branch` signal is needed. The signal should be 1 when a `b_p` instruction that will be taken is in the ID stage (zero otherwise). Show how the pipeline below would have to be modified to provide this signal. Be sure that the modified pipeline executes the code below correctly and without adding an unnecessary stalls. (8 pts)

```
sub_c r0, r1, r2  ! Set condition code.
b_p TARGET
add_c r3, r4, r5  ! Set condition code.
add r6, r7, r8    ! Doesn't affect condition code.
b_p TARGET2
```



Problem 3: Answer each question below.

(*a*) In class execution time has been modeled using the equation $t = \frac{1}{\phi} \sum_i \mathrm{IC}_i \mathrm{CPI}_i$. Given what has been covered in class so far, to what degree is CPI_i a function of the implementation and to what degree is CPI_i a function of the program? Explain. (9 pts)

(*b*) Synthetic instruction `neg r1` assigns register `r1` to the negation of its previous value, that is, `r1 = -r1`. How would such a synthetic instruction be defined in DLX? (8 pts)

(*c*) Packed-operand instructions (as found in MMX or VIS) and elaborate procedure call instructions (that perform extensive stack-frame preparation actions including register saves) are similar in that they can replace many individual instructions. Provide contrasting reasons on whether or not such instructions should be added to an ISA, including implementation factors. For example, _____instructions should be added because the implementation _____while _____should not be added because _____though certainly if _____it would _____, but that's not enough[1]. (9 pts)

[1] This is just an example, don't try to fill in the blanks!

Name _____

Computer Architecture

EE 4720

Final Examination

11 May 1998, 10:00–12:00 CDT

Modified

Problem 1	_____	(20 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: An extended version of DLX is to include a new *morph* instruction, mnemonic `mrph`. Morph takes two arguments, an instruction address and a new instruction. After executing `mrph IADDR INSTR`, when execution reaches IADDR instead of executing the instruction at IADDR, INSTR is executed. This substitution only happens once, if execution reaches IADDR a second time the instruction at IADDR executes normally (unless `mrph` was executed again). For example consider:

```
 mrph POINTA, [subd f0, f2, f6]
 ...
LOOP:
 subi r2, r2, #3
POINTA:
 muld f0, f0, f2  ! On the first iteration subd will execute.
 bneq r2, LOOP
 addd f0, f0, f4
```

In the first iteration of the loop the `subd` instruction specified by `mrph` will be executed instead of the `muld`. After that the loop will execute normally. Morph might be useful for debuggers (the substituted instruction would be a jump to a debug routine).

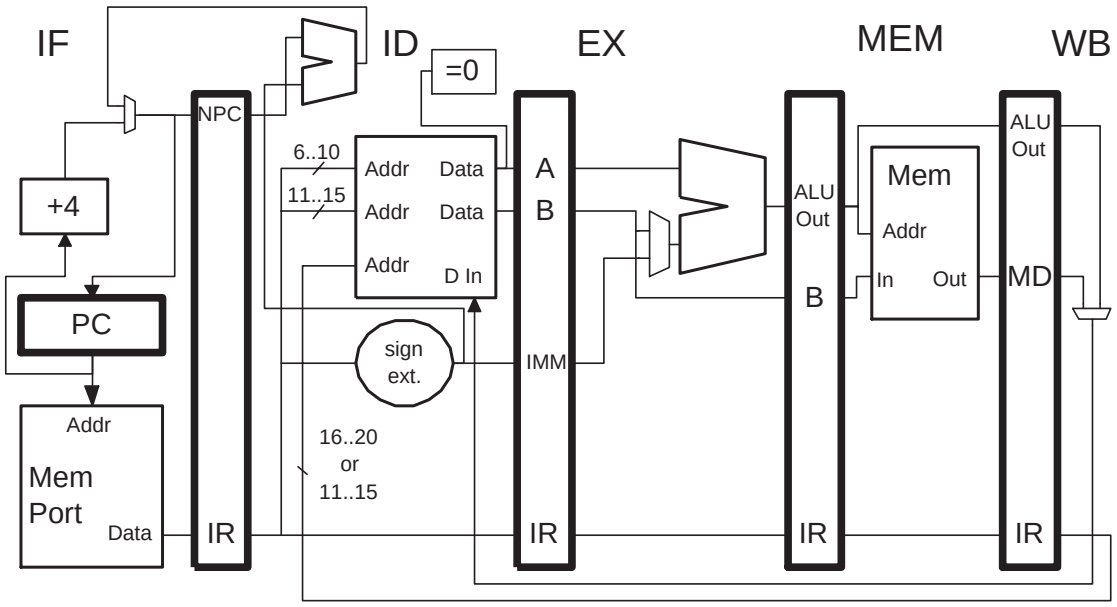
A system can have at most one morph active at any time. The morph instruction cannot modify memory[1].

(*a*) Determine a format for the morph instruction that fits naturally into the DLX ISA. (An instruction fits naturally if it uses an existing type and its implementation requires little new hardware.) The format should include the type and how the arguments are specified, that is how mrph's arguments relate to IADDR and INSTR. In other words, how is the address specified (not too difficult since many existing DLX instructions specify addresses) and how is the instruction specified (the interesting part). (The three DLX formats types are type-R, used by `add r1,r2,r3`; type-I, used by `addi r1,r2,#3`; and type-J, used by `j LOOP`. Don't confuse the assembly language instruction with the actual instruction format, the assembly language form used above may be misleading.) *Hint: If you're not sure what to do in this part, attempt the next part first.* (5 pts)

[1] Morph could be implemented in software by replacing the instruction at IADDR by a jump to a morph routine. The morph routine would execute the substituted instruction and then return.

(*b*) Modify the pipeline below to implement `mrph` using the format chosen above. The pipeline must always execute the code below correctly. (The code itself is correct.) (For partial credit if your design will not execute the code below correctly explain why and how it might be fixed.) (15 pts)

```
 bneq r10,L1
 mrph A,[add r1,r2,r3]
 j L2
L1:
 mrph SKIP,[add r0,r0,r0]
L2:
 ...
 bneq r1, SKIP     ! Hint: May cause trouble.
A:
 sub r1, r2, r3    ! Maybe add r1, r2, r3 substituted.
 lw r4, 0(r1)      ! Hint: May cause trouble.
SKIP:
 addi r4, r4, #12  ! Maybe add r0, r0, r0 substituted.
```



Problem 2: The program below executes on a 2-way, dynamically scheduled superscalar processor. The processor's features are summarized in the list below, the table gives details of the functional units. (The load/store unit computes the address in its first cycle and does the actual access in its second cycle. An operation does not enter the load/store unit until all its operands are ready.)

- ♢ Two-way superscalar instruction issue.
- ♢ Dynamically scheduled (see table), register renaming.
- ♢ CDB can accommodate results from any two functional units in any cycle.
- ♢ Reorder buffer for speculative execution and precise exceptions.
- ♢ Reorder buffer can retire as many as two instructions per cycle.
- ♢ Reservation stations, *not* reorder buffer, used for renaming.
- ♢ Zero-delay branch and branch target prediction. **No** branch folding.
- ♢ Branches do not have delay slots.

Name	Abbreviation	Latency	Initiation Interval	Reservation Station Nums
Load/Store	L	1	1	0-1
Integer	EX	0	1	2-3
F.P. Add	A	1	1	4-5
F.P. Mul.	M	5	2	6-7
Branch	BR	0	1	8-9
F.P. Divide	DIV	22	23	10-11

```
! When loop first entered r2-r1 large (loop iterates many times).
LOOP: ! LOOP = 0x1000
 lf   f0, 0(r1)     ! Don't overlook true dependencies on f0!
 mulf f0, f0, f2
 addf f0, f0, f3
 sf   4(r1), f0
 addi r1, r1, #8
 slt  r3, r1, r2
 bnez r3, LOOP
 div  f4, f5, f6
```

(*a*) Using the grid on the next page show the execution of the code above up to and including the last cycle shown on the grid. Assume perfect branch and branch target prediction, and no cache misses. Include instructions even if they have not yet finished executing at the end of the grid. (10 pts)

(*b*) Either determine and justify the CPI of an execution of a large number of iterations of the loop (ignoring cache misses and assuming perfect target prediction) or explain why it cannot easily be determined from the pipeline execution diagram. (A correct explanation of why it cannot be easily determined will get full credit, a correct CPI that left no time for problems 3 and 4 will get full credit for this subproblem and sympathy—but not credit—for omitting the others.) (4 pts)

First Iteration, Third Iteration, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`lf f0,0(r1)`																						
`mulf f0,f0,f2`																						
`addf f0,f0,f3`																						
`sf 4(r1),f0`																						
`addi r1,r1,#8`																						
`slt r3,r1,r2`																						
`bnez r3,LOOP`																						
`div f4,f5,f6`																						

Second Iteration, Fourth Iteration?, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`lf f0,0(r1)`																						
`mulf f0,f0,f2`																						
`addf f0,f0,f3`																						
`sf 4(r1),f0`																						
`addi r1,r1,#8`																						
`slt r3,r1,r2`																						
`bnez r3,LOOP`																						
`div f4,f5,f6`																						

(*c*) Suppose the second time the load (`lf`) executes it triggers a page fault exception, perhaps due to a bad load address.

In what cycle will the exception occur? Show the contents of the reorder buffer at that cycle and place a check mark next to instructions that have completed execution.

Explain how the reorder buffer will allow the exception to be precise. Specify what happens to the reorder buffer as a result of the exception, when it happens, and the contents of the buffer before and after it happens. At what cycle will the trap be inserted in the pipeline? (If solutions to the previous parts are not ready, make up a pipeline execution diagram just for this question.) (8 pts)

(*d*) What are the minimum number of reservation stations of each type needed to attain a minimum CPI on the code above? What is that CPI? (There is a grid on the next page to work this out, other methods may be faster.)(8 pts)

First Iteration, Third Iteration, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`lf f0,0(r1)`																						
`mulf f0,f0,f2`																						
`addf f0,f0,f3`																						
`sf 4(r1),f0`																						
`addi r1,r1,#8`																						
`slt r3,r1,r2`																						
`bnez r3,LOOP`																						
`div f4,f5,f6`																						

Second Iteration, Fourth Iteration?, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`lf f0,0(r1)`																						
`mulf f0,f0,f2`																						
`addf f0,f0,f3`																						
`sf 4(r1),f0`																						
`addi r1,r1,#8`																						
`slt r3,r1,r2`																						
`bnez r3,LOOP`																						
`div f4,f5,f6`																						

Problem 3: A system has a 64-bit address space ($a = 64$), addresses 16-bit characters ($c = 16$), and has a 256-bit data bus ($w = 256$).

(a) Show how memory devices could be connected to construct a 2-way set-associative cache with 1024-bit lines and 64 sets. Memory devices of any size can be used, be sure to specify their sizes (*e.g.*, $x\,\mathrm{b} \times 2^y$). Show only the connections needed to retrieve the data and tag information, determine if the access is a hit, and pass the data to the CPU. (10 pts)

(b) Suppose the cache is write-through. What is the capacity of the cache and how much memory is needed to implement it? Be sure to specify units. (5 pts)

(c) Find the addresses specified below. (5 pts)

- *Three* different address that are part of the same line and require exactly two loads to access. (Assume there are 256-bit load instructions.)

- Two addresses that can be in different lines but the same set.

- Three addresses that are in different sets.

Problem 4: Answer each question below.

(*a*) What is branch folding? In the implementations of DLX covered in class, why must the predictions used for branch folding always be correct? (6 pts)

(*b*) Why might the cost of a functional unit with an initiation interval of 2 be less than one performing the same operations but with an initiation interval of 1 but having the same latency? Given such a cost relationship, what should the minimum number of reservation stations be for a functional unit with an initiation interval of ι and a latency of λ? Explain. (6 pts)

(*c*) What are some difficulties that might be encountered in developing a superscalar implementation of a stack ISA? (For partial credit, list some distinguishing features of a stack ISA.) (6 pts)

(*d*) Explain how a reservation register can be used to detect MEM-stage structural hazards while an instruction is in the ID stage. (6 pts)

(*e*) What is the difference between a RAW hazard and a true dependency? (6 pts)

40 Spring 1997

Name _____

Computer Architecture
EE 4720
Midterm Examination

14 March 1997, 12:40–13:30 CST

Problem 1 _____ (30 pts)

Problem 2 _____ (30 pts)

Problem 3 _____ (40 pts)

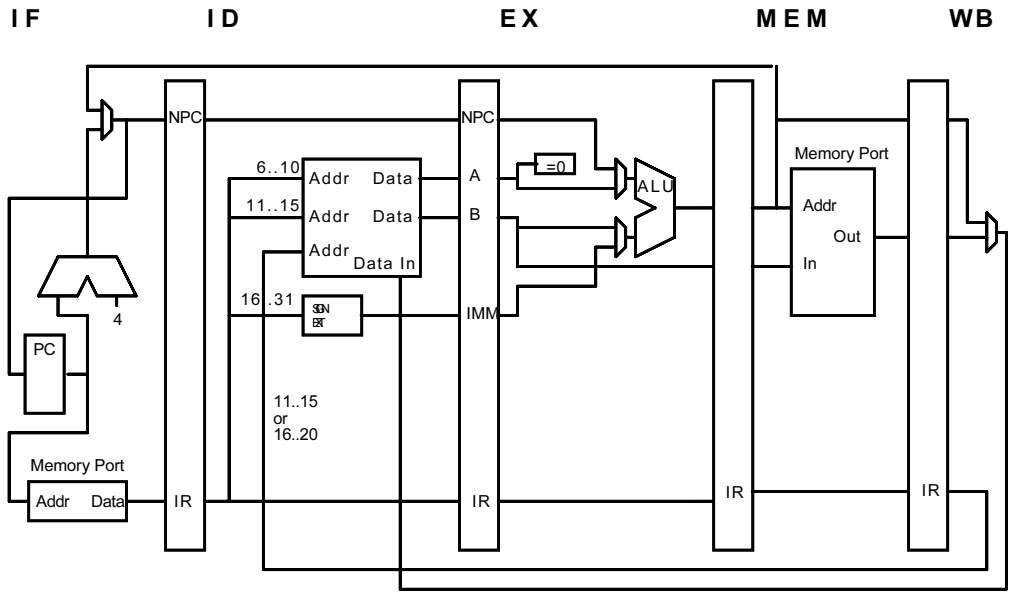
Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: Consider a new instruction, `mems r1,(r2)`, which loads and examines consecutive memory words, starting at the address stored in r2, until a word equal to the contents of r1 is loaded. When the instruction finishes, r2 will have the address of the matching word. (The loaded words are not written to registers, although they might be temporarily stored somewhere.) For example, if the contents of r1 was 123 and the contents of r2 was 1000, and a 123 were stored at memory location 1016 (and not at memory locations before 1016), then after `mems r1,(r2)` executed there would be a 1016 in r2.

(*a*) Modify the pipeline to implement this instruction and show its execution on the modified pipeline (ID, EX, etc.). Show only the data connections on the modified pipeline, not control or changes in the IR path. When showing the execution of `mems` briefly explain what is happening in each stage. The implementation can be either low cost or high speed. (10 pts)

(Hint: first do part b. Then returning to this part, try to figure out how the instruction might execute. Use this to modify the pipeline and to solve part c. Even if the execution is based on a wild guess, it can still be used for part c.)



Problem 1, continued.

(*b*) Write a DLX program that performs the same function as `mems`. (10 pts)

(*c*) Compare the time needed to execute mems to the time needed to run the program when the word is found after 1000 loads. (10 pts)

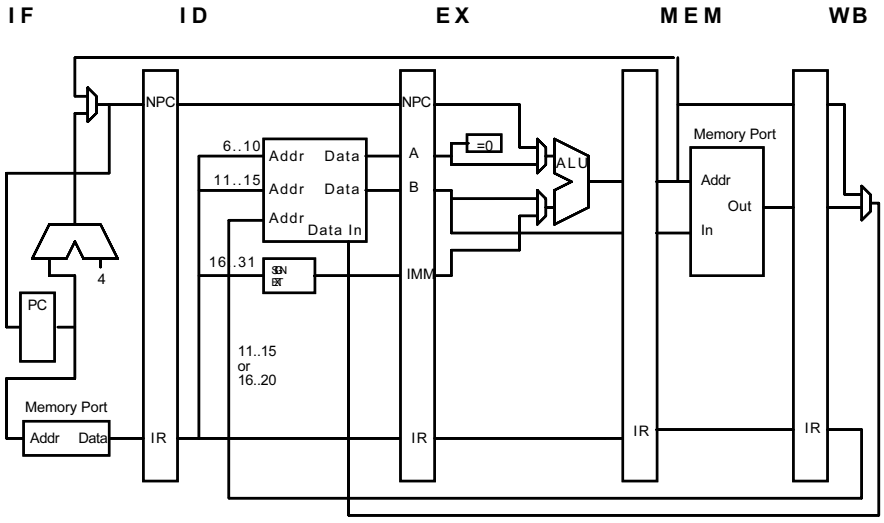
Reference Information:

Instruction type/opcode	**Instruction Meaning**
LW, SW	Load word, store word (to/from integer registers)
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
AND, ANDI	And, and immediates
LHI	Load high immediate—loads upper half of register with immediate
S__	Set conditional: "__" may be `LT, GT, LE, GE, EQ, NE`
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 `(J)` or target in register `(JR)`
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative `(JAL)` or a register `(JALR)`

Problem 2: Suppose arithmetic instructions skipped the MEM stage, going straight from EX to WB, saving a cycle.

(*a*) Because the register memory has only one write port, this scheme introduces a structural hazard in the WB stage. Find a sequence of instructions that encounter the hazard, show how they would execute (IF ID, etc.) if nothing were done about the hazard, and show where the hazard occurs. (7 pts)

(*b*) Ignoring the hazard, show how the pipeline would have to be modified to accommodate such instructions. Show both data and control (IR path) modifications. (8 pts)



(*c*) Describe two ways the structural hazard could be dealt with without delaying execution. (8 pts)

(*d*) Is this skipping of the MEM stage for arithmetic instructions worthwhile? *(Hint: It depends on whether register forwarding is used.)* Explain, stating any reasonable assumptions about pipeline features. (7 pts)

Problem 3: Answer each question below and on the next page.

(*a*) An ISA defines 200 out of 256 possible opcodes. An implementation can raise an illegal instruction exception when any of the unused instructions are encountered, or it might allow the instruction to execute (with the result being undefined). The first option is more expensive than the second. Why is the first option still the better choice? (8 pts)

(*b*) Describe two differences between a trap instruction and a subroutine call (jump and link). (8 pts)

(*c*) Why is the pipelined DLX implementation able to start fetching registers before decoding the instruction? (8 pts)

It's not over yet, more problems on next page.

(*d*) What is the advantage of the geometric mean over the harmonic and arithmetic means when combining normalized execution rates? (8 pts)

(*e*) Consider two implementations of an ISA, one old and one new. Describe why each factor, ϕ, IC, and CPI, might change from the old to the new implementation. (8 pts)

Name _____

Computer Architecture
EE 4720
Final Examination

10 May 1997, 12:30–14:30 CDT

Problem 1 _____ (25 pts)

Problem 2 _____ (25 pts)

Problem 3 _____ (25 pts)

Problem 4 _____ (25 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: DLX's immediate instructions use 16-bit immediates. Because in many cases larger immediates are needed, new *larger-immediate* instructions are to be added to DLX. Larger-immediate instructions specify an integer arithmetic operation and an immediate, but no registers. The operation is performed using the immediate as one source and the most-recent destination register as both the other source and the destination. For example, consider larger-immediate instruction `addli` in the code fragment below:

```
lw r6, 0(r7)
sub r1, r2, r3
sw 0(r4), r5
addli #0x1ffff  ! Operation: r1 = r1 + 0x1ffff
```

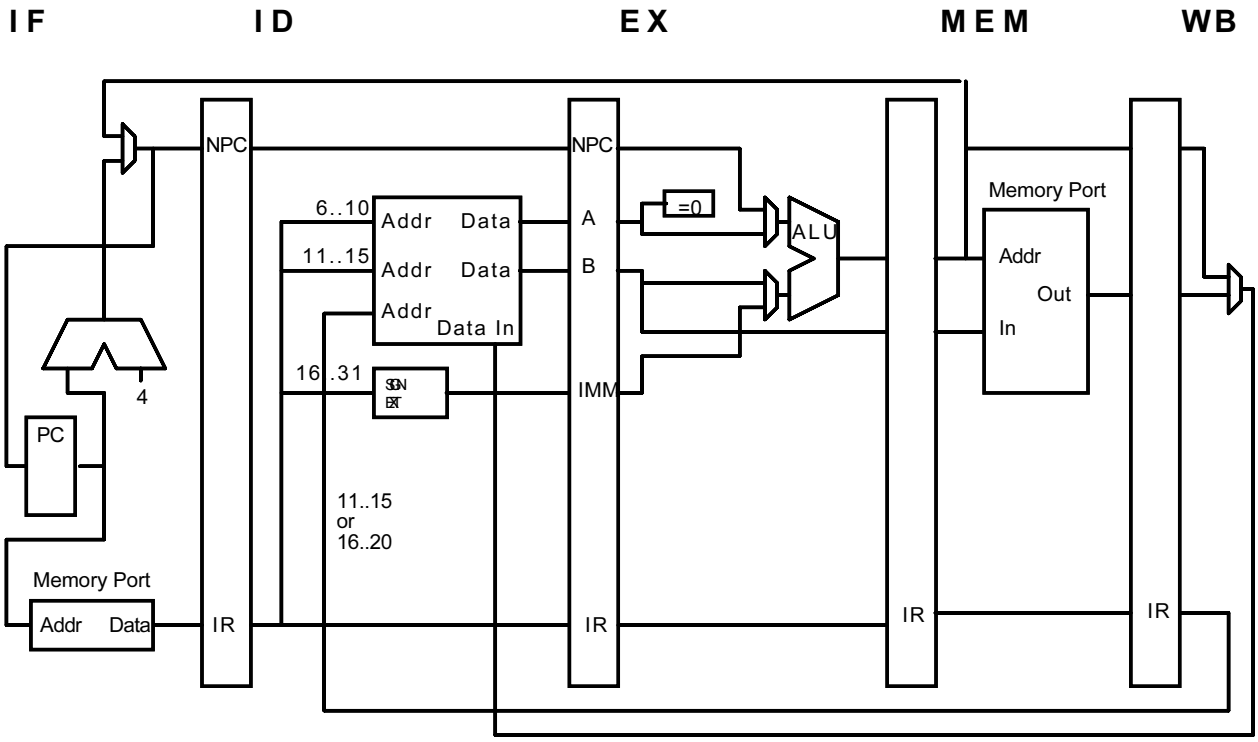
The `addli` instruction uses `r1` because it was the most-recent destination register used. (The `sw` does not modify `r4` or `r5`, so they aren't used. The `lw` does modify `r6`, but it is executed before the `sub`.) The larger-immediate instructions use the J-type format: a 6-bit opcode followed by a 26-bit immediate.

(*a*) Let register `r10` hold the memory address of an integer. The sum of that integer and 0x981234 is to be written to `r11`. Write two code fragments to perform this operation, one with and one without a larger-immediate instruction. (5 pts)

Problem 1 continued: (*b*) Show the modifications needed to implement larger-immediate instructions on the pipeline illustrated below. (Read the next part before solving.) (10 pts)

(*c*) Show how bypassing can be implemented for the instruction. *(Hint: This is easier than regular bypassing since one source is the most-recent destination.)* (10 pts)

- Addresses must be presented to the register file at the beginning of a cycle and the data won't be available until the end of the cycle.
- Be sure to label the function of each connection to registers and other devices. (*E.g.*, address, write, data.) Avoid magic boxes and clouds.
- Explain your modifications using an annotated timing diagram showing sample code executing. A detailed logic and timing diagram are preferred over a lengthy verbal description.



Problem 2: The code fragment repeated below is to be executed on several machines similar to DLX. All machines have an integer execution unit, a multiply FP unit, an add FP unit, and a divide FP unit. The integer unit takes one cycle, the add unit takes two cycles, the multiply unit takes 4 cycles, and the divide unit takes nine cycles. The multiply unit has an initiation interval of 2, the divide unit has an initiation interval of 9, all other functional units have an initiation of 1. In all cases make sure distinct pipeline segments have distinct names. All registers and functional units are available when the fragment starts. State any assumptions. (25 pts)

(*a*) Show how the sequence would execute on a single-issue pipeline without register renaming and which is fully bypassed.

```
div f0,f10,f11

mul f0,f3,f2

add f8,f9,f0

add f0,f1,f2

mul f3,f0,f4

mul f5,f0,f6
```

(*b*) Show how the sequence would execute on a single-issue pipeline using Tomasulo's approach as described in class, with two reservation stations per functional unit.

```
div f0,f10,f11

mul f0,f3,f2

add f8,f9,f0

add f0,f1,f2

mul f3,f0,f4

mul f5,f0,f6
```

Problem 2 continued:

(*c*) Show how the sequence would execute on a dynamic issue two-way superscalar machine using Tomasulo's approach with two reservation stations per functional unit. The superscalar machine has the same functional units as the others. (*I.e.*, there is not two of every functional unit.) Of course, there are two common data busses; every functional unit can write either one.

```
div f0,f10,f11

mul f0,f3,f2

add f8,f9,f0

add f0,f1,f2

mul f3,f0,f4

mul f5,f0,f6
```

(*d*) Show how the sequence would execute on a static issue two-way superscalar machine. The superscalar machine has the same functional units as the others. (*I.e.*, there is not two of every functional unit.) The pipelines are fully bypassed.

```
div f0,f10,f11

mul f0,f3,f2

add f8,f9,f0

add f0,f1,f2

mul f3,f0,f4

mul f5,f0,f6
```

Problem 3: Consider a 3-way, set-associative cache with 4096 sets and 256-byte blocks using LRU replacement to be used with a CPU reading data in aligned 8-byte units. The size of the address space is 64 bits, the cache uses virtual addresses.

(*a*) (1) Show the bit positions used for the tag, index, and offset. (2) How many bytes can the cache cache? (3) How many bytes of memory does it take to implement the cache? (5 pts)

(*b*) Modify the program below (or re-write completely) so that it moves as many blocks as possible (without replacing other blocks) into each of 2048 sets and moves exactly one block into each of the other 2048 sets. The program should accomplish this using the **minimum** number of array accesses. The cache is empty before the program is run; assume that only array accesses generate memory references. (5 pts)

```
double *a = 0x10000000;  /* Assume the entire address space is ours. */
for( i=0; i<5; i++ ) total += a[i];
```

The size of a double is 8 bytes. The address of `a[0]` is 0x10000000.

Problem 3 continued:

(*c*) Because of the 64-bit address space, a large amount of memory is used by the cache for storing tags. If the number of distinct tags present in the cache is much less than the number of blocks, that memory is wasted. To reduce the storage needed, tags can be converted to smaller *htags* using a hash function. A hash function is given[1] which hashes a tag to an 8-bit htag. (If you're hopelessly confused, see the last point below.) Design a cache with the above specifications (3-way ...) that uses the hash function to reduce the amount of storage needed for tags. (15 pts)

- Show hardware to convert addresses from the CPU to addresses for the cache's memory devices, to detect cache hits, to detect hash-function collisions, and to connect data read from the cache to the CPU. (The hash function can map two different tags to the same htag, that's called a collision.) *Do not* show hardware needed for writes, hardware to handle misses, and connections to main memory.
- Show the hash function itself as a box with a single input and output. Assume memory devices of any size are available (any number of address bits and any number of data bits). Where appropriate, specify the bits used in a connection (*e.g.*, 1..3). Omit controller details.
- Briefly describe what happens when (1) an address with a new tag is presented; (2) an address with an already-cached tag is presented (Note that this can be a miss or a hit.); and (3) an address with a tag colliding with a tag in the cache is presented.
- **For reduced credit:** (1) design just the cache described on the previous page (showing the memory devices, how the memory address is converted to addresses for the devices, etc.) and (2) assuming the cache is write-back, describe the sequence of events during a cache hit and cache miss requiring replacement. *This part can be skipped if the tag-hashing design is presented.*

[1] Read this footnote after the exam. The hash function would have to be more elaborate than the simple bit mask used in the BHT. The hash function properties and design are not a part of the problem, so don't think them until the test is over.

Problem 4: Answer each question below and on the next page.

(*a*) As presented in class, a system using Tomasulo's approach attempts to write the common data bus in the cycle *after* execution is complete. The DLX pipeline using bypassing was able to transmit a result in the last cycle of execution. Why would it be more difficult (but far from impossible) to write the common data bus in the last cycle of execution than it is to do bypassing? (5 pts)

(*b*) Computer engineers are hard at work on the next implementation of an architecture. Consider two benchmarks, one is a synthetic benchmark written by the engineers, the other is an application benchmark suite used by *P.O. Signers*, a computer magazine the engineers' customers read and respect. Describe a situation in which the benchmark suite would be a better choice for the engineers and another situation where the synthetic benchmark would be the better choice. Briefly justify your answers. (5 pts)

Problem 4 continued:

(*c*) During program execution on the Chapter 3 DLX implementation, an exception is raised in the IF stage while fetching an `add` instruction. No other exceptions occur on this or previous cycles. Show a timing diagram in which this occurs and in which the trap handler for a *different* instruction ends up being called. Explain why a different handler was called. Point out important steps in the timing diagram. The timing diagram should end with the fetch of the first instruction of the handler. (5 pts)

(*d*) How is the code scheduling done for static-issue superscalar machines similar to, and different from, the re-compiling needed for VLIW machines? (5 pts)

(*e*) Suppose the value of m were to be increased in an (m, n) correlating branch predictor while keeping the size of the branch history table constant. Give a reason why branch prediction might improve, and a reason why branch prediction might get worse. (Both are possible, depending upon how large m was before the increase.) (5 pts)

Yes, it's finally over.

41 Spring 2025 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

Friday, 21 March 2025 9:30-10:20 CDT

Problem 1		(30 pts)
Problem 2		(30 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

Alias Does it have to be good?

Good Luck!

Problem 1: [30 pts] Appearing on the facing page is the MIPS implementation that includes the `addsc` from Homework 3.

(*a*) The first four code fragments below will execute as shown with the illustrated control logic (from the Homework 3 solution), but the logic won't generate the stall for the last fragment.

☑ Add control logic to the implementation so that *all* of the code fragments execute as shown. That is, add logic to generate the stall signal ☑ for the last fragment ☑ without changing whether the others stall.

Solution and discussion on the next page.

```
lw R5, 8(r2)          IF ID EX ME WB         # Correctly stalls with existing logic.
addsc r3, r4, R5, 7      IF ID -> EX ME WB

lw R4, 8(r2)          IF ID EX ME WB         # Correct with existing logic.
addsc r3, R4, r5, 7      IF ID EX ME WB

xori R4, r2, 8        IF ID EX ME WB         # Correct with existing logic.
and r6, R4, r5           IF ID EX ME WB

lw R5, 8(r2)          IF ID EX ME WB         # Correctly stalls with existing logic.
xor r6, r4, R5           IF ID -> EX ME WB

# Cycle               0  1  2  3  4  5  6
lw R4, 8(r2)          IF ID EX ME WB         # Should stall but doesn't with existing logic.
xor r6, R4, r5           IF ID -> EX ME WB
```

(*b*) Notice that in the first two fragments below the `addsc` shift amount is zero, and so those instructions just add. In the first fragment `addsc` executes in `ME` due to the load dependence, but in the second fragment it executes in `EX` so it can avoid stalling the `or`. *Note: The material about* `doADDSC` *described below was not in the original exam.*

☑ Modify the control logic so that an `addsc` with a zero shift executes as shown below. Do so by ☑ relabeling the `isADDSC` pipeline latches to `doADDSC`. Set this signal to `1` only if there is an `addsc` in `ID` that needs to execute in `ME`. ☑ The logic should not break correct behavior for other cases, such as the ones above.

Solution and discussion on the next page.

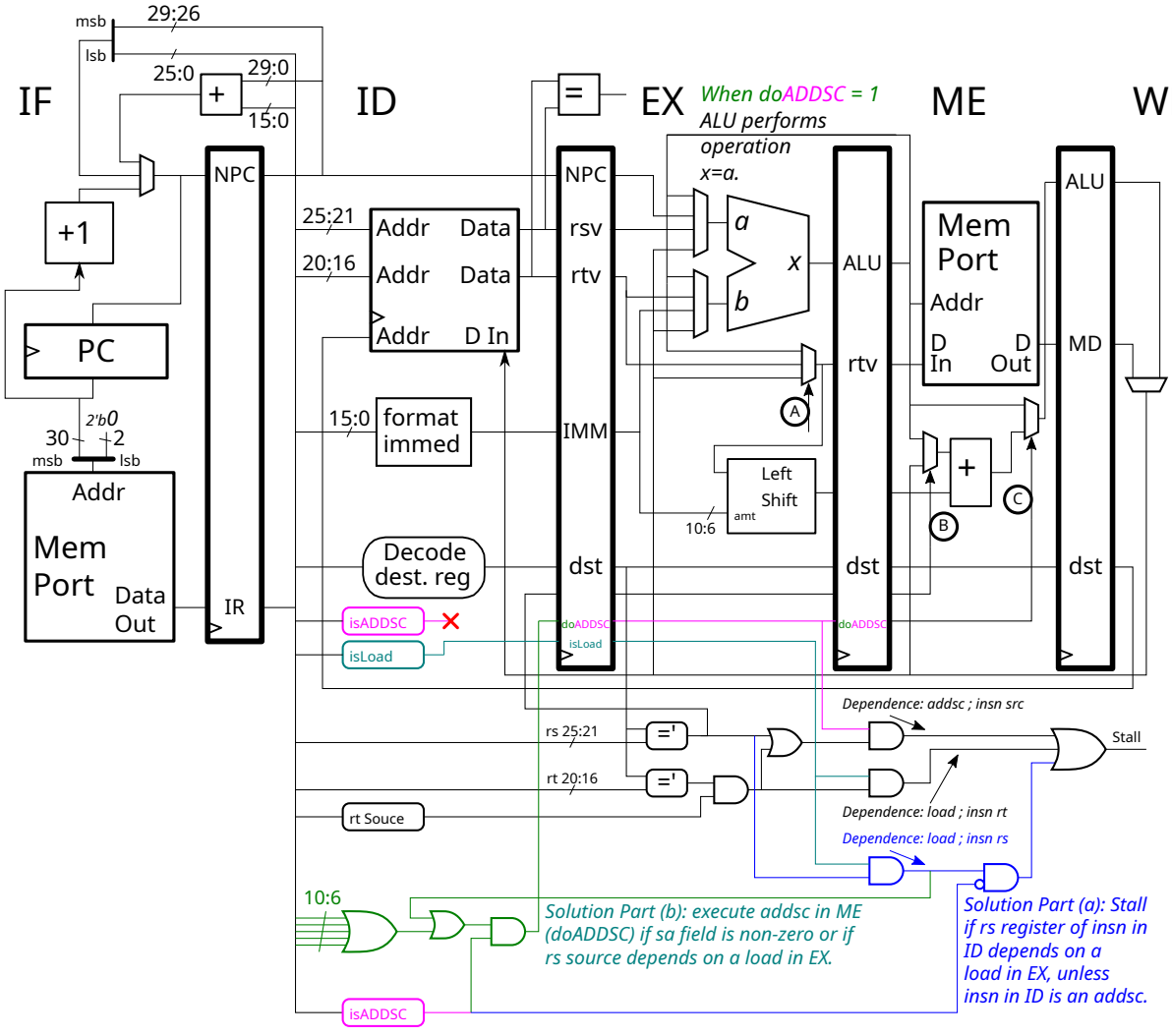
```
# Cycle               0  1  2  3  4  5  6  7  Fragment b1 - addsc adds in ME.
lw R2, 8(r9)          IF ID EX ME WB
addsc r1, R2, r3, 0      IF ID EX ME WB
or    r5, r10, r6           IF ID EX ME WB

# Cycle               0  1  2  3  4  5  6  7  Fragment b2 - addsc adds in EX
andi R2, r9, 8        IF ID EX ME WB
addsc R1, R2, r3, 0      IF ID EX ME WB
or    r5, R1, R6            IF ID EX ME WB

# Cycle               0  1  2  3  4  5  6  7  Fragment b3 - addsc adds in ME.
lw R6, 8(r9)          IF ID EX ME WB
addsc R1, r2, r3, 4      IF ID EX ME WB
or    r5, R1, R6            IF ID -> EX ME WB
```

The solution to Part a appears below in blue. The last code fragment stalls because of a dependence between a load, `lw` in particular, and the rs register of an ALU instruction, `xor`. The stall occurs in cycle 2, when the `xor` is in `ID` and the `lw` is in `EX`. (The `xor` is also in `ID` during cycle 3, but it is not stalled, it's free to leave in the next cycle.) For this discussion refer to cycle 2 in the last code fragment. A new AND gate checks for the dependence, its output is commented "Dependence: load; insn rs." There should *not* be a stall if the instruction in `ID` is an `addsc` because it is possible to bypass the loaded value without a stall. That case is handled by the second blue AND gate which suppresses the stall if there is an `addsc` in `ID`.

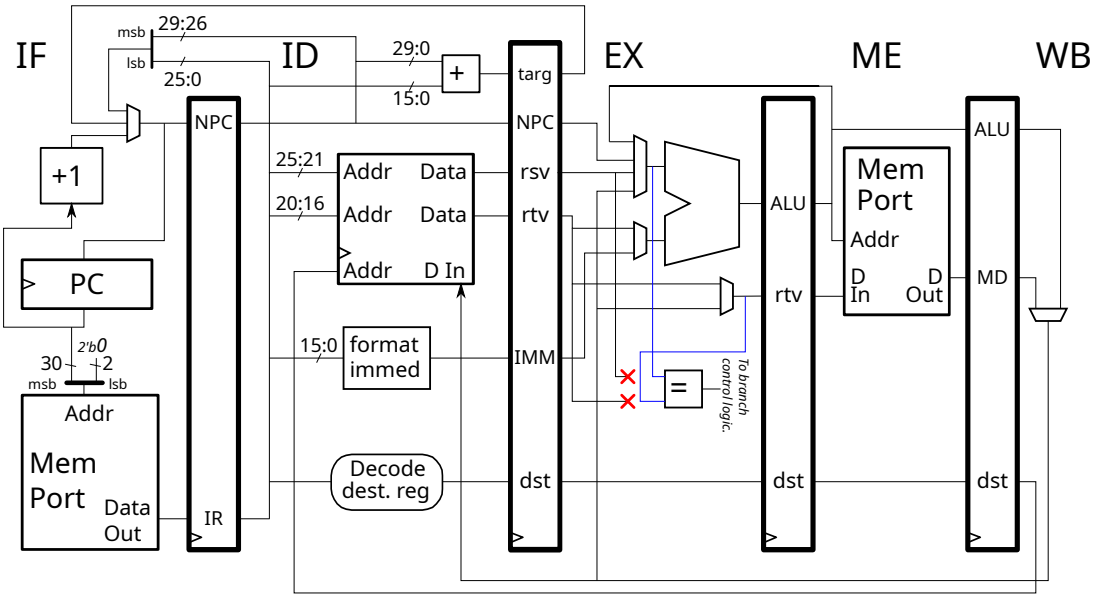
The solution to Part b appears in green. The goal is to execute `addsc` in `EX` in some cases and `ME` in other cases. When `addsc` is executed in `EX` it is treated like an ordinary `add` instruction. The control signal `isADDSC` that appeared in the pipeline latches has been renamed to `doADDSC`. When `doADDSC=0` an `addsc` is treated like an `add` instruction, executing in `EX`. When `doADDSC=1` the ALU will execute the `x=a` operation and the C select signal will route the `ME`-stage adder output to the `WB.ALU` pipeline latch (otherwise it routes the `EX.ALU` value to the `WB.ALU` latch).

The value of `doADDSC` is set to `1` if there is an `addsc` in `ID` and either the `sa` field is non-zero (and so it must use the `ME`-stage adder) or the `rs` source of the `addsc` depends on a load in `EX` (and so uses the `ME`-stage adder to avoid a stall). The logic computing this is on the lower left.

Grading Note: Many students mistakenly thought that a stall signal was needed. The problem as given did not mention the `doADDSC` *signal which might have contributed to this mistake.*



Problem 2: [30 pts] In the MIPS implementation below pay attention to bypass paths and how the branch is resolved.



☑ Show the execution of this code on the implementation above. ☑ Don't forget to check for dependencies!

Solution appears below. The `and` stalls because its `rt` register, `R4`, could not be bypassed in cycle 4 since in this particular implementation there is no bypass path to the lower ALU input. The other dependencies are with the `rs` register, and those can be bypassed (to the upper ALU input).

```
## SOLUTION
#  Cycle        0  1  2  3  4  5  6  7  8  9
add R1, r2, r3  IF ID EX ME WB
sub R4, R1, r5     IF ID EX ME WB
and R6, r7, R4        IF ID ----> EX ME WB
sw  r1, 0(R6)            IF ----> ID EX ME WB
#  Cycle        0  1  2  3  4  5  6  7  8  9
```

☑ Show the execution of this code on the implementation above. ☑ Don't forget to check for dependencies!

Solution appears below. There is a dependency from the `addi` to the two `sw` instructions. The EX-stage mux routing a value to the `ME.rtv` input can bypass a value from WB but not ME, and so the first `sw` must stall one cycle. Note that there is no problem for the second `sw` because it is in ID when `addi` is in WB and so it can get the value from the register file.

```
## SOLUTION
#  Cycle         0  1  2  3  4  5  6  7
addi R1, r1, 1   IF ID EX ME WB
sw R1, 0(r2)        IF ID -> EX ME WB
sw R1, 4(r2)           IF -> ID EX ME WB
```

☑ Show the execution of this code on the implementation above. ☑ Don't forget to check for dependencies!

There are no stalls here because each **sw** can use the bypass from **WB**.

```
## SOLUTION
lw r1, 0(r2)  IF ID EX ME WB
lw r4, 4(r2)     IF ID EX ME WB
sw r1, 0(r3)        IF ID EX ME WB
sw r4, 4(r3)           IF ID EX ME WB
```

☑ Show the execution of the code below with ☑ the branch taken on the implementation above. ☑ Don't forget to check for dependencies! ☑ Pay attention to branch behavior.

Solution appears below. The **beq** stalls two cycles because there are no bypass paths to the **EX**-stage comparison unit needed by the branch. Also notice that the branch is resolved in **EX**, and so the correct target is not fetched until the branch is in **ME**. As a result the **and** instruction is fetched and then squashed.

```
## SOLUTION
#  Cycle          0  1  2  3  4  5  6  7  8  9  10
add R1, r2, r3    IF ID EX ME WB
beq R1, r4, TARG     IF ID ----> EX ME WB
sw R1, 0(r8)            IF ----> ID EX ME WB
and r5, r1, r6                   IFx
ori r5, r5, 0x6
sw r5, 4(r8)

TARG:
lw r1, 8(r8)                        IF ID EX ME WB
#  Cycle          0  1  2  3  4  5  6  7  8  9  10
```

☑ Show how the inputs to the [=] box in **EX** can be changed to eliminate stall(s) ☑ in the example above, and stalls for other kinds of ☑ dependenices. ☑ Do not add hardware, just change the inputs.

Solution appears on the diagram in blue. The **rs** value is taken from the mux at the upper ALU input. That works here because the ALU is not computing the branch target. The **rt** value is taken from the mux for the store value, which can bypass from **WB**. In contrast the lower ALU mux can't bypass at all.

With these changes the branch would not stall. However, because it is resolved in **EX** there would still be a squashed instruction.

Problem 3: [40 pts] Answer each question below.

(*a*) In the routine below `r4` holds an integer, call its value x, and `f1` holds a single-precision float, call its value y. Complete the routine so that register `f9` holds $x \times y$ in single-precision floating point.

☑ Complete the routine so that `f9` is written with the product of the values of `r4` and `f1`. The solution only requires a few instructions, ☑ don't try to fill the entire page.

```
add r4, r5, r5
add.s f1, f2, f3

# At this point r4 holds an integer and f1 holds a single-precision float.

# SOLUTION
#
# Let x denote the value in r4.
# At this point r4 holds an integer representation of x.
#
# Let y denote the value in f1.
# At this point f1 holds a single-precision floating point representation of y.
#

mtc1 f4, r4          # Move x to f4.

# At this point f4 holds an integer representation of x.

cvt.s.w f5, f4       # Convert x to a single-precision FP representation.

# At this point f4 still holds an integer representation of x ...
# ... but f5 now holds a single-precision FP representation of x.

mul.s f9, f1, f5     # Perform the multiplication.
```

(*b*) The three MIPS code fragments below each do the same thing, and infinite loops are not the problem.

```
loop:   # Fragment A
        sw $t4, 0($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4

loop:   # Fragment B
        sw $t4, 0($t5)
        sw $t4, 4($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 8

loop:   # Fragment C
        sb $t4, 0($t5)
        sb $t4, 1($t5)
        sb $t4, 2($t5)
        sb $t4, 3($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4
```

☑ Which code fragment is the fastest, ◯ *Fragment A*, ⊗ *Fragment B*, or ◯ *Fragment C*?

☑ Which code fragment is the slowest, ◯ *Fragment A*, ◯ *Fragment B*, or ⊗ *Fragment C*?

☑ Explain choice of ☑ fastest and ☑ slowest fragment, and ☑ include a good definition of fast.

Here *fast* will be defined as executing few instructions to complete a task. The task here is initializing an area of memory from `t5` to `t3` with the value in `t4`. (This is taken from the solution Homework 1.) So the fastest fragment is the one that executes the fewest instructions.

The number of iterations performed by the Fragment A and Fragment C loops is the same, call the number n. Because `t5` is incremented by 8 rather than 4, Fragment B runs for just $n/2$ iterations.

An iteration of Fragment A is 3 instructions, an iteration of B is 4 instructions, and an iteration of C is 6 instructions. The total number of instructions is $3n$ for A, $4\frac{n}{2} = 2n$ for B, and $6n$ for C. So B is the fastest and C is the slowest.

☑ Assume that the contents of `t5` and `t3` refer to a range of valid memory addresses. Which fragment(s) put a restriction on the value of `t5`? ☑ Explain. Assume that `t3` is always chosen to avoid an infinite loop.

MIPS memory accesses are aligned, meaning the memory address must be a multiple of the access size. The access size for a `sw` is 4 and so `t5` must be a multiple of 4. That restriction applies to Fragments A and B. The access size for `sb` is 1 and so Fragment C imposes no restriction on `t5`.

(*c*) When designing a RISC ISA what is the most important criterion when considering possible instructions based on the material presented in class?

☑ Most important factor when deciding whether an instruction should be added to a RISC ISA.

In class it was explained that the most criterion for RISC is easy pipelining. Other RISC characteristics, such as fixed instruction size and restricting memory access to load and store instructions are ways of facilitating easy pipelining.

☑ Give an example of an instruction unsuitable for RISC and ☑ explain how the criterion makes it unsuitable.

An instruction such as `add r1, (r3), (r4)` because that would require two memory accesses before the arithmetic operation. A pipelined implementation that could implement it would need two memory ports before the ALU. Memory ports are expensive, so it would not be worthwhile to pipeline it.

(*d*) CISC ISAs have powerful instructions, such as `add 4(r1), (r2), ((r3))` or a call instruction that automatically saves registers.

☑ What is the benefit of powerful instructions, especially in the days when memory was made by people sewing wires around little metal rings.

Programs using such powerful instructions take up less space. Here is the equivalent MIPS code for the example instruction:

```
# add 4(r1), (r2), ((r3))

lw r12, 0(r2)  # Put value of (r2) into r12
lw r4, 0(r3)
lw r13, 0(r4)  # Put value of ((r3)) into r13
add r11, r12, r13
sw r11, 4(r1)
```

The MIPS equivalent uses 5 instructions taking up 20 bytes of memory. The size of the CISC instruction would be less, maybe 4 or 5 bytes.

One cannot easily claim that the *implementation* of a RISC ISA would be faster or slower than a CISC ISA. Current CISC implementations work by *cracking* CISC instructions into RISC-like *micro-ops* where they proceed through pipelined hardware. This adds to complexity (and the latency of cracking) but if your sales are high enough you can pay for the computer engineers to handle it.

(*e*) Intel has updated IA-32 (a.k.a. x86) since the 1980s, and later added a 64-bit variant, Intel-64. Recall that nobody actually likes IA-32.

☑ So why did Intel's customers continue to buy implementations of IA-32 and Intel 64 rather than switching to a better-designed ISA? (Note that Apple is an exception to the rule that computer makers don't switch ISAs.)

Those Intel customers (or the customers' customers) had lots of software for IA-32 (and later Intel 64). (One notable Intel customer is IBM, using the processor in their PC. It is IBM's customers that had lots of software.) When they buy a newer implementation of IA-32 they can run their old software as is. But, if they switch to a new ISA then they must recompile or port their old software, a time-consuming distraction. To get a customer to switch ISAs the implementation of the new ISA would need to be so much better than the old ISA that its is worth the trouble.

The Apple Mac (Macintosh) did switch ISAs several times. It started with the Motorola 68020, then PowerPC, then Intel 64, and now uses Arm A64. This worked for Apple in part because many layers of the software is Apple-written, and they'd make the effort to port it to the newer ISA. Third-party software would call Apple-written libraries for many time-sensitive tasks (such as for the user interface) and the remaining parts could run under emulation until ported. (Emulated code is run by simulating the older ISA for which it was written.) And yes, there's also the reality distortion field that insured everyone would go along without complaining.

(*f*) MIPS uses the `func` field as an opcode extension field.

☑ Why is an opcode extension field needed?

The opcode field is 6 bits, allowing for only 64 instructions. The `func` field enables a greater number of instructions to be encoded.

☑ Why didn't they just make the opcode longer when designing MIPS?

In some formats, including Format I, they wanted to have as much space as possible for an immediate. If the opcode field were made larger the immediate would be smaller.

42 Spring 2024 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

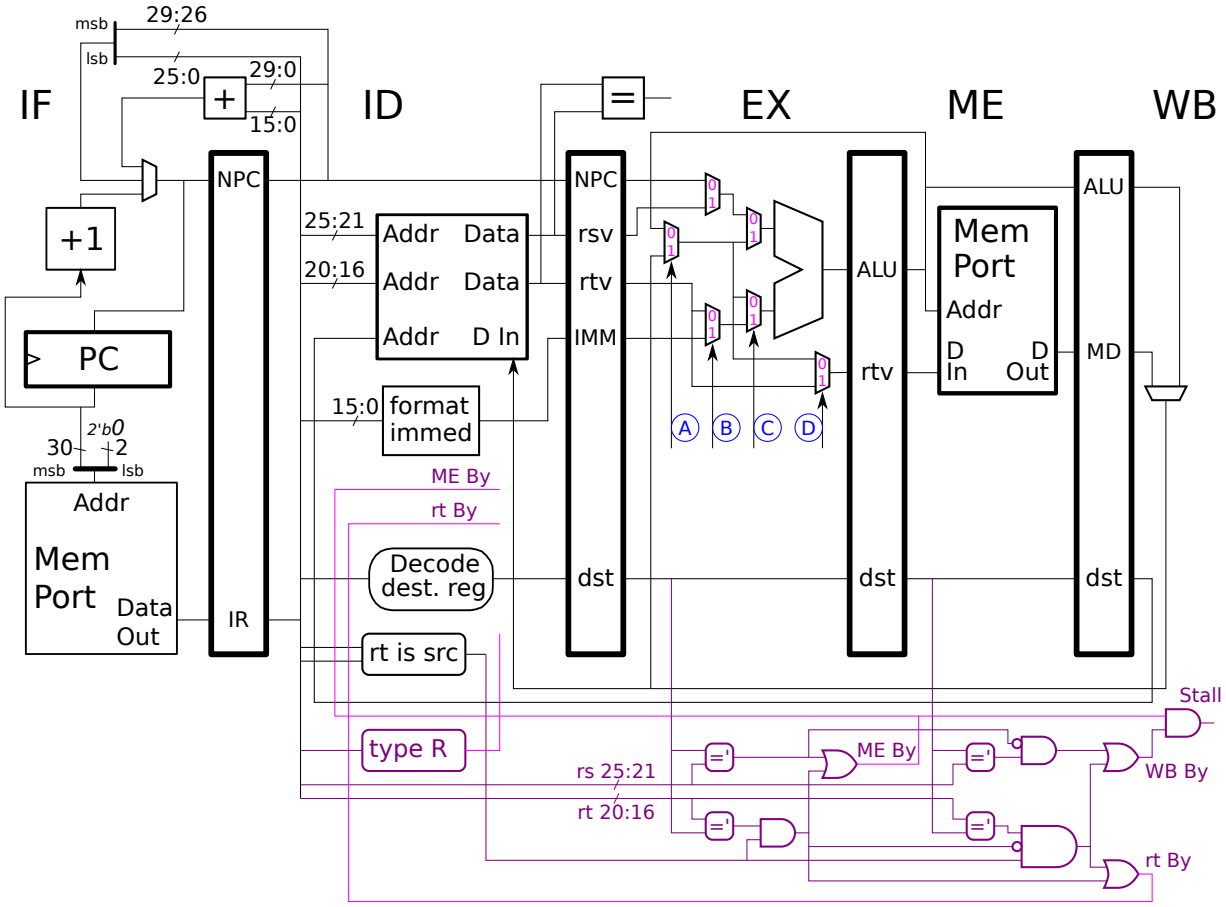
Wednesday, 3 April 2024 9:30-10:20 CDT

Problem 1		(18 pts)
Problem 2		(17 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(20 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias Clouds, be nice!

Good Luck!

Problem 1: [18 pts] Appearing below is a **changed** version of the MIPS implementation appearing in Homework 3 and the 2020 midterm exam.



☑ In the table show the select signal values expected for the execution shown below. ☑ Use `X` for select signals that don't matter (that can be either `0` or `1`). ☑ **Don't forget** to check for dependencies

```
# Cycle          0     1     2     3     4     5     6     7
add r1, r2, r3   IF    ID    EX    ME    WB
ori r7, r1, 9          IF    ID    EX    ME    WB
sub r8, r9, r7               IF    ID    EX    ME    WB
sw r7, 5(r6)                       IF    ID    EX    ME    WB
# Cycle          0     1     2     3     4     5     6     7
# SOLUTION
# Cycle          0     1     2     3     4     5     6     7
A                            X     0     0     1
B                            0     1     X     1
C                            1     1     0     1
D                            X     X     X     0
# Cycle          0     1     2     3     4     5     6     7
```

Problem 2: [17 pts] Appearing below is the implementation from the previous problem. It **is not identical** to the Homework 3 implementation. *See the last page of this exam for the Homework 3 Problem 3 solution.*

☑ Design the control logic for the A, B, C, and D select signals.

☑ Take advantage of existing logic, not much more logic is needed. ☑ Make sure that C works for the code fragment in the previous part. ☑ Don't forget that execution is pipelined.

Solution appears below in green (in the `ID` stage).

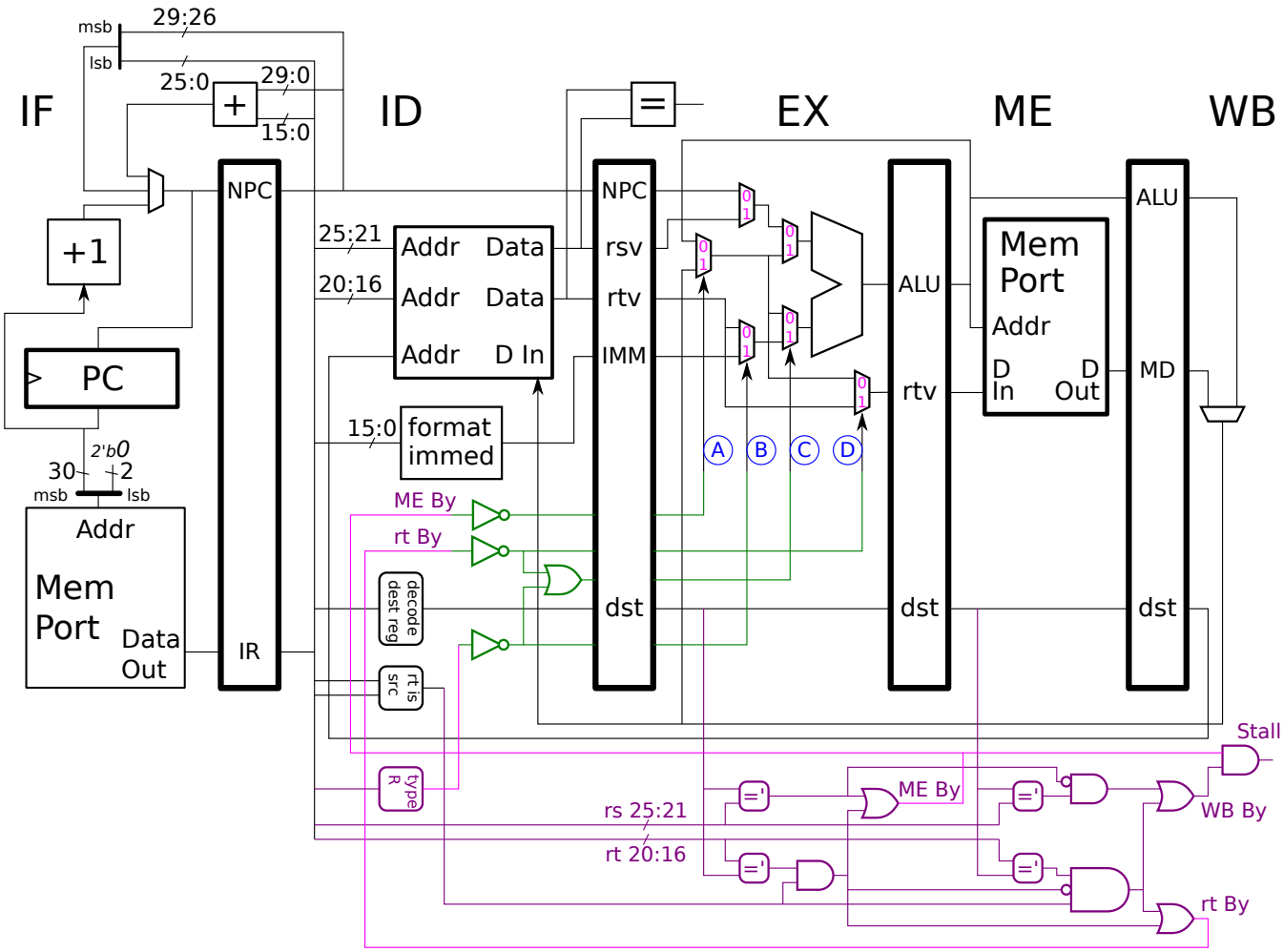
Select signal A uses the existing `ME By` signal, though inverted since a select signal of `1` connects the ME-stage bypass connection to the output. (In the Homework 3 solution the multiplexor inputs were numbered differently. In this exam all multiplexor inputs were numbered in the default way [0 at the top] to avoid confusion.)

Select signal B uses the existing `type R` signal, also inverted because a `1` selects an immediate, something not used in any type-R instruction.

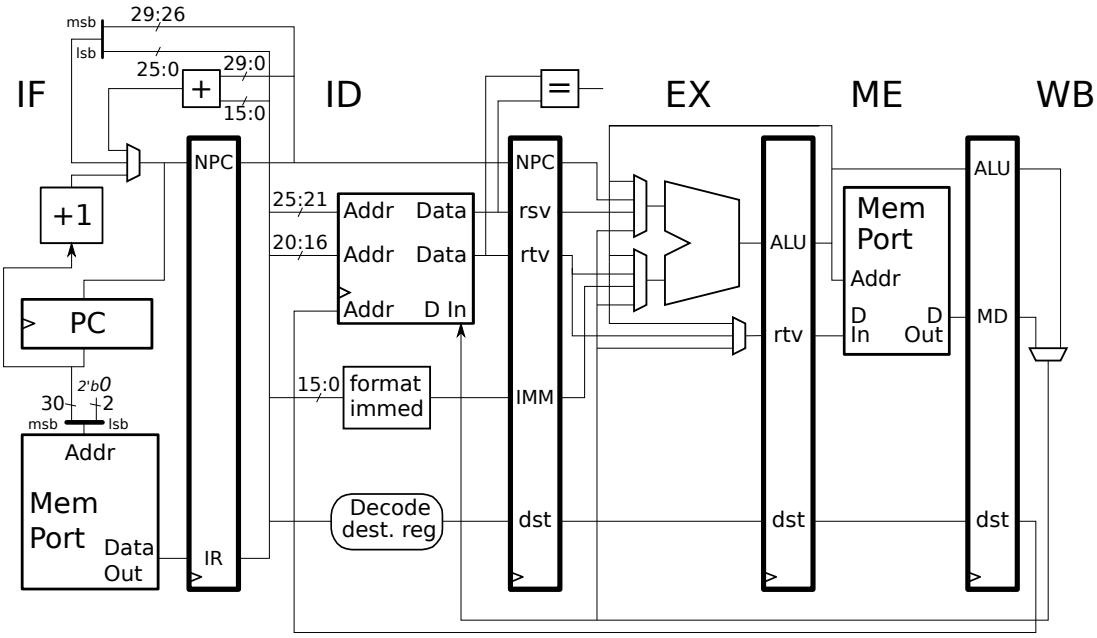
Select signal D uses the existing `rt By` signal, inverted too. Note that there is no need to check whether a store instruction is present, because if one weren't it would not matter what D was.

Select signal C requires a bit more thought. It should be `1` if either the immediate is needed (not `type R`) or the unbypassed `rt` value is needed (not `rt By`). The OR gate computes that.

Note that the logic for the select signals is in the `ID` stage so that the select signals are ready at the very beginning of `EX`.



Problem 3: [15 pts] Show the execution of the MIPS code fragments on the implementation.



☑ Show the execution of the fragment below with ☑ the branch taken. ☑ Pay close attention to branch behavior.

The solution appears below. In MIPS there is a delay slot, which is why `add` executes. In the implementation above the branch target is fetched when the branch is in **EX**.

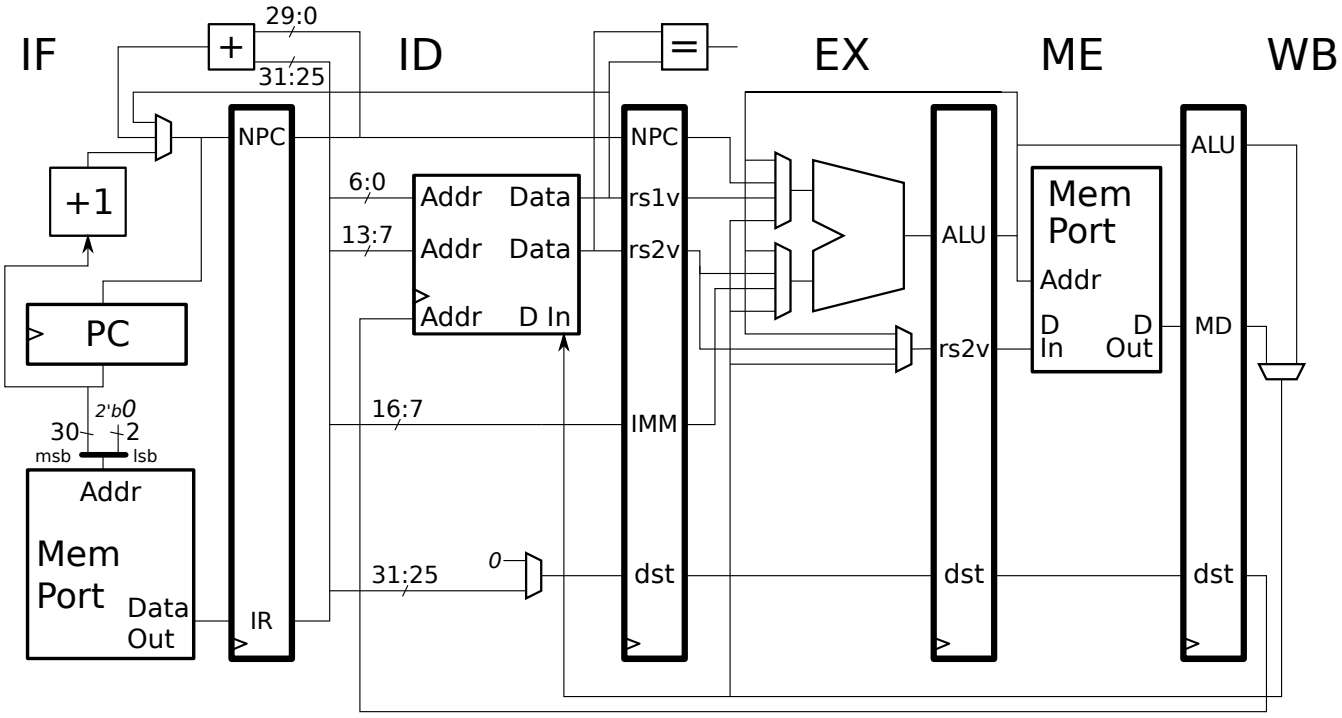
```
 # SOLUTION
 # Cycle             0  1  2  3  4  5  6
 beq r1, r1, SKIPA   IF ID EX ME WB
 add r2, r3, r4         IF ID EX ME WB
 sub r5, r6, r7
 ori r8, r9, 100
 xori r10, r11, 101
SKIPA:
 lw r12, 0(r14)            IF ID EX ME WB
 # Cycle             0  1  2  3  4  5  6
```

☑ Show the execution of the fragment below. ☑ Be sure to check for dependencies.

The solution appears below. There is no way to bypass a load value from **ME** to **EX** in cycle 4, so the `add` stalls. Bypass paths can be used for all other dependencies.

```
 # SOLUTION
 # Cycle             0  1  2  3  4  5  6  7  8
 addi R5, r5, 4      IF ID EX ME WB
 lw R2, 0(R5)           IF ID EX ME WB
 add R1, R2, r3            IF ID -> EX ME WB
 sub r4, R1, r4               IF -> ID EX ME WB
 # Cycle             0  1  2  3  4  5  6  7  8
```

Problem 4: [15 pts] Appearing below is the implementation of Another RISC ISA (ARI) and incomplete diagrams for the encoding of its MIPS-like R and I formats.



☑ How many registers does ARI have?

The address inputs to the register file read ports (in `ID`) each are connected to 7 bits ($6+1-0=7$ and $13+1-7=7$), and so there are $\boxed{2^7 = 128 \text{ registers}}$.

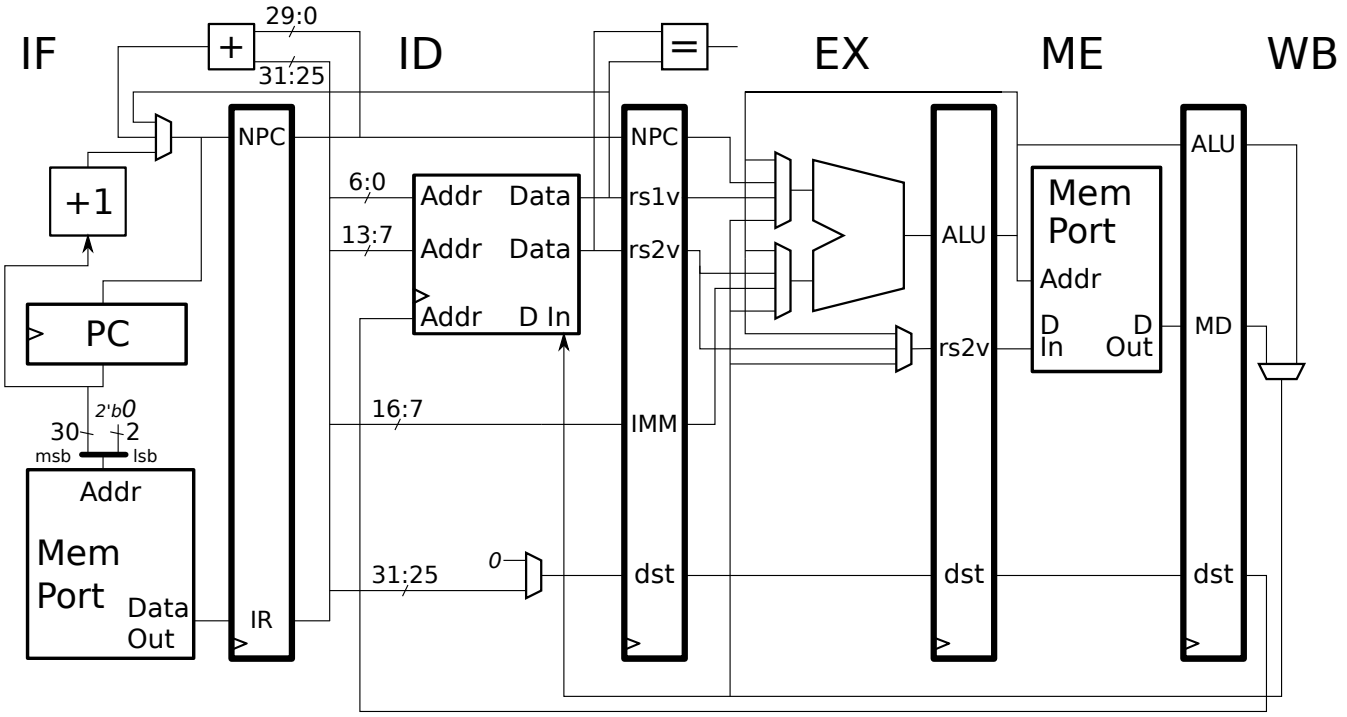
☑ What is ARI's immediate size?

The immediate size is $\boxed{16+1-7=10 \text{ bits}}$.

☑ Why is it possible to implement an instruction like `lw r1, 4(r2)` but not an instruction like `sw r1, 4(r2)` on the implementation above?

Answer: Because the instruction bits for the immediate and `rs2` overlap.

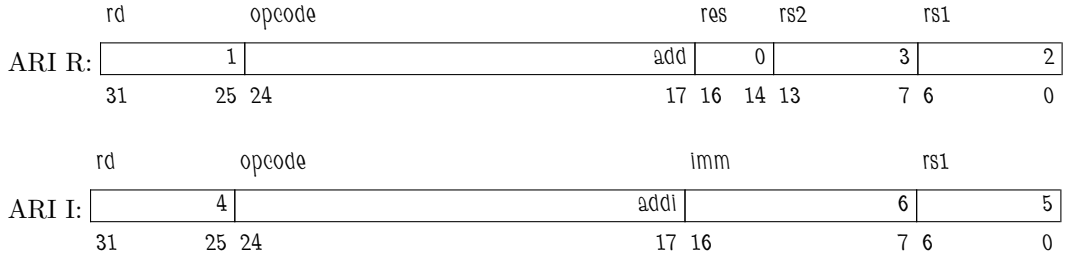
Explanation: Both the `lw` and `sw` use the immediate (along with the `rs1` value) to compute the memory address. The `sw` would need to store a register value, `r1` in the example, to memory. The register to store can't be encoded in the `rs1` field because the `rs1` field is needed for the base address (`r2` in the example). Furthermore the register to store can't be encoded in the `rs2` filed because that overlaps the immediate field. So there is no way to encode a `sw` that includes an offset (the `4` in the example). There's no problem with the `lw` since `r1` is a destination, and those bits don't overlap the immediate.



☑ In the spaces below complete ☑ ARI R and ☑ ARI I instruction formats consistent with the implementation. ☑ Be sure to show the opcode field and any opcode extensions that are needed.

The solution appears below. The opcode is chosen by using available **overlapping** bits in both formats after the bit positions for `rs1`, `rs2`, `rd`, and the immediate are claimed. That leaves bits 24:17, which is big enough for 256 instructions. The three leftover bits in the type R format are labeled `res`, for reserved. They might be used as an opcode extension field.

How realistic is ARI? Not very. Not many ISAs have 128 general-purpose registers. (Itanium is an exception.) Also, the 10-bit immediate is on the small size, it could have been made a few bits larger without shrinking the opcode field too much.



Problem 5: [20 pts] Answer each question below.

(*a*) Show the contents of the destination register after each MIPS I instruction below executes.

```
#           Initially r1 = 0x12345678
sll r2, r1, 16
#                 ☑  r2 = 0x56780000   # SOLUTION

srl r3, r1, 16
#                 ☑  r3 = 0x1234       # SOLUTION

or  r4, r2, r3
#                 ☑  r4 = 0x56781234   # SOLUTION
```

(*b*) Given the MIPS code below, why might execution never reach the `or` instruction?

```
lw $a0, 0($t0)
jal SOME_CONVENTIONAL_STANDARD_LIBRARY_FUNCTION
addi r31, r31, -8
or $s1, $s1, $v0
```

☑ The `or` instruction won't be reached because:

Each time the `jal` executes the return address (automatically written to `r31`) is changed by the `addi` from the `or` instruction address to the `jal` instruction address.

☑ What will happen instead is:

Each time `SOME_CONVENTIONAL_STANDARD_LIBRARY_FUNCTION` returns it is immediately called again, forming some kind of an infinite loop.

(*c*) Register `r9` holds the address of the middle of a large memory allocation, and so all the MIPS `lb` instructions below execute with no problem. Not so for the `lw` instructions.

```
lb r11, 0(r9)  # Will execute correctly.
lb r12, 5(r9)  # Will execute correctly.
lb r13, 10(r9) # Will execute correctly.
lb r14, 15(r9) # Will execute correctly.

lw r1, 0(r9)
lw r2, 5(r9)
lw r3, 10(r9)
lw r4, 15(r9)
```

☑ Why won't the rest of the MIPS code execute to completion?

Because memory address in the first or second `lw` will be unaligned (not a multiple of 4) and so the unlucky load will raise an unaligned access exception.

☑ What are the maximum and minimum number of `lw` instructions that will execute before an error occurs, and ☑ briefly explain how the maximum and minimum number are determined by the exact value of `r9`.

The maximum number is one, and the minimum number is zero.

If the value of `r9` is a multiple of 4, say `0x1000`, then the first `lw` will execute correctly but the second, attempting to load from `0x1005`, will raise an unaligned address exception and never finish.

If the value of `r9` is not a multiple of 4, say `0x1001`, then the first `lw` will raise the exception and execution will never even reach the second `lw`.

(*d*) Simplify MIPS the code fragment below.

```
lbu r1, 0(r10)
lbu r2, 1(r10)
sll r1, r1, 8
or r1, r1, r2
sh r1, 2(r10)
# Note: r1 and r2 not used again.
```

☑ Simplify the code fragment ☑ without changing what it does.

```
# SOLUTION
lh r1, 0(r10)
sh r1, 2(r10)
```

Problem 6: [15 pts] Answer each question below.

(*a*) In class we described three families of ISAs, CISC, VLIW, and RISC.

☑ How do VLIW ISAs differ from both RISC and CISC ISAs?

In a VLIW ISA multiple instructions are grouped into *bundles*, typically containing three instructions. Branch and jump targets are always to the first instruction in a bundle. In contrast CISC and RISC ISAs jump and branch targets can be to any instruction. Also in VLIW ISAs each slot of a bundle can have different restrictions on the kind of instruction that can be placed there. For example, loads and stores may not be allowed in slot 2, and branches might not be allowed in slots 0 and 1. Lacking bundles, RISC and CISC ISAs lack such restrictions.

(*b*) Identify the ISA family of the following ISAs:

MIPS:	◯ *CISC*	◯ *VLIW*	⊗ *RISC*
Arm A64:	◯ *CISC*	◯ *VLIW*	⊗ *RISC*
Itanium:	◯ *CISC*	⊗ *VLIW*	◯ *RISC*
Intel 64 /IA-32:	⊗ *CISC*	◯ *VLIW*	◯ *RISC*
VAX:	⊗ *CISC*	◯ *VLIW*	◯ *RISC*

(*c*) The statement below is wrong.

CISC ISAs can have large immediate values, but at the cost of having large instructions. That is why programs in CISC ISAs are large compared to those in RISC ISAs.

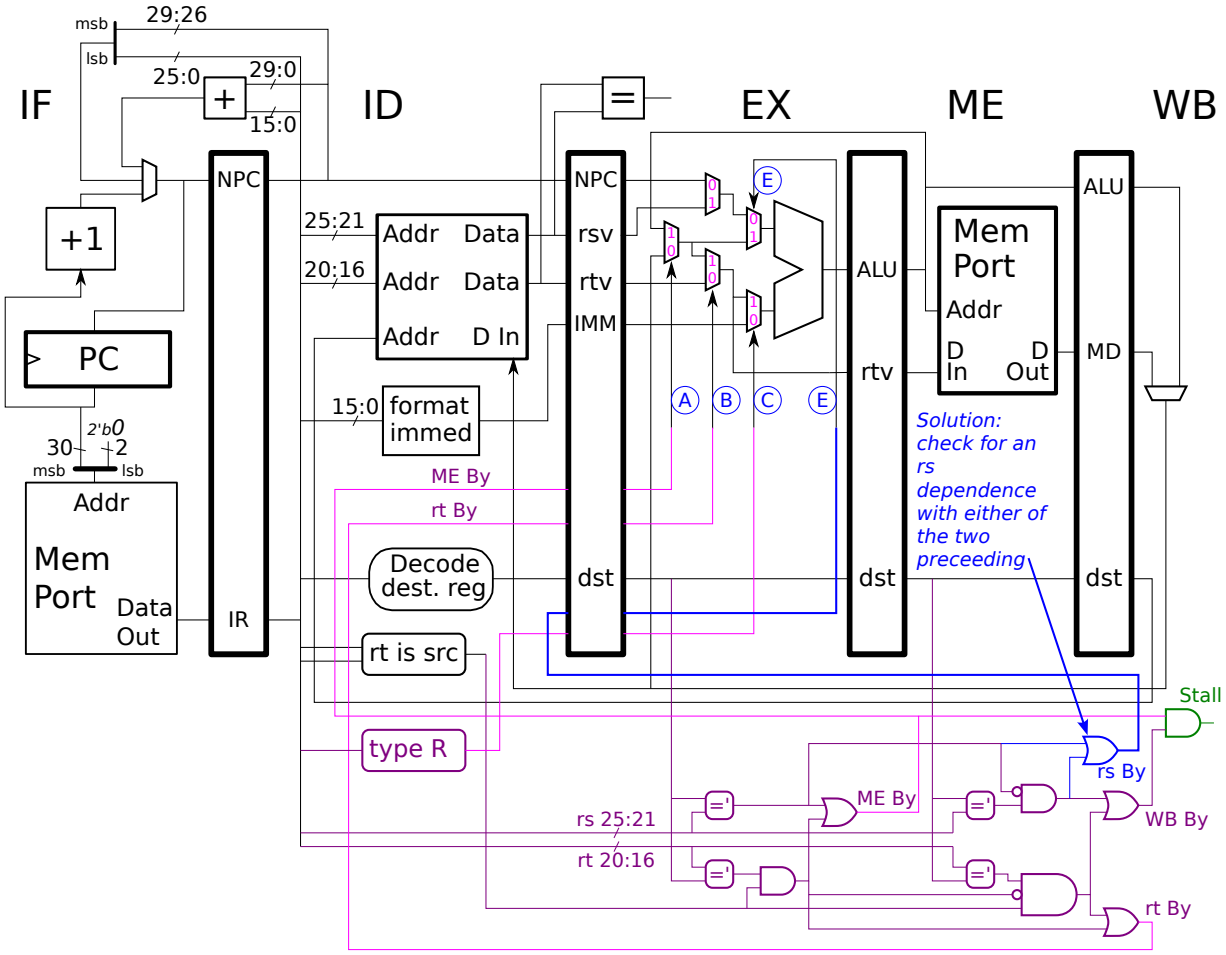
☑ What is correct in the statement above?

The first sentence is correct, but perhaps a little misleading. Misleading because CISC instruction sizes vary, and so though large-immediate instructions are of course large, other CISC instructions can be small, say one byte for a `nop`.

☑ What is wrong in the statement above?

First of all, CISC instruction sizes vary, and so some instructions are smaller than typical RISC instructions. Also, since CISC instructions are more powerful, a single CISC instruction does the same work as multiple RISC instructions, so though that CISC instruction is larger than one RISC instruction, it is smaller than the total size of the RISC instructions it replaces.

Appearing below is part of the solution to Homework 3 Problem 3. It may be helpful in solving Problem 2 in this exam.



Name Solution

Formatted For 2-Sided Printing

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Thursday, 9 May 2024 17:30-19:30 CDT

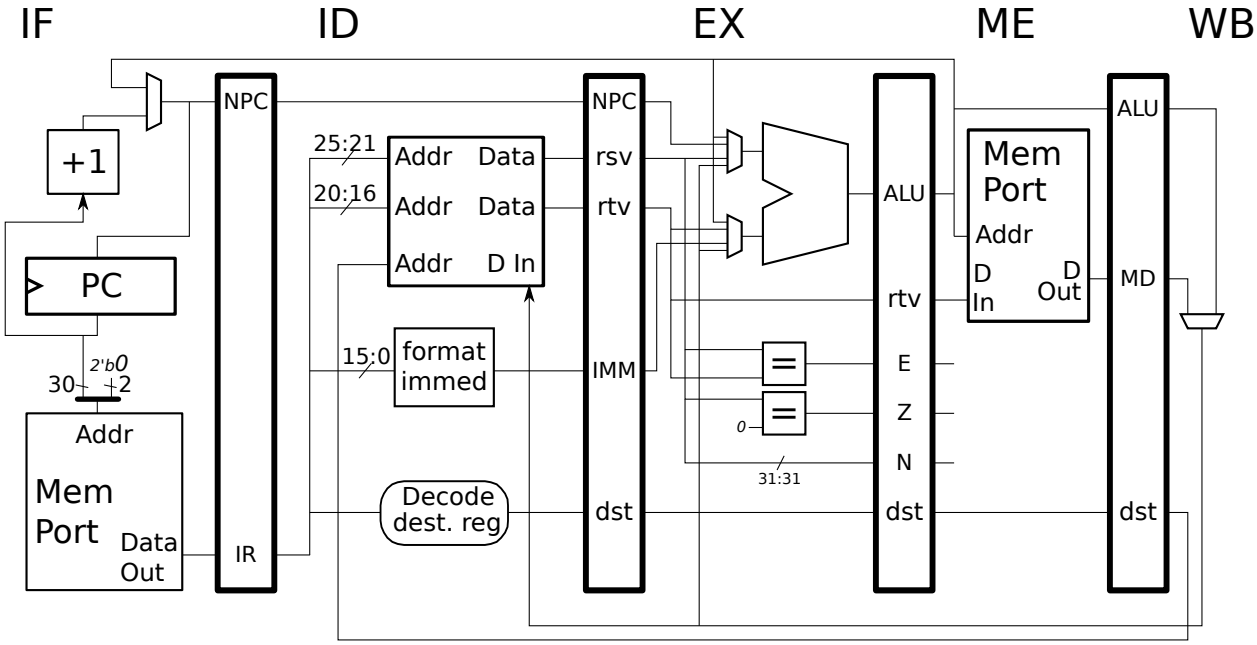
Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(25 pts)
Problem 4		(12 pts)
Problem 5		(8 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Naturally Solved

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

Problem 1: (20 pts) Appearing below are MIPS implementations and code fragments. Show execution (a pipeline execution diagram) of the code on the accompanying implementations.

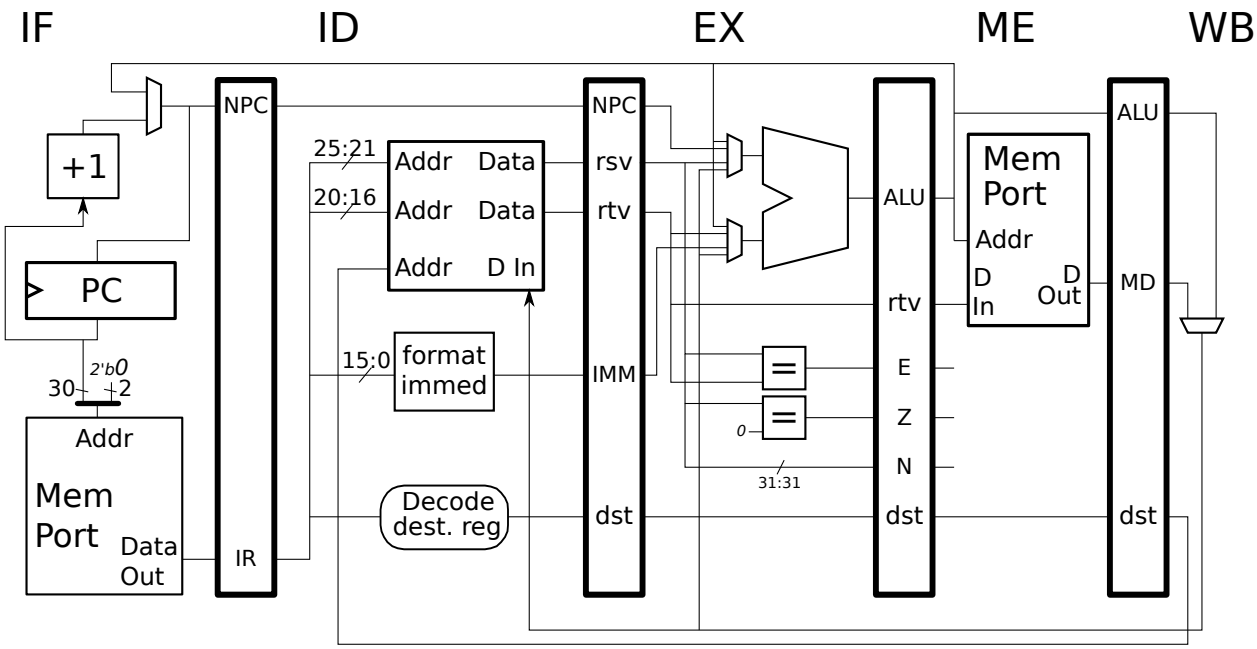


☑ Show execution of the code below on the implementation above. ☑ Check for dependencies.

The solution appears below. The implementation above can bypass to the ALU, which is why `lw` and `sub` do not stall. But, it can't bypass to the input of the ME-stage memory port, which is why the `sw` stalls.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11
add r6, r7, r8   IF ID EX ME WB
lw r9, 0(r6)        IF ID EX ME WB
add r1, r2, r9         IF ID -> EX ME WB
sub r4, r1, r5            IF -> ID EX ME WB
sw r4, 0(r5)                    IF ID ----> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11
```


Staple This Side

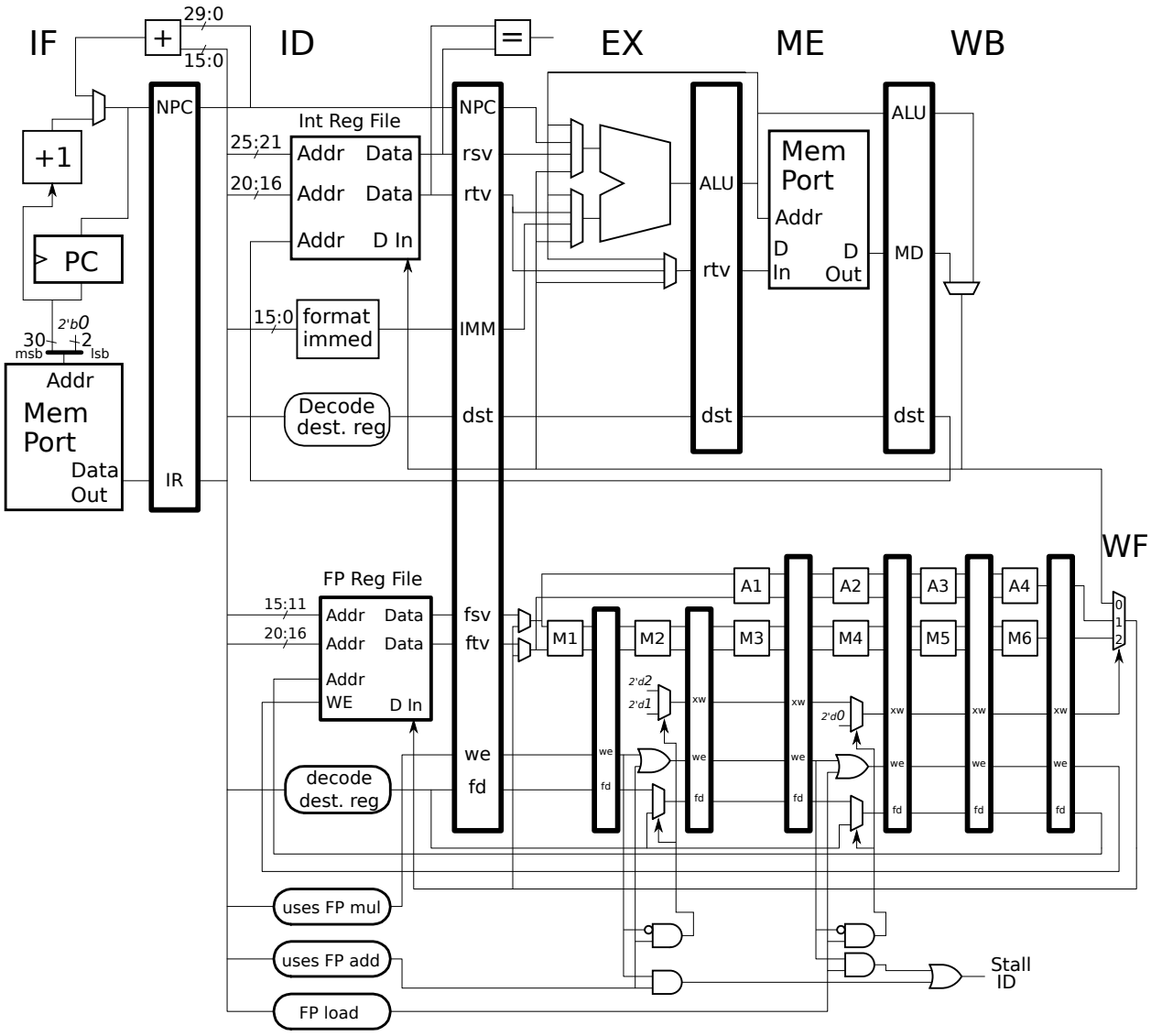


☑ Show execution of the code below on the implementation above ☑ until the second fetch of `lw`. ☑ Show when and where instructions are squashed (with an `x`). ☑ The branch must be taken. ☑ Pay close attention to branch behavior. ☑ Check for dependencies.

The solution appears below. In the implementation above the branch resolves in ME, so the branch target is fetched when the branch is in WB, which is cycle 6 in the execution below.

In MIPS the delay-slot instruction (`addi r1` below) executes whether or not the branch is taken. Because the branch resolves in ME two instructions had to be squashed.

```
 # SOLUTION
LOOP: # Cycle      0  1  2  3  4  5  6  7
 lw r3, 0(r4)      IF ID EX ME WB    IF
 addi r4, r4, 4       IF ID EX ME WB
 bne r1, r2, LOOP        IF ID EX ME WB
 addi r1, r1, r3            IF ID EX ME WB
 sw r3, 0(r9)                  IF IDx
 sw r4, 4(r9)                     IFx
 sw r2, 8(r9)
#       Cycle      0  1  2  3  4  5  6  7
```



☑ Complete the execution of the code below for the implementation above. ☑ Check for dependencies, ☑ don't overlook the first two `mul.s` instructions.

The solution appears below. The `add.s f8` has to stall two cycles to avoid the structural hazard at `WF`. Note that the stall occurs in `ID`, not in `A4`. The `add.s f7` stalls because of the dependence carried by `f4`, and the `add.s f10` stalls due to the dependence carried by `F11`.

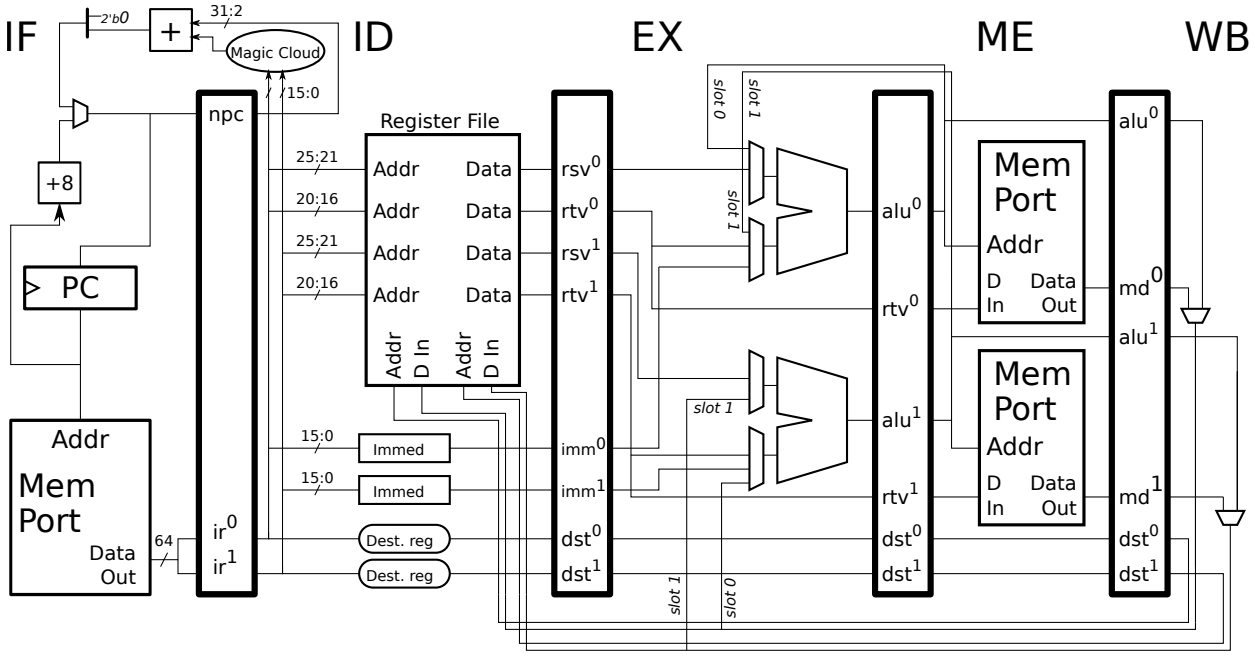
```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12
mul.s f1, f2, f3   IF ID M1 M2 M3 M4 M5 M6 WF
mul.s F4, f5, f6      IF ID M1 M2 M3 M4 M5 M6 WF
add.s f8, f15, f16       IF ID ----> A1 A2 A3 A4 WF
add.s f7, F4, f9            IF ----> ID ----> A1 A2 A3 A4 WF
lwc1 F11, 0(r1)                      IF ----> ID EX ME WF
add.s f10, F11, f12                           IF ID -> A1 A2 A3 A4 WF
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```


Staple This Side

This page left mostly blank to provide room for the pipeline execution diagram.

Problem 2: (15 pts) Answer the following questions on superscalar MIPS implementations.

(*a*) The superscalar MIPS implementation below has only four bypass paths, one per ALU multiplexor. The paths have labels, slot 0 and slot 1 (showing where they originate).



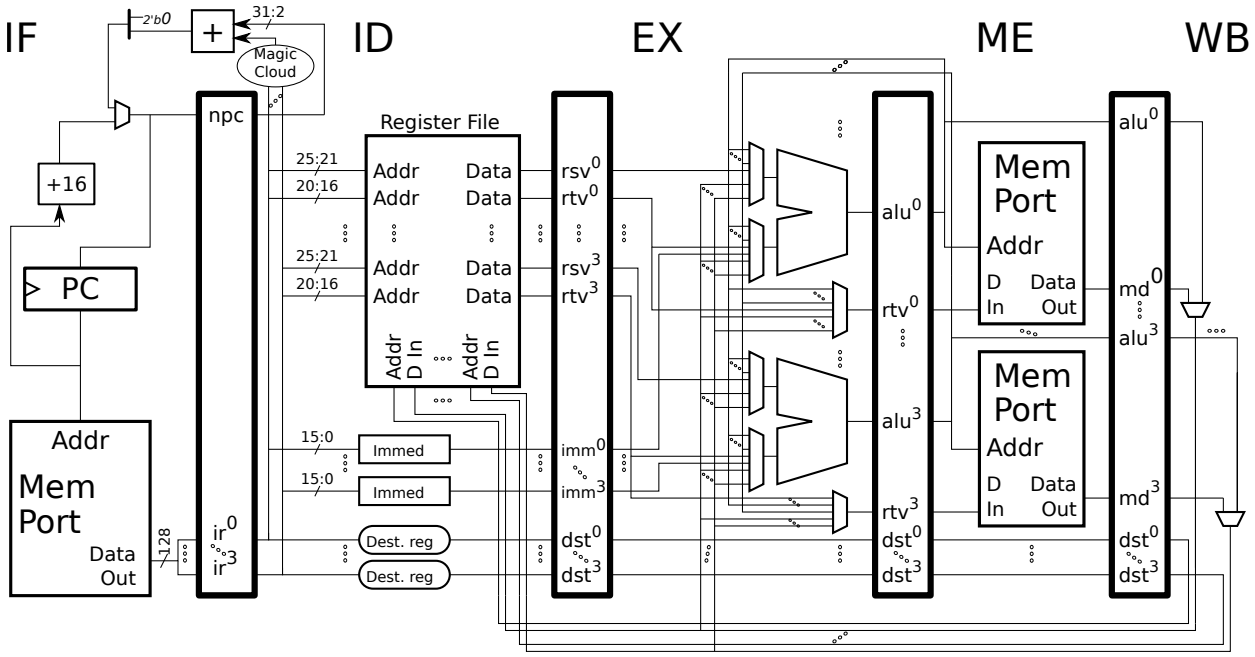
☑ Complete the code fragment below (by adding registers) so that it ☑ uses all four bypass paths **in cycle 4**.

The solution appears below. Since the bypasses need to be used in cycle 4, the bypasses will be to the last two instructions. The upper, slot 0, ALU has bypasses from ME. The upper ALU input (of the upper ALU) is to slot 0 in ME, that is used by R7. The lower ALU input (of the upper ALU) is to slot 1 in ME, that is used by R10. Similar reasoning is used for the operands to slt.

```
# SOLUTION
# Cycle                 0  1  2  3  4  5  6
add   R1, r2, r3        IF ID EX ME WB
sub   R4, r5, r6        IF ID EX ME WB
or    R7, r8, r9           IF ID EX ME WB
xor   R10, r11, r12        IF ID EX ME WB
and   r20, R7, R10            IF ID EX ME WB
slt   r21, R4, R1             IF ID EX ME WB
# Cycle                 0  1  2  3  4  5  6
```


Staple This Side

(*b*) Show the execution of the code below on the following 4-way superscalar MIPS implementation. As we have been doing this semester, instruction fetches are not aligned (the address can be any multiple of 4).



☑ Show execution on this ☑ **4-way** superscalar MIPS implementation, with ☑ the **branch taken**. ☑ Show all squashes with an `x`. ☑ Check for dependencies and ☑ pay attention to branch behavior.

The solution appears below. The branch resolves in `ID` and so the target is fetched when the branch is in `EX`, cycle 2 below. In cycle 1 `add` and `sw` are in the `ID` stage. But since the branch is taken they need to be squashed along with the four instructions in `IF`. (The reason why `sw` is squashed and refetched is discussed below.) Later, `lw` stalls for the dependence with `addi` and `xori` stalls for the `lw`.

Those who are paying attention may be thinking, sure `add` needs to be squashed in cycle 1, but why `sw`, that's at the branch target. And for the same reason, in cycle 1 why squash those four instructions in `IF` since they are on the correct path? The reason is that this would make the control logic much more complicated. For example, in cycle 1 the control logic considered `sw` to be dependent on the `add` (through `r2`), after the branch resolves the dependencies are different. This would put the control logic after branch resolution which might lengthen the critical path.

```
 # SOLUTION
 # Cycle          0  1  2  3  4  5  6  7  8  9  10
 beq r2, r3 SKIP  IF ID EX ME WB
 addi R1, r1, 8   IF ID EX ME WB
 add r2, r2, r5   IF IDx
SKIP:
 sw r2, -4(R1)    IF IDxIF ID EX ME WB
 add R1, r1, r5      IFxIF ID EX ME WB
 lw R2, 0(R1)        IFxIF ID -> EX ME WB
 xori r2, R2, 0xaa   IFxIF ID -------> EX ME WB
 slt r8, R1, r9      IFx   IF -------> ID EX ME WB
 # Cycle          0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: (25 pts) When the modifications to the MIPS implementation on the facing page are complete an `add.s` instruction will not have to stall to avoid a structural hazard with preceding `mul.s` instructions. Here an `add.s` instruction can pass through the same number of stages as a `mul.s`, so there is no possibility of a stall due to a structural hazard at `WF`. In the execution below `add.s f14` avoids such structural hazard stalls by passing through the two extra stages, `a5` and `a6`. But to avoid the necessity of *always* having to pass through those two extra stages an `add.s` can use *hop* multiplexors to skip ahead to `WF` early. The `add.s f10` skips over two stages, and `add.s f7` skips over one stage.

To keep the problem description from getting too long the following interesting material was not included in the original final exam. Hopping ahead this way is probably not the best way to deal with structural hazards, even if the avoided structural hazard stalls justified the cost of the pipeline latches. The reason is that those two new hop multiplexors could instead be used to implement bypass paths from `a5` *and* `a6` *into the* `M1` *and* `A1` *functional units. (See the Fall 2006 final exam.) With such bypasses possible there would no longer be a need to write back early. Hmmm, this may turn into a Fall 2025 question.*

New and important hardware is shown in blue. The hardware generating signals `xw`, `we`, and `fd` is from the old design and needs to be modified. Hardware for `lwc1` has been removed, it is not part of this problem. When solved correctly code should execute as shown below:

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
mul.s f1, f2, f3   IF ID M1 M2 M3 M4 M5 M6 WF
mul.s f4, f5, f6      IF ID M1 M2 M3 M4 M5 M6 WF
add.s f14, f15, f16      IF ID A1 A2 A3 A4 a5 a6 WF
sub r1, r2, r3              IF ID EX ME WB
add.s f7, f8, f9               IF ID A1 A2 A3 A4 a5 WF
or r4, r5, r6                     IF ID EX ME WB
add.s f10, f11, f12                  IF ID A1 A2 A3 A4 WF
a4/a5.wa                                  1     1     1
a5/a6.wa                                     1     1     0
a6/WF.wa                                        1     0     0
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

In cycle 11 the `add.s f7` uses the lower input of Hop6 to hop from `a6` to `WF` (which is why `a6` is not shown). In cycle 12 `add.s f10` uses the upper input of Hop6 to hop from `a5` to `WF`.

☑ Connect logic to the inputs of AND gates W5 and W6 so that the `wa` (write add) signal will be `0` for an instruction that hopped out of a stage, preventing a hopping `add.s` from writing back twice. See sample `wa` values in the execution above.

☑ Add select signals to the two hop multiplexors. ☑ *Little or no logic is required. For partial credit assume* `wa` *is correct.*

☑ Modify the design so that `fd` is correct for ☑ `mul.s` and ☑ the hopping `add.s` instructions.

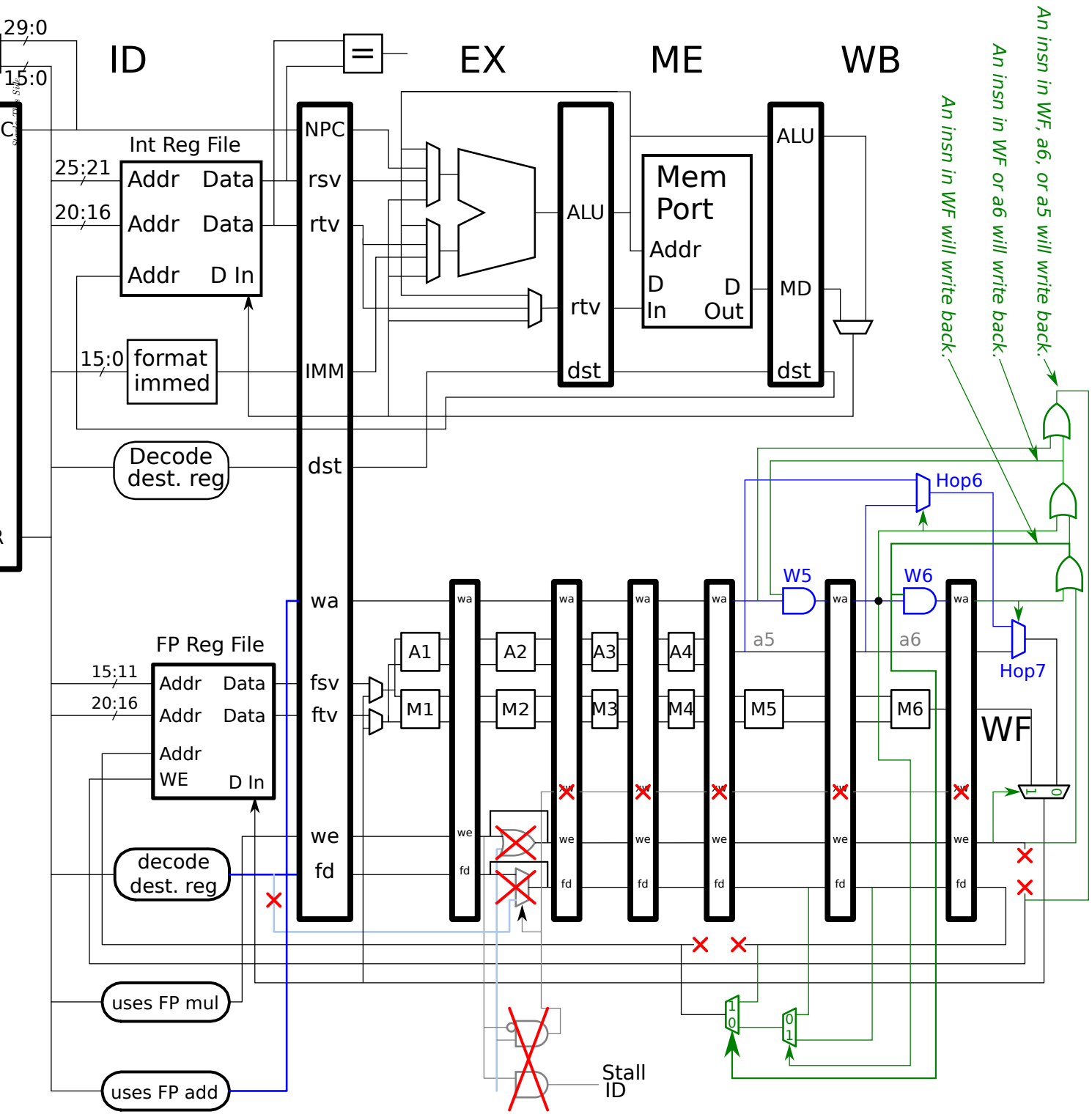
☑ Modify the `xw` logic so that it works for hopping `add.s` instructions and the `mul.s`.

☑ Modify `we` so that it is correct for hopping `add.s` and `mul.s` instructions.

☑ Remove logic that is no longer needed. That can include logic for `xw` or `we`, depending on the solution.

Solution appears in green. *Grading Note:* Many student solutions were more complicated than they had to be, even on the 2025 Homework version of this problem in which there was time to clean up a hurried first attempt. For example, the `Hop6` select input need only be set to `a6.wa`. There is no need to check whether `WF` is occupied, let alone anything in stage `a5`. Many did not notice that `xw` would not be needed if `we` was used only for multiply instructions.

Staple This Side



Because FP add instructions go through every FP stage the logic injecting `fd` and `we` in the `M2` stage has been removed. With the removal `we` really means *write multiply* not *write any FP result* as it had before. Because the `WF` mux has only two inputs, we can use `we` for the select signal.

An instruction can't hop out of `a6` if WF is occupied, that's checked for by the bottom OR gate and its output connects to the `W6` AND gate. The `a5` instruction can't hop if there's an instruction in `WF` or `a6`, that is detected by the middle OR gate and is the input to the `W5` AND gate. If there is an instruction in `a6` then either it will need the lower `Hop6` mux input, or the output of the `Hop6` mux is ignored, and so one need only use `a6.wa` for the select signal. Similar reasoning is used for the `Hop7` select signal.

Staple This Side

Problem 4: (12 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor, and the other uses a 3-outcome local history predictor.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  N  N  T  T  T  N  T    N  N  T  T  T  N  T     N  N  T  T  T  N  T   <- Outcome

B2:   N  T  N  T  N  T  N    T  N  T  N  T  N  T     N  T  N  T  N  T  N  <- Outcome
```

☑ What is the accuracy of the bimodal predictor on branch B1? ☑ Be sure to base the accuracy on a repeating pattern.

The prediction accuracy is $\frac{3}{7}$. See the work below. Note that the prediction outcomes in the second repetition of the pattern will repeat because the two-bit counter is the same, 3, at the start of the second and third repetition.

```
   0  0  0  1  2  3  2  3    2  1  2  3  3  2  3                      <- 2 bit counter
B1:  N  N  T  T  T  N  T    N  N  T  T  T  N  T     N  N  T  T  T  N  T<- Outcome
    n  n  n  n  t  t  t    t  t  n  t  t  t  t                       <- Prediction
          X  X     X       X  X  X        X                         <- Mispred
                           --------------------                      <- Repeating pattern
```

☑ What is the accuracy of 3-outcome local history predictor on B1 ignoring B2.

The accuracy is 100%. This is easy because the seven 3-consecutive-outcome patterns are unique (no two repeat). Therefore each pattern will have a separate entry in the PHT and that entry can reach the correct value. The patterns are shown in the solution to the next part.

☑ What is the accuracy of 3-outcome local history predictor on B1 **taking into account** B2. ☑ Note that the B2 pattern repeats faster than the B1 pattern.

The table below shows the patterns of B1 followed by the next outcome in both B1 and B2. Pattern TNT occurs in both branches, fortunately the next outcome, N, is the same for both branches. Pattern NTN also occurs in both branches but the next outcome for B1 is N and the next outcome for B2 is T. Uh oh! This isn't good for B1 because the B2 pattern occurs more frequently and so the PHT entry for NTN will reach 3 and so B1 will be mispredicted. Therefore the prediction accuracy for B1 will be $\frac{6}{7}$. Note that the prediction accuracy for B2 will be 100%.

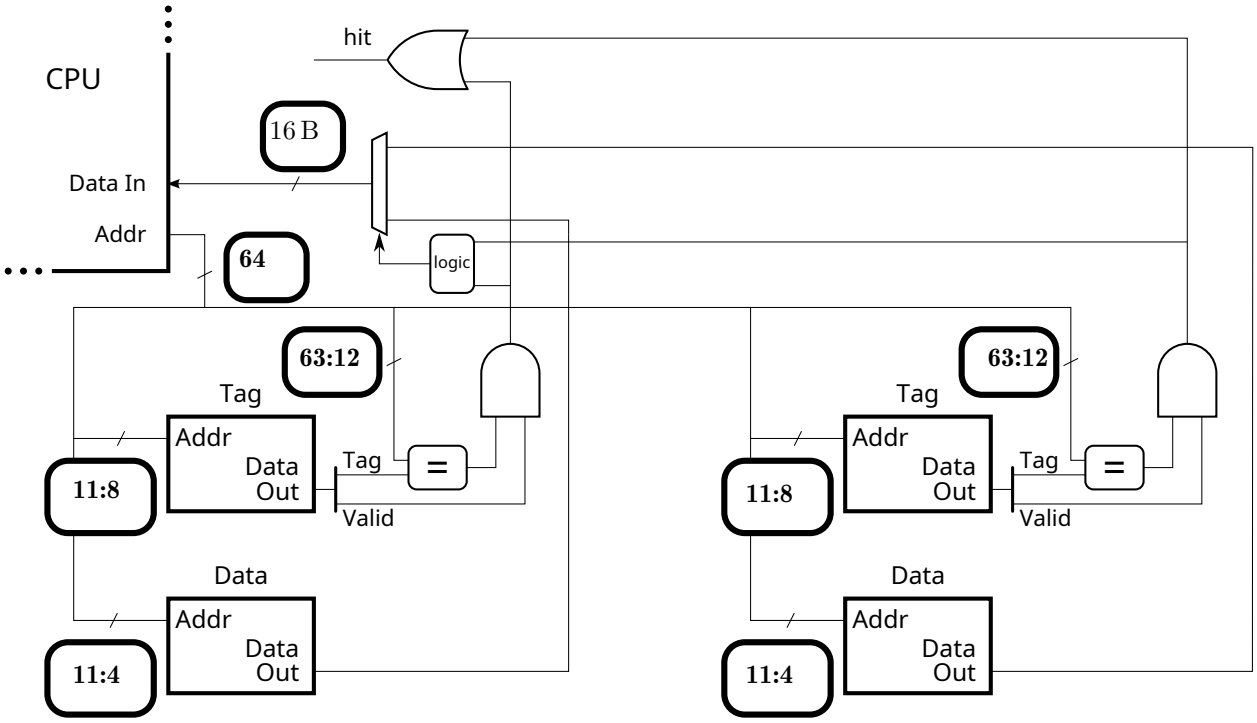
Pat	B1	B2
NNT	T	
NTT	T	
TTT	N	
TTN	T	
TNT	N	N
NTN	N	T
TNN	T	

Staple This Side

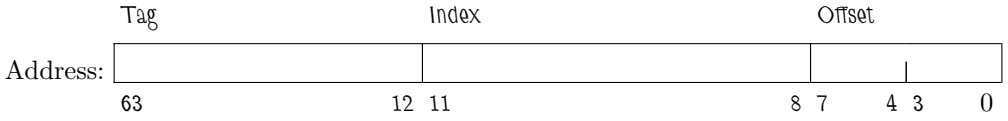
Problem 5: (8 pts) The diagram below is for a two-way set associative cache.
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.

Solution appears below.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)



☑ Cache Capacity, in Bytes (how much data can it cache).

The cache capacity can be determined from the following pieces of information: The lowest tag bit position is 12, which means that the size of the combined index and offset is 12 bits, and so each data store holds 2^{12} characters. Based on the diagram the character size is one byte. (If it weren't there would be a big hint that this is an unusual system.) The diagram shows two ways (for example, the mux feeding Data In has two inputs). So the total $\boxed{\text{cache capacity is } 2 \times 2^{12}\,\text{B} = 8192\,\text{B}}$. Note: a cache size of $8192\,\text{B}$ is considered small, even for a level-one cache.

☑ Line Size ☑ Indicate Unit!!:

Lower bit position of the address going into the tag store gives the line size, $2^8 = 256$ characters.

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 bytes (characters). The code fragment starts with the cache cold (empty); consider only accesses to the array. Of course, $2^6 = 64$.

(*b*) Find the hit ratio executing the code below.

```
int64_t sum = 0;
int64_t *a = 0x2000000; // sizeof(int64_t) == 8
int ILIMIT = 1 << 14;    // = 2^14

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $64 = 2^6$ bytes is given. The size of an array element, which is of type `int64_t` is $8 = 2^3$ B, and so there are $2^6/2^3 = 2^{6-3} = 2^3 = 8$ elements per line. The first access, at `i=0`, will miss but bring in a line with 8 elements, and so the next $8 - 1 = 7$ accesses will be to data on the line, hits. The access at `i=8` will miss and the process will repeat. Therefore the hit ratio is $\frac{7}{8}$.

Staple This Side

Problem 6: (20 pts) Answer each question below.

(*a*) For each item below provide a code fragment exhibiting the indicated dependency. For your solving convenience one instruction is already shown.

☑ Complete the code fragment so that it exhibits a true dependence. ☑ Circle the register carrying the dependence.

```
#    SOLUTION      Register carrying dependence is bold and upper-case.
add  R1,  r2,  r3
sub  r4,  R1, r5
```

☑ Complete the code fragment so that it exhibits an output dependence. ☑ Circle the register carrying the dependence.

```
#    SOLUTION      Register carrying dependence is bold and upper-case.
add  R1,  r2,  r3  # Note: This add instruction does nothing useful.
sub  R1,  r5,  r6
```

☑ Complete the code fragment so that it exhibits an anti-dependence. ☑ Circle the register carrying the dependence.

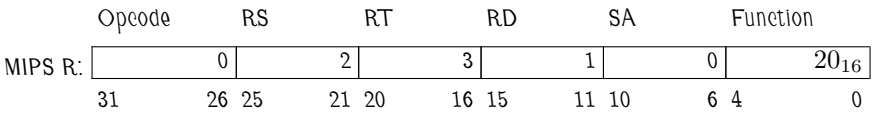
```
#    SOLUTION      Register carrying dependence is bold and upper-case.
add  r1,  R2,  r3
sub  R2,  r4,  r5
```

(*b*) Show the encoding of the instructions below. For the FP instruction infer the encoding from the implementation diagrams in other problems.

☑ Show encoding. ☑ Label fields, and show specific values where possible.

```
add r1, r2, r3
```

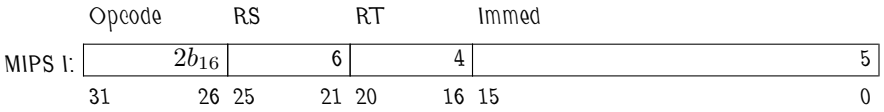
Solution appears below. Those who might have drawn a blank on the bit positions could look at one of the MIPS implementations. To receive full credit all fields except Function would need to have correct values. In the Function field it would be okay to write "add" or to note that the numeric value was just a wild guess.



☑ Show encoding. ☑ Label fields, and show specific values where possible.

```
sw r4, 5(r6)
```

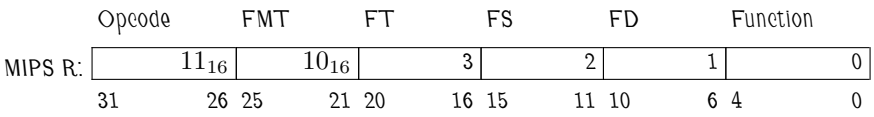
Solution appears below. To receive full credit all fields except Opcode would need to have correct values. In the Opcode field it would be okay to write "sw or to note that the numeric value was just a wild guess.



☑ Show encoding. ☑ Label fields, and show specific values where possible.

```
add.s f1, f2, f3
```

Solution appears below. Those who have no idea how FP instructions are encoded could have looked at one of the FP instructions for help. For full credit students would have to correctly show the location of the Opcode, FS, and FT fields. The values in the FS, FT, and FD fields would need to be correct (even if FD were in the wrong place). Note: The FMT field indicates whether the operands are single- or double-precision, the field values are documented in Section 5.10 of Volume I of the MIPS Architecture for Programmers. The Opcode field value of 11_{16} is written as COP1 (co-processor 1). Many FP instructions use this value.



(*c*) Based on the execution below, why can't the exception raised by the `mul.s` instruction be precise? What can't the handler do after it returns that could be done if the exception were precise?

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10
mul.s f4, f5, f6  IF ID M1 M2 M3 M4 M5*M6*WF
addi r6, r6, 1       IF ID EX ME WB

Handler:
sw r1, 4(fp)                          IF ID ..
```

☑ `mul.s` exception can't be precise because:

... because the `addi` instruction wrote its register value in cycle 5. For an exception to be precise the handler must start with registers and memory in the state they would be in if execution were correct up to just before the faulting instructions, `mul.s` in this case, and without the faulting instruction or any instructions after it modifying registers or memory (architectural state). This requirement is violated by the `addi` writing `r6`.

☑ If the exception were precise the handler could:

... the handler could perhaps put some useful value in `f4` and then resume execution at the `addi` instruction. If it were to do so in the execution above `r6` would have been incremented twice. (We know the `addi` executed before the handler started, but it would be tricky for the handler to know that for sure since it would depend on several details of the architecture, such as the number of stages used by the faulting instruction. Also, even if the hardware knew exactly which instructions past the faulting instruction executed, it would not be possible in general to restore registers and memory to their original values, at least without help from the hardware. The easiest way for the hardware to help is to just provide precise exceptions.)

(*d*) Many early RISC ISAs avoided branch instructions that compared registers, such as `blt r1, r2, TARG`, (branch less than) because that would lower the clock frequency. Later ISAs have included them. Assume that the time needed to compute `r1 < r2` has not changed.

☑ Explain why newer ISAs can have instructions like `blt r1, r2, TARG` without slowing the clock frequency, given that comparison is no faster? ☑ In your answer indicate where branches might resolve and how penalty is avoided.

Full-credit answer: Since branches are predicted they can be resolved later, in `EX`, and so register read and magnitude comparison don't have to be squeezed into the same cycle.

Explanation: If there were no prediction one would want to reduce the number of squashed instructions when a branch is taken by resolving early, such as in `ID`. To resolve in `ID` the hardware would need to read registers and do the comparison in the same cycle. With prediction there is no reason to resolve branches early. A magnitude comparison branch, like `blt`, can resolve later, in `EX` where the hardware would only need to do the comparison since the register values would be available at the beginning of the cycle.

(*e*) A design team is trying to decide between including bypass path A or bypass path B. With the target workload compiled without optimization the implementation with path A is faster. With the workload compiled with optimization the implementation with path B is faster.

☑ Which should be used ◯ *bypass path A* or ⊗ *bypass path B*. ☑ Explain.

Unless one is debugging, code is compiled with optimization on since such code runs faster and uses less energy (with typical optimization goals).

Staple This Side

43 Spring 2023 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

Wednesday, 29 March 2023 9:30-10:20 CDT

Problem 1		(17 pts)
Problem 2		(20 pts)
Problem 3		(16 pts)
Problem 4		(16 pts)
Problem 5		(16 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias With $\ll 10^{12}$ tokens.

Good Luck!

Problem 1: [17 pts] Candidate MIPS instruction `subir r1, 22, r3` is to compute `r1=22-r3`, which can't be done with a single existing MIPS instruction. The `22` is taken from instruction bits 15:0, which is the immediate field of Type-I instructions.

The `subir` instruction is to be encoded so that it can be executed by the implementation to the right **with the ALU computing** $X = A - B$, the same operation used by existing subtract instructions. Notice that in the implementation the **immediate connects to both** ALU inputs.

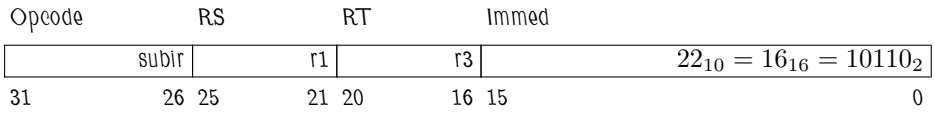
(*a*) Show how `subir r1, 22, r3` instruction would be encoded for this hardware.

☑ Show encoding of `subir r1, 22, r3`. Be sure to show ☑ the position of the fields and ☑ the field values for the sample instruction.

☑ Be sure that the encoding fits with the illustrated hardware and other MIPS instructions.

The solution appears below. The location of the immediate was given in the problem, bits 15:0. The opcode of every instruction in an ISA must be in the same place, for MIPS that is bits 31:26. The instruction has two register operands, a source, `r3` in the example, and a destination, `r1` in the example. Since the ALU will be computing $A - B$ the value of the source register, `r3`, must be delivered to the B ALU input. Only the `rt` value can reach the B input and so the source register must be encoded in the `rt` field. If the source is put in `rt`, the destination register number must be put in the `rs` field, which no other MIPS instruction does.

This encoding is shown below, with the destination, `r1` in the `rs` field and the source, `r3`, in the `rt` field. The immediate is shown in three different radices, full credit would have been given for any one of them, even decimal.



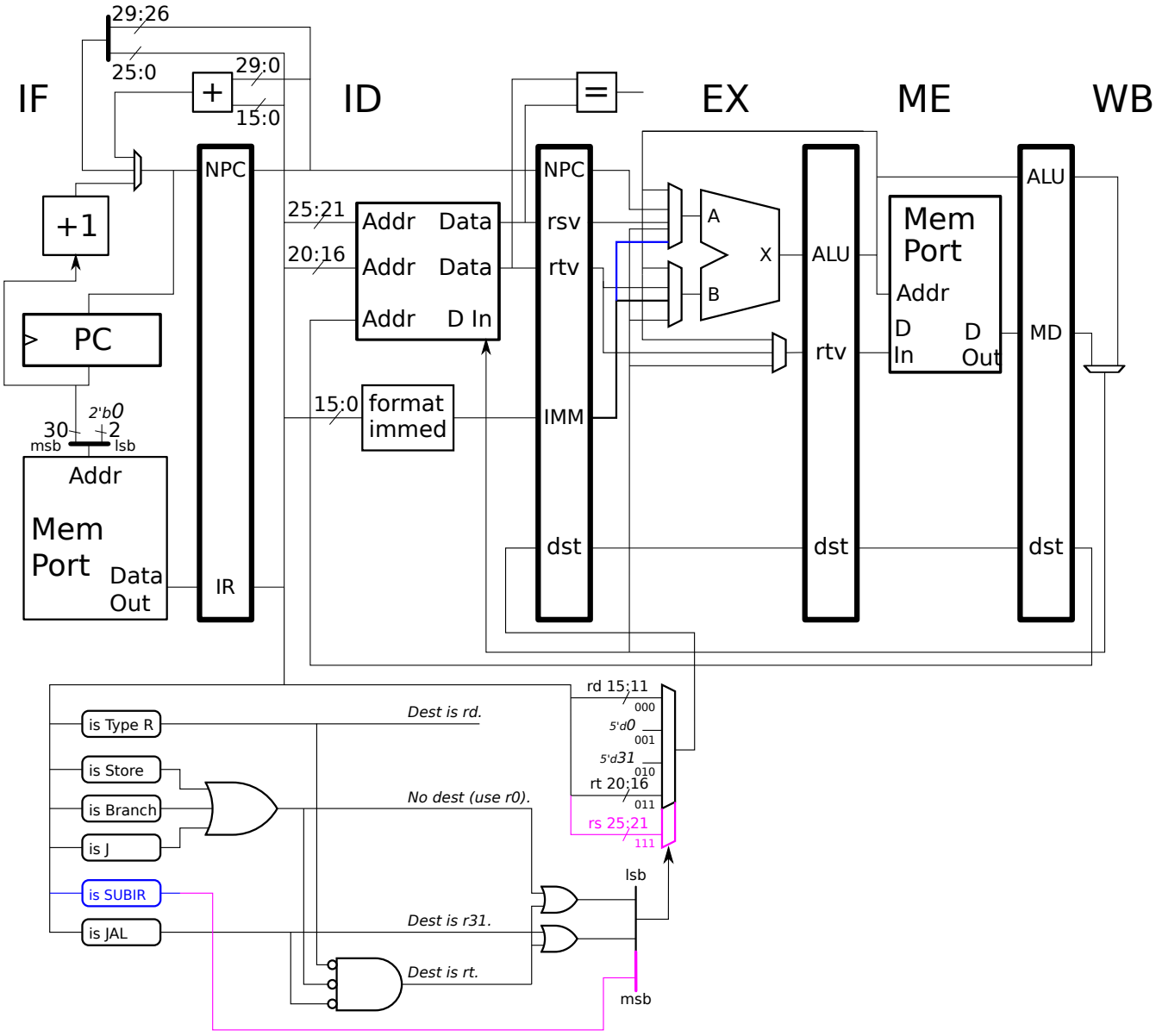
(*b*) Some control logic is shown for the implementation.

☑ Modify the control logic that computes `dst` so that `subir` executes correctly. ☑ **Do not** design control logic for the ALU multiplexors.

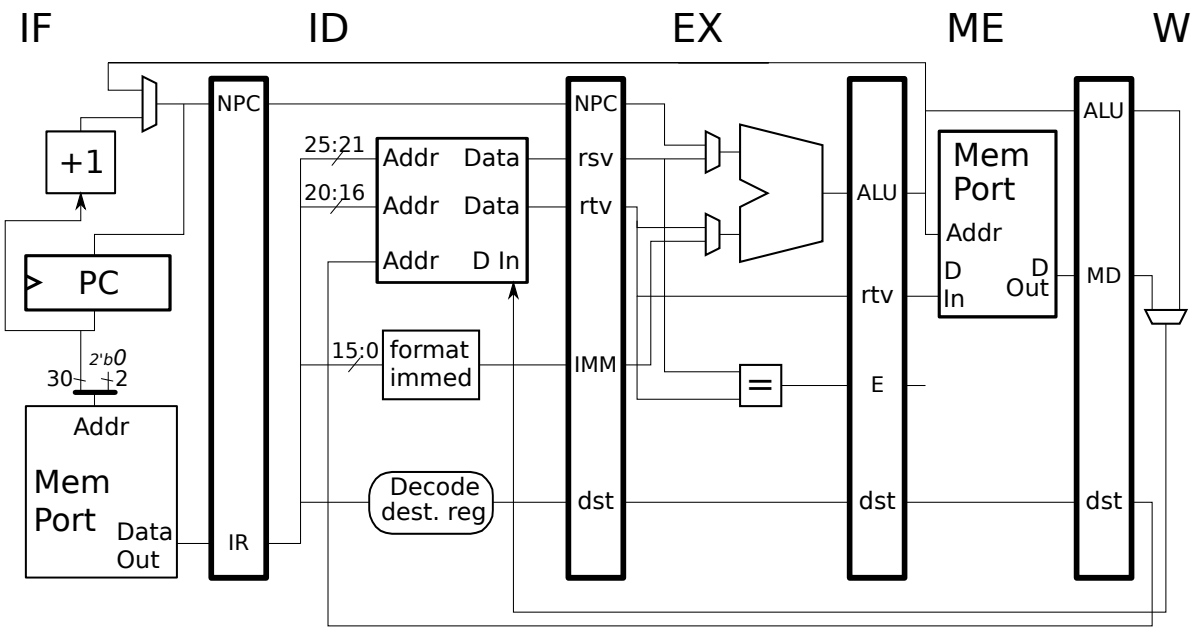
☑ The control logic should not break existing instructions.

☑ The control logic changes should be consistent with your answer to the previous part.

As described above, the destination register for `subir` must be placed in the `rs` field, something that no other MIPS instruction does. So, the logic that computes the destination register, `dst`, must be modified so that `dst` is set to the `rs` field when a `subir` is in `ID`. To do so a fifth input is connected to the mux and that input is set to the `rs` bits, 25:21. That fifth input is numbered 111_2 rather than 100_2 to simplify the logic.


IF
ID
EX
ME
WB
29:26
25:0
29:0
15:0
+
=
+1
PC
NPC
IR
Addr
Mem
Port
Data
Out
30
msb
2'b0
2
lsb
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format
immed
NPC
rsv
rtv
IMM
dst
A
B
X
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
is Type R
is Store
is Branch
is J
is SUBIR
is JAL
Dest is rd.
No dest (use r0).
Dest is r31.
Dest is rt.
rd 15:11
000
5'd0
001
5'd31
010
rt 20:16
011
rs 25:21
111
lsb
msb

Problem 2: [20 pts] Show the execution of the code fragments below on their accompanying MIPS implementations.



☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

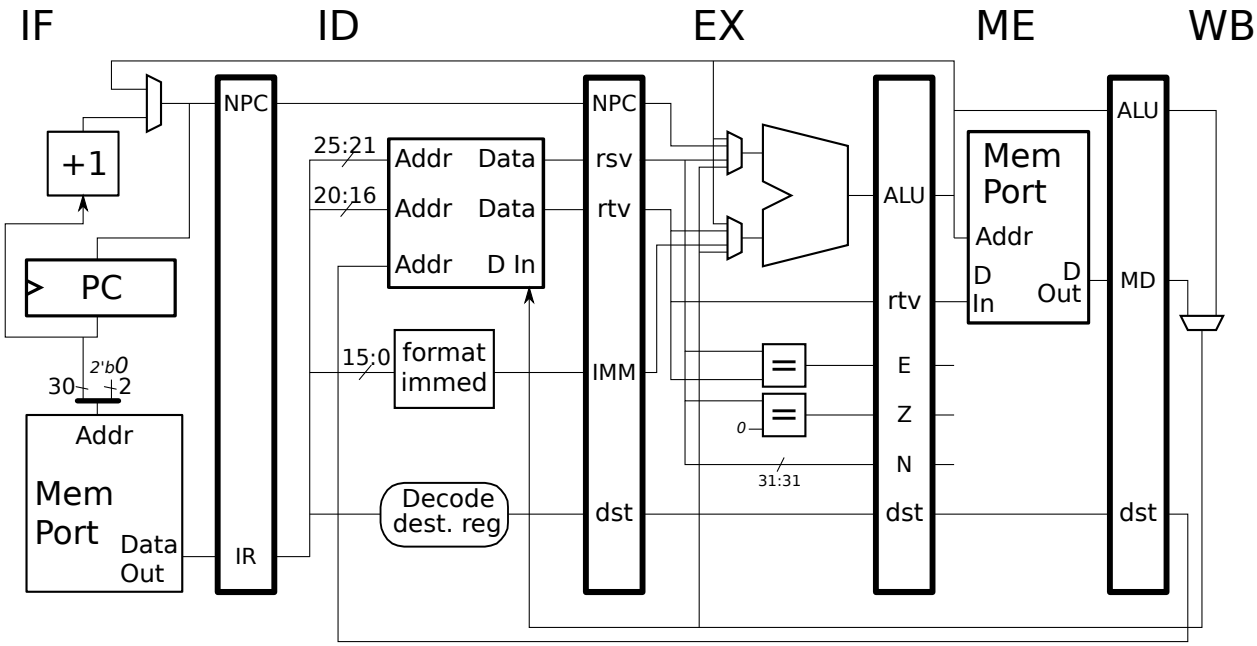
```
# SOLUTION
# Cycle         0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4  IF ID EX ME WB
lw   R3, 0(R1)     IF ID ----> EX ME WB
sw   r1, 4(R3)        IF ----> ID ----> EX ME WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10
```

☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

```
# SOLUTION
# Cycle         0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4  IF ID EX ME WB
sw   R1, 4(r3)     IF ID ----> EX ME WB
lw   r3, 0(r1)        IF ----> ID EX M WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10
```

The solution appears above. A common mistake was to not realize that register `r1` in the `sw` instruction is a **source** register, **not** a destination. Therefore there is no true dependence between `sw` and `lw`.

Problem 2, continued:



☑ Show the execution of the code fragment below on ☑ **the implementation above.** ☑ Be sure to check for dependencies.

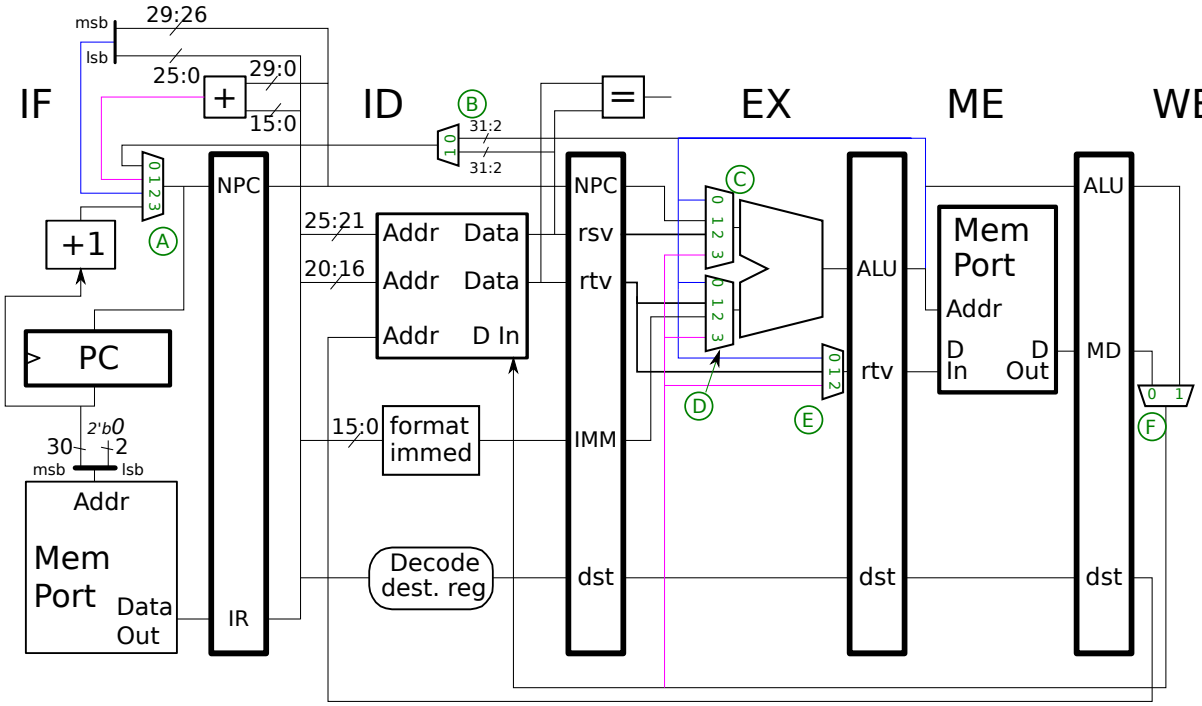
```
# SOLUTION
# Cycle         0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4  IF ID EX ME WB
lw   R3, 0(R1)     IF ID EX ME WB
sw   r1, 4(R3)        IF ID -> EX ME WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10
```

☑ Show the execution of the code fragment below on ☑ the implementation above. ☑ Be sure to check for dependencies.

The solutions appear above and below. Note that this implementation has bypass paths to the ALU but lacks bypass paths to the `EX/ME.rtv` latch, and so the `sw` below must stall until the fresh value of `r1` arrives in `ID`.

```
# SOLUTION
# Cycle         0  1  2  3  4  5  6  7  8  9  10
addi R1, r2, 4  IF ID EX ME WB
sw   R1, 4(r3)     IF ID ----> EX ME WB
lw   r3, 0(r1)        IF ----> ID EX ME WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: [16 pts] Appearing below is the MIPS implementation with labeled multiplexor select signals from Homework 3. Following that is an execution diagram along with a row showing select signal values for the D multiplexor. The first instruction, `add`, is shown.



☑ Complete the code fragment so that it produces the values shown for D.

```
#   SOLUTION
#   Cycle            0  1  2  3  4  5  6  7  8
add R1, r2, r3       IF ID EX ME WB
sub r4, r5, R1          IF ID EX ME WB
xor R6, r7, R8             IF ID EX ME WB
addi r9, r10, 11              IF ID EX ME WB
or  r11, r12, R6                 IF ID EX ME WB
#   Cycle            0  1  2  3  4  5  6  7  8
D:                          1  0  1  2  3
```

Problem 4: [16 pts] Rewrite each code fragment below so that it uses fewer instructions.

☑ Simplify code fragment.

```
addi r1, r0, 123
add  r1, r1, r2

# SOLUTION
addi r1, r2, 123
```

☑ Simplify code fragment.

```
lw r1, 0(r2)
addi r2, r2, 4
lw r2, 0(r2)

# SOLUTION
lw r1, 0(r2)
lw r2, 4(r2)
```

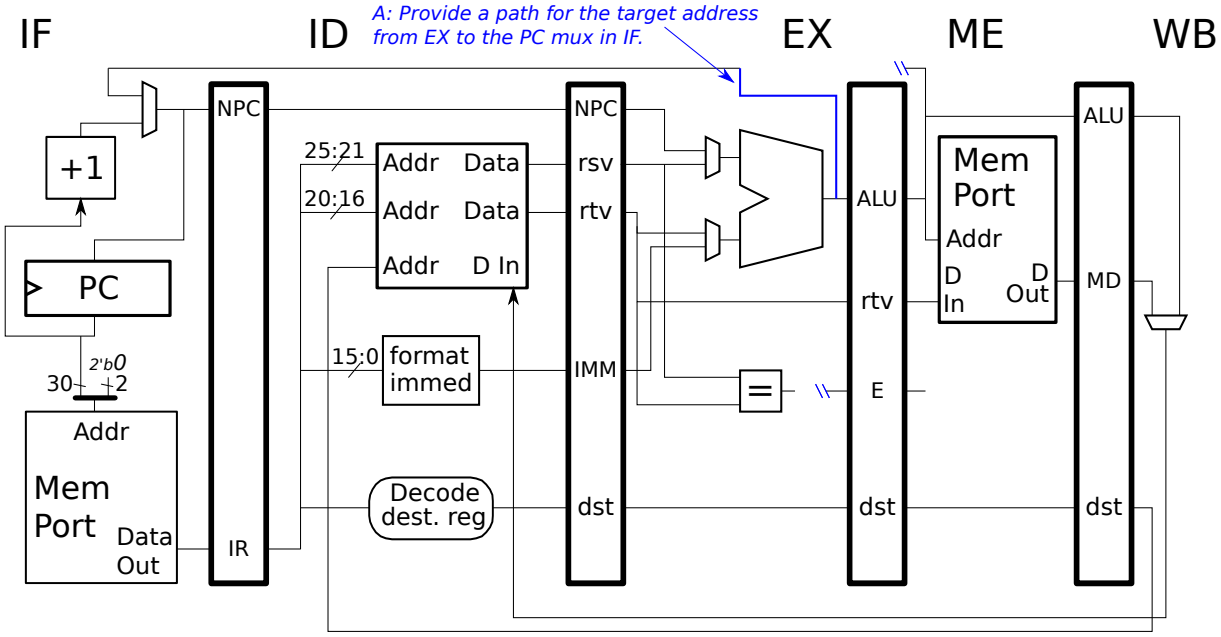
☑ Simplify code fragment.

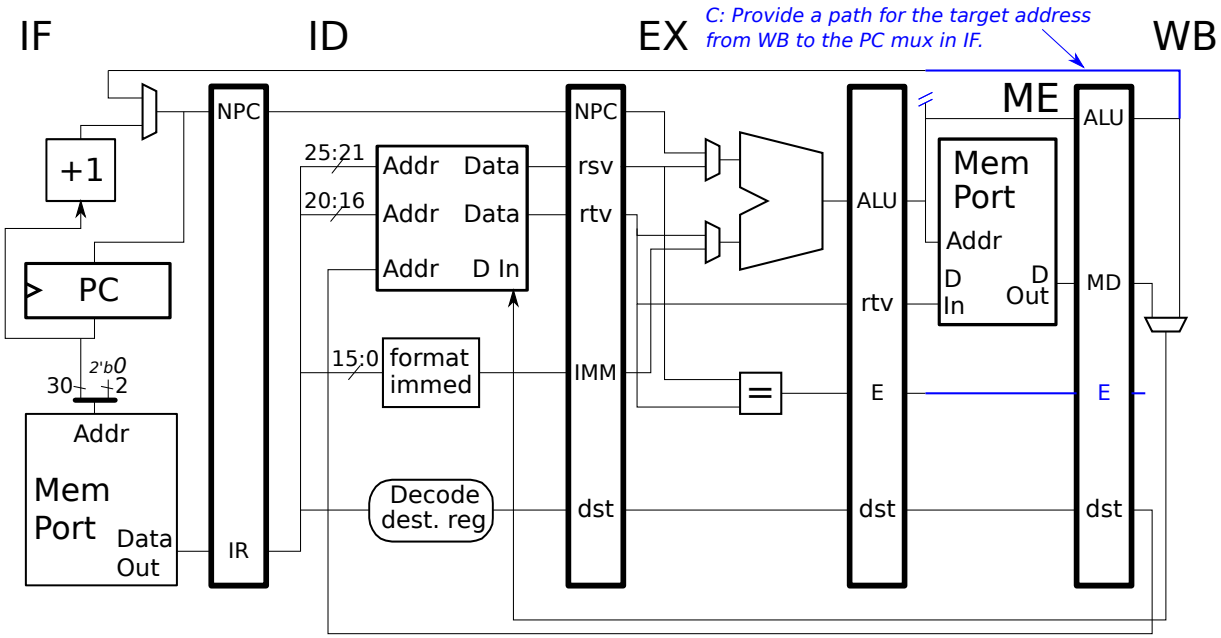
```
sub r1, r2, r3
beq r1, r0,  TARG
lw r1, 0(r4)

# SOLUTION
beq r2, r3,  TARG
lw r1, 0(r4)
```

Problem 5: [16 pts] Appearing below are two identical illustrations of one of our MIPS implementations. To the right are three executions of a code fragment, only one of which is possible on the implementation.

Identify the execution that is possible. For each of the executions that is not possible modify one of the illustrations below so that it is. The modification is very simple, just consider the target address. A few well chosen lines will suffice. No logic gates.

The solution appears to the right and below. The branch target address must appear at the PC mux in `IF` in the cycle before the target is in `IF`. In Execution A the target is in `IF` in cycle 3, so the target address must appear at the PC mux in cycle 2, when the `bne` is in `EX`. Therefore for Execution A the path to the PC mux must be moved from `ME` to `EX`, that's shown in blue. Similar reasoning applies to Execution C.





☑ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ⊗ *No*.

☑ If not, modify the implementation so that it is and ☑ label your modifications *A*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IFx
 lui r5, 0x1234
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10              IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION A
```

☑ Is the execution below consistent with the unmodified implementation? ⊗ *Yes* or ◯ *No*.

☑ If not, modify the implementation so that it is and ☑ label your modifications *B*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF IDx
 lui r5, 0x1234             IFx
 ori r5, r5, 0x6789
TARG:
 xor r8,r9,r10                 IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION B
```

☑ Is the execution below consistent with the unmodified implementation? ◯ *Yes* or ⊗ *No*.

☑ If not, modify the implementation so that it is and ☑ label your modifications *C*.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
 bne r1, r2, TARG  IF ID EX ME WB
 add r1, r1, r3       IF ID EX ME WB
 sw r1, 0(r4)            IF ID EXx
 lui r5, 0x1234             IF IDx
 ori r5, r5, 0x6789            IFx
TARG:
 xor r8,r9,r10                    IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  EXECUTION C
```

Problem 6: [15 pts] Answer each question below.

(*a*) Company A and B both come out with a new computer each year. Company A changes both the ISA and implementation each year. Company B changes only the implementation each year but uses the same ISA.

☑ Which company is following accepted practice?

Company B.

☑ Which company's customers are more likely to stay with the company when it is time to upgrade to a new model? ☑ Explain.

Company B, because they do not need to re-write their software.

(*b*) In MIPS `nop` is a pseudo instruction.

☑ What is a pseudo instruction?

It is something that can be used as though it were a machine instruction in assembly language, even though no such instruction is defined by the ISA or recognized by implementations. Instead, an assembler will translate a pseudoinsruction into a machine instruction (or if necessary multiple machine instructions) that performs the operation defined for the pseudoinsruction. Pseudoinstructions are provided as a convenience. For example, in MIPS it is easier to type `nop` than to type some other instruction that does nothing, such as `sll r0, r0, 0`.

☑ Does having too many pseudo instructions make implementations too expensive? ☑ Explain.

No, since they do not affect the hardware. For example, consider a collection of 500 pseudo-instructions including `plus1 RS` which is translated to `addi RS, RS, 1`, pseudo-instruction `left1 RS` which is translated to `sll RS, RS, 2`, ..., and `less1 RT, RS` which is translated to `slti RT, RS, 1`. All of these pseudoinstructions translate into an existing machine instruction so their presence does not affect the ISA and therefore the implementation.

(*c*) The first code fragment below, from code presented in the course, loads element `i` of an array of integers. (Here integers are four bytes.) Complete the second code fragment so that it loads element `i` from an array of shorts (A short is two bytes.).

```
# C CODE                        #  ASM REGISTER = C VARIABLE NAME
# int *a; ...          # $s1 = a;  $t0 = i    sizeof(int) = 4 chars.
# x = a[i];

sll $t5, $t0, 2       # $t5 -> i * 4;  Each element is four characters.
add $t5, $s1, $t5     # $t5 -> &a[i]  (Address of a[i].)
lw $t1, 0($t5)        # x = a[i];   $t1 -> a[i]
```

☑ Complete code below so that it loads a short.

```
# C CODE                        #  ASM REGISTER = C VARIABLE NAME
# short *a; ...          # $s1 = a;  $t0 = i    sizeof(short) = 2 chars.
# x = a[i];

# SOLUTION
sll $t5, $t0, 1       # $t5 -> i * 2;  Each element is two characters.
add $t5, $s1, $t5     # $t5 -> &a[i]  (Address of a[i].)
lh $t1, 0($t5)        # x = a[i];   $t1 -> a[i]
```

Name Solution

Formatted For 2-Sided Printing

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Monday, 8 May 2023 10:00-12:00 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(10 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

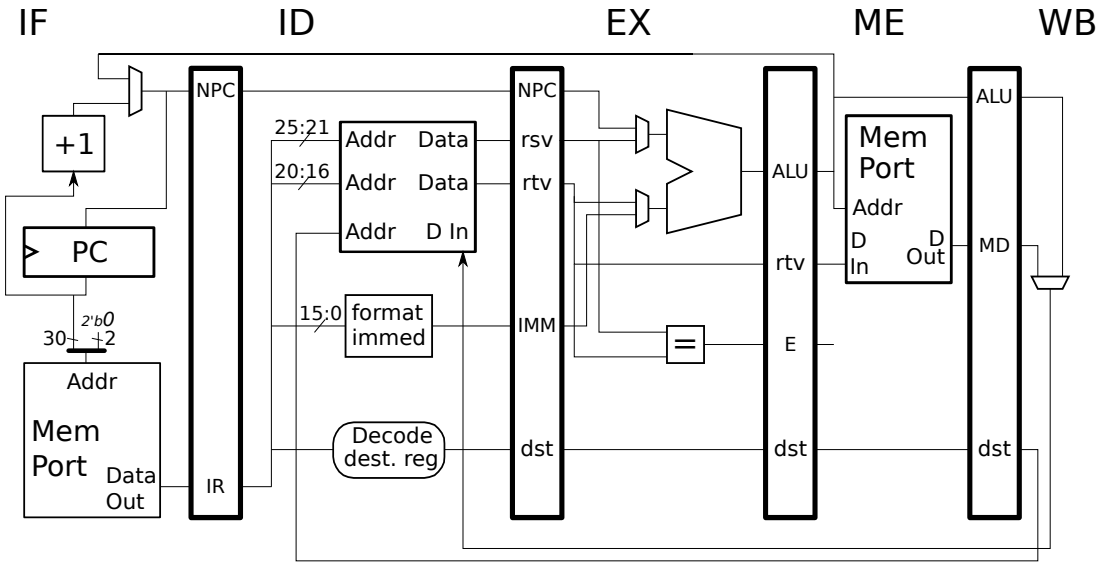
Alias

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

Problem 1: (25 pts) Show the execution of the code fragments on the following implementations. In each case the branch is taken.

(*a*) Show the execution on this basic MIPS implementation.



☑ Show execution for the case where the branch is taken. ☑ Check for dependencies. ☑ Base execution on hardware shown. ☑ Pay close attention to branch behavior.

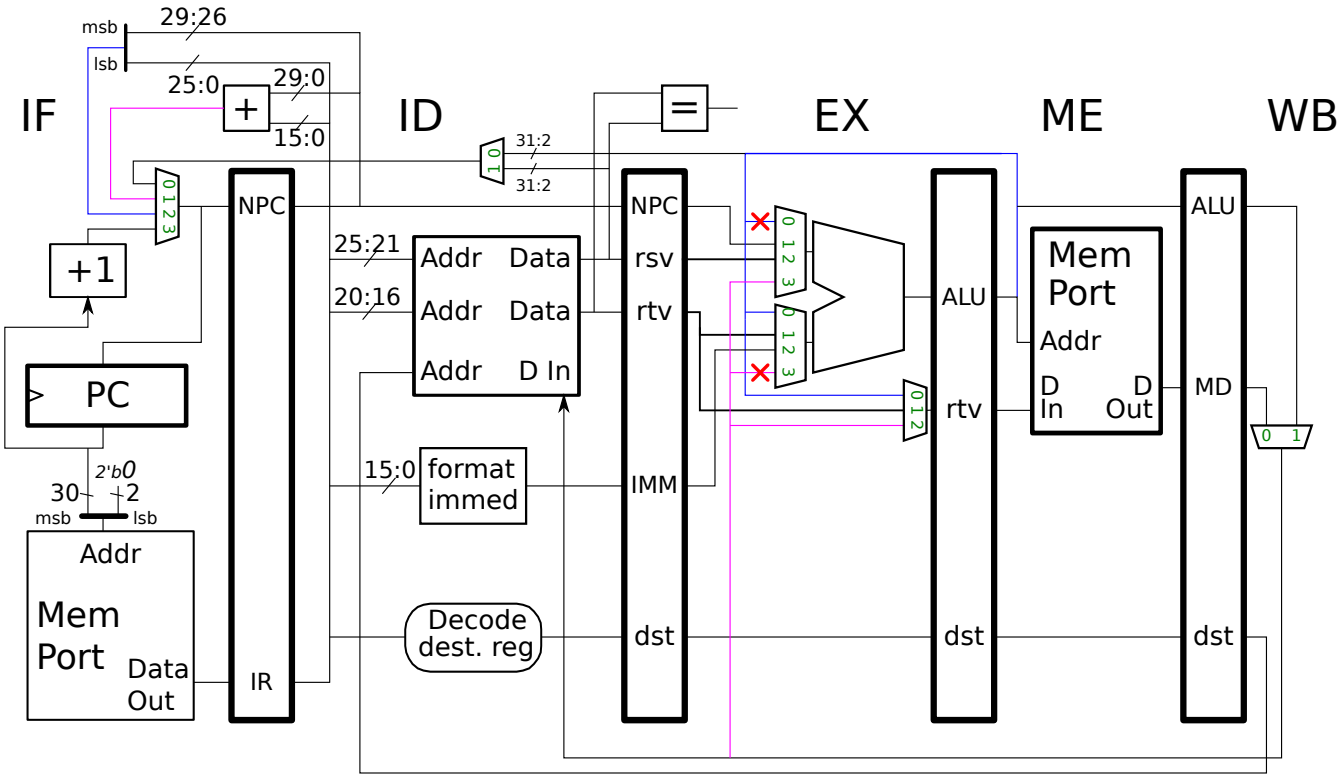
The solution appears below. To help in understanding the solution registers carrying dependencies that result in stalls are shown in uppercase and **bold**, for example as R5 instead of r5.

Notice that in this implementation there are no bypass paths, which is why ALU-to-ALU dependencies (such as between `sll` and `add`) are two cycles. If this does not make sense then please please please study more carefully and ask for help if you don't get it!

Also notice that we can tell that the branch resolves in ME because a connection to the multiplexor in the IF stage comes from the ME stage. Because the branch resolves in ME the target will be fetched in the next cycle, when the branch is in WB, which is cycle 9 in the example below. (In most five-stage MIPS implementations used in class the branch resolved in ID.)

```
 # SOLUTION
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r6, r6, 1     IF ID EX ME WB
 lui R5, 0xf00d        IF ID EX ME WB
 lw R3, 0x81b4(R5)        IF ID ----> EX ME WB
 bne r1, r2, TARG            IF ----> ID EX ME WB
 or r8, R3, r6                        IF ID -> EX ME WB
 sw r8, 0x8120(r5)                       IF ->x
 lw r3, 0x8200(r5)
 addi r5, r5, 16
TARG:
 sll R10, r3, 8                                  IF ID EX ME WB
 add r11, R10, r12                                  IF ID ----> EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

(*b*) Appearing below is a MIPS implementation and an **incorrect** execution of a code fragment on that implementation. The code executes more slowly than it would on the implementation. Modify **the implementation** so that the execution is correct. Your modifications will reduce the cost of the implementation.



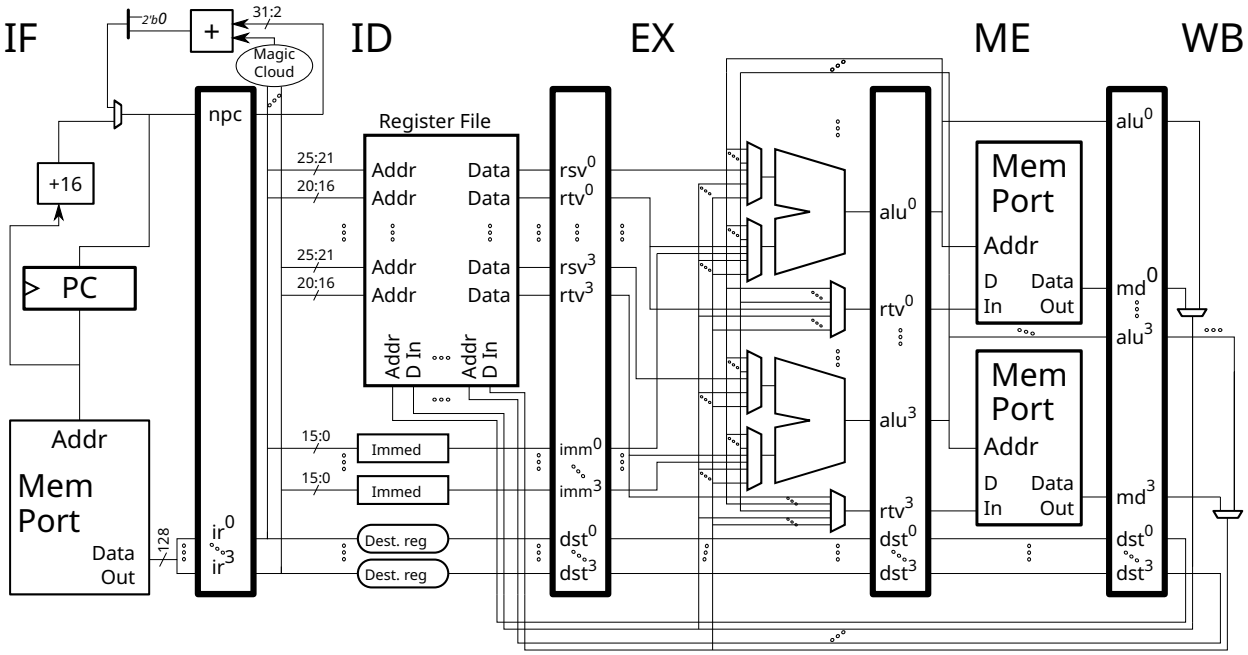
☑ Modify the implementation above so that the code below executes as shown.

☑ **Make as few** changes as possible. For example, remove a bypass path from a mux input, rather than the entire bypass path.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10
addi r6, r6, 1     IF ID EX ME WB
lui R5, 0xf00d        IF ID EX ME WB
lw R3, 0x81b4(R5)        IF ID -> EX ME WB
bne r1, r2, TARG            IF -> ID EX ME WB
or r8, r6, R3                     IF ID -> EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10
```

Solution shown above in red in the EX stage. The crossed out input on the upper mux is for the bypass from ME that would have provided the value of R5 needed by the `lw`. The crossed out lower mux input would have been used by the `or` instruction to bypass the in WB from the `lw` instruction.

(*c*) Appearing below is a **4-way** superscalar MIPS implementation.



☑ Show execution on the 4-way superscalar implementation ☑ with branch taken.

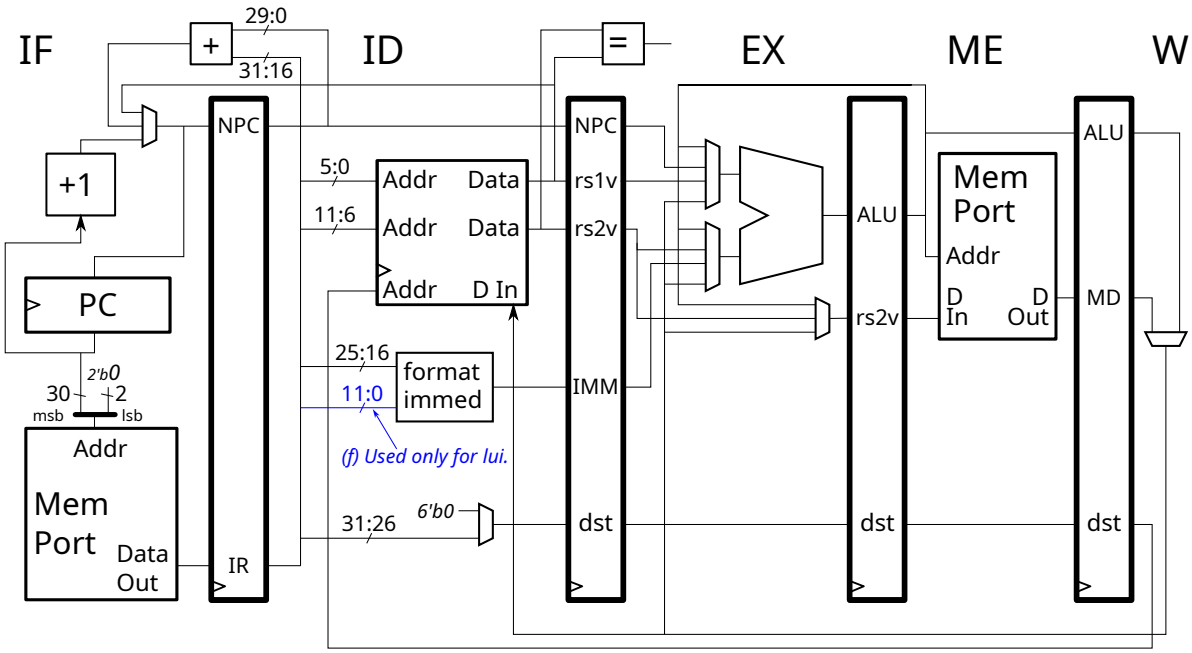
☑ Pay attention to ☑ branch behavior and ☑ the order of instructions within a stage.

Solution appears below. Notice that the branch target, `sll`, is fetched when the branch is in `EX`.

Grading Note: A common mistake (in 2024 when this appeared in Homework 5) was to mistake `lui` for some kind of load. Remember that `lui` is the load upper immediate instruction, it puts the immediate into the 16 high bits of the destination register and zeros the low 16 bits.

```
 # SOLUTION
 # Cycle            0  1  2  3  4  5  6  7  8  9
 addi r6, r6, 1     IF ID EX ME WB
 lui r5, 0xf00d     IF ID EX ME WB
 lw r3, 0x81b4(r5)  IF ID -> EX ME WB
 bne r1, r2, TARG   IF ID -> EX ME WB
 or r8, r3, r6         IF -> ID -> EX ME WB
 sw r8, 0x8120(r5)     IF ->x
 lw r3, 0x8200(r5)     IF ->x
 addi r5, r5, 16       IF ->x
 # Cycle            0  1  2  3  4  5  6  7  8  9
TARG:
 sll r10, r3, 8           IF -> ID EX ME WB
 add r11, r10, r12        IF -> ID -> EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9
```

Problem 2: (25 pts) The implementation of *ANRI*, a new RISC ISA, appears below. Many instructions are similar to those of MIPS, though they differ in format and other features. Like MIPS, ANRI registers are named `r0`, `r1`, ...



(*a*) First, an easy question:

☑ How does ANRI differ from MIPS in the ☑ number of registers and ☑ the immediate size?

Full-Credit Answer: ANRI has 64 registers, while MIPS has 32, and the ANRI immediate size is 10 bits while MIPS' is 16 bits.

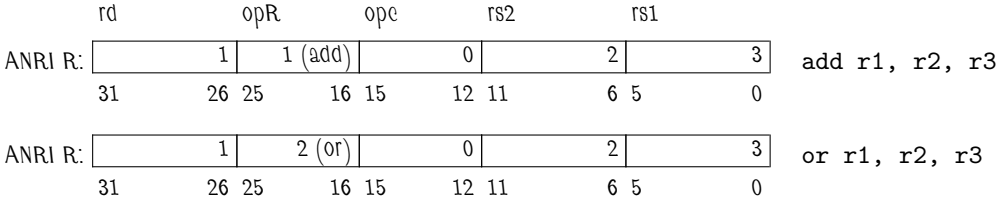
Explanation: The number of registers (integer [general-purpose] registers to be exact) is determined by looking at the number of bits used as an address in the ports of the register file (in the `ID` stage). The width of each of the `Addr` inputs is 6 bits, and so the number of registers is $2^6 = 64$. The size of the immediate can be determined by looking at the number of bits at the input to the original (black in the diagram) `format immed` unit in `ID`, that's 10 bits. (The input labeled (f) is part of the solution to part (f).)

(*b*) Like MIPS, ANRI's Format R is used for three-register instructions such as `add r1, r2, r3`. Show a possible ANRI Format R consistent with the hardware.

☑ Show the bit positions and the name of each field in ANRI Format R based on reasonable guesses.

☑ Show possible field values for `add r1, r2, r3` ☑ and for `or r1, r2, r3`. (Two instructions for a reason.)

Solution:



Explanation: The position of the `rs1` and `rs2` fields can be determined by looking at the read-port `Addr` inputs of the register file (in the `ID` stage). Similarly, the `rd` field can be determined by looking at the bits, `31:26`, that travel through the pipeline `dst` latches on their way to the register file write `Addr` port.

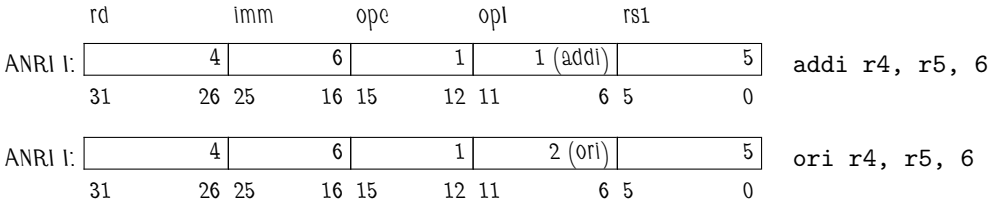
The position of the opcode field needs to be inferred. The opcode field must appear in the same place in every instruction. Bits `25:16` are used for the immediate, and so they can't be used for the opcode field. The only remaining bits are `15:12`, and so those must be the location of the opcode field. A 4-bit opcode field, with only 16 distinct values, is not enough to encode every instruction. So like MIPS an opcode extension field will be needed, that's called `opR` (opcode extension for format R) here. Also like MIPS, the opcode field is set to zero for type-R integer instructions. Values of 1 and 2 in the `opR` field are used for `add` and `or` instructions.

Note: MIPS uses field names `opcode, rs, rt, rd, sa, func, immed, ii`. There is no reason to use exactly the same names in ANRI. Most ISAs use `rd` for a destination and `opcode` for the opcode, so that's retained in this solution. But MIPS' ambiguously named `rt` does not apply because `rd` is always a destination and the two sources are always sources (if used at all). For that reason, the sources were named `rs1` and `rs2` (which is how many other ISAs name sources). Many ISAs use an immediate field to specify a constant shift amount, MIPS is an exception using a dedicated `sa` field. Given that no shift unit was shown there was no reason to expect a dedicated shift amount field.

(*c*) Like MIPS, ANRI's Format I is used for immediate instructions such as `addi r4, r5, 6`. Assume that like MIPS, **each of the dozens of ANRI arithmetic and logical instructions has an immediate variant**.

☑ Show the bit position and name of each instruction field in ANRI Format I based on reasonable guesses and ☑ heeding the bold text above. ☑ Show possible field values for `addi r4, r5, 6` ☑ and for `ori r4, r5, 6`.

Solution:

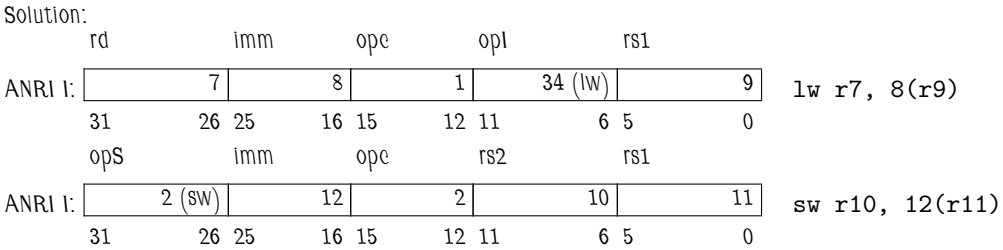


Explanation: The type I instruction uses an `imm` field, whose position is determined by the input to the `format immed` unit in `ID`. Because there are dozens of type I instructions (see **the bold text**) the 4-bit opcode field will not be sufficient to code them all. For that reason bits `11:6` are used as an opcode extension field, called `opI`, occupying the same position as `rs2`, which is not used in type-I instructions.

(*d*) Consider load and store instructions.

☑ Show encoding of `lw r7, 8(r9)` and ☑ `sw r10, 12(r11)` in ANRI Format I, or something similar. Don't forget to ☑ base the encoding on the hardware.

Solution:



Explanation: The `lw` is encoded the same way as the other type I ANRI instructions. However, that approach won't work for `sw` because the `rs2` field is needed for the register holding the value to be written to memory (`r10` in the example). Therefore for store instructions bits `31:26` are used for an opcode extension field, called `opS` here.

(*e*) Consider procedure call instructions.

☑ Based on the implementation, why does it appear that ANRI would lack the equivalent of MIPS `jal SomeProcedure` though it could still encode the equivalent of `jalr r1, r2`.

Full-Credit Answer: Because the only inputs to the `dst` mux are zero and `rd`, so there is no way to encode an implicit destination register such as `r31`.

Explanation: The MIPS `jal` instruction writes the return address in `r31`. The instruction has only an opcode and an immediate field, the 31 is not encoded. The ID-stage mux selecting the writeback register (`dst`) has only two inputs, a zero or the `rd` field. Therefore every instruction in ANRI that writes a register must use the `rd` field to specify the destination register, explicitly. That makes it impossible to have an implicit register such as `r31` in MIPS. (Note that `r0` must be the zero register because the register file lacks a write-enable input.)

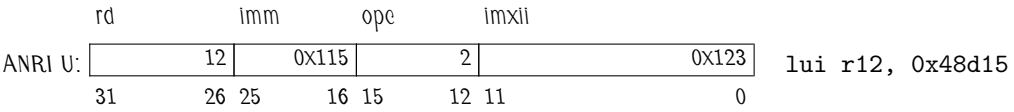
Grading Note: Many solutions to 2024 Homework 2, based on this problem, incorrectly answered that there was no way for an ANRI `jal` to write a return address into a register. The return address in MIPS is `PC + 8` (the same as `NPC+4`). If that were the case for ANRI, then a return address could easily be saved by having the ALU add 4 (or maybe 1) to the `NPC` value available in the upper input.

(*f*) Based on the hardware above, ANRI would lack a means of loading an arbitrary 32-bit constant into a register using two instructions. Modify the hardware so that ANRI could encode an instruction like MIPS `lui`, one that could be used to load an arbitrary 32-bit constant into a register using two instructions.

☑ Modify hardware to implement an instruction to help loading a 32-bit constant.

☑ Show the format and encoding of this new instruction. ☑ The format must fit in as much as possible with existing formats.

The encoding appears below and the hardware modifications appear in the diagram in blue. The format is called type U, following RISC-V terminology. Since a `lui` instruction does not need source registers the `rs1` and `rs2` fields are used for an immediate extension field, called `imxii` here. With the 10-bit `imm` field and the 12-bit `imxii` field the ANRI `lui` can accommodate 22-bit immediates. That is sufficient to load an arbitrary 32-bit contents using the instruction pair `lui r1, hi(0x12345678); ori r1, ri, lo(0x12345678);`, where assembler macros `hi` and `lo` extract the high 22 bits and low 10 bits of their arguments. Because ordinary type-I instructions use `imm` for their entire immediate it would be more efficient hardware-wise to use `imm` for the least-significant 10 bits of the immediate in `lui` instructions and `imxii` for the remaining 12 bits. In this example `hi(0x12345678)` evaluates to `0x048d15` or $00\,0100\,1000\,1101\,0001\,0101_2$. So the `imm` field would be set to the lower ten bits, $01\,0001\,0101_2 = 115_{16}$. The `imxii` would be set to the high 12 bits, $0001\,0010\,0011_2 = 123_{16}$.



`lui r12, 0x48d15`

Problem 3: (20 pts) In the incomplete MIPS implementation to the right the FP multiply unit has its own write port to the FP register file, shown in blue and labeled WM in several places. Because of this new MW port the sub.s instruction in the execution below does not stall, both the sub.s and mul.s can write back in cycle 8. The control logic has not yet been updated for MW.

```
# Cycle           0  1  2  3  4  5  6  7  8  9
mul.s f1, f2, f3  IF ID M1 M2 M3 M4 M5 M6 WM       # Uses MW, the mult-only write port.
add.s f4, f5, f6     IF ID A1 A2 A3 A4 WF
sub.s f7, f8, f9        IF ID A1 A2 A3 A4 WF       # No stall!
lwc1 f10, 0(r11)           IF ID ----> EX ME WF    # Stall due to WF str hazard.
add.s f12, f1, f14            IF ----> ID -> A1... # Stall due to dep with mul.s
```

(*a*) With the illustrated hardware a result cannot be bypassed from a mul.s to another instruction. The last add.s suffers a stall because of that. Add bypass hardware for such cases.

☑ Add the bypass hardware. ☑ Try to keep cost down by using one mux.

Solution appears in green. Notice that the bypass can either be from the multiply unit or the value headed to WF, but not both. Because of this stalls are still possible, which might be the topic of a followup question in 2024.

(*b*) Modify the control logic so that it no longer stalls instructions that would write back through WF at the same time as a preceding mul.s. *Hint: This is just a matter of crossing things out.*

☑ Modify logic to eliminate stalls due to mul.s ☑ **but retain stalls** for instructions contending for WF, such as lwc1 in execution above.

Gates shown crossed out with red exes. The exed-out gates checked for an instruction in M2 to avoid a structural hazard on WF. The only instruction that can be in M2 is a multiply, and that now uses its own write port, so there is no need to check.

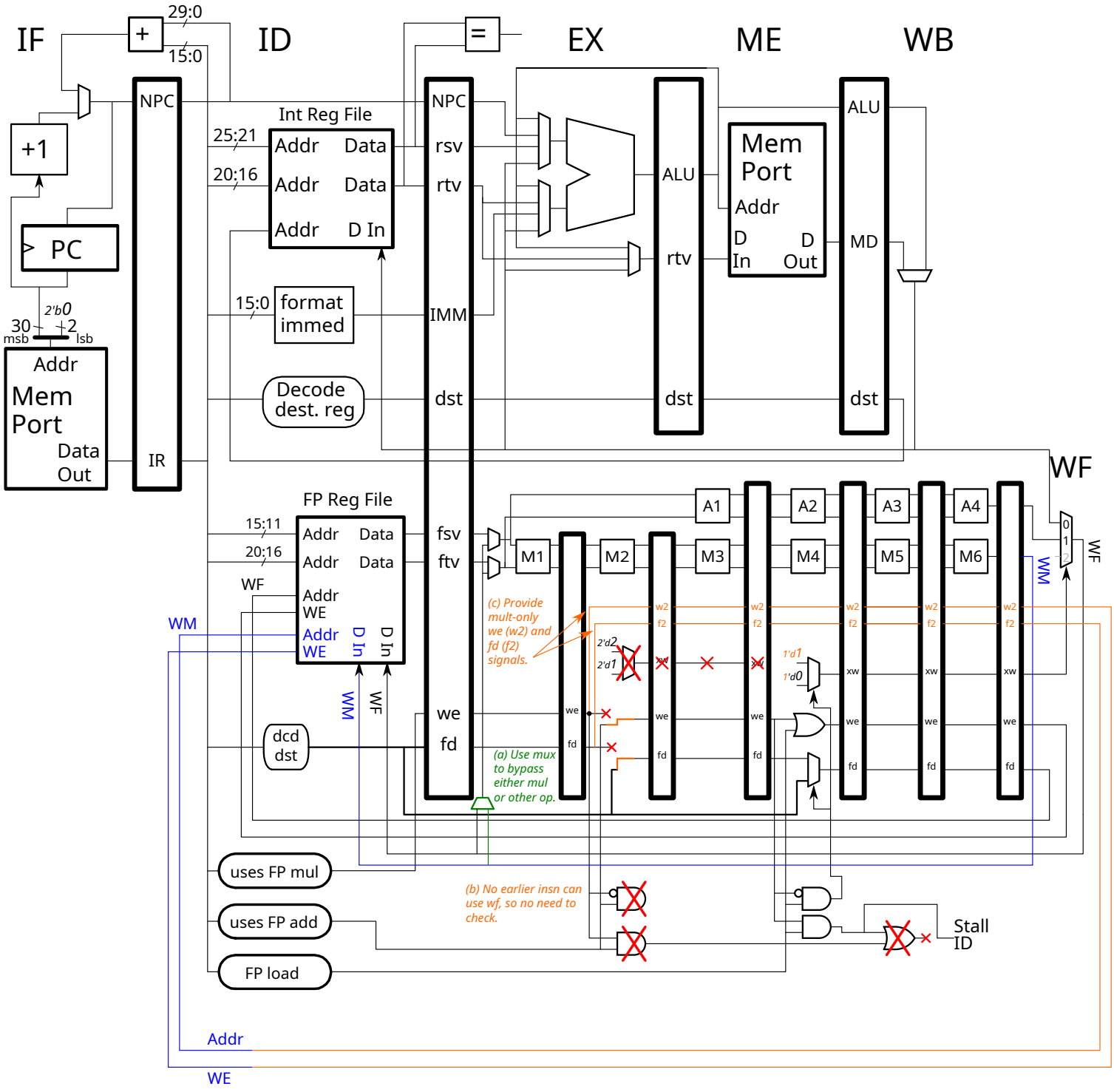
(*c*) Provide the correct Addr and WE signals to the WM and WF ports of the FP register file. Note that the WF ports are connected, but based on the original version. The WM port wires are shown unconnected on the lower-left of the diagram.

☑ Add hardware for the MW Addr and WE signals, ☑ and make changes to the WF Addr and WE signals.

Solution appears in orange. Because the multiply is the only instruction that uses the WM port, multiply-only pipeline latches are provided for the write-enable and destination register signals. The multiply's write-enable signals are labeled w2 and the multiply's destination register signals are labeled f2.

☑ Cross out unneeded hardware and ☑ simplify remaining hardware where possible.

Because a multiply has its own write-enable and destination register pipeline latches, there is no need for the fd multiplexor, the xw multiplexor, and OR gate in the M2 stage. Also notice that the xw signal can be made 1 bit since the WF mux now has two inputs.



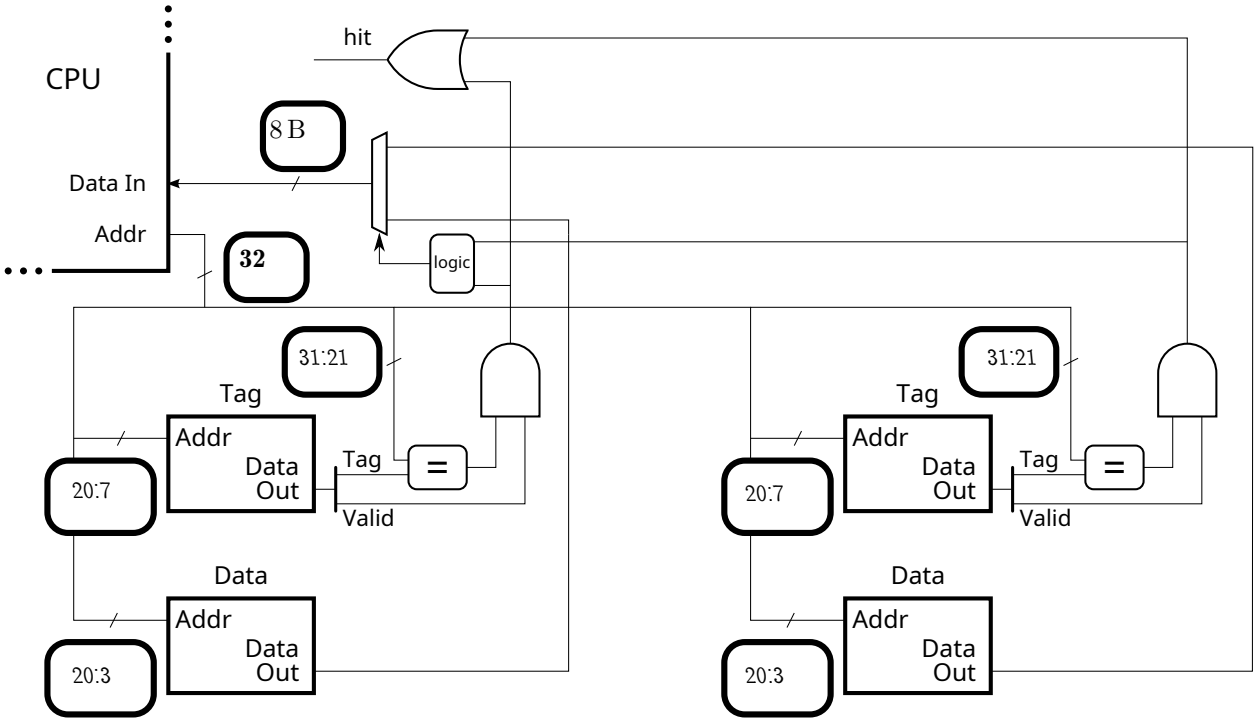
Staple This Side

Problem 4: (10 pts) The diagram below is for a 4 MiB two-way set-associative cache with a line size of 128 B. The character size is the usual 8 bits. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $128 = 2^7$.

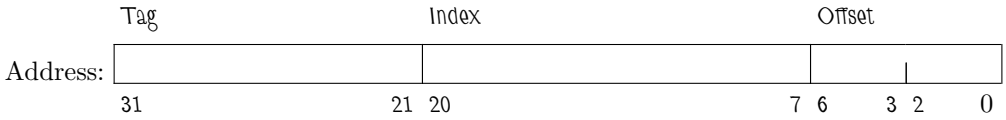
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.

Solution appears below.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)



The code in the problem belows run on a cache with a line size of 128 B (which is 2^7 B). The code fragment starts with the cache cold (empty); consider only accesses to the array.

(*b*) Find the hit ratio executing the code below.

```
float sum = 0;
bfloat16_t *a = 0x2000000; // sizeof(bfloat16_t) == 2
int ILIMIT = 1 << 11;    // = 2^11

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^7 = 128$ bytes is given. The size of an array element, which is of type `bfloat16_t` (a 16-bit floating point type called *brain float*), is $2 = 2^1$ B, and so there are $2^7/2^1 = 2^{7-1} = 2^6 = 64$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^6 elements, and so the next $2^6 - 1 = 63$ accesses will be to data on the line, hits. The access at `i=64` will miss and the process will repeat. Therefore the $\boxed{\text{hit ratio is } \frac{63}{64}}$.

Problem 5: (20 pts) Answer each question below.

(*a*) How does the ARM A64 `fcvtzs` instruction differ from MIPS `trunc.w.s` instruction? These were the instructions used in `sum_thing_unusual` from Homework 5.

☑ The difference between `fcvtzs` and `trunc.w.s` is:

Full-Credit Short Answer: Both read a FP register and convert the truncated FP value to an integer, the result is written to an integer register by `fcvtzs` and to a FP register by `trunc.w.s`.

Long Answer: Both instructions read a floating-point value from a floating-point register, truncate the value, and convert it from a floating-point representation to an integer representation. (Truncating a floating-point value means discarding the fractional part. For example, truncating 2.1 yields 2 and truncating -3.9 yields -3. The representation before *and after* truncation is floating point, which is way conversion to an integer representation is specifically mentioned above as something each instruction does. So that both 2.1 and 2.9 are converted to 2. Or, put another way, both `0x40066666` and `0x4039999a` are converted to `0x2`.) The ARM A64 `fcvtzs` writes the result to an integer register. In contrast, the MIPS `trunc.w.s` writes its result to a floating-point register (even though the result is in an integer representation). If the converted value is to be used by an integer instruction the MIPS `trunc.w.s` needs to be followed by a `mfc1` (move from co-processor 1) instruction.

(*b*) With the SPEC CPU benchmarks it is the testers responsibility to compile and run the benchmarks.

☑ A brand-new implementation has many more bypass paths than the old implementation. Why might the results of a tester-compiles test (like SPEC CPU) show better performance on the new implementation than on the old implementation, while with a pre-compiled test the old and new implementations would show the same performance?

Suppose that the only change in the new implementation are the new bypass paths. Further, suppose the compiler used in the pre-compiled test was targeting the old implementation. (It could not target the brand new implementation because the compiler would not have been available or if it were, it would make no sense to have compiled the code for an implementation that was not yet available.) Consider an instruction pair that could make use of the new bypass path. To avoid a stall, the compiler would separate those two instructions. Suppose the compiler separates every pair of instructions that would otherwise use the new bypass paths. Then the code would run at the same rate on both processors. The bypass paths would go unused.

If the code were compiled for the new implementation there would be no reason to separate instruction pairs that could use the bypass paths. Such code would suffer stalls on the old implementation that would not occur on the new implementation.

(*c*) Appearing below are some hypothetical CISC instructions.

```
 # Some Hypothetical CISC Instructions
I1: add r1, r2, 0x12345678          # r1 = r2 + 0x12345678
I2: add (r1), r2, 0x1234            # Mem[r1] = r2 + 0x1234
I3: add (r1), 0xff04(r2), 0x1234    # Mem[r1] = Mem[r2+0xff04] + 0x1234
I4: add r1, (r2), 8(r3)             # r1 = Mem[r2] + Mem[r3+8]
I5: add (r1), r2, ((r3))            # Mem[r1] = r2 + Mem[ Mem[r3] ]
```

☑ Which of these instructions could not easily be included in an ISA with 32-bit fixed-length instructions? ☑ Explain.

Instructions I1 and I3, because the total size of the immediate(s) in each instruction would be 32 bits, leaving no room for anything else.

☑ Which of these instructions would be difficult to implement in a pipelined implementation, even if IF and ID could easily handle variable-length instructions? ☑ Explain.

Short, Full-Credit Answer: Instructions I3, I4, and I5 because they each require more than one memory access. A pipelined implementation would need multiple memory ports, which would be costly.

Explanation: A typical scalar RISC pipeline has one memory port for data access, using our notation, in the ME stage. To implement these three instructions in a pipelined fashion there would need to be multiple memory ports, inflating the cost by a substantial amount. For example, to implement I4 two memory ports would be needed to read the sources. Those could be in the same stage, but for I5 they would need to be in separate stages, one to read `Mem[r3]` and one to read `Mem[ Mem[r3] ]`, and finally one to write the result to memory. It is possible to implement I3-I5 with one data memory port by using the port multiple times per instruction. But such an implementation would not be considered pipelined by most people.

What about I2? That could be implemented by our 5-stage pipeline. RISC ISAs lack such an instruction because they would be more difficult to compile code for. Difficult because sources would still have to come exclusively from registers, so dependent instructions would need to have a companion load to get the value that was written to memory.

(*d*) Consider a 4-way superscalar implementation of a conventional ISA, like MIPS, that has **just one memory port** in the ME stage. Also consider *Hy4VI*, a hypothetical 4-slot VLIW ISA in which only slot 1 can contain a load or store instruction.

☑ Why might the memory port and related hardware in the ME stage of a Hy4VI implementation cost less than that hardware in the ME stage in the 4-way RISC implementation?

In the superscalar implementation the memory port would need to connect to each of the four slots in the ME stage. The multiplexors to make that connection aren't free. In Hy4VI the memory port only connects to slot 1, avoiding the multiplexors.

☑ Why might code written in the conventional ISA enjoy an advantage over code in Hy4VI when running on respective **future** implementations even when the performance of that code is the same on current implementations? The reason given ☑ must have something to do with load and store instructions.

Because sometimes code has lots of loads and stores. In a conventional ISA there's no problem with, say, eight consecutive load instructions. But in Hy4VI each load instruction would have to be accompanied by three non-load instructions. If no such useful instructions could be found those non-load instructions would be no-ops. In the current one-memory port superscalar implementation there would be stalls to limit one load at a time entering ME, and so it would have no advantage over Hy4VI. But a future superscalar implementation might have two memory ports (or even four) in ME, and so there would be fewer or no stalls and thus a performance advantage over Hy4VI.

44 Spring 2022 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

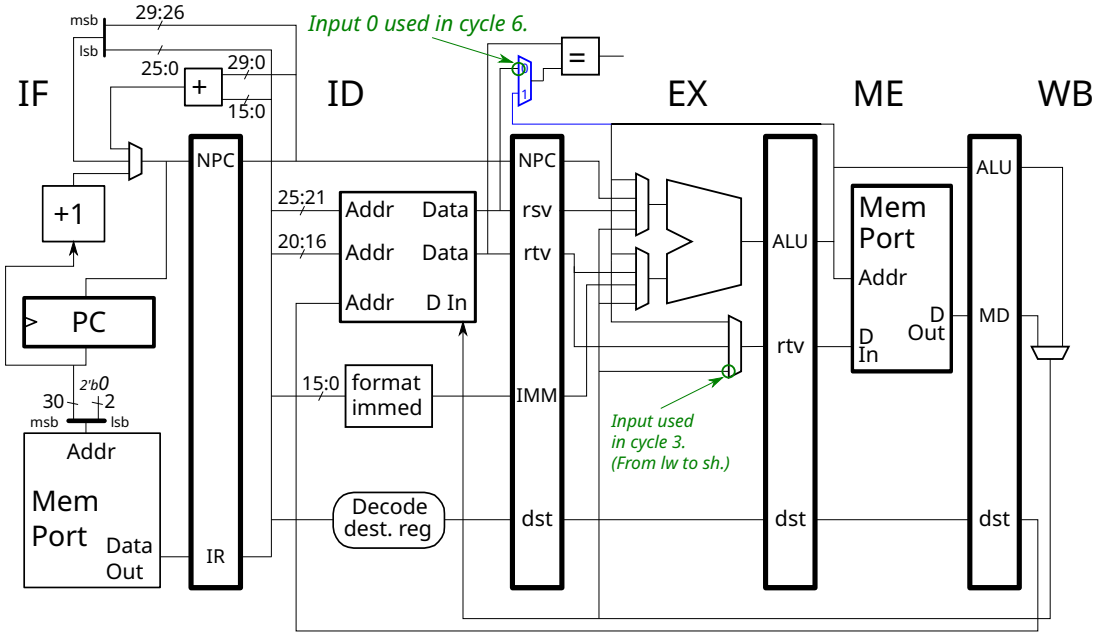
Wednesday, 30 March 2022 9:30-10:20 CDT

Problem 1		(30 pts)
Problem 2		(30 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

Alias Paper!

Good Luck!

Problem 1: [30 pts] The code fragment below is to execute on the illustrated implementation. Show its execution and compute the instruction throughput (IPC) for a large number of iterations. (Note: `sh` is store half.)



☑ Show execution of code below.

☑ Mark each input to the `rtv` mux (in `EX`) ☑ and by the branch comparison (blue) mux used by the code below.

Solution shown above in green. For the branch the value is taken from the register file in cycle 6, using input 0 of the mux. For the `sh` the value from `WB` is used.

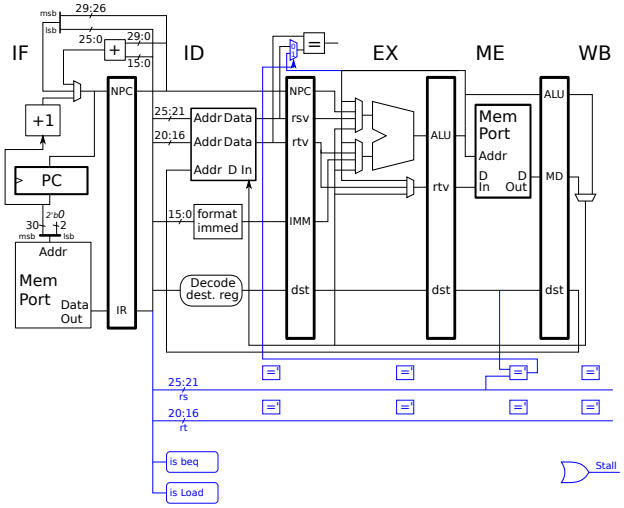
☑ Compute instruction throughput (IPC) for a large number of iterations.

As can be seen in the execution below, the first iteration starts in cycle 0 and the second iteration starts in cycle 7. Each iteration consists of 5 instructions and so the instruction throughput is $\frac{5}{7}$ insn/cycle.

```
 #                SOLUTION
 lw r1, 0(r2)
LOOP:  #  Cycle    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r2, r2, 4    IF ID EX ME WB                      # 1st Iteration
 sh r1, -2(r2)        IF ID EX ME WB
 lw r3, -4(r2)           IF ID EX ME WB
 bne r3, r1, LOOP           IF ID ----> EX ME WB
 lw r1, 0(r2)                  IF ----> ID EX ME WB
LOOP:  #  Cycle    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 addi r2, r2, 4                        IF ID EX ME WB # 2nd Iteration
```

Problem 2: [30 pts] Appearing below (and larger on the next page) is a MIPS implementation based on the solution to Homework 4 Problem 2, in which control logic for a branch bypass was designed. The diagram includes a `Stall` signal in the lower right. Add control logic to set the stall signal to `1` when a `beq` needs to stall due to a dependence that can't be bypassed.

Appearing below are some code fragments. Complete executions are shown for the first two, in the others the executions are incomplete. The control logic should work with these code fragments. It may be helpful to complete the executions.

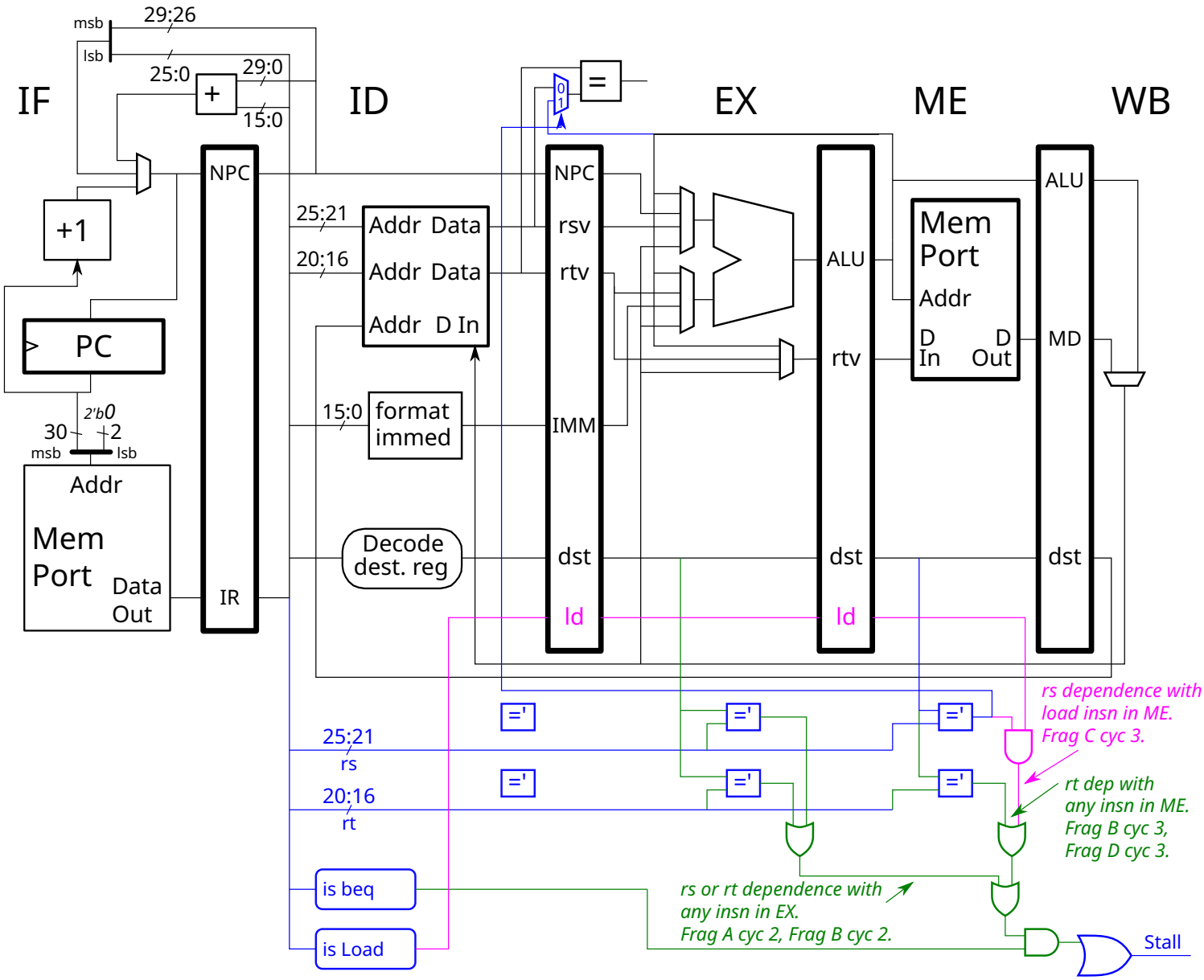


Use next page for solution.

```
# Cycle              0   1   2   3   4   5   6   7   8  Frag A
  addi r1, r2, 3     IF  ID  EX  ME  WB
  beq r1, r4, TARG       IF  ID  ->  EX  ME  WB
  nop                    IF  ->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8  Frag B
  addi r1, r2, 3     IF  ID  EX  ME  WB
  beq r4, r1, TARG       IF  ID  ----->  EX  ME  WB
  nop                    IF  ----->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8  Frag C
  lw r1, 0(r2)       IF  ID  EX  ME  WB  #   Execution below part of solution.
  beq r1, r4, TARG       IF  ID  ----->  EX  ME  WB
  nop                    IF  ----->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8  Frag D
  lw r1, 0(r2)       IF  ID  EX  ME  WB  #   Execution below part of solution.
  beq r4, r1, TARG       IF  ID  ----->  EX  ME  WB
  nop                    IF  ----->  ID  EX  ME  WB

# Cycle              0   1   2   3   4   5   6   7   8  Frag E
  lw r9, 0(r2)       IF  ID  EX  ME  WB  #   Execution below part of solution.
  beq r1, r4, TARG       IF  ID  EX  ME  WB
  nop                    IF  ID  EX  ME  WB
```

☑ Design control logic to generate the stalls for a `beq`. Show connections to the input of the OR gate on the lower right. ☑ Make sure that the logic handles the cases above and for similar situations. ☑ Use as many or as few comparison units, [=’], as you need.

The solution appears below. The branch can't bypass anything in `EX`, and so logic in `EX` checks for a dependence with the branch `rs` or `rt` sources. Examples of such a stall are Frag A and Frag B in cycle 2. A branch can't bypass from `ME` to its `rs` if the instruction in `ME` is a load. An example of such a stall is Frag C in cycle 3. The logic shown in purple checks for this case. This logic needs to know whether a load instruction is in `ME`, and it does so using a new `ld` pipeline latch which carries the output of the [is Load] through the pipeline. The branch needs to stall if there is an `rt` dependence with any instruction in `ME`. An example is Frag D cycle 3. In all cases the logic checks whether there is a branch in `ID`. Otherwise the logic would stall non-branch instructions.



Problem 3: [40 pts] Answer each question below.

(*a*) The MIPS code below loads, stores, and loads again. The two sets of tables further below show the contents of memory before and after the code executes. Numbers in the table are hexadecimal. The code runs on a big-endian system.

```
 # Initially r2 = 0x1200
LOOP:
 lw r1, 0(r2)
 sb r1, 1(r2)
 lw r3, 0(r2)
 bne r1, r3, LOOP
 addi r2, r2, 4
```

☑ Modify the *After* column so that it shows the contents of memory after the code executes.

Solution appears below, emphasized with → arrows ←.

Memory Address	Before Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

Memory Address	After Memory Contents
0x1200	0xa0
0x1201	→ 0xa3 ←
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	→ 0xa7 ←
0x1206	0xa6
0x1207	0xa7

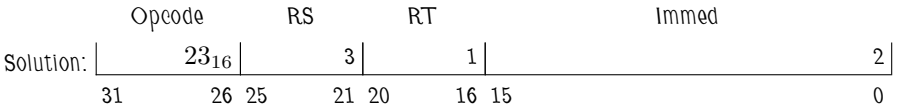
☑ Modify **one row** in the *Before* column below so that the code above executes just one iteration.

Memory Address	Before Memory Contents
0x1200	0xa0
0x1201	→ 0xa3 ←
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

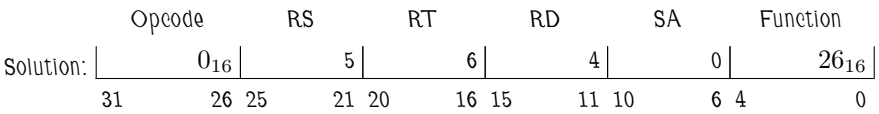
Memory Address	After Memory Contents
0x1200	0xa0
0x1201	0xa1
0x1202	0xa2
0x1203	0xa3
0x1204	0xa4
0x1205	0xa5
0x1206	0xa6
0x1207	0xa7

(*b*) Show the encoding of each MIPS instruction below. (That is, show the layout of the 32 bits in the instruction.) Fill fields with numeric values whenever possible, such as for register numbers and immediate values. For unknown opcodes and func field values show some kind of name.

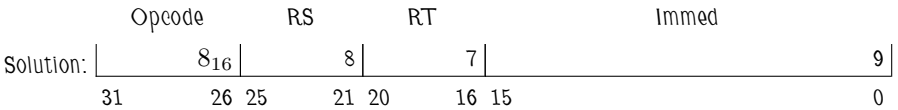
☑ Show encoding of: `lw r1, 2(r3)`.



☑ Show encoding of: `xor r4, r5, r6`.



☑ Show encoding of: `addi r7, r8, 9`.



(*c*) Arm A32 is a 32-bit ISA, Arm A64 (Aarch64) is a 64-bit ISA.

☑ What does the n in n-bit ISA refer to?

Full-Credit Answer: It refers to the number of bits in a memory address.

Discussion: Memory addresses, of course, are what are used by instructions such as MIPS instruction `lw r1,2(r3)`. For this instruction the memory address is `r3+2`. In 32-bit versions of MIPS (including the default MIPS used in classroom examples) that address is 32 bits. In MIPS64 the address would be 64 bits.

For those that already know the difference between a virtual address and a physical address, the n is the number of bits in a virtual address.

☑ Name an application or kind of device for which a 32-bit ISA has an advantage, and ☑ describe the advantage.

Full-Credit Answer: An embedded processor controlling a simple device, such as a coffee maker. In these devices a less-expensive 32-bit processor makes sense because the processor is a big chunk of the cost and the large address space of a 64-bit processor is not needed.

☑ Name an application or kind of device for which a 64-bit ISA is a requirement or a big advantage, and ☑ describe the requirement/advantage.

One which needs to access more than 2^{32} bytes of data. With 64-bit addresses this can easily be done. Though accessing this much data using 32-bit addresses is possible, it is extremely tedious.

(*d*) In the statement below the description of how ISAs and implementations are developed is different than how they are typically developed in accepted practice.

> *By finalizing an ISA after its implementation is complete it is assured that the ISA exactly describes the implementation and that the implementation makes the best use of the technology at hand.*

☑ How is this statement of ISA and implementation development different than accepted practice? ☑ What is the disadvantage of the approach described in the statement (ignoring the "technology at hand" part)?

In accepted practice the ISA is designed first, then implementations are designed. The disadvantage of the approach is that since each implementation has its own ISA, code compiled for one implementation cannot be run on a different (perhaps newer) one.

☑ The phrase "makes the best use of the technology at hand" is correct. Explain why accepted practice of ISA and implementation development may not make the best use of technology. *Hint: think about the number of bits in a register.*

The ISA must be followed. This means that one cannot add things, such as wider registers, just because space is available. The number of bits in a register is specified by the ISA, and that can't be increased in an implementation just because the area is available and wider registers would be beneficial.

(*e*) Answer the following about CISC ISAs.

☑ What feature of CISC ISAs allow them to have large, say 32-bit, immediate values?

Variable-size instructions.

☑ Why can't a RISC ISA like MIPS practically have 32-bit immediates?

Because the instruction size is only 32 bits, and so they would not fit.

Name Solution

Computer Architecture
LSU EE 4720
Final Examination

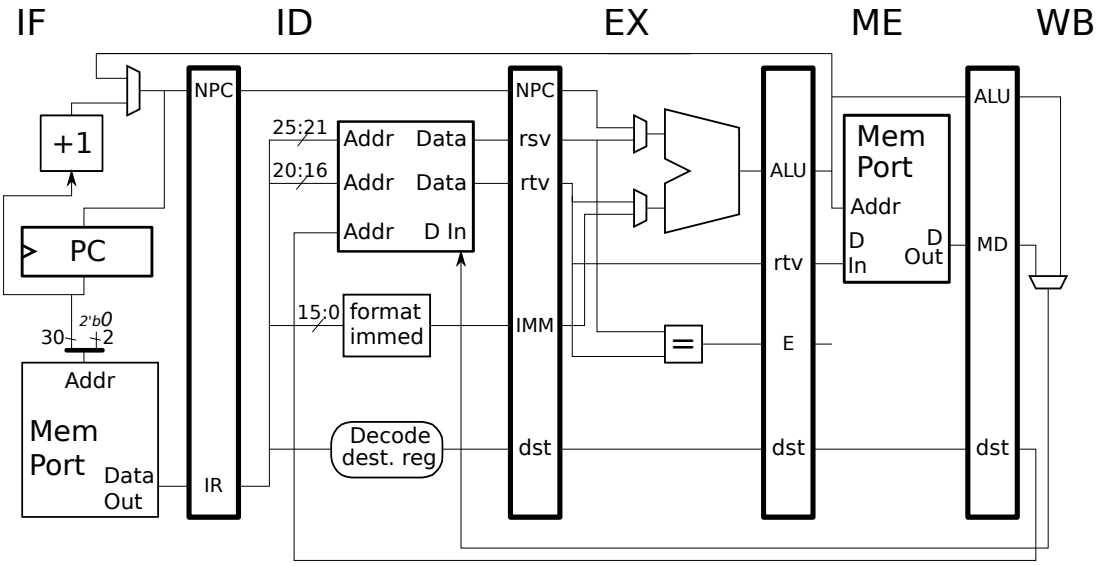
Monday, 9 May 2022 10:00-12:00 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias Purple Mode.

Good Luck! Thank you for your effort in EE 4720!

Problem 1: (20 pts) Show the execution of the code fragments on the following implementations for enough iterations to determine the instruction throughput (IPC). **As always, base the behavior of branches and the availability of bypasses on the implementations. Also, don't forget that MIPS branches have a delay slot.** Sorry for yelling, but I hate it when students miss things.

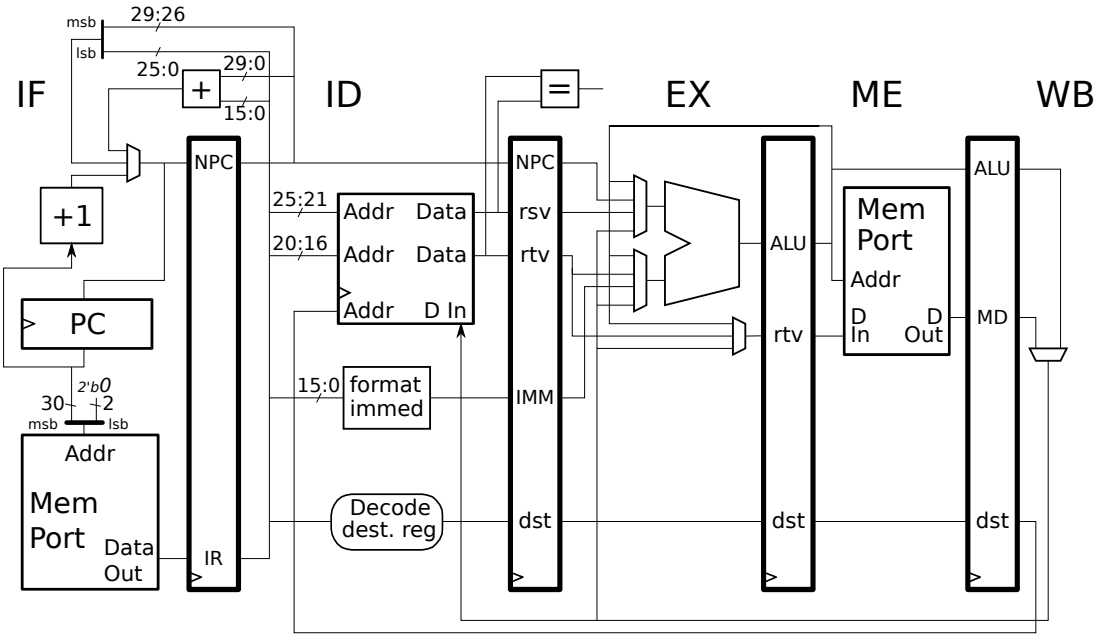


☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

The solution appears below. The branch is resolved in ME, and so the target is fetched (in IF) in the next cycle, when the branch is in WB. Two wrong-path instructions are fetched, `xor` and `sub`. They are squashed when the branch is resolved. (Of course, they would not be squashed if the branch were not taken.)

The instruction throughput is $\frac{2\,\text{insn}}{(8-4)\,\text{cyc}} = \frac{2}{4}\,\text{insn/cycle}$ based on the second iteration starting at cycle 4 and the third iteration starting at cycle 8.

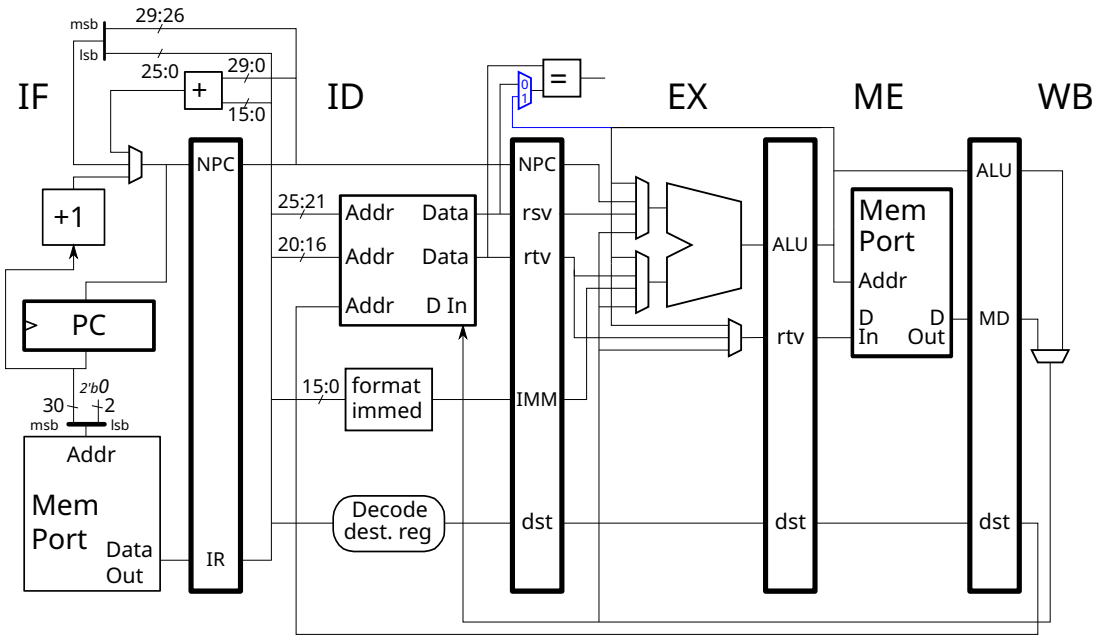
```
 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP  IF ID EX ME WB                # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
 xor r5, r6, r7          IF IDx
 sub r8, r9, r10            IFx
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP              IF ID EX ME WB    # Second Iteration
 addi r1, r1, 4                   IF ID EX ME WB
 xor r5, r6, r7                      IF IDx
 sub r8, r9, r10                        IFx
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12
 bne r1, r2, LOOP                          IF ID EX ME WB
 ...

 # These instructions will be completely executed after the last iteration.
 xor r5, r6, r7
 sub r8, r9, r10
```



☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

The solution appears below. The good news in this pipeline the branch is resolved in `ID`, meaning that zero wrong-path instructions are fetched. The bad news is that there is a dependence carried by `r1` that stalls `bne` in `ID` for two cycles. For this reason, the instruction throughput is the same: $\frac{2\,\text{insn}}{(6-2)\,\text{cyc}} = \frac{2}{4}\,\text{insn/cycle}$ based on the second iteration starting at cycle 2 and the third iteration starting at cycle 6.

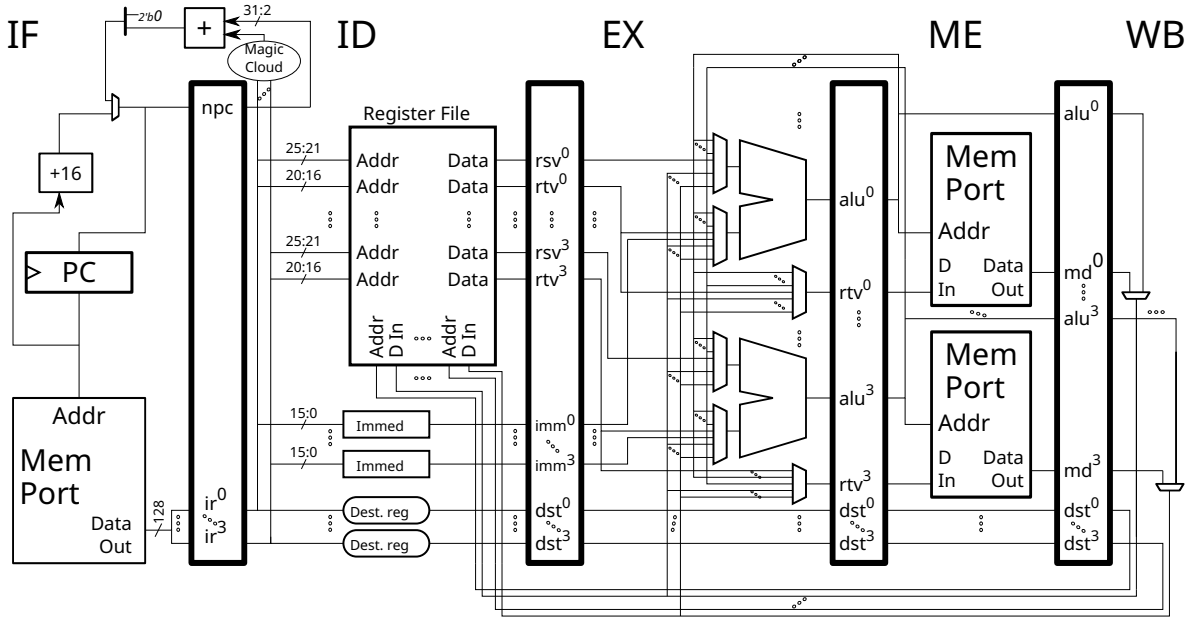
```
LOOP:  # Code in Static Instruction Order
 bne r1, r2, LOOP
 addi r1, r1, 4
 xor r5, r6, r7
 sub r8, r9, r10


 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP  IF ID EX ME WB                  # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP        IF ID ----> EX ME WB       # Second Iteration
 addi r1, r1, 4             IF ----> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 bne r1, r2, LOOP                    IF ID ----> EX ME WB
 addi r1, r1, 4                         IF ----> ID EX ME WB

 # These instructions will be executed after the last iteration.
 xor r5, r6, r7
 sub r8, r9, r10
```



☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

In this implementation there is a bypass that helps with the branch condition dependence, reducing the stall from two cycles to one cycle. The instruction throughput is higher, $\frac{2\,\text{insn}}{(5-2)\,\text{cyc}} = \frac{2}{3}\,\text{insn/cycle}$ based on the second iteration starting at cycle 2 and the third iteration starting at cycle 5.

```
LOOP:  # Code in Static Instruction Order
 bne r1, r2, LOOP
 addi r1, r1, 4
 xor r5, r6, r7
 sub r8, r9, r10

 # SOLUTION -- Dynamic Instruction Order
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP  IF ID EX ME WB                    # First Iteration
 addi r1, r1, 4       IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP        IF ID -> EX ME WB          # Second Iteration
 addi r1, r1, 4             IF -> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11
 bne r1, r2, LOOP                 IF ID -> EX ME WB
 addi r1, r1, 4                      IF -> ID EX ME WB
```



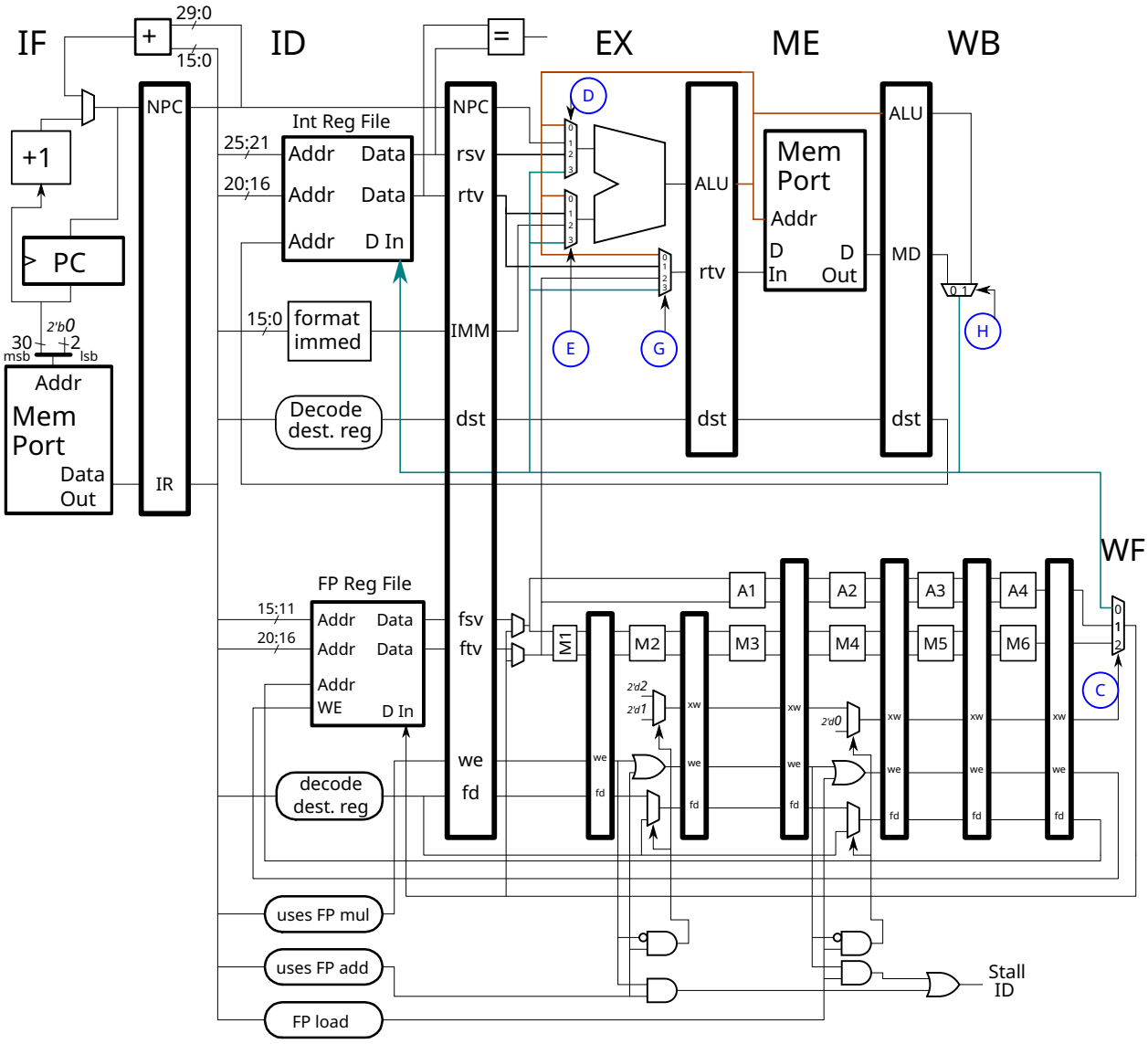
☑ For the **4-way** superscalar MIPS above show execution until the fetch of the `lw r1` in the second iteration.
☑ Show instruction throughput (IPC) assuming a large number of iterations.

Solution appears below. The `add` stalls due to the dependence carried by `r3` and the `sw` stalls due to the dependence carried by `r4`. In this implementation there is a bypass to the memory port `D In` connection and so the `sw r4` need only stall one cycle. In cycle 1 the `sub` and `sw r5` are stalling only so that instructions in `ID` remain in program order.

The instruction throughput is $\frac{9\,\text{insn}}{(7-0)\,\text{cyc}} = \frac{9}{7}\,\text{insn/cycle}$.

```
 # SOLUTION -- Dynamic Instruction Order
LOOP:                0  1  2  3  4  5  6  7  8  9  10 11 12
 lw r1, 0(r2)        IF ID EX ME WB                         # 1st Iteration
 lw r3, 4(r2)        IF ID EX ME WB
 add r4, r1, r3      IF ID ----> EX ME WB
 sw r4, 0(r6)        IF ID -------> EX ME WB
 sub r5, r1, r3         IF -------> ID EX ME WB
 sw r5, 4(r6)           IF -------> ID -> EX ME WB
 addi r6, r6, 4         IF -------> ID -> EX ME WB
 bne r2, r9, LOOP       IF -------> ID -> EX ME WB
 addi r2, r2, 8                     IF -> ID EX ME WB
 xor r10,r11,r12                    IFx
LOOP:                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 lw r1, 0(r2)                          IF ID EX ME WB     # 2nd Iteration
```

Problem 2: (20 pts) Appearing below is our MIPS implementation with a floating-point pipeline. The select inputs of some multiplexors are labeled with a letter. Also, the inputs to some multiplexors have been colored to make them easier to follow.



☑ Show the values on the select inputs (D, E, and H) expected from the execution shown below. ☑ Leave a signal **blank** if it does not affect execution.

```
# Cycle          0    1    2    3    4    5    6
## SOLUTION
D:                          2    0    3
E:                          1    2    2
H:                                    1    1    0
# Cycle          0    1    2    3    4    5    6

# Cycle          0    1    2    3    4    5    6
add R3, r5, r6   IF   ID   EX   ME   WB
addi r2, R3, 4        IF   ID   EX   ME   WB
lw r1, 0(R3)               IF   ID   EX   ME   WB
# Cycle          0    1    2    3    4    5    6
```

☑ Show instructions that could have produced the select input (D,E,G, and H) values shown below. ☑ Take dependencies into account when choosing register numbers.

```
# Cycle                0    1    2    3    4    5    6
D:                                2    2    3
E:                                2    1    2
G:                                               0
H:                                               0    1
# Cycle                0    1    2    3    4    5    6

## SOLUTION
lw R3, 1(r5)           IF   ID   EX   ME   WB
add R2, r5, r4              IF   ID   EX   ME   WB
sw R2, 0(R3)                     IF   ID   EX   ME   WB
# Cycle                0    1    2    3    4    5    6
```

☑ Show the values on the select inputs expected from the execution shown below. ☑ Leave an input **blank** if it does not affect execution.

```
# Cycle          0    1    2    3    4    5    6    7    8    9
## SOLUTION
G:                          2
H:                                    0         1
C:                                    0         0              1
# Cycle          0    1    2    3    4    5    6    7    8    9

lwc1 f1, 0(r5)   IF   ID   EX   ME   WF
swc1 f2, 0(r7)        IF   ID   EX   ME   WB
mtc1 f3, r8                IF   ID   EX   ME   WF
add.s f4, f5, f6           IF   ID   A1   A2   A3   A4   WF
# Cycle          0    1    2    3    4    5    6    7    8    9
```

Problem 3: (20 pts) Appearing to the right is the *early writeback* 2-way superscalar implementation from the 2021 Final Exam and 2022 Homework 6. Recall that in this implementation if slot 1 contains a load instruction then slot 1 writes back when it reaches WB but slot 0 writes back early, in ME/MW. If slot 1 does not contain a load instruction slot 0 writes back when it reaches WB and slot 1 writes back in ME/MW. This is illustrated in the execution below.

```
# Cycle          0  1  2  3  4  5  6
add r2, r3, r4   IF ID EX ME WB    # Slot 0 uses WB since slot 1 isn't a load.
and r1, r5, r6   IF ID EX MW
or r10, r9, r7      IF ID EX MW    # Slot 0 uses MW since slot 1 is a load.
lw r6, 8(r11)       IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
```

(*a*) Consider the execution of the code below:

```
# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX ME WB
sub r1, r2, r5   IF ID -> EX MW
```

The sub stalls because it needs to wait for r2 from the add. Add control logic to generate a stall when slot 1 depends on slot 0. The output of rs src and rt src are 1 when the slot-1 instruction uses the rs and rt registers as sources. Use these in your solution.

☑ Add hardware to generate a stall (see the big OR gate) when slot 1 depends on slot 0.

The solution is on the next page.

(*b*) In the first code fragment below the lw r5 stalls, but because the lw has a zero immediate it could have just used the value computed by the add instruction (since there is no need to add anything to it). In the second execution Mux A is used to perform a *lateral bypass* from the add to the lw during cycle 2, avoiding the stall.

Modify the control logic so that a lateral bypass will be performed when there is a zero-immediate load in slot 1 that depends on the slot-0 instruction. Other code, such as the examples at the top of this problem, should continue to work correctly.

```
# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX ME WB
lw r5, 0(r2)     IF ID -> EX ME WB

# Cycle          0  1  2  3  4  5
add r2, r3, r4   IF ID EX MW      # add executes normally.
lw r5, 0(r2)     IF ID EX ME WB   # Lateral Bypass: lw uses slot-0 alu value.
```

☑ Add logic to detect whether a lateral bypass is possible, and if so, suppress the stall from part a.

☑ Modify the control logic to implement a lateral bypass, in part using Mux A.

☑ Make sure that ☑ early writeback continues to work correctly in other cases and ☑ that the destination of the slot 0 and slot 1 instructions are written to the correct registers.

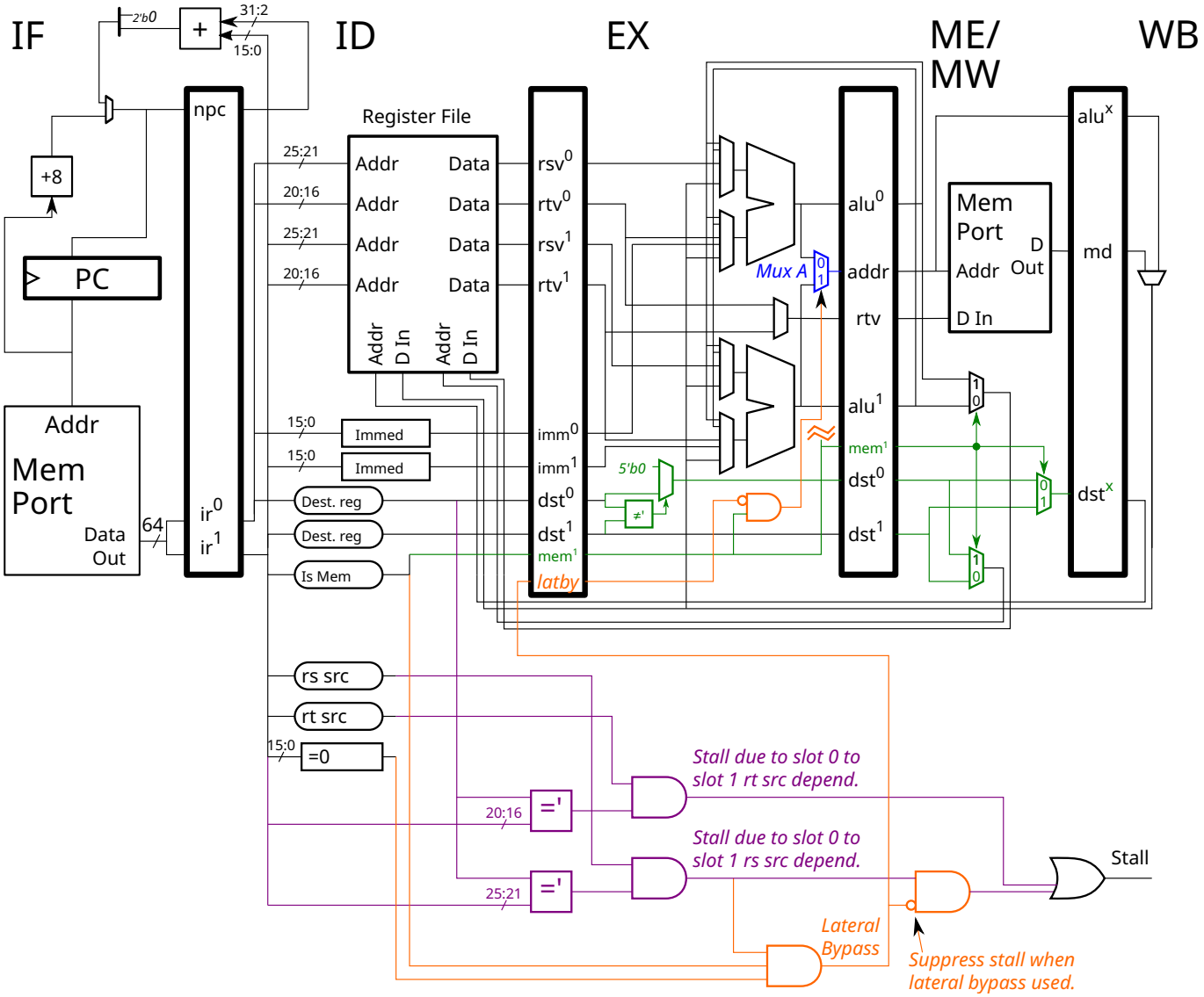
☑ As engineers always do, pay attention to cost and performance.

The solution is on the next page.

Solutions appears below. The hardware for part (a) appears in purple. To detect the dependence the destination of the slot-0 instruction is compared to the `rs` and `rt` sources of the slot-1 instruction. The rs src and rt src blocks are used to check whether the `rs` and `rt` fields of the instruction are used for sources. (In most type I instructions the `rt` is not used for a source. Some instructions, such as floating-point instructions don't have any integer sources.)

The hardware for part (b) appears in orange. A lateral bypass is possible if there is a memory instruction in slot 1 with a zero immediate, and if the `rs` source depends on the slot-0 instruction. If these conditions are true the stall is suppressed.

The logic for implementing the lateral bypass is very simple. If there is a lateral bypass Mux A uses input 0, otherwise Mux A uses the input it would have used without a lateral bypass.



Problem 4: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with a 6-outcome local history.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  N  N  T  T  N  T    T  N  N  T  T  N  T    T  N  N  T  T  N  T     <- Outcome

          All N's          All T's            All N's          All T's
      ---------------  --------------     ---------------  --------------  ...
B2:    N  N  ...  N  N  T  T  ... T  T     N  N  ...  N  N  T  T  ... T  T  <- Outcome
       1  2       7  8  9 10        16     1  2       7  8  9 10        16  <- Position
```

☑ What is the accuracy of the bimodal predictor on branch B1? ☑ Be sure to base the accuracy on a repeating pattern.

The accuracy is $\boxed{\frac{3}{7}}$. The work is shown below. The accuracy is based on the repeating pattern, shown underlined below.

```
   SOLUTION WORK
    0 1  0  0  1  2  1  2    3  2  1  2  3  2  3    3  2  1  2  3  2  3  <- 2-bit Counter
B1:  T  N  N  T  T  N  T    T  N  N  T  T  N  T    T  N  N  T  T  N  T  <- Br Outcome
     x        x  x  x  x       x  x  x     x          x  x  x     x     <- Mispreds
                                                    ----------------------  <- Repeating Ptrn
```

☑ What is the accuracy of the local predictor on B1 ignoring B2.

Though the B1 pattern is 7 outcomes long, no 6-outcome sequence appears twice and so the 6-outcome history is long enough to achieve an accuracy of 100%.

☑ What is the accuracy of the local predictor on B2 ignoring B1.

Full Credit Answer: The accuracy is $\frac{14}{16}$ because out of the 16 positions the local history starting at positions 3 and 11 will fool the predictor, as can be seen based on the table below.

Explanation for students studying: The way to solve this is to construct a table for all possible six-outcome local histories of branch B2, that table is shown below. The local history in the second row, NNNNNT, occurs one per repeating pattern, at position 4. In fact, the local histories for all but two rows appears exactly once per repeating pattern. The exceptions are NNNNNN and TTTTTT. Each of those local histories can appear in three possible places. For NNNNNN local history can start at positions 1, 2, and 3. If it starts at positions 1 and 2 the next outcome is an N, but if it starts at position 3 the next outcome is a T. Therefore the 2-bit counter at the PHT entry for NNNNNN will have values $0, 0, 1, 0, 0, 1, \ldots$. Since the counter value is 0 or 1 the branch will be predicted N for this local history, and the prediction will be correct for positions 1 and 2 and wrong for 3. Something similar happens with local history TTTTTT. The other local histories for this branch each correspond to one starting position and each have one possible outcome, so all will be predicted perfectly. The table below shows the number of predictions for each entry and the number of correct predictions. The sum of these last two columns is used to compute the correct prediction ratio $\frac{14}{16}$.

Local History	Possible Start Pos.	Counter Values	Number Pred.	Number Correct
NNNNNN	1, 2, 3	0,0,1,0,..	3	2
NNNNNT	4	3,3,3,..	1	1
NNNNTT	5	3,3,3,..	1	1
NNNTTT	6	3,3,3,..	1	1
NNTTTT	7	3,3,3,..	1	1
NTTTTT	8	3,3,3,..	1	1
TTTTTT	9, 10, 11	3,3,2,3,.	3	2
TTTTTN	12	0,0,0,..	1	1
TTTTNN	13	0,0,0,..	1	1
TTTNNN	14	0,0,0,..	1	1
TTNNNN	15	0,0,0,..	1	1
TNNNNN	16	0,0,0,..	1	1

Total:			16	14

☑ What is the accuracy of the bimodal predictor on branch B2?

The bimodal predictor will mispredict the first two Ns and the first two Ts of B2, other predictions will be correct. So the overall accuracy will be $\frac{12}{16}$. The two-bit counters and outcomes appear below.

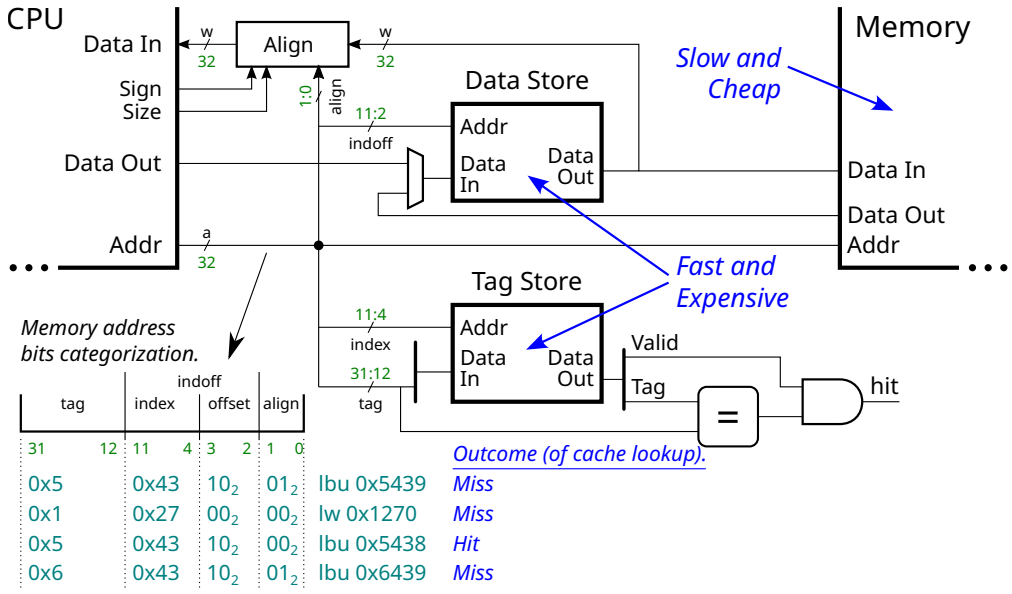
```
          All N's          All T's            All N's          All T's     SOLUTION WORK
      ---------------  --------------      ---------------  --------------  ...
    0  0  0    0  0  0  1  2   3  3  3      2  1      0  0  1  2  3   3  3
B2:   N  N  ...  N  N  T  T  ... T  T      N  N  N. ... N  T  T  T ...  T  <- Br Outcome
                     x  x                x  x              x  x            <- Mispred
      1  2       7  8  9 10        16      1  2  3      8  9 10 11     16  <- Position
```

☑ What is the shortest history size for which the local history predictor is better than the bimodal predictor on branch B2?

One outcome. With a one-outcome local history the local predictor will, with this pattern, predict that the next outcome will match the previous one. It will still just be wrong 2 out of 16 times.

Problem 5: (20 pts) Answer each question below.

(*a*) The diagram below shows a simple direct-mapped cache and the address bit categorization of four lookup addresses (`0x5439, 0x1270, ..`).



Two kinds of memory are used in the diagram above, fast/expensive and slow/cheap.

☑ On the diagram above show which blocks are fast and which blocks are slow.

The blocks are labeled in blue. The Memory block is labeled slow. (If it weren't slow there would be no need for a cache.) Note that both the Data Store and Tag Store must use fast memory.

Suppose that the cache is initially cold (there is nothing in the cache). Show the outcome, hit or miss, of each of the four lookups.

☑ Show outcome, hit or miss, on diagram above.

The outcomes appears in the diagram above, in blue.

Find the addresses requested below.

☑ After the four lookups, what is the smallest address that will hit the cache.

The smallest address is `0x1270`.

☑ After these four lookups, what is the largest address that will hit the cache.

Answer: The largest address is `0x643f`.

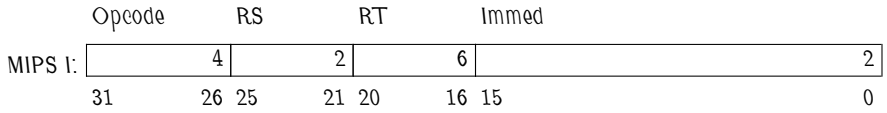
Explanation: The largest *lookup* address is `0x6439`. Since it is the last of the four lookup addresses it will surely be in the cache after the four lookups are complete. Each cache miss, including the one for lookup address `0x6439`, brings in a line's worth of data. The starting address of a line is found by setting the offset and align bits (bits 3 to 0 here) to zero. For `0x6439` the line starting address is `0x6430`. The last, or largest, address in a line can be found by setting all of the offset and align bits to 1. That yields the answer to the question, `0x643f`.

(*b*) Show the encoding for the `beq` and `lw` as used in the code below. Be sure to include the immediate value.

```
 addi r6, r0, 10
 beq r2, r6, SKIP
 lw r1, 4(r3)
 add r1, r1, r5
SKIP:
 and r9, r9, r1
```

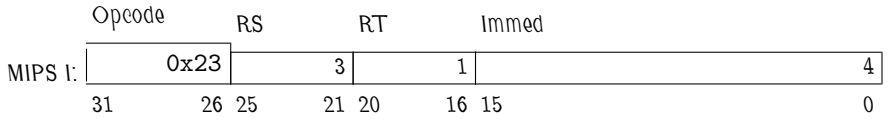
☑ Encoding of `beq`. ☑ Be sure to show a value for the immediate field.

The encoding is shown below. Though the solution shows the opcode of `beq`, $100_2 = 4$, full credit would be received for an answer that showed `beq` or something like that in the opcode field.



☑ Encoding of `lw`. ☑ Be sure to show a value for the immediate field.

The encoding is shown below. Though the solution shows the opcode of `lw`, $10011_2 = 23_{16} = 35_{10}$, full credit would be received for an answer that showed `lw` or something like that in the opcode field.



(*c*) Answer the following about ISA families.

☑ Which style of implementation are RISC ISAs designed for?

RISC ISAs are designed for pipelined implementations.

☑ Which style of implementation are VLIW ISAs designed for?

VLIW ISAs are designed for multiple issue (sort of like superscalar) implementations.

(*d*) Some early RISC ISAs omitted useful magnitude-comparison branch instructions such as `bgt r1, r2, TARG` in which the branch is taken if `r1 > r2`. As branch prediction became more common magnitude-comparison branch instructions were added to RISC ISAs. One might argue that with branch prediction the cost and performance impact of magnitude-comparison instructions was lower.

☑ Explain how the cost of implementing `bgt` is lower with branch prediction than without.

Because without branch prediction it is likely that the branch would be resolved in the `ID` stage and so would require the use of a magnitude comparison unit used only for resolving branches. With branch prediction the magnitude comparison could be done later, in `EX`, and could be done using the ALU, and so no extra magnitude comparison unit would be needed.

☑ Explain how the performance impact of implementing `bgt` is lower with branch prediction than without.

Without branch prediction the magnitude comparison would likely be done in the `ID` stage. That magnitude comparison could not start until the register values were retrieved from the register file, and the time to do both might lengthen the critical path, and so lower performance. With branch prediction the register values would be retrieved in one cycle `ID`, and the comparison would be done in the next cycle, `EX`, assuming something like the MIPS five-stage implementation used in class.

45 Spring 2021 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Friday, 26 March 2021 to Monday, 29 March 2021 16:00 CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(40 pts)
Exam Total	_____	(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \;\wedge\; r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [30 pts] One instruction that might have come in handy for Homework 2 is the proposed `lbit`, load bit, instruction. Consider `lbit r1, (r2..r3)`. This instruction will load a single bit from memory into `r1`. Register `r2` holds a base address and `r3` holds a bit offset. The bit offset is relative to the most-significant bit of the byte at address `r2`. So if `r3` is zero the MSB is loaded into `r1`. If `r3` is 7 the LSB of the byte at `r2` is loaded into `r1`, if `r3` is 8 the MSB of the byte at `r2+1` is loaded into `r1`, etc. (As with Homework 1 and 2, bit ordering is big-endian.) To help understanding `lbit` there are two code fragments below. They do the same thing, the first uses `lbit`, the second uses existing MIPS instructions.

```
# Proposed Instruction
lbit r1, (r2..r3)

# Equivalent MIPS Code
sra r9, r3, 3
add r9, r2, r9
lbu r1, 0(r9)
sll r1, r1, 24
andi r9, r3, 0x7
sllv r1, r1, r9
srl r1, r1, 31
```

(*a*) Modify the illustrated MIPS implementation so that it implements `lbit`, omitting control logic. Assume that the memory port will be set to perform a read byte unsigned operation (the same operation as would be performed for the `lbu` instruction) and the ALU will be set to perform an add operation. (That is, don't assume or try to add new operations for the memory port nor for the ALU.) The modifications should provide the appropriate address to the memory port and should place the appropriate bit in the destination register.

As always, assume that the critical path is through the memory port. For this problem it is okay to put additional non-control logic in the `WB` stage.

☑ Add logic to compute the correct load address.

☑ Add logic to extract the needed bit.

☑ There is no need to show control logic.

☑ Don't assume or implement new Mem Port or ALU operations.

☑ It's okay to add logic to the `WB` stage.

☑ Pay attention to performance.

☑ Pay attention to cost. ☑ Do not show functional units that are more complicated than necessary. ☑ Use existing pipeline latches and other data carrying paths when possible.

☑ As always, do not break other instructions.

The solution appears on the next page.

The solution to parts a, b, and c appears below in blue and part-c-specific (**ebit**) changes appear in green.

For **lbit** assume that control logic is set initially for an **lbu** instruction. To actually implement **lbit** some control logic would need to be changed, but that's not part of this problem. In the **EX** stage we need to change how the address is computed. An **lbu** would compute **rsv+IMM**, but for **lbit** we need **rsv+rtv/8**. The added logic computes **rtv/8** and connects the value to a new input on the lower ALU mux. Note that **rtv/8** is computed by effectively shifting to the right by 3 bits while sign extending. (With sign extension **rtv** can be negative.) Note that the shifting itself uses no hardware, it just relabels bit positions. The only added hardware in **EX** is the new mux input.

Note that the value of **rtv** is taken from the output of the mux used by store instructions. By doing so we can get bypassed values without having to add new hardware. Bypassed values are needed for part b.

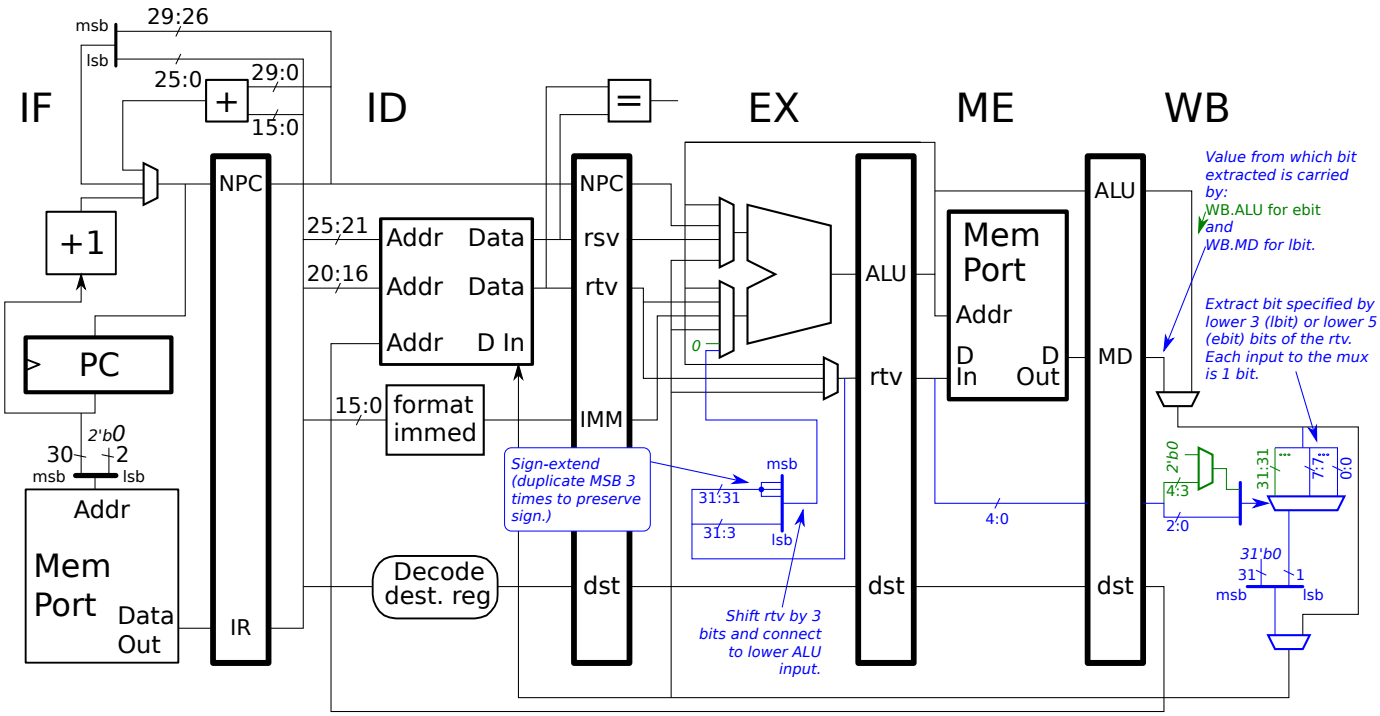
In **WB** a mux is used to extract the needed bit, it will be called the *bit-extract mux* in this discussion. (That's the mux with more than two inputs.) For **lbit** the mux needs 8 inputs, each of one bit, connected to the 8 LSB of **WB.MD**. The hardware shown, rather than connecting directly to **WB.MD**, connects to the mux that selects either **WB.MD** or **WB.ALU**. This is needed for part c. Also, for part c the bit extract mux has 32 inputs.

The select signal for the bit-extract mux is taken from the LSB of **rtv**. In the existing hardware **rtv** only makes it as far as **ME**. So for this problem a new pipeline latch is added carrying the 5 LSB of **rtv** to **WB**. For part a only the 3 LSB are used, for part c all 5 bits are used.

The output of the bit-extract mux is just 1 bit. Thirty-one zeros are appended to make a 32-bit quantity.

For part c, in which **ebit** is implemented, there are several differences with the **lbit** implementation. First, we need to deliver the **rsv** to **WB**. To do so without requiring a new ALU operation, a new zero input to the lower mux has been added. The ALU will perform an add operation and use the zero input, so the output of the ALU will be **rsv**. In **WB** the **rsv** is in **WB.ALU**, and as mentioned earlier, it has a path to the bit extract mux. The select signal needs to be 5 bits for **ebit**, so a mux is used to set the 2 MSB of the bit-extract mux's select signal: to zero for **lbit** (valid values are 7-0) and for **ebit** use bits **4:3** of **rtv** so we can get a range of 31-0 at the select signal.

Common Mistakes: One common mistake was to use the equivalent MIPS code as a blueprint for the hardware. That equivalent code was provided to be clear on what the **lbit** instruction should do, not how it should do it. So a solution with a box for each instruction in the equivalent code, such as a four shift-unit boxes for **sra**, **sll**, **sllv**, and **srl** would be very wasteful.



(*b*) Show the execution of the code fragments below on your implementation. Add reasonable bypass paths to eliminate stalls.

☑ Add reasonable bypass paths to avoid stalls that would be suffered by the code below.

☑ Show execution of each code fragment (with reasonable bypass paths).

The executions appear below, with highlighting used to emphasize registers carrying dependencies.

```
## SOLUTION
# Fragment A        0  1  2  3  4  5  6  7
addi R3, r3, 1      IF ID EX ME WB
lbit R1, (r2..R3)      IF ID EX ME WB
add r4, r4, R1            IF ID -> EX ME WB

## SOLUTION
# Fragment B        0  1  2  3  4  5  6
lbit R1, (r2..r3)   IF ID EX ME WB
addi r3, r3, 1         IF ID EX ME WB
add r4, r4, R1            IF ID EX ME WB

## SOLUTION
# Fragment C        0  1  2  3  4  5  6  7  8  9
lbit R1, (r2..r3)   IF ID EX ME WB
bne R1, r0  TARG       IF ID ----> EX ME WB
addi r3, r3, 1            IF ----> ID EX ME WB

TARG:
 xor r8, r9, r10                  IF ID EX ME WB
 # Cycle:           0  1  2  3  4  5  6  7  8  9
```

(*c*) Consider another instruction `ebit`, extract bit. Consider `ebit r1, r2, r3`. This instruction extracts the bit at position `r3` from `r2` and writes it to `r1`. The MSB is at position 0. The bit position is in the least significant five bits of `r3`, other bits of `r3` are ignored.

Both `lbit` and `ebit` extract a bit from a value, so it is possible to use some of the hardware for `lbit` to implement `ebit`. One difference is that `lbit` extracts an 8-bit quantity while `ebit` extracts a bit from a 32-bit quantity. If the hardware were shared, the `lbit` hardware would have to be upgraded to handle 32-bit values.

Ignoring whether such sharing really is a good idea, modify the implementation of `lbit` so that it could implement `ebit` using hardware shared with `lbit`.

☑ Modify MIPS hardware to implement `ebit` using hardware shared with `lbit`.

☑ No need to show control logic.

The solution to this part is shown and discussed several pages back on the page showing the hardware changes.

(*d*) Explain why an implementation sharing `ebit` and `lbit` hardware would execute the code fragment below slowly and describe a faster alternative.

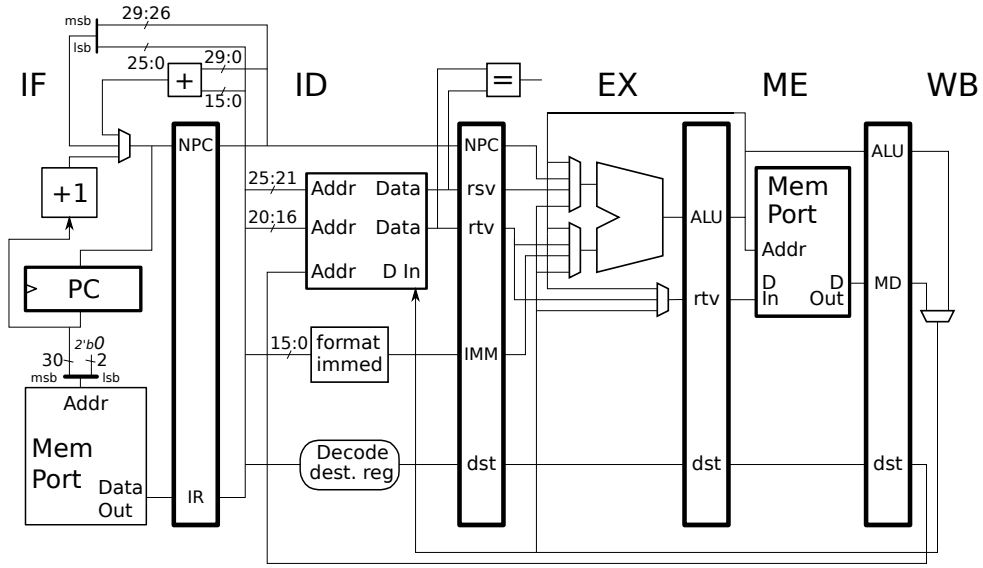
```
ebit r1, r2, r3
add r4, r4, r1
```

☑ Why does the shared hardware implementation slow code below?

☑ Why is an implementation of `ebit` that is similar to other computation instructions faster?

It is slower because the result is computed in **WB**. This will force a dependent instruction, such as the **add** in the example above to stall one cycle. If a bit-extraction mux for **ebit** had been placed in **EX** then it would be possible to bypass in the example above without a stall.

Problem 2: [15 pts] Consider the pointer-chasing loop below. Assume that the loop executes many iterations on the illustrated hardware.



(*a*) Show an execution of the loop below for enough iterations—at least two—to compute the IPC (inverse of CPI). The IPC is the number of executed instructions divided by the number of cycles. Compute it for a very large number of iterations.

☑ Show execution.

☑ Compute IPC for a large number of iterations.

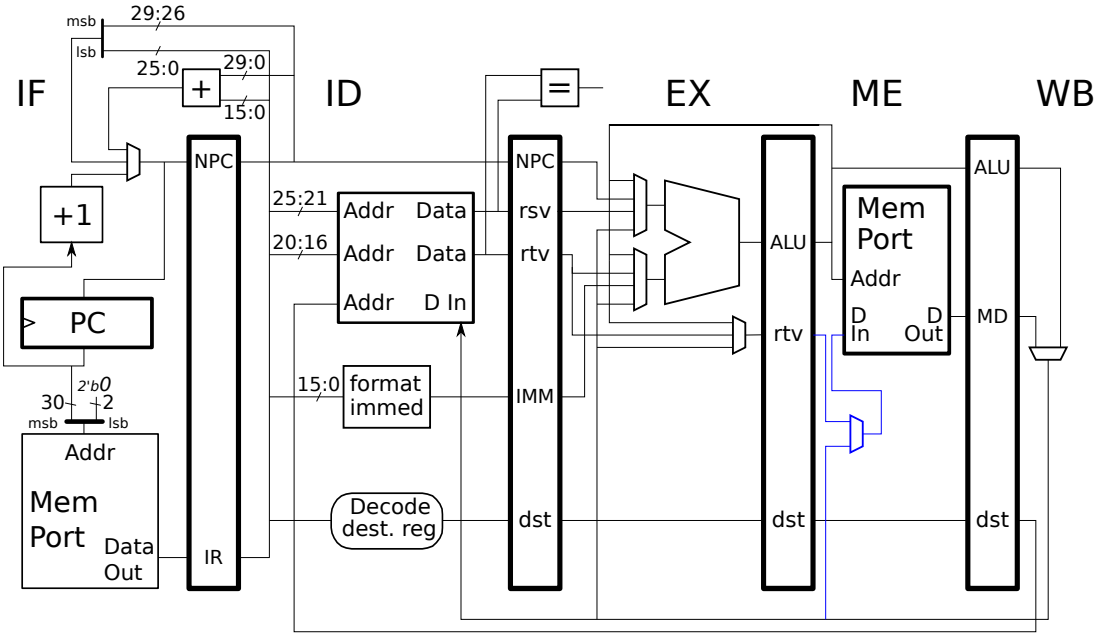
☑ Check for dependencies and available bypass paths.

The execution appears below. The first iteration starts in cycle 1, the second starts in cycle 7, so the time for an iteration is $7 - 1 = 6$ cycles. (The start time is when the first instruction is in `IF`.) Since there are 4 instructions in the loop body the IPC is $\frac{4}{6} = \frac{2}{3}$ insn/cycle.

Common Mistakes: One common mistake is forgetting the delay slot instruction (`lw r3`).

```
## SOLUTION
 lw r3, 8(r3)       IF ID EX ME WB                                  BEFORE LOOP
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)          IF ID -> EX ME WB                            FIRST ITERATION
 sw r1, 0(r3)             IF -> ID -> EX ME WB
 bne r1, r5 LOOP                IF -> ID EX ME WB
 lw r3, 8(r3)                         IF ID EX ME WB
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)                            IF ID -> EX ME WB       SECOND ITERATION
 sw r1, 0(r3)                               IF -> ID -> EX ME WB
 bne r1, r5 LOOP                                  IF -> ID EX ME WB
 lw r3, 8(r3)                                           IF ID EX ME WB
       # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

 add r5, r3, r9                                                  AFTER LOOP
```

(*b*) If the previous part were solved correctly, then there should be two stalls per iteration. One stall could be eliminated by a bypass path, but the other could not (without increasing critical path). For each stall in your execution (even if there are more or less than two) show a reasonable bypass path that would avoid the stall or else explain why such a bypass is not reasonable.



☑ Show reasonable bypass paths needed to avoid stalls on your code.

☑ For each stall that could not be eliminated with a bypass path, explain why:

As stated in the problem, the code suffers two stalls. The first stall is due to a `lw/lw` dependence. On the execution above it ends in cycle 3 in the first iteration and 9 in the second iteration. (A stall ends at the arrow head in the diagram.)

The second stall is due to the `lw/sw` dependence. That ends at cycle 5 in the first iteration.

To examine which bypass paths are possible consider the stall-less, hypothetical, **not necessarily correct** execution below.

```
## WARNING: HYPOTHETICAL EXECUTION WITHOUT STALLS. WILL NOT RUN CORRECTLY.
 lw r3, 8(r3)       IF ID EX ME WB                                  BEFORE LOOP
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)          IF ID EX ME WB
 sw r1, 0(r3)             IF ID EX ME WB
```

Consider first the `lw r3` / `lw r1` stall. In the hypothetical stall-less execution above the `lw r1` needs the value of `r3` in the beginning of cycle 3 when it is in `EX`. But at that time `lw r3` is just starting `ME` and so the value has not been loaded. So there is no reasonable way to bypass the value.

Next, consider the `lw/sw` dependence, carried by `r1`. The `sw` needs `r1` at the beginning of its time in the `ME` stage in cycle 5. At that time the `lw` is in `WB`, and so it is possible to bypass it. That bypass path is shown in blue. It has been pointed out that the Mem Port is assumed to be on the critical path. But the critical path is from the `Addr` input to the `D Out` output. We can assume that there is some slack on the `D In` input, meaning that it can be delayed for a short time, which is why the mux is not a problem. It would be a problem to put a mux before the Mem Port `Addr` input or after the Mem Port `D Out` output.

Problem 3: [15 pts] Appearing below are two candidate MIPS instructions, `jca`, *jump case add*, and `jcc`, *jump case concatenate*, that can be used to implement C-style `switch` statements. The instructions are designed for case statements that each consist of up to eight instructions. In both instructions the `rs` register (register `r1` in the examples) holds the address of case statement zero. Case statement 1 starts at address `r1+32`, case statement 2 starts at address `r1+32*2`, etc. The `rt` register (`r2` in the examples) holds the number of the case statement to jump to, so the address to jump to is `r1+32*r2`. The only difference between the two instructions is that in `jca` the value of `r1` must be a multiple of 4 (since instruction addresses are aligned) while in `jcc` the value of `r1` must be a multiple of 4096 (the 12 least-significant bits must be zero) and `r2` must be less than 128. Like other MIPS control transfers, both instructions have a 1-instruction delay slot. Note that `jca r1, r0` is equivalent to `jr r1`.

The code below shows the use of `jca` and an equivalent code fragment that uses only existing MIPS instructions.

```
# Candidate Instruction
jca r1, r2   # Jump to r1 + r2 * 32
nop

# Another Candidate Instruction
jcc r1, r2   # Jump to { r1[31:12], r2[6:0], 5'b0 }
nop

# Equivalent code to jca (and partly jcc) using existing MIPS instructions.
sll r9, r2, 5
add r9, r9, r1
jr r9
nop
```

A resolve-in-`ID` implementation of `jcc` can be designed at low cost and with no risk of lengthening the critical path. In contrast, a resolve-in-`ID` implementation of `jca` would add to cost and risk critical path impact.

(*a*) Show the datapath changes to the MIPS pipeline on the next page needed for resolve-in-`ID` implementations of the two instructions.

☑ Show datapath changes (not control logic) for resolve-in-`ID` implementation of ☑ `jca` and ☑ `jcc`.

☑ As always, pay attention to cost and performance.

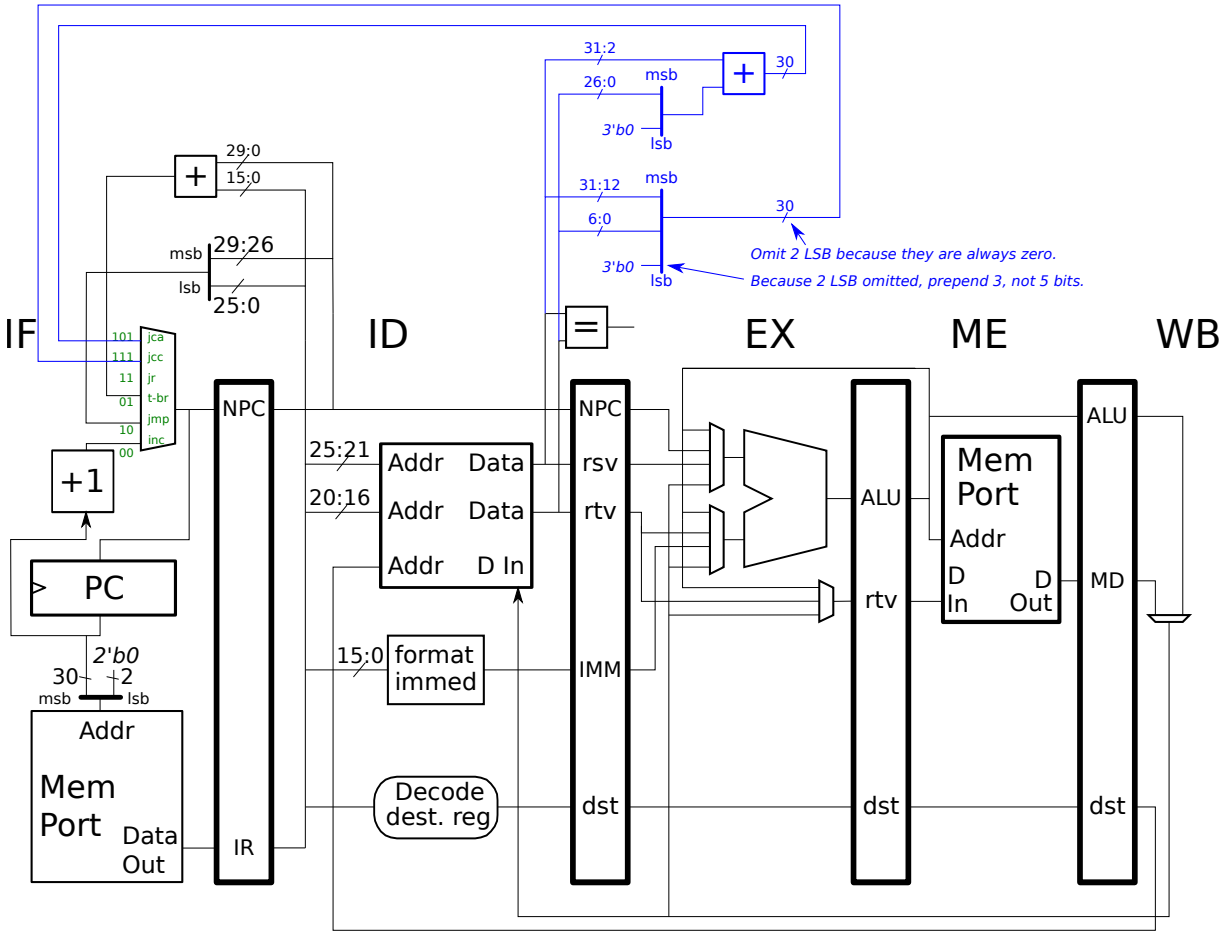
The solution and a discussion of the solution appears on the next page.

(*b*) Explain why computing a branch target, which is done using an adder, has no critical path impact while there is critical path impact for `jca`.

☑ Why can a branch safely use an adder in `ID`, but not `jca`?

Because the inputs to the adder for the branch are available at the beginning of the clock cycle, in contrast the inputs to the adder for `jca` are available later, after they are retrieved from the register file. Because of this one cannot automatically assume that there is enough time for the `jca` adder to compute its sum.

Solution to part a appears below in blue. Note that since the PC holds only 30 bits, it is not necessary to compute the two LSB of the target. (They would always be zero anyway.) For that reason 3 bits, instead of 5 bits, are prepended to the `rt` register value.



Problem 4: [40 pts] Answer each question below.

(*a*) MIPS branches have one delay slot. That enables five-stage scalar MIPS implementations to fetch the delay-slot instruction while resolving the branch. So, is the delay slot a feature of the ISA or a feature of the implementation?

☑ Is a delay slot an ISA feature or an implementation feature? ☑ Explain.

It's an ISA feature because it describes what instructions do. That is, the feature says that the instruction after a branch executes whether or not the branch is taken. Because of this ISA feature some implementations, including our five-stage MIPS pipeline, execute branches with zero penalty. But for others, such as superscalar pipelines (which will be covered soon), a delay slot adds to complexity and provides little benefit. Because it is an ISA feature, a superscalar implementation must execute the one delay slot instruction, even if a different number of delay slots would have made more sense.

(*b*) There are 32 MIPS integer (general-purpose) registers, usually called `r0` to `r31`. But these registers are also given names, which are shown in the table below. Suppose we wanted to rearrange the names. For example, suppose we wanted to name register `r16 t8` (instead of name `r24 t8`) and make `r24` the new `k0`. Which registers could we rearrange without changing the ISA? It must be possible to use the registers for the purpose suggested by their names after rearranging.

```
Names     Numbers Suggested Usage
$zero:    0       The constant zero.
$at:      1       Reserved for assembler.
$v0-$v1:  2-3     Return value
$a0-$a3:  4-7     Argument
$t0-$t7:  8-15    Temporary (Not preserved by callee.)
$s0-$s7: 16-23    Saved by callee.
$t8-$t9: 24-25    Temporary (Not preserved by callee.)
$k0-$k1: 26-27    Reserved for kernel (operating system).
$gp      28       Global Pointer
$sp      29       Stack Pointer
$fp      30       Frame Pointer
$ra:     31       Return address.
```

☑ Which register numbers can get new names without having to change the ISA? ☑ Explain.

The ISA dictates special behavior for `r0` (it's always zero) and for `r31` (the `jal` instruction writes it with return address). There is no special behavior for the other registers so they can be renamed. That is, there's no problem with making `r1-r4` the argument registers and `r5-r6` the return value registers.

Common Mistake: Some incorrectly supposed that the important-sounding registers such as `sp` and `at` could not be renamed. When it comes to the ABI (which includes rules for how registers are used when making procedure calls) `sp` is important, but no more important than `t0` or `s0`. But there is nothing about the MIPS ISA that makes `r29` a good choice for the stack pointer (`sp`), any other register except `r0` and `r31` could be chosen.

(*c*) The code fragment below adds 1 to the floating-point value in register `f2` and puts the sum in `f3`.

```
addi $t1, $0, 1    # Integer 1
mtc1 $t1, $f1
cvt.s.w $f1, $f1
add.s $f3, $f1, $f2
```

☑ Explain what the `cvt.s.w` instruction does in the code above.

The instruction converts the value in the `fs` register (`f1` in the example) from a signed integer to a single-precision floating-point value and writes the result in the `fd` register (also `f1` in the example, but it could be different than the source).

☑ Re-write the code so that it adds a 1, but does so without a `cvt` instruction. *Note: The original exam used the wording "without a* `cvt.s.w` *instruction.".*

```
## SOLUTION
lui $t1, 0x3f80   # Note: An IEEE 754 Single 1.0 = 0x3f800000
mtc1 $t1, $f1
add.s $f3, $f1, $f2
```

(*d*) In early RISC ISAs, including MIPS I, floating-point registers were 32 bits and yet many of these ISAs had double-precision (64-bit) floating-point instructions. Where do these instructions find their 64-bit operands?

☑ MIPS-I gets 64-bit floating-point operands . . .

. . . from a pair of registers consecutive. The instruction specifies the even-numbered register of the pair. Consider `add.d f0, f2, f8`. The first source is in registers `f2` and `f3`, the second source is in `f8` and `f9`. Registers `f0` and `f1` are written with the result.

(*e*) ARM A64 and RISC-V RV64 are both late RISC ISAs. But ARM A64 has many more instructions than RISC-V. How does having more instructions help A64 and fewer instructions help RISC-V?

☑ Lots of instructions help A64 because . . .

They enable programs to be written with fewer instructions. This reduces program size and presumably reduces execution time and execution energy. (Certainly reduced execution time and energy were a goal, since A64 is successful we presume that these goals have been reached in many A64 implementations.)

☑ Fewer instructions help RISC-V because . . .

. . . it was designed for teaching research purposes and having fewer instructions would reduce the difficulty of carrying out investigations. (Though intended for teaching and research an important design goal is that it be complete enough to be used in practical applications.)

Some pointed out that RISC-V could have fewer instructions in its base version, say RV64I, because its modular design enabled additional instructions to be added only by those who needed them. Even so, RISC-V is still much simpler than A64. Bit extraction instructions are still in draft form, and there is nothing like A64's scaled indexing. Though not 100% correct, an answer along these lines would get full credit.

Some pointed out that RISC-V was intended for embedded applications where low hardware cost is important and easier to achieve with fewer instructions. That's not a bad argument, but it ignores several contributors to cost: the core itself (such as our pipeline), program storage, and execution energy. With a smaller instruction set comes lower-cost cores. But programs will be longer, requiring more storage and potentially more energy executing them. That's only a problem if the code for the embedded application is beyond a certain size and if energy is an issue.

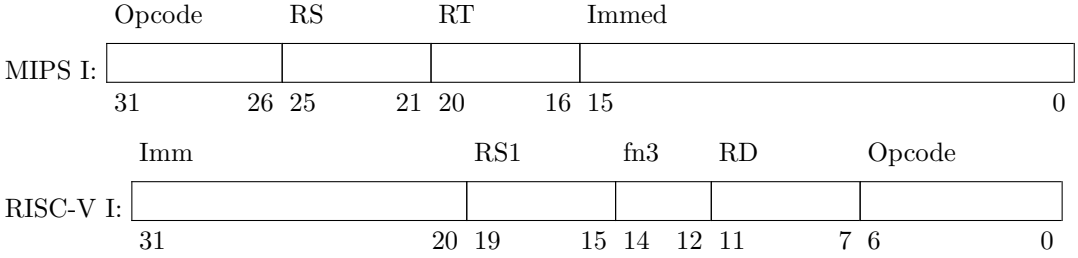
(*f*) Explain the problem with this statement:

Implementations of CISC ISAs were slow because of complex instructions. Only later did computer engineers discover that with simpler RISC ISAs implementations could be made faster.

☑ The statement is misleading or incorrect because . . .

. . . it ignores that the implementation technology used when CISC ISAs were developed was much more costly, so pipelined designs (for which RISC ISAs were intended) were out of the question and memory could not be wasted. For that reason an implementation of a RISC ISA would be slower than an implementation of a CISC ISA back then. Those complex instructions helped reduce redundant work (such as reading and writing registers or memory). In a non-pipelined implementation complex instructions are not particularly difficult to implement. For example, and arithmetic instruction could retrieve its operands from memory, compute arithmetic on them, and store them back in memory. There's no problem with complex memory addressing modes, because a single ALU could be used as many times as needed.

(*g*) Appearing below are MIPS and RISC ISAs' immediate formats.



☑ What advantage does the MIPS format have?

☑ Show an example of a MIPS instruction that could not be encoded in the RISC-V format.

The MIPS format has a larger immediate field and so it can be used with larger immediate values. For example, the MIPS instruction `addi r1, r2, 0x1234` has no RISC-V counterpart. That is `addi a1, a2, 0x1234` is not a RISC-V instruction because the immediate would not fit.

☑ What advantage does the RISC-V format have? (Another question on this exam implies this advantage is wasted.)

There are more opcode bits in the RISC-V format, a total of 10 and so based on this format there can be more format-I opcodes than in MIPS.

(*h*) Compilers optimize by scheduling (rearranging) instructions to avoid stalls due to true dependencies. In that case, why do we need to have bypass paths?

☑ Bypass paths are needed despite optimizations because:

Because it is not always possible to find an instruction to put between a dependent pair to avoid a stall. This is particularly difficult in code with frequent branches. Suppose, for example, one out of six instructions were a branch. Then scheduling could only easily be done with about five instructions, making it difficult to find instructions to move.

Name Solution

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Tuesday, 27 April 2021 to Friday, 30 April 2021 16:00 (4 PM) CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. For example, don't Web-search the text of a problem unless the problem specifically allows it. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman. **Suspected violation of these rules will be reported to the Dean of Students as a violation of the Student Code of Conduct.**

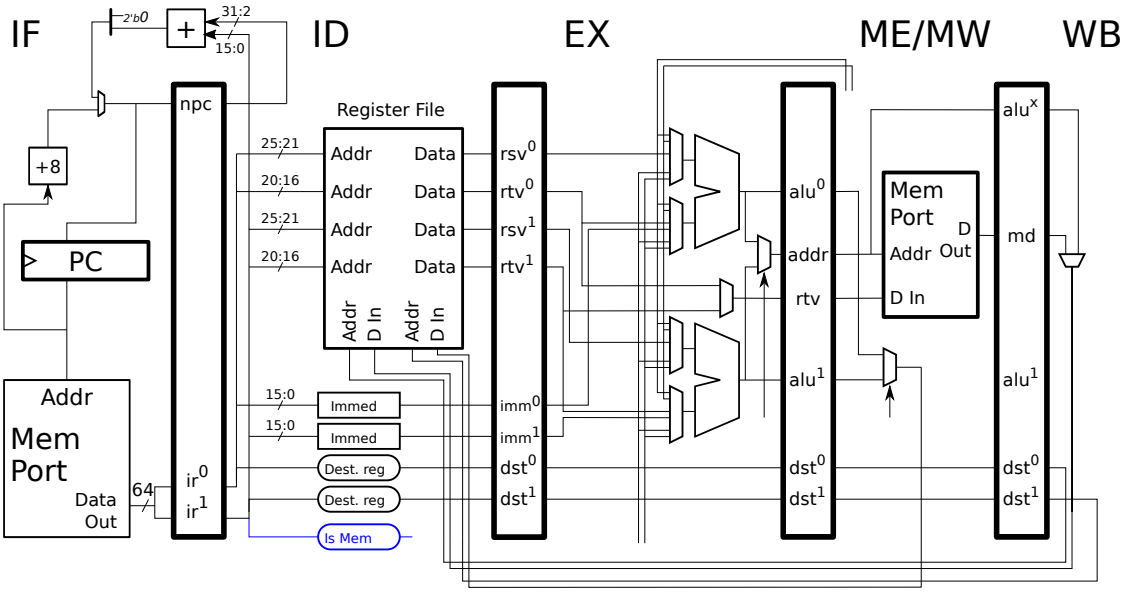
Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \;\wedge\; r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Thank you for your effort in EE 4720!

Problem 1: (25 pts) Appearing below (and larger a few pages ahead) is an idea for a 2-way superscalar MIPS implementation with an unorthodox feature: The register file is written both by an instruction in the `ME/MW` stage (the former `ME` stage) and by an instruction in the `WB` stage. A memory instruction would have to write back in the `WB` stage, but a non-memory instruction could write back in either stage. Consider the execution below. For the first pair, `lw` and `addi`, the `lw` must use `WB` and the `addi` must use `MW`. The situation is similar for the second pair except that the memory instruction, `lb`, is in slot 1 rather than slot 0. When neither slot holds a memory instruction (the pair fetched in cycle 2) either one (but not both!) could use `MW`. If both use the memory port, the later one should stall. In execution diagrams label `MW` is used by an instruction that writes back in that stage, and `ME` is used by an instruction that will write back in the `WB` stage.

Put solution on diagram several pages ahead!



Put solution on diagram several pages ahead!

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID EX ME WB
addi r2, r2, 4   IF ID EX MW
sub r3, r4, r5      IF ID EX MW
lb r7, 5(r2)        IF ID EX ME WB
and r9, r10, r11       IF ID EX ME WB
or r12, r13, r14       IF ID EX MW
lb r16, 0(r7)             IF ID EX ME WB
lh r17, 2(r7)             IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

This idea has a potential cost benefit, but it must be thought through because the order in which registers are written can vary. (Which is a scary thing to those worried about correctness.) One cost benefit can be seen in the diagram. The `WB.alu1` pipeline latch is no longer needed. The label is still there but it is not connected (and won't be). Other cost-saving changes are part of the subproblems below.

☑ For all parts of this problem remember to pay attention to cost and performance. ☑ For example, don't connect a bypass path that will never be needed.

(*a*) The `D In` connections to the write ports of the register file have been changed, but the register number inputs, `Addr`, are the same. Modify the hardware so that the `Addr` inputs to the write ports get the correct register number. For this part don't worry about two instructions writing the same register number.

☑ Modify hardware (next page, not above) so that `Addr` inputs of the register file write ports get the correct register number.

Solution shown in green. If `mem1` is true then the slot 0 instruction writes in `MW` and the slot 1 instruction writes in `WB`. New multiplexors wrote the signals appropriately. Note that since `dst1` is not needed in `WB` it is not sent to `WB`.

(*b*) The diagram shows one cost savings: the `WB.alu1` pipeline latch is no longer needed. Notice that the four bypass connections to the `EX` stage are unconnected. Reconnect them as needed so that any dependency that could be bypassed in the unmodified superscalar can be bypassed here. Leave a bypass unconnected if not needed.

☑ Re-connect bypass paths (on next page, not above), possibly adding or modifying other hardware to bypass values.

Solution shown in purple. Note that because only one value is written back in `WB` only one bypass path is needed from `WB`. The bypass values from `ME` are taken from the pipeline latches, `ME.alu0` and `ME.alu1`, which is good because their values are available at the beginning of the clock cycle.

(*c*) Notice that there is an unconnected select signal in `EX` and `MW`. Design control logic for these and for any multiplexors used to provide the correct register numbers (the first subpart above). The Is Mem logic in `ID` should be helpful. *Hint: There is not that much to do for this part. The Is Mem block should come in handy.*

☑ Connect select signals in `EX` and `MW`.

☑ Connect select signals for any multiplexors used for register numbers.

Solution appears in turquoise. Fortunately, that Is Mem logic block provides the exact signal needed for the select inputs.

Grading Note: some solutions, by also considering the case in which a memory instruction is in slot 0, were more complicated than they needed to be. There is no need to consider whether a memory instruction is in slot 0 because if there is not a memory instruction in slot 1 the execution will be correct whether or not there is a memory instruction in slot 0. (And what if there's a memory instruction in both slots? Should I answer that now (11 May 2021, 17:50:37 CDT) or should I make this a problem on the 2022 final exam?)

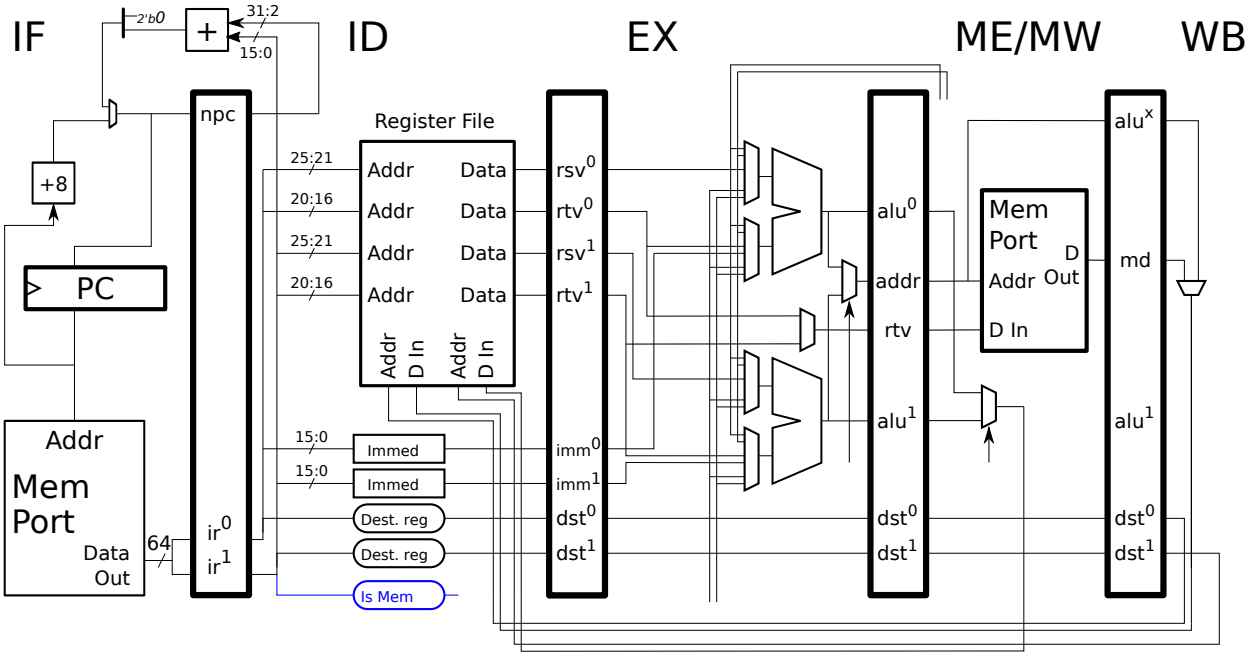
(*d*) Notice that in the execution below the `and` writes `r9` after the `or`. It looks like the `add` instruction will get the value written by the `and` rather than the `or`. That's not right!

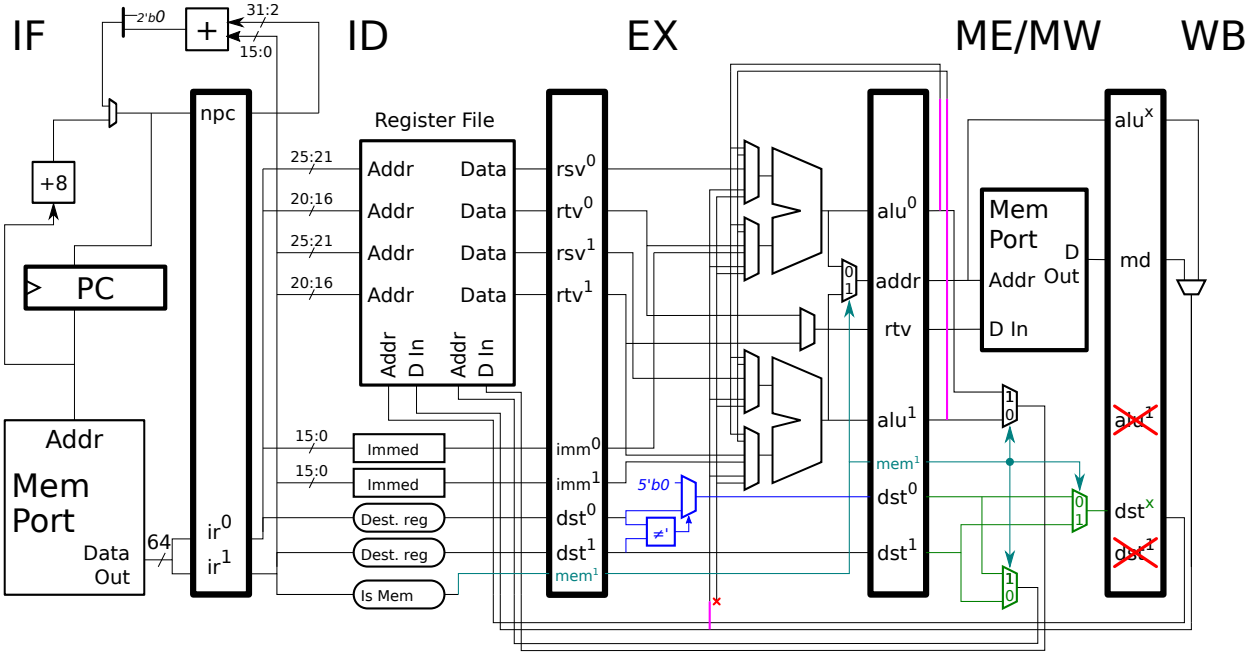
```
# Cycle            0  1  2  3  4  5 ... 9  10 11 12 13
and r9, r10, r11   IF ID EX ME WB
or r9, r7, r14     IF ID EX MW
# .. later..
add r1, r1, r9                          IF ID EX MW WB
```

☑ Add control logic to detect such WAW hazards and which will substitute `r0` for the destination of the earlier instruction.

Solution shown in blue. If both registers are the same the slot-0 register is changed to zero. The comparison unit is put in the **EX** stage to provide more time. (Okay, I'll be honest, I didn't want to squeeze it in.)

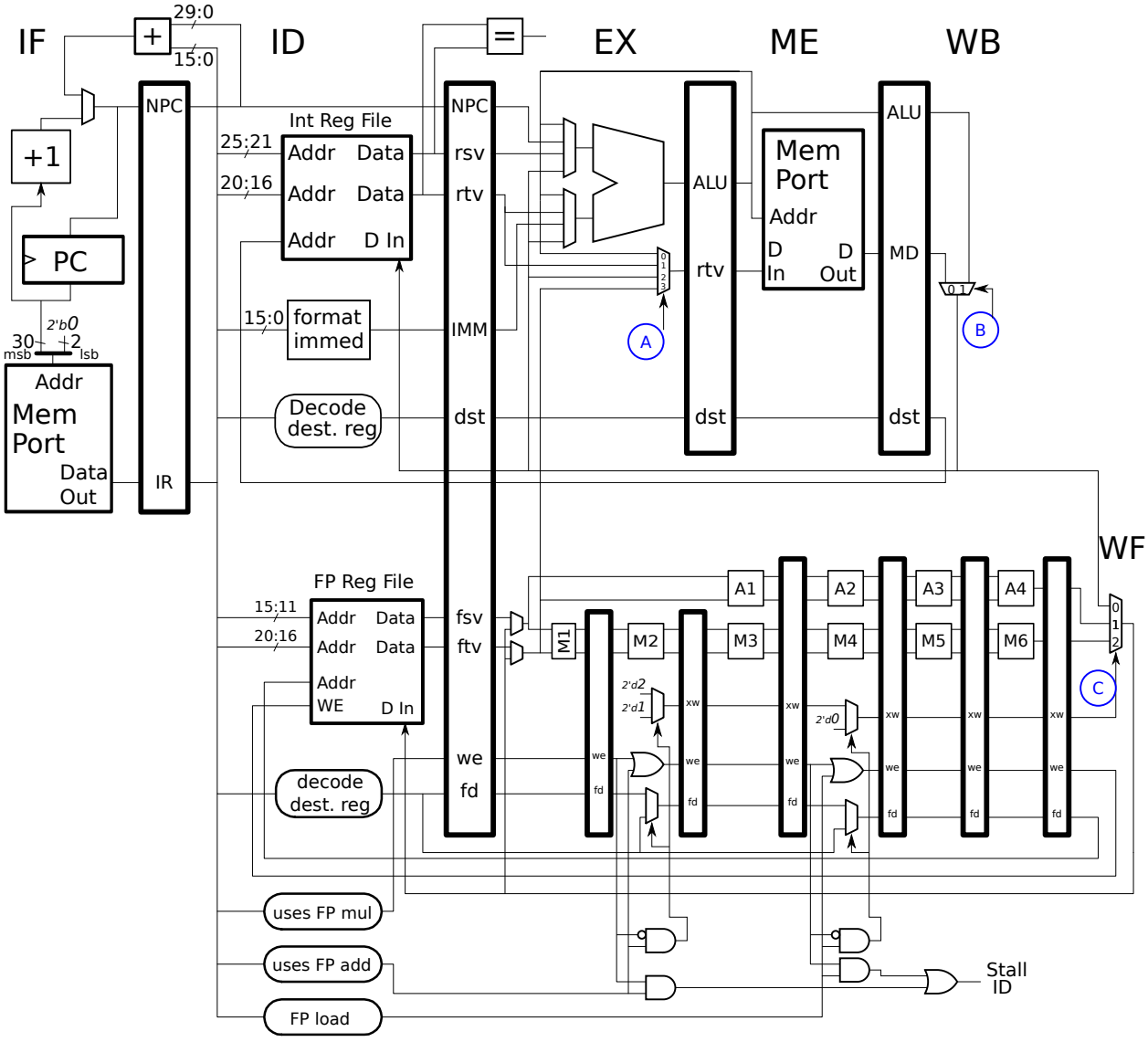
Use your favorite SVG or plain text editor on the SVG source for the implementation `https://www.ece.lsu.edu/ee4720/2021/fe-ss-px.svg` and logic gates `https://www.ece.lsu.edu/ee4720/2021/g.svg`.





Problem 2: (25 pts) Show the execution of the code fragments as requested below.

(*a*) The MIPS implementation below is similar to the one used in class, but with some added bypass paths for FP instructions and some labeled multiplexor select signals.



Appearing on the next page is a code fragment, and above the code fragment are the labels `A`, `B`, and `C`. These labels correspond to those used in the implementation.

Show the execution of the code below on this pipeline long enough to determine the IPC for a long number of iterations. Of course, that means the branch is taken. Show the value of each labeled select signal in those cycles it is being used.

☑ Show the execution on the illustrated implementation.

Solution appears below. The `add.s` stalls one cycle waiting for the loaded value. The `swc1` stalls until the `add.s`, writing `f3`, reaches `WF` and then bypasses the value to the input of `ME.rtv`.

The following reminders are based on common mistakes made on the exam: Don't forget that since MIPS has delay slots the `addi` executes. Also don't forget that since the branch is taken the `xor` should not be executed. Based on the diagram the branch resolves in `ID`, meaning that in the next cycle, when the branch is in `EX`, the target will be in `IF`. That the branch resolves in `ID` can be seen by noting that the branch target input to the `PC` mux originates from the `ID` stage.

☑ Show the values of the labeled select signals, `A`, `B`, and `C` when they are in use.

The values are also shown below. Those values are only shown for cycle in which they are needed.

Signal `A` routes the correct store value for a store instruction in `EX`. In the execution below `swc1` is in `EX` in cycle 9 and its store value, specified by `f3`, is being written by the `add.s` which is in `WF`. Input 3 connects to the value in `WF` (admittedly taking a long path). `A` is only used in cycle 9 (below) and so a blank is shown for the other cycles. In real life `A` can have any value at those other cycles and execution would still be correct.

Signal `B` selects the value from the integer pipeline that will proceed to `WB` or `WF`. In cycles 4 and 5 (and again in 14) when the two loads write back `B` is zero to select the output of the memory port. In cycle 13, when `addi` is in `WB`, `B` is 1 to select the output of the ALU. Nothing is written back in the other cycles and so they are shown blank. (The `swc1` and `bne` reach the `WB` stage but they don't write a register.)

Signal `C` selects the value to be written to the FP register file. In cycles 4, 5, and 14, the value is 0 so that the load value has a path from the integer pipeline. In cycle 9 the `add.s` reaches `WF`, and the value of `C` is 1 to select the output of the `A4` functional unit.

☑ Find the IPC for a large number of iterations.

The instruction throughput, or IPC, is computed by dividing the number of instructions by the number of cycles. The number of instructions is 6. The number of cycles is based on the fetch of the first instruction of the loop (the `lwc1 f1`). That's at cycles 0 and 10, so the number of cycles for an iteration is 10. The instruction throughput is then $\boxed{\frac{6}{10-0} = 0.6\,\text{insn/cycle}}$.

Grading Note: In many—too many—answers the number of cycles is computed incorrectly. Just remember to use a common reference point on the two iterations, usually the fetch of the first instruction. This way the time for one iteration does not overlap the time for the next.

```
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14    SOLUTION
 A                                                3
 B                              0  0                        1  0
 C                              0  0              1            0
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f1, 0(r1)     IF ID EX ME WF                                 # 1st Iter
 lwc1 f2, 4(r1)        IF ID EX ME WF
 add.s f3, f1, f2         IF ID -> A1 A2 A3 A4 WF
 swc1 f3, 8(r1)              IF -> ID -------> EX ME WB
 bne r1, r2  LOOP                  IF -------> ID EX ME WB
 addi r1, r1, 12                               IF ID EX ME WB
 xor r5, r1, r6
LOOP:  # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lwc1 f1, 0(r1)                                   IF ID EX ME WF  # 2nd Iter
```

(*b*) There should have been stalls in the execution of the code above. Re-write the code so that it executes with as few stalls as possible and still computes the same result. It is okay to add extra instructions before and after the loop. For convenience assume that the code executes for at least two iterations. But don't unroll the loop.

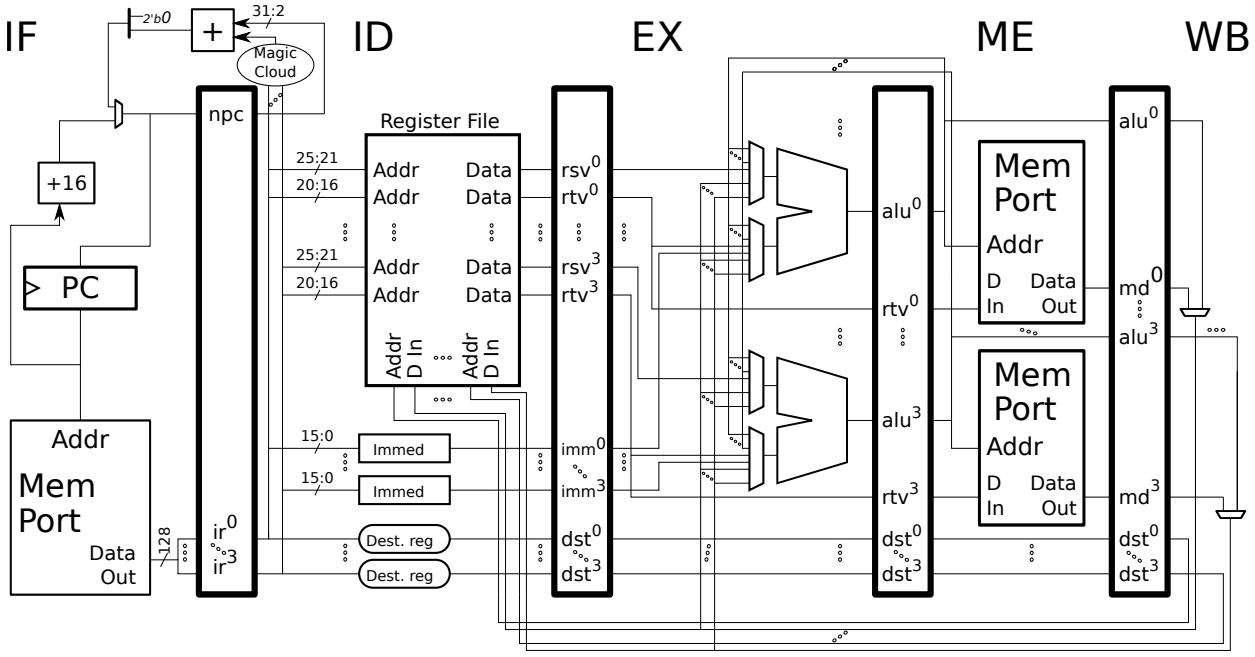
☑ Re-write code to avoid stalls. ☑ Code must compute the same values. ☑ Can re-arrange instructions, change registers, and put instructions before the start of the loop.

The solution appears below. The code is shown, followed by the execution of the code shown in dynamic instruction order.

To avoid the `swc1` stall the `swc1` stores the value computed in the *previous iteration* (or by the prologue code in the first iteration). The `swc1` is put between the second `lwc1` and the `add.s` eliminating the load/use stall. Also note that the offsets on the stores have changed because they are loading a value that will be needed for the next iteration. The first iteration has stalls, but that's due to the prologue code. The second and subsequent iterations execute stall-free.

```
 # Solution Code
 #
 lwc1 f1, 0(r1)
 lwc1 f2, 4(r1)
 add.s f3, f1, f2
LOOP:
 lwc1 f1, 12(r1)
 lwc1 f2, 16(r1)
 swc1 f3, 8(r1)
 add.s f3, f1, f2
 bne r1, r2  LOOP
 addi r1, r1, 12
 xor r5, r1, r6

 # Solution Code Execution (Shown in dynamic instruction order.)
 #
 lwc1 f1, 0(r1)      IF ID EX ME WF
 lwc1 f2, 4(r1)         IF ID EX ME WF
 add.s f3, f1, f2          IF ID -> A1 A2 A3 A4 WF
 # Loop
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lwc1 f1, 12(r1)           IF -> ID EX ME WF
 lwc1 f2, 16(r1)                 IF ID -> EX ME WF
 swc1 f3, 8(r1)                     IF -> ID EX ME WB
 add.s f3, f1, f2                         IF ID A1 A2 A3 A4 WF
 bne r1, r2  LOOP                            IF ID EX ME WB
 addi r1, r1, 12                                IF ID EX ME WB
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
 lwc1 f1, 0(r1)                                    IF ID EX ME WF
 lwc1 f2, 4(r1)                                       IF ID EX ME WF
 swc1 f3, 8(r1)                                          IF ID EX ME WB
 add.s f3, f1, f2                                           IF ID A1 A2 A3 A4 WF
 bne r1, r2  LOOP                                              IF ID EX ME WB
 addi r1, r1, 12                                                  IF ID EX ME WB
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
 lwc1 f1, 0(r1)                                                      IF ID EX ME WF
```

(*c*) Appearing below is a 4-way superscalar MIPS implementation. In this implementation fetch is not aligned (which makes things easier). Also, there is no branch prediction, which is how we have been doing things in class.



☑ Show the execution of the code below for enough iterations to determine IPC. (Note: There is no need to put slot numbers on the stage labels.) ☑ Don't forget that it is 4-way superscalar.

Solution appears below. To keep instructions in order in `ID`, the stall of one instruction in `ID` stalls all instructions ahead in `ID`. So, in cycle 1 and 2 the `add` is stalling for the `lw` value. That forces the `sw` and `addi` to stall too, even though the `bne` and `addi` are not waiting for anything.

The branch resolves in `ID` and so the target is not fetched until the branch is in `EX`, resulting in the squash of six instructions. Just because it would be nice to fetch the branch target in cycle 7, does not mean the hardware can actually do so.

The instruction throughput is $\frac{6}{8-0} = .75$ insn/cycle.

```
 # SOLUTION
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lw r10, 0(r1)       IF ID EX ME WB
 add r3, r10, r3     IF ID ----> EX ME WB
 sw r3, 0(r5)        IF ID -------------> EX ME WB
 addi r5, r5, 4      IF ID -------------> EX ME WB
 bne r1, r9, LOOP       IF -------------> ID EX ME WB
 addi r1, r1, 4         IF -------------> ID EX ME WB
 lb r8, 0(r9)           IF -------------> IDx
 xor r11, r8, r10       IF -------------> IDx
 some insn                                IFx
 some insn                                IFx
 some insn                                IFx
 some insn                                IFx
LOOP:  # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 lw r10, 0(r1)                               IF ID EX ME WB
 add r3, r10, r3                             IF ID ----> EX ME WB
```

(*d*) The code fragment below is to execute on the same 4-way superscalar MIPS implementation. Notice that loop body in the code fragment below contains two copies of the loop body from the loop in the previous subproblem. So one iteration of the loop below does the work of two iterations of the loop from the previous problem. This is the first step in the application of a technique called *loop unrolling*. The loop from the previous problem has been unrolled by *degree 2*. The next step is to re-arrange the instructions, and possibly eliminate those that are no longer needed. Complete this step, of course without changing what the code does. Finally, to eliminate all stalls *software pipelining* will need to be used: values computed used in one iteration will have to come from instructions executed in the prior iteration.

☑ Re-write the loop above so that it runs more efficiently on the 4-way MIPS implementation. ☑ Use fewer instructions, even if doing so does not help with the degree 2 unroll, in the expectation that it might be beneficial at higher unroll degrees.

The solution appears below. A single load/add/store calculation is spread across three iterations. For example, suppose `lw r10` loads something in iteration x. The `add r19` and `add r3` instructions will operate on that `r10` in iteration $x+1$ and the value computed by the `add r3` and the `add r13` instructions in iteration $x+1$ will be written by the `sw` instructions in iteration $x+2$. This technique is called *software pipelining*. The prologue code starts up the process. As can be seen from the execution, this avoids all stalls.

An iteration takes just 4 cycles now, compared to 8 cycles in the original code. Also, each iteration computes twice as many values, and so the code runs 4 times faster than the original.

```
 # SOLUTION - Code
 #
 # Prologue
 lw r10, 0(r1)
 lw r12, 4(r1)
 add r3, r10, r3
 add r13, r12, r3
 lw r10, 8(r1)
 lw r12, 12(r1)
 #
 # Main Loop
LOOP:
 add r19, r10, r12
 sw r3, 0(r5)
 sw r13, 4(r5)
 add r3, r13, r10
 lw r10, 16(r1)
 lw r12, 20(r1)
 add r13, r13, r19
 addi r5, r5, 8
 bne r1, r9, LOOP
 addi r1, r1, 8
 #
 # Post-Loop Code
 lb r8, 0(r9)
 xor r11, r8, r10
```

```
 # Solution Code Execution (Omitting prologue)
 #
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
 add r19, r10, r12    IF ID EX ME WB
 sw r3, 0(r5)         IF ID EX ME WB
 sw r13, 4(r5)        IF ID EX ME WB
 add r3, r13, r10     IF ID EX ME WB
 lw r10, 16(r1)          IF ID EX ME WB
 lw r12, 20(r1)          IF ID EX ME WB
 add r13, r13, r19       IF ID EX ME WB
 addi r5, r5, 8          IF ID EX ME WB
 bne r1, r9, LOOP           IF ID EX ME WB
 addi r1, r1, 8             IF ID EX ME WB
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
 add r19, r10, r12                IF ID EX ME WB
 sw r3, 0(r5)                     IF ID EX ME WB
 sw r13, 4(r5)                    IF ID EX ME WB
 add r3, r13, r10                 IF ID EX ME WB
 lw r10, 0(r1)                       IF ID EX ME WB
 lw r12, 4(r1)                       IF ID EX ME WB
 add r13, r13, r19                   IF ID EX ME WB
 addi r5, r5, 8                      IF ID EX ME WB
 bne r1, r9, LOOP                       IF ID EX ME WB
 addi r1, r1, 8                         IF ID EX ME WB
LOOP:  Cycle          0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: (25 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor and the other system has a local predictor with a 10-outcome local history.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  T  T  T  N  T  N    T  T  T  N  T  N    T  T  T  N  T  N   <- Outcome
     1  2  3  4  5  6    1  2  3  4  5  6    1  2  3  4  5  6   <- Outcome Pos.

BA:

B2:   N  N  N ...  N  N  T  T  T    N  N  N ...  N  N  T  T  T
      1  2  3     11 12 13 14 15    1  2  3     11 12 13 14 15  <- Outcome Pos.
```

☑ What is the accuracy of the bimodal predictor on branch B1?

The accuracy is $\boxed{\frac{4}{6}}$. The work is shown below. The accuracy is based on the repeating pattern, shown underlined below.

```
   SOLUTION WORK
   0   1  2  3  2  3   2   3  3  3  2  3   2   3  3  3  2  3    <- 2b Counter
B1:  T  T  T  N  T  N    T  T  T  N  T  N    T  T  T  N  T  N   <- Branch Outcome
     x  x     x     x            x     x            x     x   <- Pred. Outcome
                     --------------------                     <- Repeat
```

☑ What is the accuracy of the local predictor on B1 ignoring B2.

The 10-outcome history is longer than the period of B1, six outcomes, and so the accuracy is 100%.

☑ What is the accuracy of the local predictor on B2 ignoring B1.

In B2 there are twelve consecutive Ns, and so a local history of 10 Ns, that is, NNNNNNNNNN, will occur three times: when predicting positions 11, 12 and 13. The outcome of positions 11 and 12 will be N, decrementing the counter, and the outcome at 13 will be T, incrementing the counter. After warmup the counter will be 0 or 1 and so position 13 will be mispredicted. The overall accuracy will be $\frac{14}{15}$. *B2 in the original exam had 11 consecutive* N*s, and in that case the accuracy would likely be* $\frac{13}{14}$ *but in the worse case could be* $\frac{12}{14}$.

☑ What is the longest local history size for which branch B1 and branch B2 will interfere with each other on the local predictor? (The question is for a local predictor, not a global predictor.)

Five outcomes. Local history NTTTN precedes a T in B1 and a N in B2. Any B1 local history of length 6 would have to contain at least four Ts, no local history of B2 can contain four Ts, so there can be no interference with a six outcome local history.

(*b*) If branch B1 at position 6 is mispredicted `T` (taken) then data at address $a + x$ will be brought into the cache. An adversary would like to learn the value of x. To do so the adversary, running in another process on the same core as the process using B1, will force a misprediction of 6. Then the adversary will make careful timing measurements of loads to addresses starting at a to determine what address was loaded, and so learning x.

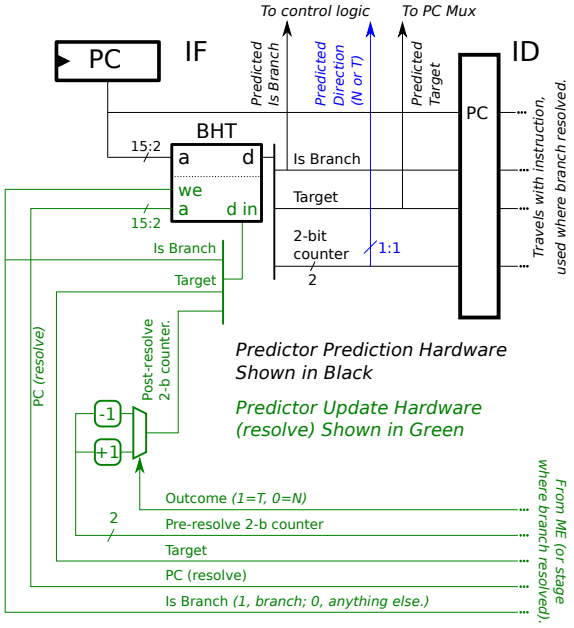
Branch BA is part of the adversary's code. Find a pattern for BA that will force B1 to be mispredicted by the 10-outcome local predictor at position 6 (but not at other positions). The misprediction does not need to occur every time position 6 in B1 is executed.

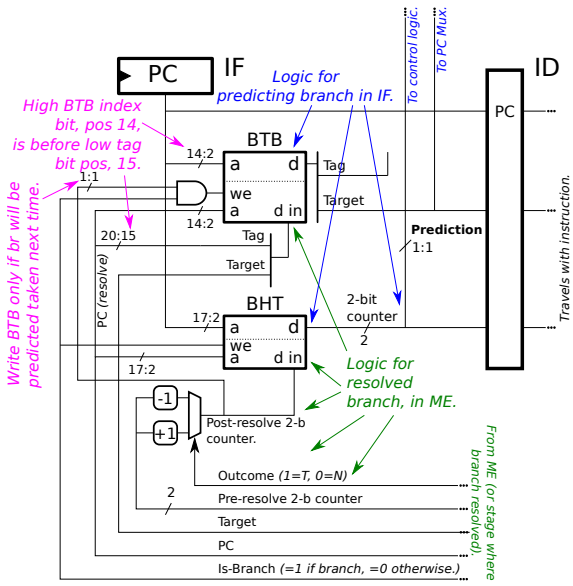
☑ Pattern for BA that forces misprediction of B1 at position 6 on 10-outcome local predictor.

For B1 outcome 6 to be mispredicted `T` branch B2 must have a matching local history that precedes an outcome of `T`, and that pattern must occur more frequently. The B1 local history when predicting outcome 6 is `TTNTN TTTNT` (the space is for readability). One choice for BA would be the pattern repeating `TTNTN TTTNT T`, but that can be shortened to `NTN TTTNT T`. To induce a misprediction BA must occur more frequently than B1. In the timing shown below two repeats of the BA pattern occur between each pattern of B1. (There is no reason why the branch outcomes of B1 and BA needed to be neatly interleaved with each other.)

```
B1:  T  T  T  N  T  N                        T  T  T  N  T  N   <- Outcome
     1  2  3  4  5  6                        1  2  3  4  5  6   <- Outcome Pos.

BA:                     NTNTTTNTTNTNTTTNTT                     NTNTTTNTTNTNTTTNTT

     Outcomes of BA shown below for clarity, shown above with intended timing.
BA:  N  T  N  T  T  T  N  T  T    N  T  N  T  T  T  N  T  T
     1  2  3  4  5  6  7  8  9    1  2  3  4  5  6  7  8  9
                        -------------------------------     <- Local History
```

(*c*) Below on the left is a plain bimodal predictor as first described in class. Below on the right is a refined version that makes better use of storage by splitting the BHT into two tables, a *branch target buffer*, holding the target and a *tag*, and a BHT holding only the two-bit counters. The tag field replaces the `IsBranch` field and is used like a cache tag. (Though a cache tag would include bits `31:15`.) The control logic will compare the tag retrieved from the BTB with bits `20:15` of the PC, if they match a prediction will be made using the 2-bit counter from the BHT, if the tag doesn't match it is assumed that the instruction is not a branch. See 2017 Homework 8 Problem 3 for additional explanation.





The memory address and behavior of two branches from two programs, A and B, are shown below. One program runs better on the plain predictor than the BTB predictor, the other program runs better on the BTB predictor. Identify which is better, and why, as requested below.

```
# Program A
B1: 0x9000:  T T T ...
B2: 0x1000:   T T T ...
```

☑ Prediction of branches in A better on ⊗ *Plain* or ◯ *BTB* predictor. ☑ Explain why other predictor does worse on this program. ☑ Pay attention to address of branches.

For Program A the two branches, B1 and B2, differ in bit position 15 and so they would use different entries in the plain predictor's BHT. For that reason the plain predictor would correctly predict them.

On the BTB predictor the same BTB entry would be used for B1 and B2 since their addresses are identical at bit positions 14:2. So, when predicting B2 the BTB would return a tag for B1. Because the tag would not match no prediction would be made. (And, if a prediction were made the target address would be wrong since it's the target of B1.) Something similar happens when B1 is predicted.

```
# Program B
B3: 0x11000:  N N N T  N N N T  N N N T    ...
B4: 0x21000:   T T T N  T T T N  T T T N    ...
```

☑ Prediction of branches in B better on ◯ *Plain* or ⊗ *BTB* predictor. ☑ Explain why other predictor does worse on this program. ☑ Pay attention to address of branches.

The addresses of branches B3 and B4 are identical in bit positions 15:2 (and so also 14:2) but differ in bits 17:2. A prediction of B3 in the plain predictor's BHT will retrieve the 2-bit counter and target address of B4, and so there will possibly be a misprediction. B4 could similarly be mispredicted by the plain predictor. In a worst case both would be mispredicted.

In the BTB predictor branches B3 and B4 will use separate BHT entries, so their 2-bit counters will be entirely their own. The BHT is always written, but the BTB is only written if the branch would be predicted taken. For that reason only branch B4 is written to the BTB. So, when B3 arrives there will be a miss in the BTB (the tag won't match), and so the predictor will assume that the instruction being fetched is either not a branch or a not-taken branch. For B3 that will be correct three out of four times. When B4 is predicted the BTB will hit. The 2-bit counter from the BHT will be 3 or 4 and so the branch will be predicted taken. That will be correct 3 out of 4 times, and the target will also be correct.

Problem 4: (25 pts) Answer each question below.

(*a*) A program runs with fewer stalls on an 2-way superscalar than on an 8-way superscalar statically scheduled processor. Both have the same clock frequency and are otherwise comparable.

☑ Does that mean the program runs faster on the 2-way processor? ☑ Explain.

No, of course not. Consider a stall of one instruction (the **sub** in the example below) waiting for a value computed by another (the **add** in the example below). Because the pipeline is the same depth (five stages) on both the 2-way and 8-way systems (because they are otherwise comparable), the number of cycles it takes for the value to be bypassable will be the same on both. That is, on both systems the **sub** can bypass the value two cycles after the **add** is in **ID**. There is no stall in the 2-way system because the **sub** is fetched one cycle later.

```
# Execution on 2-way. No Stall!
add R1, r2, r3     IF ID EX ME WB
xor r9, r10, r11   IF ID EX ME WB
sub r4, R1, r5        IF ID EX ME WB
or r12, r13, r14      IF ID EX ME WB
# Cycle            0  1  2  3  4  5

# Execution on 4-way. No slower!!
add R1, r2, r3     IF ID EX ME WB
xor r9, r10, r11   IF ID EX ME WB
sub r4, R1, r5     IF ID -> EX ME WB
or r12, r13, r14   IF ID -> EX ME WB
```

(*b*) Shorten the code fragments below.

☑ Shorten code fragment.

```
addi r1, r0, 0x4755
sll r1, r1, 16
addi r1, r1, 0x4720
lw r1, 0(r1)
```

Solution appears below. Many forgot that they could use an offset in the load.

```
# SOLUTION
lui r1, 0x4755
lw r1, 0x4720(r1)
```

☑ Shorten code fragment.

```
lw r3, 0(r1)
addi r2, r0, 4
add r1, r1, r2
lbu r1, 0(r1)

# SOLUTION
lw r3, 0(r1)
lbu r1, 4(r1)
```

(*c*) Appearing below are three code fragments. These are to run on a machine with vector instructions and four-lane vector units. Indicate whether each fragment can easily be replaced by a vector instruction, and the reason why or why not.

☑ Fragment below ◯ *can* or ⊗ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
mul.s f7,  f8,  f9
sub.s f10, f11, f12
```

The code performs three different operations, **add**, **mul**, and **sub**. A vector instruction performs the same operation on multiple sets of operands (bit positions within vector registers).

☑ Fragment below ◯ *can* or ⊗ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s F1,  f2,  f3
add.s f4,  F1,  f6
add.s F7,  f8,  f9
add.s f10, F7, f12
```

Typical vector instructions do not support dependencies like that. Certainly not arbitrary dependencies.

☑ Fragment below ⊗ *can* or ◯ *cannot* be replaced by a vector instruction. ☑ Explain.

```
add.s f1,  f2,  f3
add.s f4,  f5,  f6
add.s f7,  f8,  f9
add.s f10, f11, f12
```

The code sequence can easily be replaced by a vector instruction.

(*d*) Let's break something! As we know, an implementation of an ISA must execute a program as defined in the ISA. The Broeken Company is going to sell a 4-way superscalar statically scheduled five-stage "implementation" of ARM A64, call that the Broeken4 implementation. (There is no space between the Broeken and the 4. That's so you don't notice the 4.) We know how smoothly our five-stage scalar MIPS implementation handles branches: zero penalty because of the delay slot. But A64 branches don't have a delay slot. That didn't stop the Broeken Company from changing things a bit. In the Broeken4 implementation the number of delay slots for a branch can vary from 4 to 7. If a branch is in slot 0 of its fetch group there are 7 delay slots. Branches in slots 1, 2, and 3 have 6, 5, and 4 slots, respectively. If the branch is not stalled by dependencies there will be zero branch penalty. This chip performs about 20% better than the (rule-abiding) competition on performance and energy efficiency benchmarks.

Computer engineering professors obviously will hate the Broeken company for ignoring what an implementation of an ISA should do. But what about others? Gauge the reaction of each group below.

Note: If this were an in-class exam questions would be shorter.

☑ Compiler writers will be ◯ *positive, happy, and supportive* or ⊗ *negative, irritated, and obstreperous* . ☑ Explain.

It will be difficult to fill all those delay slots. Depending on personality type, putting in `nop` instructions will make compiler writers feel like failures or make them feel like they are being set up to take the blame for an unworkable idea (the large number of delay slots).

☑ Buyers of laptops and desktops using Broeken4 will be ◯ *positive, satisfied* or ⊗ *negative, seeking a refund* . ☑ Explain.

They will be negative because most software won't run. They expected their new computer to run their old software.

☑ Engineers using Broeken4 in embedded devices, such as the controller chip in a microwave oven, will be ⊗ *positive, satisfied* or ◯ *negative, seeking a new supplier* . ☑ Explain.

Software is usually custom written for an embedded system, and so compatibility is not an issue. They will be happy because of the performance and efficiency improvements.

(*e*) The execution below is taken from an illustration of how exceptions work from a class lecture. The `lw` raises an exception in cycle 4 and in response the handler starts in cycle 5. The first instruction that the handler executes is `sw`. It is likely that the handler is saving registers to the stack before attending to the event causing the exception. Presumably the handler will save every register that it plans to modify (or to be safe, all registers). To reduce the number of registers saved, why not split the work between the interrupted code and the handler. As with the ABI rules for procedure calls, why not have the interrupted code save the caller-save registers it wants preserved (`t0-t9`, etc.) so that the handler would only need to save the callee-save registers it plans to overwrite (`s0-s7`, etc.)?

```
# Cycle:                0  1  2  3  4  5 ...          99 100 ...
        add  r1, r2, r3  IF ID EX ME WB
        lw   r6, 0(r1)      IF ID EX ME*x             IF ID EX ME WB
        or   r5, r6, r7        IF ID EXx                 IF ID EX MEx
        xor  r10, r11, r12        IF IDx                    IF ID EXx
        and  r20, r21, r22           IFx                       IF IDx
...
Handler:
        sw ...                          IF ...                      IF ...
        ...
        eret (exception return)                 IF ID EX ME WB
```

☑ Why can't interrupted program save the caller-save registers before the handler starts, potentially reducing work?

Short answer: This is an absurd idea! Where would all that register save code be placed? Surely the effort needed to find it when there was an exception or to jump over it when there wasn't will be much greater than any reduction in the number of registers saved.

Explanation: For caller-save registers to be saved, routines to save those registers need to be written and put somewhere. It would be no problem for a compiler to write such code. The problem is a register-save routine would need to be written for every instruction that could raise an exception, which would be a huge number. Putting such register-save routines before each instruction would be a waste for instructions like loads, because they only raise exceptions infrequently, but the register save code would be executed each time, and for nothing. Or, some scheme could be developed to either locate a register save routine for each exception raising instruction, or to skip over such code if it were placed just before the instruction.

Also, instructions in the register save code itself could raise exceptions.

46 Spring 2020 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Tuesday, 14 April 2020 to Friday, 17 April 2020 23:59 CDT

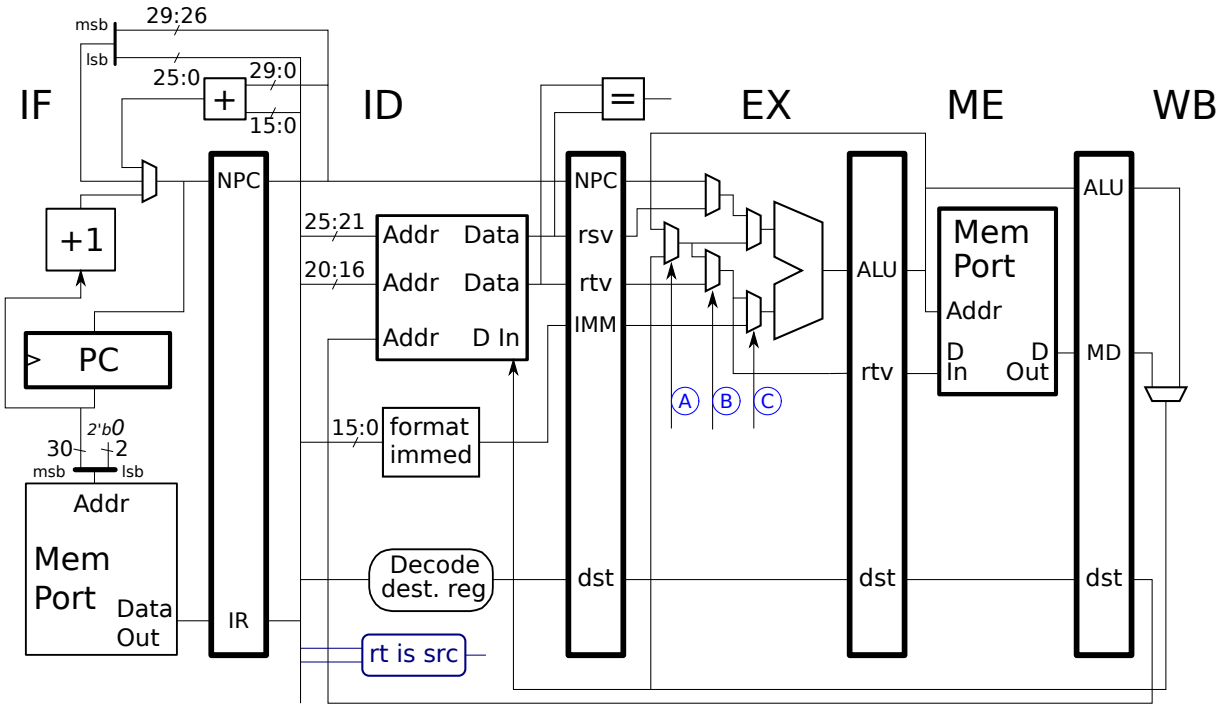
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(15 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Don't be Foolish!

Problem 1: [15 pts] The pipeline below is a slightly lower cost version of the bypassed MIPS implementation that we've been using. The cost saving is achieved by not allowing an instruction to use a bypassed value from both the ME and WB stage, the value must come from one stage or the other. Select inputs are shown for three of the re-done EX stage multiplexors, they are labeled A, B, and C. For this problem assume that they are connected to properly designed control logic.



Problem continues on next page.

(*a*) Show the values on the labeled select signals for an execution of the code below for those cycles in which an instruction below is in the `EX` stage. If the value on a select signal does not matter, show an X.

☑ Show values of A, B, and C for when `EX` occupied by code below. ☑ Use X if value does not matter, blank when no insn in `EX`.

Solution appears below.

The following is an explanation to help those studying. There is no need to provide such a long-winded answer on an exam. The A select signal is set to 1 when a bypass from `WB` is needed and is set to 0 when a bypass from `ME` is needed. The `add` instruction does not use a bypassed value, so A is shown as X (meaning it could be either 0 or 1) in cycle 2 (when `add` is in `EX`). The `sub` bypasses from `ME` so A is 0 in cycle 3 and `sw` bypasses from `WB` in cycle 4 so A is 1.

The B select signal is 0 if the `rt` source is bypassed. Only the `sub` bypasses an `rt` source, so B is 0 in cycle 3, the other instructions use the value from the register file so B is 1 in cycles 2 and 4.

The C select signal is 1 if the immediate is needed at the lower ALU input. That is only true for the `sw`, where the store memory address is computed by adding the immediate, 8, to the `r1` value. The `sw` instruction uses the `rt` value too, but that's the store data which is delivered to `D In`.

```
#    Cycle        0     1     2     3     4     5     6
 add r1, r2, r3   IF    ID    EX    ME    WB
 sub r4, r5, r1         IF    ID    EX    ME    WB
 sw r6, 8(r1)           IF    ID    EX    ME    WB

#    Cycle        0     1     2     3     4     5     6     7 SOLUTION
 A                        X     0     1
 B                        1     0     1
 C                        0     0     1
#    Cycle        0     1     2     3     4     5     6
```

(*b*) Show a code fragment that would stall on the implementation above but would not stall on our usual bypassed MIPS (which appears in Problem 3).

☑ Code fragment that stalls on this implementation, but not our usual 5-stage MIPS.

Solution appears below. The `xor` instruction uses the result of both the `addi` and `or`. The execution is for "our usual bypassed MIPS," where both values can be bypassed and so no stall is needed.

```
 # SOLUTION
 # Cycle          0  1  2  3  4  5  6   Note: Execution is for usual bypassed MIPS.
 addi R1, r2, 3   IF ID EX ME WB
 or   R4, r5, r6     IF ID EX ME WB
 xor  r7, R1, R4        IF ID EX ME WB
```

Problem 2: [25 pts] Appearing below is the lower-cost MIPS implementation from the previous problem. Design the control logic specified below. The output of rt is src is 1 if the `rt` field of the instruction specifies a source value, as it does in most type R but only a few type I, such as `sw`. The Inkscape SVG source for the image below can be found at `https://www.ece.lsu.edu/ee4720/2020/mt-p1.svg`.

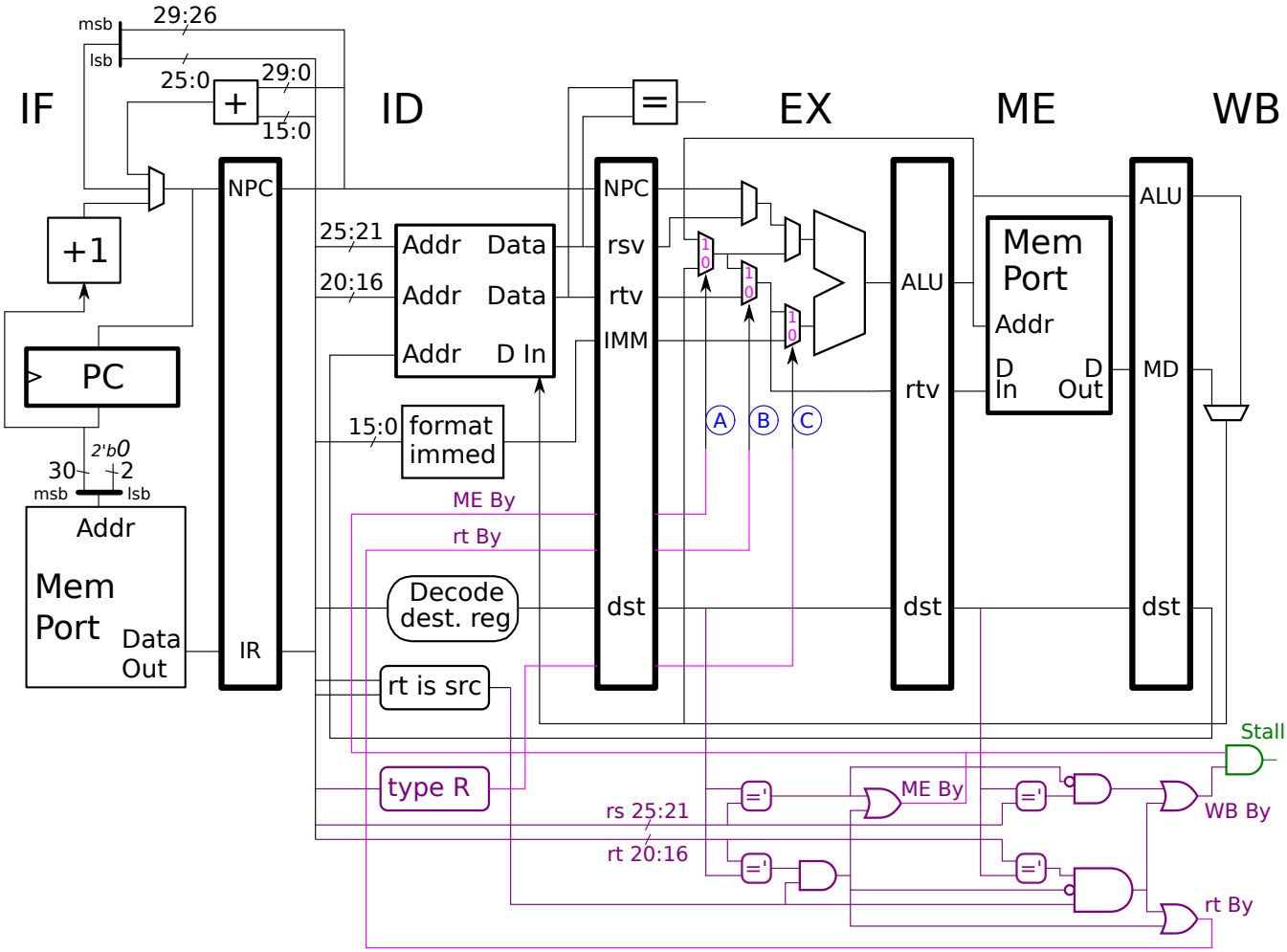
☑ Design control logic for the labeled multiplexor select signals, A, B, and C.

☑ Design control logic to generate a stall signal when a bypass would have been from both `ME` and `WB`.

☑ Pay attention to the usual stuff: ☑ Cost and critical path. ☑ The stage that instructions are in when the select signals are computed and the stage in which they are used.

Solution appears below. Logic for the select signals is shown in two shades of purple (two shades to help emphasize longer wires). Signal `ME By` is 1 if there is a dependence between the instruction in `ID` and the instruction in `EX`. It is used **in the next cycle** for select signal A. Signal `rt By` is 1 if the `rt` register of the instruction in `ID` is a source and it depends on either the instruction in `EX` or `ME`. It is used for B in the next cycle. Signal C is easiest, it is 1 if the instruction is a type R, otherwise it is zero. Type I instructions that use the ALU need the immediate at the lower input. The few type I instructions that use the `rt` value as a source, such as `sw` and `beq`, use the `rt` value in some place other than the ALU, such as the `ID`-stage comparison unit for a `beq`. Logic for the stall signal, in green, simply checks for a bypass from both stages.

The solution discussion continues on the next page.



Problem 2 Common Mistakes: Many solutions would generate a stall signal when a bypass was needed and the instructions in **EX** and **ME** wrote the same register. For example in the following code fragment . . .

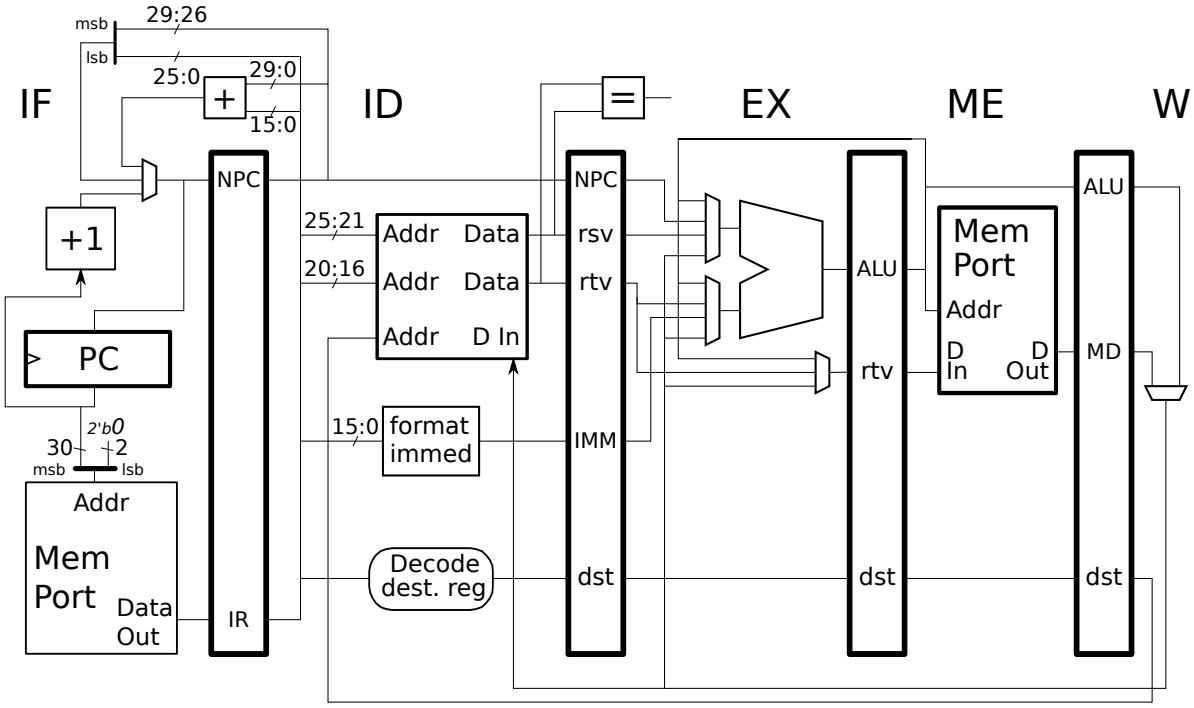
```
# Cycle          0  1  2  3  4  5  6
sub r1, r2, r3   IF ID EX ME WB
add r1, r1, r4      IF ID EX ME WB
and r5, r1, r6         IF ID EX ME WB
```

. . . the **and** instruction uses the value of **r1** computed by the **add**. That value will by bypassed to the **add** in cycle 4 from the **ME** stage. The value of **r1** carried by the **sub** instruction is irrelevant to the **and**. The correct solution here handles this case using the AND gate with a bubbled input.

In too many solutions the rt is src was optimistically assumed to compute the exact signal needed by the B select input. I'm not sure if this counts as a mistake, or as a face-saving way to avoid putting in the necessary effort or time management discipline to solve the problem.

On the original exam there was no description of what the rt is src logic block did, but it had been used in earlier assignments in the Spring 2020 semester, and students were free to ask what that block did.

Problem 3: [15 pts] Show the execution of the code fragment below on the illustrated implementation.



☑ Show execution. ☑ Note that the branch is taken. ☑ Pay attention to the timing of the branch. ☑ Check for dependencies, ☑ including for the branch.

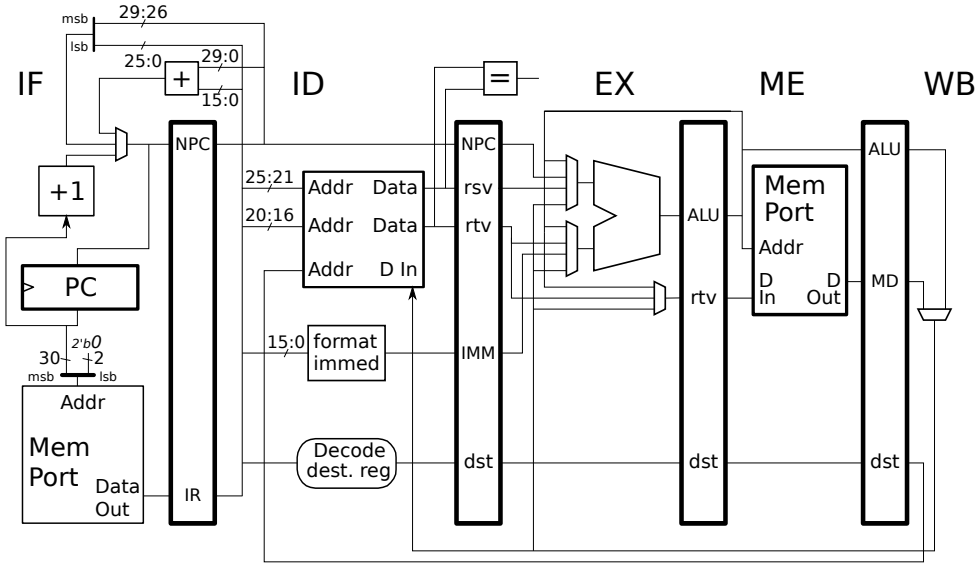
The solution appears below.

One common mistake was overlooking that there is no bypass path to supply the `beq` with `r3` (which is needed in the `ID` stage), and so the branch must stall until `slt` reaches `WB`.

Another common mistake is stalling when a bypass path is available.

```
 # Branch taken.     SOLUTION
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 lw r1, 0(r2)        IF ID EX ME WB
 slt r3, r1, r4         IF ID -> EX ME WB
 beq r3, r0  SKIP          IF -> ID ----> EX ME WB
 addi r2, r2, 4                  IF ----> ID EX ME WB
 xor r5, r5, r9
 or  r6, r6, r9
SKIP:
 addi r7, r7, 4                            IF ID EX ME WB
 sw r1, 0(r7)                                 IF ID EX ME WB
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
```

Problem 4: [15 pts] The code fragment below runs inefficiently. Modify the code so that it runs faster on the implementation below. Instructions can be re-arranged, changed, or removed, and registers can be changed. Don't forget that the modified needs to do the same thing as the original code.



☑ Re-write code so that it is faster but, of course, does the same thing as the original.

```
LOOP:
 lw r1, 0(r2)
 andi r1, r1, 0xff
 addi r2, r2, 4
 lw r3, 0(r2)
 srl r3, r3, 24
 add r9, r9, r1
 add r9, r9, r3
 addi r2, r2, 4
 sub r8, r2, r11
 bne r8, r0  LOOP
 nop
```

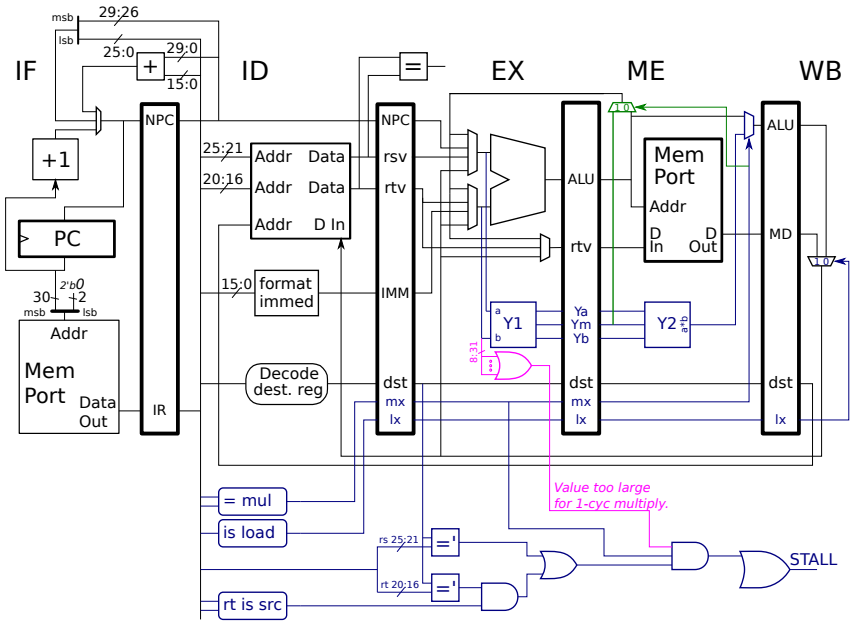
First, the easier optimizations: The two **addi** instructions were replaced by one by using offsets on the load instructions. (More about those offsets later.) Instructions were re-arranged to avoid load-use stalls (for example the stall suffered by the **andi** in the original code waiting for **r1** to reach WB), and the branch delay slot was filled.

The **sub** was eliminated by just checking for equality in the branch, **bne r2, r11**. The **andi** sets **r1** to the 8 least-significant bits of the loaded value (which was in **r1**). The **lbu r1, 3(r2)** loads just those 8 bits, and so the **andi** is no longer necessary. (The offset is 3 because MIPS is big-endian, meaning the memory address, **r2+0**, is where the 8 most significant bits are.) The **srl** instruction extracts the 8 most significant bits of **r3**. The **srl** can be avoided by again using an **lbu** to load just the needed byte. Note that the offset is **-4** because **r2** is incremented by 8 between the two **lbu** instructions.

```
LOOP:  # SOLUTION
 lbu r1, 3(r2)       # Replacement for:   lw r1,0(r2)     andi r1, r1, 0xff
 addi r2, r2, 8      # Replacement for:   addi r2, r2, 4  addi r2, r2, 4
 lbu r3, -4(r2)      # Replacement for:   lw r3,0(r2)     srl r3, r3, 24
 add r9, r9, r1
 bne r2, r11  LOOP   # Replacement for:    sub r2,r2,r11  beq r8, r0, LOOP
 add r9, r9, r3      # Fill branch delay slot.
```

Problem 5: [30 pts] Answer each question below.

(*a*) The code fragments below are to run on the implementation with the small-multiply bypass from Homework 3 and shown to the right. For each code fragment, indicate whether our small-value bypass feature always eliminates a stall, sometimes, or never? Explain.



☑ Eliminates stall on code below: ⊗ *Always* ◯ *Sometimes* ◯ *Never*

```
andi r3, r5, 0x3f    # SOLUTION: In bitwise AND result the high 24 bits all 0.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: ◯ *Always* ◯ *Sometimes* ⊗ *Never*

```
ori r3, r5, 0x63f    # SOLUTION: This bitwise OR result must be > 255.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: ⊗ *Always* ◯ *Sometimes* ◯ *Never*

```
lbu r3, 0(r4)        # SOLUTION: Loaded unsigned byte must fit in 8 bits.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

☑ Eliminates stall on code below: ◯ *Always* ⊗ *Sometimes* ◯ *Never*

```
lw r3, 0(r4)         # SOLUTION: Loaded value may or may not be < 256.
mul r1, r2, r3
add r6, r6, r1
lw r10, 0(r6)
```

(*b*) In typical practice a company decides upon an ISA, and then makes multiple implementations of that ISA. Let H_I and L_I be two implementations of ISA I, H_I is a high-end system and L_I is low-cost. Let ISA E (for expensive) be an ISA designed for high-end systems, and ISA C (for cheap) be an ISA designed for low-cost systems, and let H_E and L_C be their implementations. All three ISAs and all four implementations were designed by skilled engineers with lots of resources.

☑ Why might H_E be better than H_I and why might L_C be better than L_I? The same reason should apply to both. The answer is related to the ISAs used for the implementations.

H_E would be better then H_I because ISA E would be designed just for high-end implementations and so can include features that are good for these implementations without regard for whether such features would make it difficult to design low-cost implementations. In contrast, I might have omitted expensive-to-implement features and so H_I would not be as good. Similarly, L_C would be better (perhaps cost less) than L_I because C would omit features that are only needed in high-end implementations.

☑ Even if L_C is better than L_I, why might a user still choose L_I?

Software compatibility. The cash-strapped user buys L_I but hopes that later he can afford H_I and run his software on it unmodified. Had the user bought L_C and then later bought H_E, software would have to be re-compiled before H_E could be used.

(*c*) Consider the preparation of a set of SPECcpu results. For each item below indicate who is responsible, SPEC (the organization) or the tester. Also indicate what would be the problem if it were the other way around. For example, if you answered that SPEC chooses the benchmarks, then explain the disadvantage of having the tester choose the benchmarks.

☑ Choose the benchmarks: ⊗ *SPEC* or ◯ *The Tester*

A benchmark suite is only useful if all testers use the same benchmarks.

☑ Problem if it were the other way around:

If each tester chose the benchmarks there would be no way to compare results from two different testers. They could differ even if exactly the same system were tested.

☑ Choose the benchmark input data: ⊗ *SPEC* or ◯ *The Tester*

Changing the input data can drastically change the run time, so the reasons are the same as for benchmark choice above.

☑ Problem if it were the other way around:

Certain testers would choose inputs for which run time would be low, thus making the tested system appear fast.

☑ Choose the benchmark training data: ⊗ *SPEC* or ◯ *The Tester*

Training data is used for profiling runs. Results from the profiling run are used by the compiler to optimize code, for example, to re-arrange code so that most branches are not taken.

A case could be made that the choice of training data should be based on how the compiler will use the profiling information. Since the compiler choice is up to the tester, training data ought to be up to the tester to. But the SPEC R&R rules say no, testers SHALL use the SPEC-provided training inputs.

☑ Problem if it were the other way around:

If the tester were allowed to choose the training data then certain unscrupulous testers might use the reference data (the data used to compute SPEC scores) for training. That would yield the best results, but does not reflect a real-world situation because for all kinds of reasons developers don't know in advance the exact inputs their programs will be run with. For one thing, each run the program usually get different inputs. How many people write the exact same letter each time they use a word processor?

☑ Choose the compiler: ◯ *SPEC* or ⊗ *The Tester*

SPECcpu is designed to test new ISAs and implementations, and to show their full potential. If an ISA is truly new then there is no way SPEC could choose the compiler, the compiler would be developed in-house by the company that designed the new ISAand implementation. Furthermore, the compiler can be thought of as part of the system being tested, for example, it might optimizing assume the presence of certain features.

☑ Problem if it were the other way around:

If SPEC did choose the compiler it would be impossible or difficult to test new designs.

☑ Choose the compiler optimization flags: ◯ *SPEC* or ⊗ *The Tester*

If the tester chose the compiler, but SPEC choose the flags that would mean that SPEC is requiring compilers to support a set of SPEC flags. A case could be made for this at the base tuning level, in which the programmer is expected to use a set of flags providing good results, for example, `-O3` (optimization level 3) or `-fast`. But for the peak tuning level SPEC-provided flags would preclude experts choosing flags specially chosen for a particular benchmark on the tested system.

☑ Problem if it were the other way around:

If SPEC chose the flags then the peak results would reflect the best possible performance. That's because SPEC could not be expected to choose them for each system and benchmark. (They certainly could not do that years in advance when the suite is developed, because they couldn't know how to set flags for implementations and using compilers that don't yet exist. It would cost way too much money to have a SPEC team updating the flags for each new benchmark, and anyway would be a source of endless squabbling about the amount of effort the team puts in.)

(*d*) The IA-32 ISA has been described as Intel's golden handcuffs. Who slapped on those handcuffs? What does the gold refer to? What do the handcuffs refer to? *This was discussed in class, but it is okay to use Web searches to answer this question.*

☑ The reason for these handcuffs is:

IBM chose the Intel 8086, sort of an IA-32 implementation, for their personal computer, the IBM PC. That product became very successful, not because it was the first personal computer, but because the IBM name signaled that personal computers were now usable by anyone, not just hobbyists.

☑ They are golden because:

IBM sold lots of PCs so Intel made lots of money.

☑ They are handcuffs (a restriction) because:

Short Answer, but sufficient for full credit: A huge base of software ensured customers for future implementations of the ISA despite its many limitations.

Long Answer: Intel may have felt that the 8086 ISA (called IA-32 in class) was saddled with too many restrictions to be useful for a personal computer ISA with a decade or more of implementations ahead, and so would have wanted to develop a new ISA and scrap IA-32. But a computer with a new ISA would not be able to run all the software developed for the very successful PC, and so buyers would have to wait for new software to become available and hope that their favorite programs would be ported. A buyer then could just as easily by a computer using a non-Intel CPU. Intel and IBM were well aware of this, and so they dared not change the ISA, despite its flaws. Among the more irritating flaws was the need to use a pair of registers to specify a 32-bit memory address. That made address arithmetic much more complicated.

(*e*) Appearing below are some hypothetical instructions. Indicate whether each instruction is a better candidate for a RISC ISA or a CISC ISA. Explain why.

☑ Is the instruction below more ◯ *RISC* or ⊗ *CISC* like? ☑ Explain.

```
addi r1, r2, 0x12345678
```

RISC ISAs have fixed size instructions, usually 32 bits. That leaves no room for large immediates, including the one in the example above.

Common Mistake: Some incorrectly assumed that 64-bit ISAs had 64-bit instructions and so that immediate could be accommodated. In current use "32-bit" and "64-bit" ISAs refer to the size of a virtual memory address (the only kind used in class so far). Typically the integer register size matches the virtual address size, so 64-bit systems have 64-bit registers. However there is no need to increase the size of the instruction itself. What would a 64-bit type R instruction use all the extra space for—other than immediates? And it would not make sense to nearly double the size of program for those few cases where a larger immediate was needed.

☑ Is the instruction below more ⊗ *RISC* or ◯ *CISC* like? ☑ Explain.

```
lw r1, (r2+r3)  # Load r1 = Mem[ r2 + r3 ]
```

The instruction above could easily execute in a RISC pipeline, including our 5-stage MIPS (though MIPS lacks such an instruction) in which the ALU could add `r2` and `r3` just as easily as it could add a register value to an immediate.

☑ Is the instruction below more ⊗ *RISC* or ◯ *CISC* like? ☑ Explain.

```
bgt r1, r2, TARG  # Branch if r1 < r2
```

This can also easily be pipelined. It is omitted from MIPS I only, one assumes, to make it possible to resolve branches in `ID`. (Testing equality is easier than comparing magnitude.)

☑ Is the instruction below more ◯ *RISC* or ⊗ *CISC* like? ☑ Explain.

```
add (r1), r2, (r3)   # Mem[r1] = r2 + Mem[r3]
```

In RISC ISAs memory is accessed by memory instructions (loads and stores) other instructions must get their operands from registers, and so the instruction isn't suitable for RISC.

Name Solution_____

Computer Architecture
LSU EE 4720
Solve-Home Final Examination

Wednesday, 6 May 2020 to Saturday, 9 May 2020 5:00 (5 AM) CDT

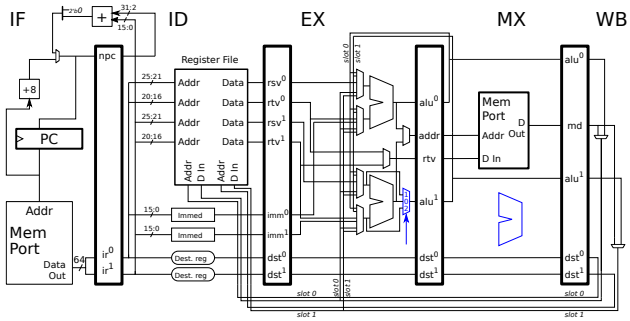
Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as MIPS tutorials, digital logic design guides, and computer architecture references can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(30 pts)
Exam Total	_____	(100 pts)

$r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck! Help Keep Everybody Safe!

Problem 1: (30 pts) The two-way superscalar MIPS implementation below has an ALU in the `MX` (née `ME`) stage, call it the *second-chance ALU*. For this problem the second-chance ALU will be connected so that two stall situations are avoided. *A slightly similar problem appeared on the Spring 2006 Midterm Exam in Problem 2. It's okay to look at the problem and solution.*



Yes, it's small! Use the next page for the solution.

(*a*) The `sub` in the code below suffers a stall due to a dependence with the other instruction in the same fetch group. Connect the second-chance ALU so that the stall is avoided. The changes must not break existing functionality and must not result in stalls for unrelated code. In particular note that the `add` does not stall in either version.

```
# Cycle           0  1  2  3  4  5  6  7   # Unmodified Implementation
ori R3, r1, 0xff  IF ID EX ME WB
xor R2, r8, r9    IF ID EX ME WB
add R1, R2, R3       IF ID EX ME WB
sub R4, R1, r5       IF ID -> EX ME WB
and r7, r8, R4          IF -> ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7

# Cycle           0  1  2  3  4  5  6  7   # With second-chance ALU.
ori R3, r1, 0xff  IF ID EX MX WB
xor R2, r8, r9    IF ID EX MX WB
add R1, R2, R3       IF ID EX MX WB        # No stall in either implementation.
sub R4, R1, r5       IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4          IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle           0  1  2  3  4  5  6  7
```

☑ Connect second-chance ALU to avoid the stall by the `sub` and allowing code to execute as in the sample above. ☑ The connections should work for any pair of dependent, ALU-using, non-memory instructions.

☑ Pay attention to cost. Assume that a pipeline latch bit costs twice as much as a multiplexor bit.

☑ Do not add unneeded bypass paths. ☑ Don't break existing functionality.

Solution appears in blue in the diagram several pages ahead. The muxes at the second-chance ALU inputs are needed because the `MX.alu0` value could be needed by the `rs` or `rt` operands (or both). Many solved the problem assuming that the `MX.alu0` value is only needed for the `rs` source.

(*b*) The code below suffers a load/use stall. Add the minimum number of connections to the second-chance ALU so that such load/use stalls (in which the using instruction is in slot 1) can be avoided.

```
add r3, r2, r3    IF ID EX ME WB
lw r4, 5(r1)      IF ID EX ME WB
ori r6, r1, 0xff     IF ID EX ME WB
sub r5, r3, r4       IF ID -> EX ME WB
```

☑ Add connections to the second-chance ALU to avoid load/use stalls when the using instruction (such as the `sub` in the example) is in slot 1.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

See solution several pages ahead.

(*c*) In the code below the `sub` does not stall due to the second-chance ALU but the `and` does stall. Add control logic to generate a stall signal for cases such as this.

```
# Cycle           0  1  2  3  4  5  6  7
ori R3, r1, 0xff  IF ID EX MX WB
xor r2, r8, r9    IF ID EX MX WB
add R1, r2, R3       IF ID EX MX WB
sub R4, R1, r5       IF ID EX MX WB        # No stall due to second-chance ALU!
and r7, r8, R4          IF ID -> EX MX WB  # Stall due to second-chance ALU.
# Cycle           0  1  2  3  4  5  6  7
```

☑ Add logic to generate a stall signal for the situation described above. ☑ The logic should work for any instruction dependent on an instruction using the second-chance ALU.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

See solution on next page.

(*d*) Generate the select signal for the `EX` stage multiplexor shown in blue. The control logic should work for the intra-group dependence case. (The control logic does not need to work for the load/use case.)

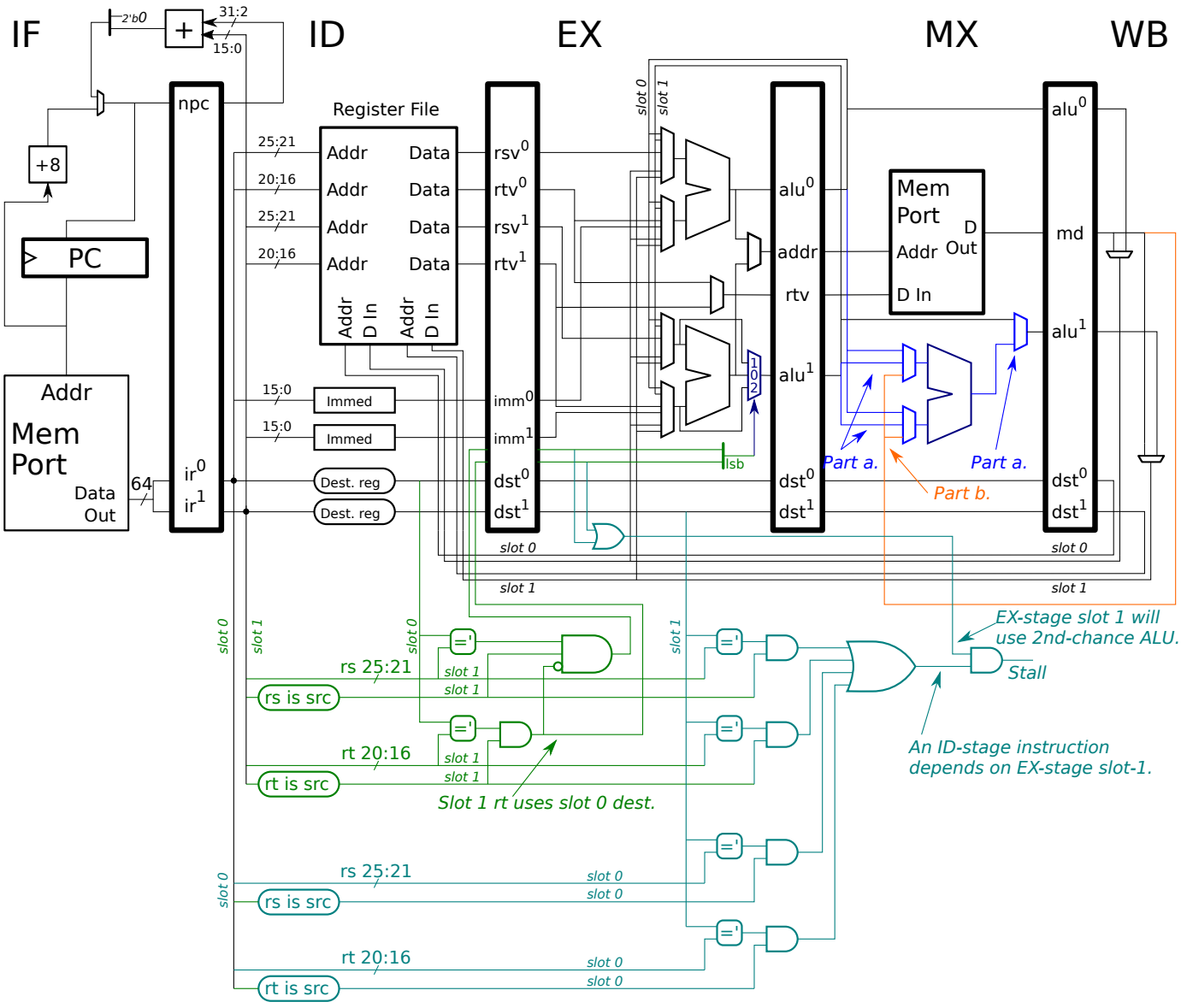
☑ Add logic for the select signal for the intra-group dependence. ☑ The logic should work for any pair of dependent, ALU-using, non-memory instructions.

☑ Pay attention to cost, use the same cost assumption as given in the previous part.

You're almost there! The solution is on the very next page!!

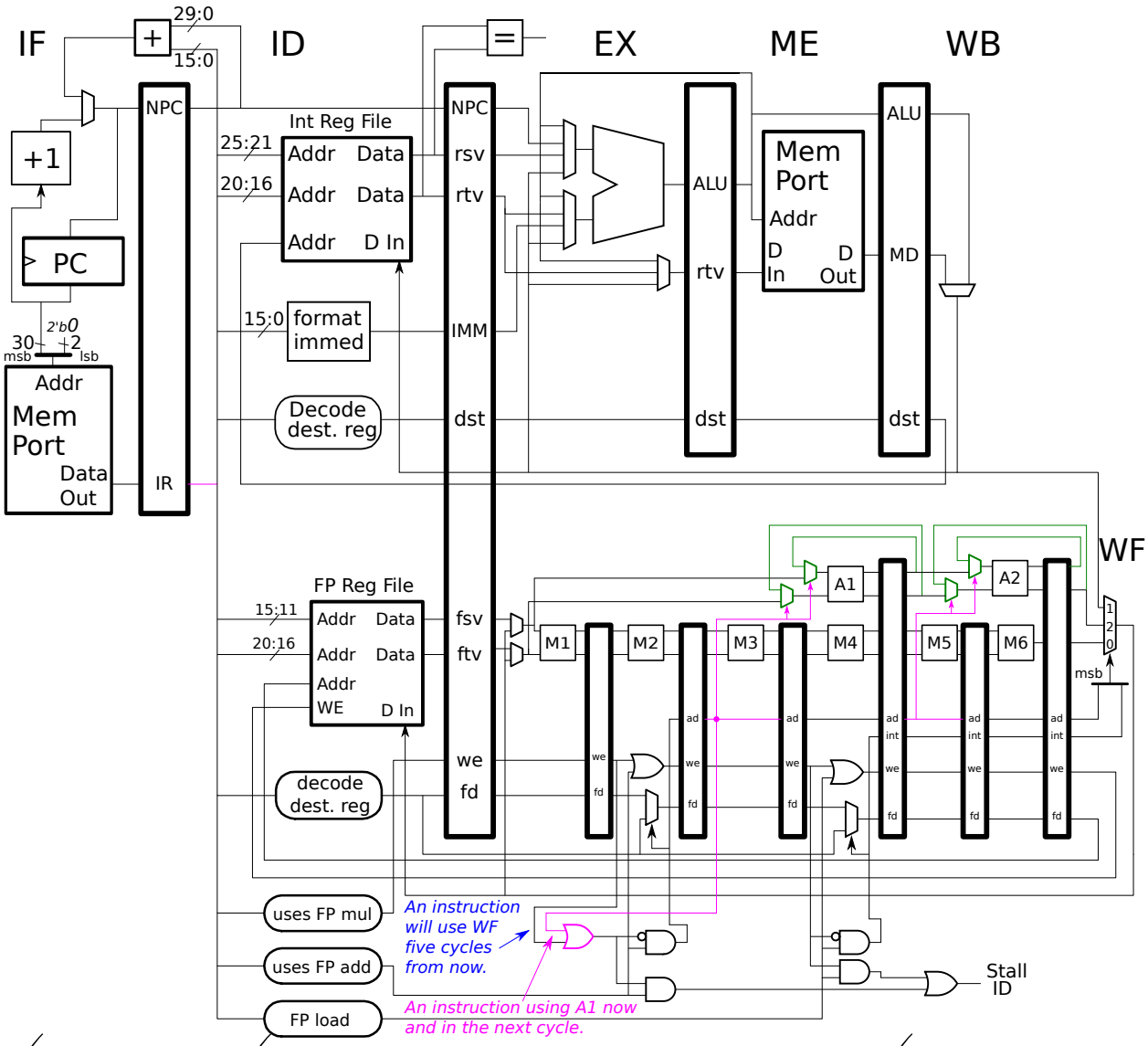
The Inkscape SVG source is at `https://www.ece.lsu.edu/ee4720/2020/fe-p1-v2-ss.svg`.

Solution appears below in blue (part a and b), green (part c or d, depending on whether I swap them), and turquoise (part d or c).



Problem 2: (25 pts) Show the execution of the code fragments as requested below.

(*a*) Show the execution on the FP pipeline below, note that the adder unit has an initiation interval of 2.

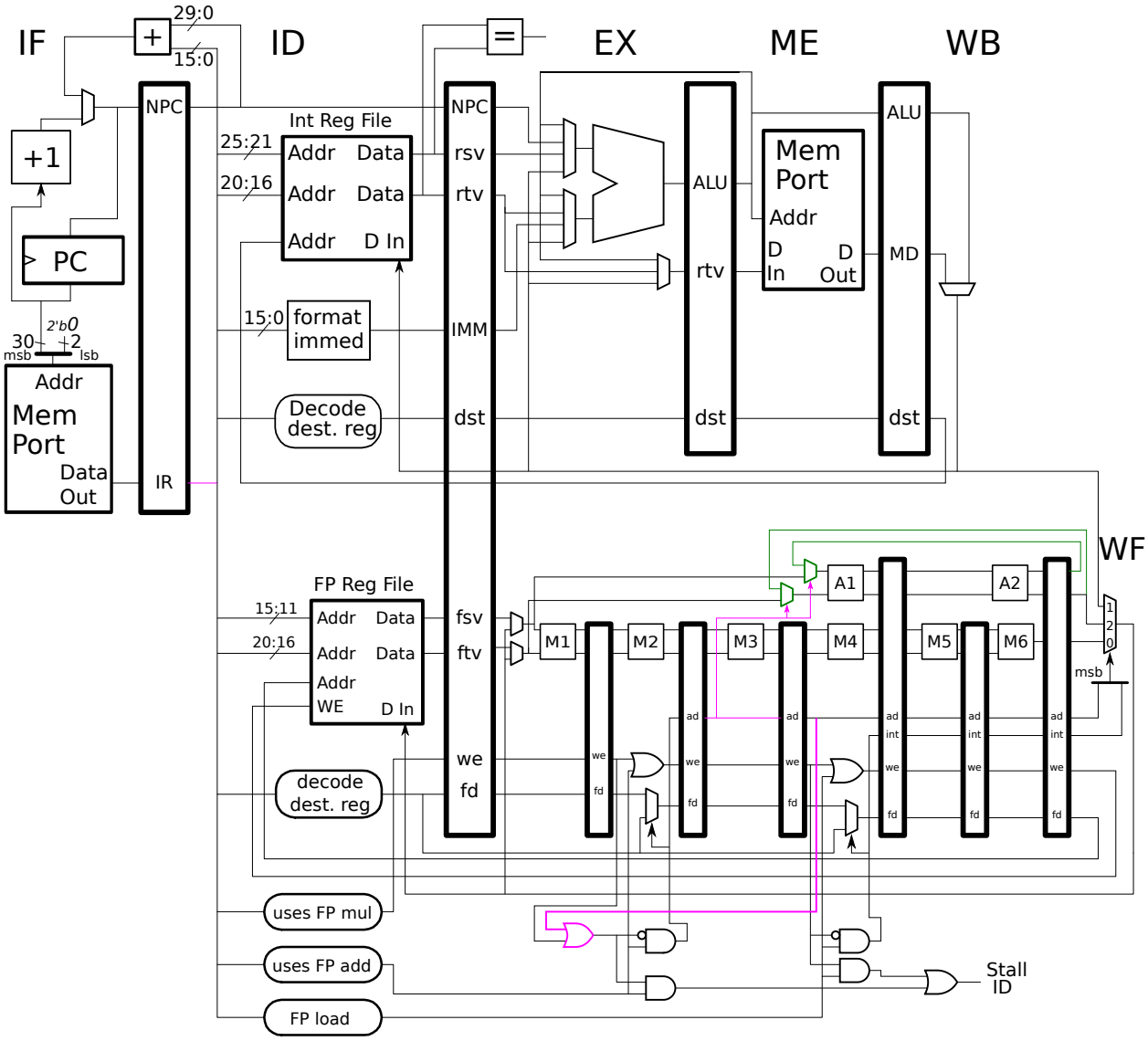


☑ Show execution. ☑ Pay attention to how the FP add unit should operate. ☑ Don't forget to check for dependencies.

Solution appears below. The second `add.s` stalls one cycle because of the initiation interval of the FP add unit. You get what you pay for.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14  SOLUTION
lwc1 f2, 0(r1)   IF ID EX ME WF
add.s f0, f2, f4    IF ID -> A1 A1 A2 A2 WF
add.s f1, f2, f5       IF -> ID -> A1 A1 A2 A2 WF
add.s f3, f1, f6             IF -> ID -------> A1 A1 A2 A2 WF
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

(*b*) Show the execution on the FP pipeline below, note that the adder unit is different than the previous problem and from other examples covered in class.

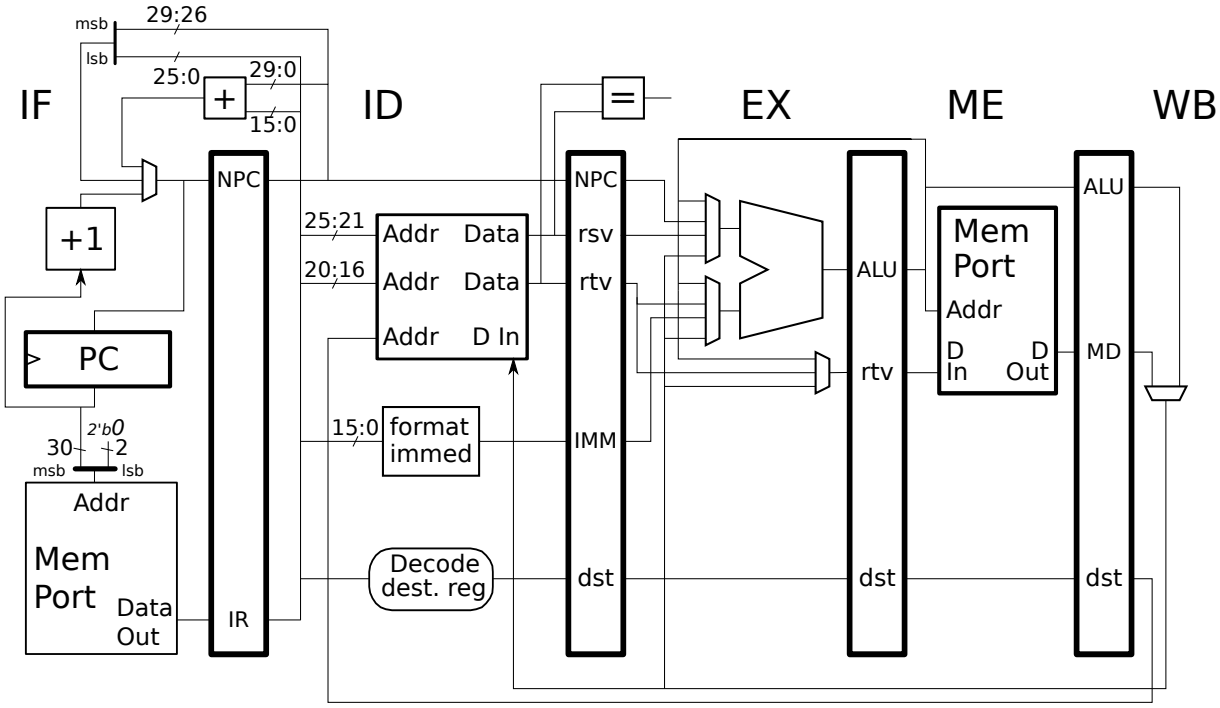


☑ Show execution. ☑ Pay attention to how the FP add unit should operate. ☑ Don't forget to check for dependencies.

Solution appears below. As can be inferred from the control signals, a FP add instruction passes through the A1 and A2 stages twice as shown in the solution below. Note that the second add.s no longer stalls waiting for A1!

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13  SOLUTION
lwc1 f2, 0(r1)   IF ID EX ME WF
add.s f0, f2, f4    IF ID -> A1 A2 A1 A2 WF
add.s f1, f2, f5       IF -> ID A1 A2 A1 A2 WF
add.s f3, f1, f6             IF ID -------> A1 A2 A1 A2 WF
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

(*c*) Show the execution of the code on the implementation below. Find the CPI for a large number of iterations.

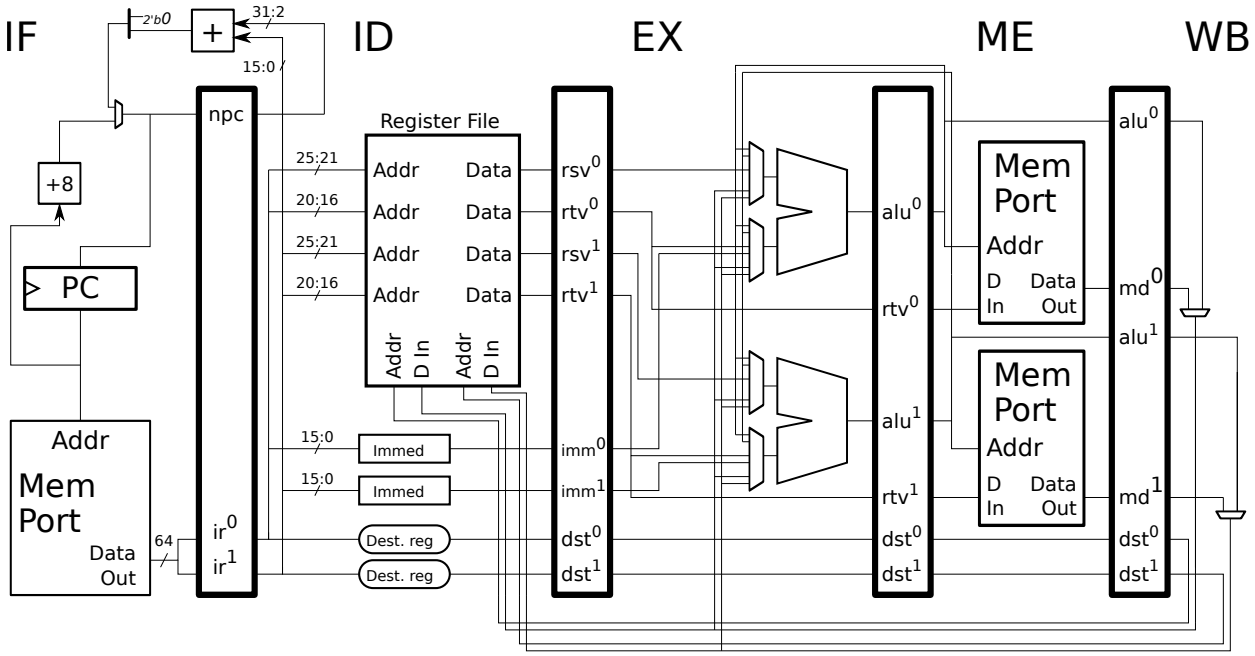


☑ Show execution on the illustrated implementation with the branch taken. ☑ Find the CPI for a large number of iterations.

The solution appears below. The only stall is a 1-cycle load/use stall suffered by the `sw`. The first iteration starts in cycle 0 (when the first instruction, `addi`, is in `IF`) and second at cycle 6. The second iteration is the same as the first, so all iterations will take $6-0=6$ cycles. An iteration has 5 instructions and so the CPI is $\frac{6}{5}=1.2$ CPI, and the instruction throughput is $\frac{5}{6}$ insn/cycle.

```
      # SOLUTION
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11
 addi r2, r2, 16    IF ID EX ME WB                      First Iteration
 lw r1, 8(r2)          IF ID EX ME WB
 sw r1, 12(r3)            IF ID -> EX ME WB
 bne r3, r4, LOOP            IF -> ID EX ME WB
 addi r3, r3, 32                   IF ID EX ME WB
 sub r10, r3, r2
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11
 addi r2, r2, 16                      IF ID EX ME WB    Second Iteration
 lw r1, 8(r2)                            IF ID EX ME WB
 sw r1, 12(r3)                              IF ID -> EX ME WB
 bne r3, r4, LOOP                              IF -> ID EX ME WB
 addi r3, r3, 32                                     IF ID EX ME WB
#       Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*d*) Show the execution of the code on the 2-way superscalar MIPS implementation illustrated below, and find the CPI for a large number of iterations. This is not the same as the implementation from Problem 1. Instruction fetch is of aligned groups.



☑ Show execution on the illustrated implementation. ☑ Find the CPI for a large number of iterations.

☑ Take aligned fetch into account, the address of `LOOP` is `0x1000`. ☑ Pay attention to available bypass paths.

Solution appears below. Because there is no bypass path for the store value the `sw` must stall until the `lw` reaches `WB`.

The CPI is $\frac{6}{5} = 1.2$, which is disappointing because the implementation is capable of a CPI of $\frac{1}{2}$.

```
LOOP:  #     Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 #  SOLUTION
 addi r2, r2, 16    IF ID EX ME WB                          #  First Iteration
 lw r1, 8(r2)       IF ID -> EX ME WB
 sw r1, 12(r3)         IF -> ID ----> EX ME WB
 bne r3, r4, LOOP      IF -> ID ----> EX ME WB
 addi r3, r3, 32             IF ----> ID EX ME WB
 sub r10, r3, r2             IFx
LOOP:  #     Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12
 addi r2, r2, 16                     IF ID EX ME WB       #  Second Iteration
 lw r1, 8(r2)                        IF ID -> EX ME WB
 sw r1, 12(r3)                          IF -> ID ----> EX ME WB
 bne r3, r4, LOOP                       IF -> ID ----> EX ME WB
 addi r3, r3, 32                              IF ----> ID EX ME WB
 sub r10, r3, r2                              IFx
#            Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*e*) The code fragment below is the same as the one from the previous problem and is to run on the same superscalar system. Re-write the code so that it runs with fewer stalls (and of course does the same thing), and compute the CPI for a large number of iterations. Extra instructions can be added before or after the loop. Do not unroll the loop.

☑ Re-write code to minimize stalls on the superscalar implementation.

☑ Compute the CPI of the re-written code for a large number of iterations.

```
LOOP:
 addi r2, r2, 16
 lw r1, 8(r2)
 sw r1, 12(r3)
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2
```

Two solutions are given here, Partial Credit and Full Credit (starting on the next page). An easy change was moving the `addi r2,r2,16` increment after the `lw` and adding 16 to the load offset. This is shown as the Partial Credit solution—partial credit because the `sw` still stalls.

```
 # SOLUTION - Partial Credit
LOOP:
 lw r1, 24(r2)     # Move lw before addi and increase offset to 24.
 addi r2, r2, 16
 sw r1, 12(r3)     # Still lots of stall cycles.
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2
```

In the full-credit solution the `sw` is put before the `lw` where it will store the `r1` value *that was loaded in the previous iteration* or by the *prologue* instruction added before the loop. The `lw` in the loop is loading a value for the next iteration, which is why its offset is increased by another 16 to 40.

To compute the instruction throughput (or CPI) a repeating pattern needs to be found. The repeating pattern is established in the second iteration and verified in the third by noting that the pipeline state at cycles 10 and 14 are identical. So the iteration time is $14 - 10 = 4$ cycles and so the instruction throughput is $\frac{5}{4} = 1.25\,\text{insn/cycle}$ or the CPI is $\frac{4}{5} = .8\,\text{CPI}$, an improvement.

```
 # SOLUTION - Full Credit
 #
 lw r1, 24(r2)     # Prologue: Load initial value of r1.

LOOP:
 sw r1, 12(r3)     # This finishes up the previous iteration.
 lw r1, 40(r2)     # Load value to be used in next iteration, if any.
 addi r2, r2, 16   # Increment r2 after lw to avoid a stall.
 bne r3, r4, LOOP
 addi r3, r3, 32
 sub r10, r3, r2

 # Pipeline Execution Diagram
 #
 lw r1, 24(r2)    IF ID EX ME WB
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)       IF ID ----> EX ME WB       First Iteration
 lw r1, 40(r2)       IF ID ----> EX ME WB
 addi r2, r2, 16        IF ----> ID EX ME WB
 bne r3, r4, LOOP       IF ----> ID EX ME WB
 addi r3, r3, 32                IF ID EX ME WB
 sub r10, r3, r2                IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)    Second Iteration  IF ID EX ME WB
 lw r1, 40(r2)                      IF ID EX ME WB
 addi r2, r2, 16                       IF ID EX ME WB
 bne r3, r4, LOOP                      IF ID -> EX ME WB
 addi r3, r3, 32                          IF -> ID EX ME WB
 sub r10, r3, r2                          IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)    Third Iteration              IF ID EX ME WB
 lw r1, 40(r2)                                 IF ID EX ME WB
 addi r2, r2, 16                                  IF ID EX ME WB
 bne r3, r4, LOOP                                 IF ID -> EX ME WB
 addi r3, r3, 32                                     IF -> ID EX ME WB
 sub r10, r3, r2                                     IFx
LOOP:   #  Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 sw r1, 12(r3)                                              IF ID EX ME WB
 lw r1, 40(r2)                                              IF ID EX ME WB
```

Problem 3: (15 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and the other systems have a local predictor, the length of the local history is given in the questions below.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   N  N  N  N  T  T    N  N  N  N  T  T   ...

B2:      T  T  T  T  T  T    T  T  T  T  T  T   ...
```

☑ What is the accuracy of the bimodal predictor on branch B1?

Short answer: $\frac{3}{6}$, see work below.

```
    SOLUTION WORK
    0  0  0  0  0  1  2    1  0  0  0  1  2   <-  Two-bit counter
B1:   N  N  N  N  T  T    N  N  N  N  T  T   ...
                  x  x    x           x  x    <-  Prediction Outcome
                        ----------------------  <-  Repeating Pattern
```

Explanation: The line just below SOLUTION WORK (above) shows 2-bit counter values for B1, for when the counter starts at zero. The prediction outcomes are shown below the branch outcomes. To compute the prediction ratio we need to use a repeating pattern. A pattern is repeating if the branch outcomes pattern repeats and if the two-bit counter value is the same before the pattern starts and after it ends. Such a pattern is underlined, the counter is two at the start and the end. The prediction accuracy is $\frac{3}{6} = \frac{1}{2}$.

☑ What is the accuracy of a local predictor with a 12-outcome local history on branch B1 and ignoring B2.

Branch B1's pattern repeats and has a length of 6 outcomes and so it can easily be predicted with 100% accuracy by a 12-outcome local predictor.

☑ What is the accuracy of a local predictor with a 2-outcome local history on branch B1 and ignoring B2.

The accuracy is worked out below. The local histories table shows the 6 local histories used when predicting B1. The labels (1-6) refer to the outcome being predicted, and the local history (LH) consists of the 2 preceding outcomes. For example, then predicting the first T, label 3, the local history is NN. The Pattern History Table Analysis table has one row for each possible PHT entry, all $2^2 = 4$ of them. (In this case the branch affects every entry in the table, but usually branches use only a small fraction of the entries. For example, if the local history size were 10 there would be 1024 PHT entries but branch B1 would use only 6 of them [after warmup].) A PHT entry holds a two-bit value (called a counter), and the table shows values for these counters for each entry as well as the outcome pattern used to update the counter. For TN the outcome pattern is just a sequence of Ns so the counter stays at zero. The NT and TT entries are similarly well-behaved. But the NN entry is used to predict three outcomes, those at label 1, 2, and 3. The counter will change form 1 to 0 and back, and so N will be consistently predicted, which will be correct 2 out of 3 times. These accuracies are shown in the last column and totaled at the bottom. Note that the total is computed by adding the numerators and denominators: $\frac{2+1+1+1}{3+1+1+1} = \frac{5}{6}$, and that it is important that the denominator is set to the number of times the local history is seen in the repeating pattern. Based on this analysis the prediction accuracy is $\frac{5}{6}$.

```
      SOLUTION WORK
B1:   N  N  N  N  T  T    N  N  N  N  T  T
      5  6  1  2  3  4    5  6  1  2  3  4  <- Label of predicted branch.

Local Histories (LH) and Outcome
   LH Outcome
1: NN N
2: NN N
3: NN T
4: NT T
5: TT N
6: TN N

Pattern History Table Analysis
LH  Pattern              Two-Bit Counter Evolution   Accuracy
--  -------------------  -------------------------   --------
NN: N N T    N N T       1,0,0,      1,0,0,          2/3
NT:      T        T            3,          3,       1/1
TT:       N        N             0,          0,     1/1
TN:        N        N              0,          0,   1/1
--  -------------------  -------------------------   --------
                                                     5/6
```

☑ What is the accuracy of a local predictor with a 2-outcome local history on branch B1 and taking into account B2.

The local history for B2 is consistently TT and of course the outcome too is consistently T. Branch B1 and B2 will share the TT entry, but not nicely. When B1 is resolved the entry is decremented but when B2 is resolved the entry is incremented. We can infer from the branch patterns that B2 will use the TT local history six times more frequently than B1 and so B1 at label 5 will retrieve a 3 and predict T and be wrong. That's shown in the table below, with the B2 outcomes shown in lower case, t, for clarity. So, the accuracy of B1 taking into account B2 is $\frac{4}{6}$. Branch B2 is predicated at 100% accuracy.

```
Pattern History Table Analysis                              B1
LH  Pattern                    Two-Bit Counter Evolution    Accuracy
--  -------------------------  -------------------------    --------
NN: N N T       N N T          1,0,0,      1,0,0,           2/3
NT:       T           T              3,          3,        1/1
TT:  t t t tNt t t t t tNt t   3,3,3,3,2,3,3,3,3,2,3,3    0/1
TN:           N           N               0,          0,   1/1
--  -------------------------  -------------------------    --------
                                                            4/6
```

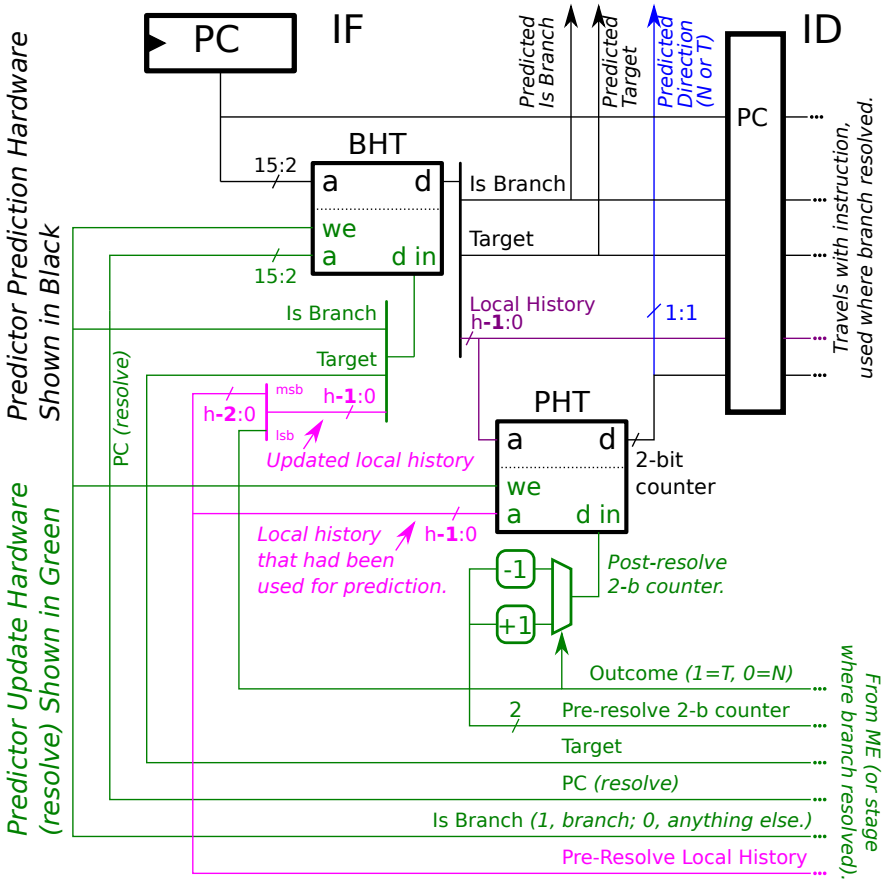
☑ What is the minimum local history size so that branch B1 is predicted with 100% accuracy, taking into account B2.

To solve this problem increase the local history size until the local histories are followed by consistent outcomes (unlike NN above). Because of NNNNT a local history of 3 is too short because NNN can be followed by N or T. A local history size of four will be sufficient. Note that this time B1 uses 6 out of $2^4 = 16$ entries and that B2 no longer causes trouble since its one local history, TTTT, does not match any of B1's six local histories.

```
Local Histories (LH) and Outcome
   LH Outcome
1: TTNN N
2: TNNN N
3: NNNN T
4: NNNT T
5: NNTT N
6: NTTN N
```

```
Pattern History Table Analysis                  B1         B2
LH    Pattern       Two-Bit Counter Evolution   Accuracy   Accuracy
----  ------------  -------------------------   --------   --------
NNNN:    T          3,3,...                     1/1
NNNT:      T        3,3,...                     1/1
NNTT:        N      0,0,...                     1/1
NTTN:          N    0,0,...                     1/1
TNNN:   N           0,0,...                     1/1
TTNN: N             0,0,...                     1/1
TTTT: t t t t t t   3,3,...                                 6/6
----  ------------  -------------------------   --------   --------
                                                6/6        6/6
```

(*b*) Appearing below is a diagram of a local predictor. The local predictor illustrated has a specific BHT size and an h-outcome local history size. The BHT size does not necessarily match the BHT in the prior part. Determine the amount of storage, in bits, used by the BHT and PHT for a 16-bit local history. Assume that Target is stored efficiently.



☑ Amount of storage for PHT is:

Short answer: The amount of storage is $2^{16} \times 2 = 2^{17}\,\text{b} = 2^{14}\,\text{B}$.

Each PHT entry holds a 2-bit counter, and for a 16-bit local history there are $2^{16} = 65536$ entries. The total size is $65536 \times 2\,\text{b} = 131072\,\text{b} = 16384\,\text{B}$. Note: full credit would be given for an answer in bits, no need to convert to bytes. That said, it is a good idea to include the unit (bits or bytes).

☑ Assumption about Target:

☑ Amount of storage for BHT is:

Short answer: $(1 + 16 + 16)\,2^{14}\,\text{b}$ assuming that the Target field is 16 bits and is used, after retrieval from the BHT, as a displacement from PC+4 to compute the full 32-bit target address.

From the diagram it can be seen that $16-2 = 14$ bits are used for the BHT address (a) input, and so there must be 2^{14} BHT entries. The diagram shows that an entry consists of three fields: *Is Branch*, *Target*, and *Local History*. Assume 1 bit for Is Branch because nothing was mentioned about predicting other kinds of control-transfer instructions. For the target assume that just 16 bits are stored, and that the full 32-bit target is computed using PC+4, just as it is for a branch. The local history is given as 16 bits. Totaling these yields the of a BHT entry, $(1+16+16)$ b. The sum of the 2^{14} entry sizes yields the BHT size, $(1+16+16)2^{14}$ b.

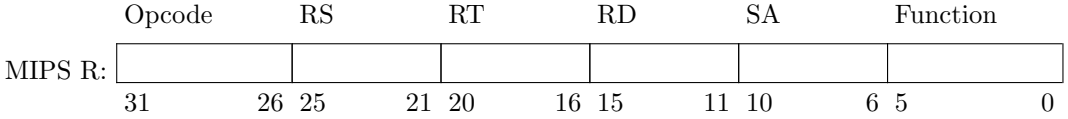
That's all that's needed for full credit. But since we're here, let's convert it to bytes and compare the size to other parts of the CPU to see how much the BHT is costing us. So, $(1+16+16)2^{14}\,\mathrm{b} = (1+16+16)2^{14}/2^3\,\mathrm{B} = (1+16+16)2^{11}\,\mathrm{B} = 67584\,\mathrm{B}$. That's roughly the size of a typical L1 data cache, so it is on the large size. Real BHTs are smaller.

Problem 4: (30 pts) Answer each question below.

(*a*) Appearing below are the three integer MIPS I instruction formats. Consider a modified form of MIPS in which there are 64 rather than 32 integer registers. A goal is compatibility with MIPS-I code and to use as few new opcodes and function field values as possible. Modify each format so that it can use 64 registers and explain what new opcodes (if any) are needed and any assumptions about existing MIPS-I instructions. *Hint: For one case there's nothing to do, for one case many opcodes will be needed, for one case only a few.*

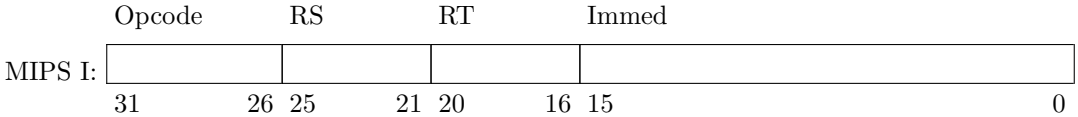
To encode 64 registers a 6-bit register field is needed. Since a goal is compatibility we want to make as few changes to the instruction format as possible.

☑ Modification for 64-register MIPS. ☑ Describe what new opcode/func values are needed for, if any.

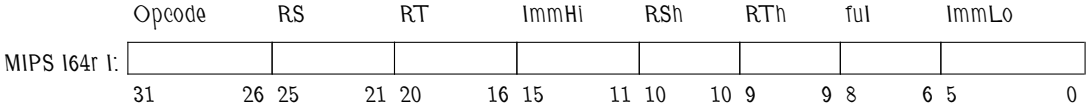


For most MIPS-I Type R instructions the **SA** field is unused and must be zero. So for MIPS-I64r use the **SA** field for the three extra register field bits, one for **RS**, one for **RT**, and one for **RD**. New **Function** values will be needed for the shift instructions because they use the **SA** field. The existing shift instructions, such as `sll r1, r2, 3` must use the **SA** field in the same way as MIPS-I (otherwise MIPS-I64r would not be compatible). New shift instructions, such as `sll64 r40, r1, 3` will use the **RT** field for the shift amount. No new opcodes or **Function** values will be needed for the other Type R instructions.

☑ Modification for 64-register MIPS. ☑ Describe what new opcode/func values are needed for, if any.



There is no free space in most type I instructions, so new opcodes will be needed. Since there are not many opcodes available a new 3-bit **fuI** field was added (see below) to provide more opcode space, serving the role that **Function** places in type R instructions. The extra bit for the **RS** and **RT** registers are in the same place they are for MIPS I64r instructions, they are labeled **RSh** and **RTh**. The immediate, now 11 bits, is split between two fields, **ImmHi** and **ImmLo**. If a compiler or human needs an instruction with an immediate value of 12 or more bits it will need to use a MIPS-I instruction and so will have to limit itself to registers **r0** to **r31**.



☑ Modification for 64-register MIPS. ☑ Describe what new opcode/func values are needed for, if any.

No changes need to be made because there are no register fields.

(*b*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock. *Yes, up until this comment the question is identical to one asked on the Spring 2019 final exam.* The SPECcpu benchmarks are run on each chip. Recall that SPECcpu can be run to compute a speed score and a rate score. (Don't confuse speed/rate with base/peak or int/FP.) Feel free to visit the SPEC site to help answering this question.

☑ Which chip would likely score higher (better) on the SPECspeed2017int benchmarks,
⊗ *Chip A* or ◯ *Chip B* . ☑ Explain.

Chip A because SPECspeed2017 is run with one benchmark at a time and the SPECcpu benchmarks use few threads, maybe one. For one thread Chip A has a peak performance of 4 insn/cycle, much better than Chip B's peak of 1 insn/cycle.

☑ Which chip would likely score higher (better) on the SPECrate2017int benchmarks,
◯ *Chip A* or ⊗ *Chip B* . ☑ Explain.

Chip B because a 1-way core would run code more efficiently (fewer stalls) than a 4-way core, and the SPECrate scores are obtained by running many copies of an individual benchmark and so all cores would be used.

(*c*) Our goal is to build a machine that can execute eight floating-point operations per cycle. Two machines are under consideration, an 8-way superscalar system implementing ISA I, and a 2-way superscalar system with an 8-lane vector unit implementing ISA IV, which is like I but with vector instructions. Both machines run at 1 GHz, and both can sustain eight billion floating-point operations per second.

☑ Which machine is likely to be more expensive? ☑ Explain.

The 8-way superscalar. Both machines will have eight sets of FP functional units. But the 8-way superscalar has hardware for decoding eight instructions versus two for the 2-way system. Also the cost of bypass paths in the 8-way superscalar system will be higher than bypass paths in the 2-way system, even when including the vector unit. That's because the vector unit would not have inter-lane bypass paths.

☑ Which machine is likely to be faster on typical code? ☑ Explain with ☑ a code example.

Short answer: The peak FP operation rate of the two systems is identical, but the ISA I implementation is likely faster on typical code because unlike ISA IV, ISA I does not need sets of eight identical operations to realize eight operation per cycle performance.

Description: The ISA I (8-way superscalar) implementation is likely faster than the ISA IV (2-way superscalar plus vector unit) because the ISA IV system can sustain 8 billion floating-point operations (plus one instruction) per second only on code containing one vector instruction per fetch group. On code which is not *vectorizable*, meaning for which vector instructions are not useful or on which vector instructions operate on only one lane, the ISA IV will execute at most only 2×10^9 insn/s. Since most code is not vectorizable the ISA I implementation will be faster. For example, the eight instructions in the code fragment below could be executed at the same time on ISA I. Those eight instructions could not be replaced by a single vector instruction because there are eight different instructions, not the same instruction, say `add`, operating on eight different sets of operands.

```
# Code not vectorizable because each instruction performs a different operation.
 add.d f0, f2, f4
 andi r1, r2, 0xeof
 sub.d f6, f8, f10
 ori r3, r2, 0xfood
 mul.d f12, f14, f16
 xori r4, r2, 0xa8
 add.s f18, f19, f20
 slt r9, r10, r11
```

(*d*) In MIPS and many other RISC ISAs memory accesses must be aligned. For example, a `lw` instruction, which loads a four-byte value, must load from an address that is a multiple of 4. The execution of a `lw` loading from an address that is not a multiple of 4 will result in an exception (and on Linux system resulting in the Bus Error signal handler being called). As we pointed out in class, integer instructions, and especially load and store instructions, in any reasonable ISA would be required to raise precise exceptions. MIPS is certainly reasonable in this respect.

Suppose that a program uses non-aligned addresses in memory accesses, but is otherwise correct. That is, the program would run correctly if the load could handle a non-aligned address. (After all, CISC ISA load instructions can do it.) But on MIPS it raises an exception as soon as a non-aligned load or store is attempted. Suppose further that re-writing the program is not feasible.

☑ Explain how we can take advantage of precise exceptions so that this program would run correctly. A code example would be nice but not necessary.

The execution of a `lw` with a misaligned address will cause execution to reach the exception handler. We will re-write the exception handler so that when a misaligned address exception occurs it will performs the unaligned `lw` using four load byte unsigned instructions (for which there is no alignment restriction). The exception handler will get the attempted load address and the address of the faulting instruction (the load) from MIPS' co-processor 0 registers (or from the appropriate place in some other ISA). Rather than using a `lw` to load the data, the handler will load the data using four `lbu` instructions since they don't have alignment restrictions. The four loaded values will be put together in one register (see the code sample below). Next, the handler routine will need to load the faulting `lw` instruction and determine which register it intended to write. (That is, the instruction will be read from memory as though it were data and the `rt` field value would be extracted using a `srl` and an `andi`.) The handler will place the value in that register, or in the part of the exception handler stack holding the saved value of that register. Finally, the exception handler will resume execution of the interrupted code at the instruction after the faulting load. See the example below.

(In this solution four `lbu` instructions were used. Some ISAs provide instructions specifically intended to load and piece together a misaligned value.)

```
# SOLUTION -- Example of code raising exception.
#
 add r1, r1, r2
 lw r3, 0(r1)       # Note: r1 may not be aligned. If not, handler called ..
 and r8, r8, r3     # .. and later returns to this instruction with r3 loaded.

# SOLUTION -- Part of handler that loads 32-bit value in four pieces.
# Register usage:
#   r1: Attempted load address.
#   r3: Register to put load value in.

# Load the value without risk of misalignment. (For big-endian byte order.)
  lbu r10, 0(r1)    # Most significant byte.
  lbu r11, 1(r1)
  lbu r12, 2(r1)
  lbu r3,  3(r1)    # Least significant byte.

  sll r10, r10, 24  # Move byte to most-significant position.
  or r3, r3, r10    # Combine with least-significant byte.
  sll r11, r11, 16
  or r3, r3, r11
  sll r12, r12, 8
  jr r31
  or r3, r3, r12
```

☑ Explain why it would be impossible if loads only raised deferred exceptions. (Assume that aligned accesses work fine with such loads.)

In a deferred exception the handler starts after the faulting instruction and at least one following instruction finishes execution. Any value written by the faulting instruction will not be correct. (In a precise exception the handler is called after instructions up to but not including the faulting instruction finish execution. All values written will be correct.)

For the handler to emulate unaligned loads it needs to resume execution at the instruction following the unaligned load. In the solution example above the `lw` is the faulting instruction and the **and** is the instruction at which execution needs to resume for the program to work correctly. If the exception were deferred then by definition at least the **and** and possibly several other instructions would execute before the handler were called. In that case the old value of `r8` would be lost and so there would be no way to return to the **and** and execute the code correctly. For example, suppose `r8` were 7 (a correct value) and `r3` were 0 (incorrect, assume that the correct value is 17) when the **and** executes the first time (before the handler is called). The execution of **and** sets `r8` to 0. Suppose that the handler is called and sets `r3` to 17 (a correct value) and then returns to the **and**. The second execution of **and** sets `r8` to 0 (the value of `0 & 17`) when it should be set to 1 (the value of `7 & 17`).

(*e*) MIPS has a `slt` (set less than) instruction, but doesn't have a `sge` (set greater than or equal to) instruction. Why not?

☑ Why doesn't MIPS have an `sge` instruction?

Because the exact same operation can be obtained by swapping the two source operands of `slt`. For example, if you need a `sge r1, r2, r3` just use `slt r1, r3, r2`. There is no need to waste a precious opcode.

(*f*) *Note: The following question was asked about two months after in-person classes were ended for the CoViD-19 pandemic.*

Perhaps many of us are wishing we could go back in time. (Not to warn people, that's obviously futile.) Wish granted. You are in a meeting (in person, not Zoom) with future Turing Prize winners discussing which features to put into their new [airquotes] "RISC" ISA, MIPS.

One attendee is advocating for the inclusion of magnitude comparison branch instructions such as `bgt r1,r2 TARG` (branch if `r1` greater than `r2`). But many others oppose the idea because it would have too much critical path impact. "We can include `bgt` with zero critical-path impact if we use a surprisingly simple but effective technique called branch prediction," you say.

☑ Explain how branch prediction can remove the critical path impact that was a concern at the meeting.

MIPS was designed so that branches could be resolved in the `ID` stage of a five-stage pipeline. For any branch instruction that tests register values, the test (such as greater-than) can't start until the register values are retrieved. Since the register values are retrieved in the `ID` stage the critical path includes both the retrieval and the test. This limited the kinds of two-register tests that branches could do to equality comparison but not magnitude comparison. With branch prediction the test is moved from the `ID` stage to the `EX` stage. In the `EX` stage the test starts near the beginning of the clock cycle (as do all other ALU operations) and so more time-consuming tests such as magnitude comparison can easily be done.

(In class we talk about performing branch prediction in the `IF` stage, which is the appropriate place to predict for most implementations.)

☑ Was the phrase *branch prediction* an anachronism at that fictional meeting? Web-search freely to answer this question.

No, branch prediction was in use for years before MIPS was designed (in the mid 1980s). For example, see James Smith's 1981 survey [1] appearing in the 8th International Symposium on Computer Architecture, one of the leading conferences in the area.

[1] Smith, J. E. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Washington, DC, USA, 1981), ISCA '81, IEEE Computer Society Press, p. 135148. `https://dl.acm.org/doi/10.5555/800052.801871`.

47 Spring 2019 Solutions

Name Solution

Computer Architecture
LSU EE 4720
Midterm Examination

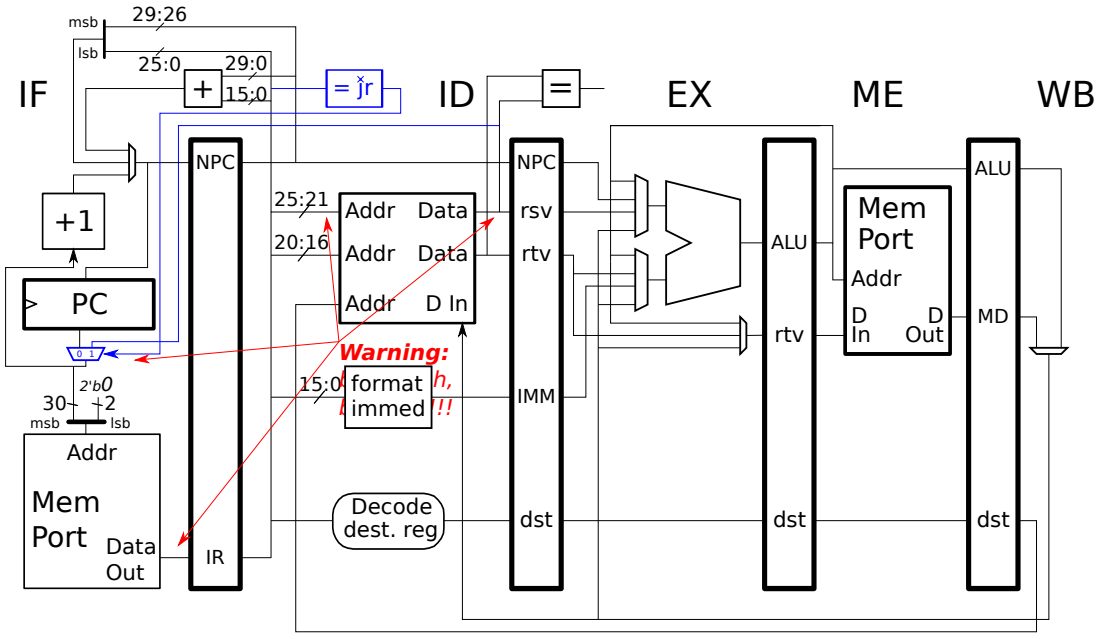
Wednesday, 27 March 2019, 9:30–10:20 CDT

Problem 1		(7 pts)
Problem 2		(17 pts)
Problem 3		(27 pts)
Problem 4		(12 pts)
Problem 5		(12 pts)
Problem 6		(25 pts)
Exam Total		(100 pts)

Alias It's okay when *we* rely on just one.

Good Luck!

Problem 1: [7 pts] The MIPS pipeline below implements a hypothetical MIPS `jr` instruction, the hardware for `jr` is shown in blue. Don't confuse `jr` with the existing MIPS `jr` instruction.



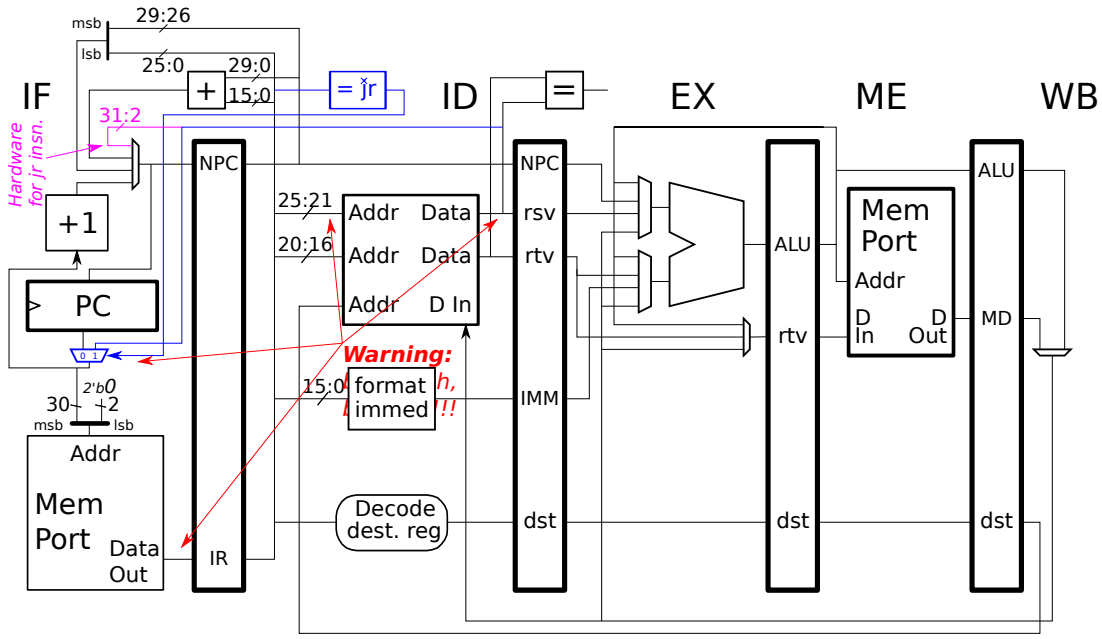
(*a*) The diagram shows a warning in red with lots of arrows and an explanation. Alas, the explanation is covered by the `format immed` box. **Assume that the hardware for `jr` is correct.** Then what can the warning be about?

☑ Reason for warning:

The warning is probably about the critical path which passes through the `IF`-stage memory port. The address input to a memory port should be available at the beginning of a clock cycle but in the `jr` implementation the `IF`-stage memory port address is taken from the register file, and that adds too much to the critical path.

For the record, the text under the `format immed` *box says, "Blah, blah, blah!!!".*

Problem 1, continued: (*b*) *Note: This part did not appear on the exam because the exam was already long enough.* There are two differences between ~~`jr`~~ and `jr`. Fragment A, below, uses `jr`. Complete Fragment B so that it uses ~~`jr`~~, making changes to account for these two differences. Fragment B must jump to the same location and perform the same computation as Fragment A. Register `r9` can be used for intermediate values. *Hint: The differences are when and where.*



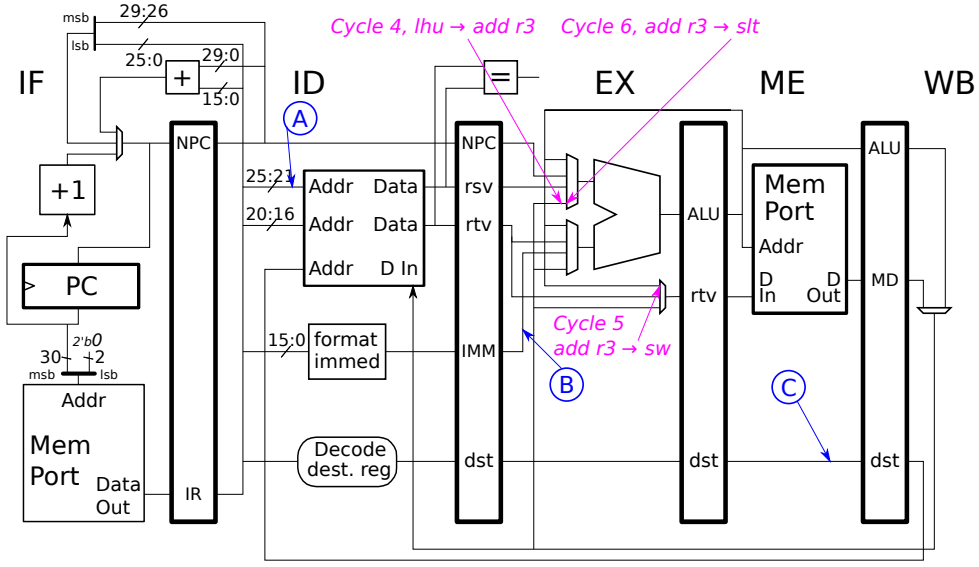
☑ Complete code, or for partial credit explain two differences.

Solution appears below, and the implementation of the ordinary `jr` instruction is shown above in purple. The `jr r1` instruction jumps to the address in `r1` after the delay-slot instruction, in this case `addi r2,r2,4`, is executed. The hardware in purple above shows the path used by the `jr` instruction. It carries `rsv` (the rs value, in the example the value of `r1`) from the register file (in ID) to the PC mux (in IF). As is done with the branch and jump targets, the two least-significant bits of the `rsv` are not used since these will be zero for any valid instruction address. The ~~`jr r1`~~ instruction is different in two ways. First, it jumps to the address `4*r1`. That's because unlike the path used by `jr`, the two-least significant bits of the register value are retained and two zeros are prepended to the less-significant side at the input to the IF-stage memory port. The second difference is that the target address for ~~`jr`~~ is at the input to the memory port when ~~`jr`~~ is in ID, meaning that the target is in IF while the ~~`jr`~~ is in ID, and so the target is fetched one cycle earlier than `jr` (and one cycle earlier than the branch instructions). This also means that ~~`jr`~~ lacks a delay-slot.

So that Fragment B jumps to the same address as Fragment A, the address must be shifted right by two bits. Because ~~`jr`~~ lacks a delay slot, the delay slot instruction for `jr`, `addi`, must be moved before the ~~`jr`~~.

```
# Fragment A -- Uses jr. Don't modify it.
lw r1, 0(r6)
jr r1
addi r2, r2, 4
xor r3, r4, r5

# Fragment B --  ☑  Finish code below, use ~~jr~~   SOLUTION
lw r1, 0(r6)     # Load address of instruction to jump to.
srl r1, r1, 2    # Divide address by 4.
addi r2, r2, 4   # ~~jr~~ lacks a delay slot, so move addi before ~~jr~~
~~jr~~ r1            # Jump to target
xor r3, r4, r5   # Not executed. (Unless reached by some other code.)
```

Problem 2: [17 pts] The code below executes on the illustrated implementation. The implementation has hardware that enables the `bne` to avoid the stall, but that hardware is not shown.



(*a*) The illustration has several circled letters pointing to wires. In the diagram below show the values on those wires on each cycle the value is used.

☑ Show values for ☑ A, ☑ B, and ☑ C, show these ☑ for each cycle used.

Solution appears below. In cycle 7 the B row (the immediate) shows the branch displacement, -6 instructions. For the C row it is important to show zeros for instructions that don't write a register, `sw` and `bne`.

```
LOOP: # Cycle    0    1    2    3    4    5    6    7    8    9    10   11
 lhu r1, 8(r2)   IF   ID   EX   ME   WB
 addi r2, r2, 2       IF   ID   EX   ME   WB
 add r3, r1, r3            IF   ID   EX   ME   WB
 sw r3, 12(r6)             IF   ID   EX   ME   WB
 slt r5, r3, r4                 IF   ID   EX   ME   WB
 bne r5, r0  LOOP               IF   ID   EX   ME   WB  # No stall? Next prob.
 addi r6, r6, 4                      IF   ID   EX   ME   WB

#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11
A:                    2    2    1    6    3    5    6                   # SOLUTION
B:                    8    2         12        -6    4                   # SOLUTION
C:                              1    2    3    0    5    0    6         # SOLUTION
#       Cycle    0    1    2    3    4    5    6    7    8    9    10   11
```

(*b*) For each dependency below highlight, on the illustration, the multiplexor input that provides the bypassed value. Also indicate the cycle in which the bypass occurs.

☑ Dependence from `lhu` to `add r3`. ☑ Cycle when bypass used.

☑ Dependence from `add r3` to `sw`. ☑ Cycle when bypass used.

☑ Dependence from `add r3` to `slt`. ☑ Cycle when bypass used.

Solution appears in the implementation diagram in purple.

Problem 3: [27 pts] In the previous problem the `bne` did not stall despite a dependence with `slt`. Code with a similar dependence appears below. Add hardware to the implementation below that correctly sets the Taken signal for such `slt rX, rY, rZ` to `bne rX, r0` dependencies. Branches without the dependence should not be effected. Note that the result of `slt` is 0 or 1. Some useful hardware is shown in blue.

```
LOOP: # Cycle      0  1  2  3  4  5  6
 slt r5, r3, r4    IF ID EX ME WB       # Note: r5 is 0 or 1.
 bne r5, r0  LOOP     IF ID EX ME WB
 addi r6, r6, 4          IF ID EX ME WB
```

☑ Add logic to set `Taken` correctly for the dependence described above.

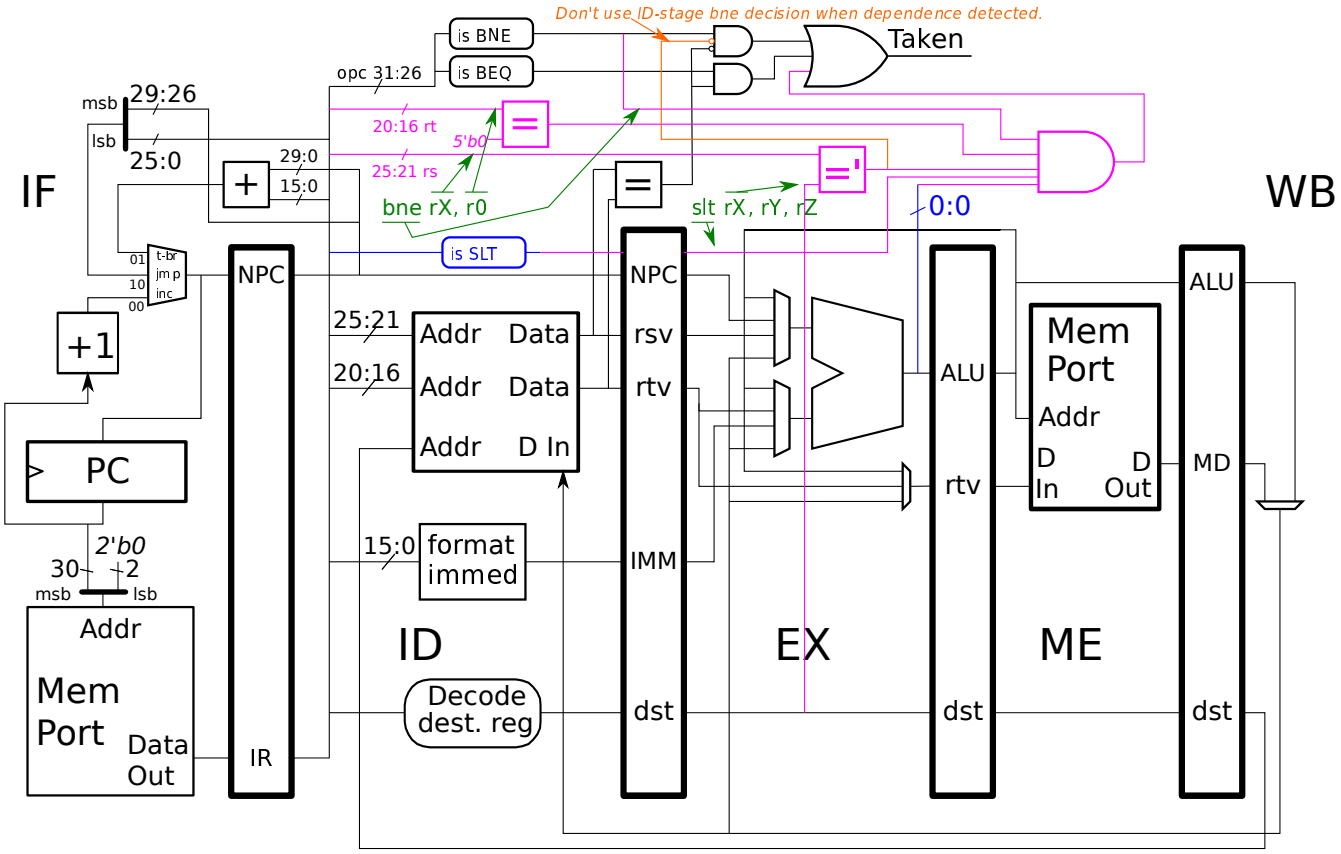
☑ Avoid costly solutions. No part of the solution should operate on 32 bits.

☑ Check that the second branch register is `r0`. ☑ Branches without the dependence should not be effected.

Solution appears below in purple and orange. Sample instructions are shown in green. The inputs to the big AND gate from top to bottom check for the following: (1) that the instruction in `ID` is a `bne`, (2) that the `rt` register of the branch is `r0`, (3) that the `rs` register of the branch is the same as the destination register of the instruction in `EX`, (4) that the instruction in `EX` is an `slt`, and (5) that the `slt` result is 1 (meaning that the condition is true).

To reduce hardware cost the logic checks whether `rt=0` (a 5-bit comparison) rather than `rtv=0` (a 32-bit comparison).

Note: The following is the solution to 2019 Homework 5 Problem 2. If the `slt/bne` dependence is detected then the branch taken condition computed using `rsv` and `rtv` (using the = logic) needs to be ignored. To do so a new bubbled input, shown in orange, has been added to the AND gate computing the `bne` taken condition. That input only checks for the dependence (`ID`-stage `rs` with `EX`-stage destination). (It doesn't matter what instruction is in the `EX` stage, if there's a dependence that taken signal should be suppressed.)



Problem 4: [12 pts] The loop below writes zeros to a range of memory.

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3  # Memory location at which to stop.
 addi r2, r2, -1
LOOP:
 sb r0, 0(r1)
 bne r1, r2, LOOP
 addi r1, r1, 1
```

(*a*) Compute the rate that it zeros memory when it runs on our bypassed 5-stage pipeline. Use an appropriate unit.

☑ Rate at which loop above copies data.

Short answer: Rate is $1/(7-3) = .25$ bytes per cycle.

Appearing below is a pipeline diagram showing the execution of the loop. The first iteration completes without a stall, but the dependence from the **addi** to the **bne** causes a one-cycle stall on subsequent iterations. The time for the second iteration is $7 - 3 = 4$ cyc and we can expect subsequent iterations to also take 4 cycles. Since only one byte is zeroed the rate is $\frac{1}{4}\frac{\text{B}}{\text{cyc}}$.

```
 # SOLUTION
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)      IF ID EX ME WB
 bne r1, r2, LOOP     IF ID EX ME WB
 addi r1, r1, 1          IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)            IF ID EX ME WB
 bne r1, r2, LOOP           IF ID -> EX ME WB
 addi r1, r1, 1                IF -> ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 sb r0, 0(r1)                        IF ID EX ME WB
 bne r1, r2, LOOP                       IF ID -> EX ME WB
 addi r1, r1, 1                            IF -> ID EX ME WB
```

(*b*) Apply loop unrolling and make other changes so that the code writes at the rate of two bytes per clock cycle, assuming favorable values of `r1` and `r3`. State those assumptions.

☑ Show unrolled loop and make other changes for 2 byte per cycle copy.

Short Answer: The unrolled loop appears below.

Explanation: To increase the rate at which memory is zeroed two `sw` instructions are used instead of one `sb`. The second `sw` uses an offset of 4, avoiding the need for a second `addi` instruction. Since the loop body has one more instruction than the original loop the branch does not need to stall, so the loop still takes 4 cycles per iteration. The loop zeroes memory at the rate of $8/4 = 2$ bytes per cycle.

☑ Assumption about `r1`: is a multiple of 4. (See below)

☑ Assumption about `r3`: is a multiple of 8. (See below)

Because the base address of `lw` must be 4-byte aligned, `r1` must be a multiple of 4. Because the address (`r1`) is incremented by 8 each iteration, the number of bytes to zero must be a multiple of 8.

```
 # Call Value: r1 is address of the start of the region to zero.
 # Call Value: r3 is the number of bytes to zero.
 add r2, r1, r3

# SOLUTION - Code
 addi r2, r2, -8
LOOP:
 sw r0, 0(r1)
 sw r0, 4(r1)
 bne r1, r2, LOOP
 addi r1, r1, 8

# Execution of solution code.
#
 add r2, r1, r3    IF ID EX ME WB
 addi r2, r2, -8      IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11  Time: 6-2 = 4 cycles
 sw r0, 0(r1)           IF ID EX ME WB
 sw r0, 4(r1)              IF ID EX ME WB
 bne r1, r2, LOOP             IF ID EX ME WB
 addi r1, r1, 8                  IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
 sw r0, 0(r1)                       IF ID EX ME WB
 sw r0, 4(r1)                          IF ID EX ME WB
 bne r1, r2, LOOP                         IF ID EX ME WB
 addi r1, r1, 8                              IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Problem 5: [12 pts] The MIPS code below adds an integer value loaded from memory to the constant π.

(*a*) Suppose no other instructions needed the value of `a0` that was loaded. Modify the code to use fewer instructions.

☑ Use fewer instructions.

Solution appears below. Since the value loaded into `a0` is only used by the FP code, it would be more efficient to load it into a FP register (using `lwc1`) and so avoid the need of an `mtc1` instruction to move the value from `a0` to a FP register.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        # lw $a0, 0($t2)  # SOLUTION - Replace lw ..
        lwc1 $f1, 0($t2)  # .. with a lwc1

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        # mtc1 $f1, $a0   # SOLUTION - Remove mtc1, since val already in f1.
        cvt.s.w $f2, $f1
        add.s $f3, $f0, $f2
```

(*b*) Modify the code below so that it would run correctly if `a0` were a floating-point value. Also use fewer instructions.

☑ Fix code so it works correctly if `a0` is FP.

Solution appears below. Since the loaded value is already FP there is no need for the conversion instruction, `cvt.s`. As in the previous problem, the value is loaded directly into a FP register, avoiding a need for the `mtc1`.

```
        .data
famous_constant: # 0x10010300
        .float 3.141592654
        .text
pie:

        # lw $a0, 0($t2)  # a0 is FP!  ☑  Fix code below for FP a0.
        lwc1 $f2, 0($t2)     # SOLUTION: Replace lw with lwc1.

        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)

        #  mtc1 $f1, $a0     # SOLUTION - Don't need move insn, val in a FP reg.
        #  cvt.s.w $f2, $f1  # SOLUTION - Don't need convert, it's already in FP repr.
        add.s $f3, $f0, $f2
```

Problem 6: [25 pts] Answer each question below.

(*a*) SPARC divides the 32 integer registers an instruction can access into four groups, `%l0 to %l7`, `%g0 to %g7`, `%o0 to %o7`, and `%i0 to %i7`. The names reflect how they might be used in programs, and how SPARC's register windowing feature affects them when `save` and `restore` instructions are executed. Explain what the first letter of each group stands for. Explain what happens to the values in those registers when `save` or `restore` instruction is executed.

☑ Word that `l`, `g`, `o`, and `i` each stand for.

They are: local, global, output, input.

☐ What happens to values on a `save` or `restore`.

Values in the global (`g`) registers aren't changed by `save` and `restore`.

FINISH.

(*b*) Instructions like `addi r1, r1, 1` occur frequently in programs. For this add-one-to-a-register operation CISC ISAs have a specialized instruction, for example `inc r1`. MIPS lacks such an increment instruction.

☑ Why do MIPS and other RISC ISAs lack such an instruction?

Because there would be no benefit to having an `inc` instruction. The `addi` instruction would do exactly the same thing. Furthermore, a new `inc` instruction would require a new opcode, and there are only a limited number of opcodes, so such an opcode would be wasted.

☑ Why do CISC ISAs have such an instruction?

☑ What is the benefit that RISC ISAs don't realize?

Short Answer: A CISC `inc` instruction would be shorter than a CISC `add` instruction, and so programs using `inc` would be shorter.

Explanation: CISC ISAs have variable-length instructions. So an `inc r1` instruction would take less space than an `addi r1, r1, 1` since there would be no need to specify the increment amount (it's always 1) and there would be no need to list a separate source and destination registers (since they'd be the same).

(*c*) A CPU design team has an area budget that they can use for bypass paths. There's not enough area for all the bypass paths they'd like. They are simulating these design alternatives to determine which is best.

Explain the role that the compiler people have in this process.

☑ Role for compiler writers in deciding on bypass paths.

They need to provide back-ends for each alternative design that optimizes taking into account the bypass paths they provide. By doing so compiled code would have fewer stalls than a compiler that optimized for an old design. This would enable the team to properly assess how well each set of bypass paths performs.

(*d*) Provide examples for the following common optimizations. Show code to which this optimization can apply and how it is optimized.

☑ Dead-code elimination. ☑ Example code and code after optimization.

Solution appears below. Because `planb` is false the `if` part is never executed, so after DCE only the else part remains.

```
// SOLUTION: Before DCE
bool planb = false;
if ( planb ) x = value >> ( log2p1(y) >> 1 );  else x = sqrt(y);

// After DCE.
// Note: Code below would be in compiler's intermediate representation.
x = sqrt(y);
```

☑ Constant propagation and folding. ☑ Example code and code after optimization.

Solution appears below.

```
// SOLUTION: Before constant propagation and folding.
int wp_lg = 5;
int wp_sz = 1 << wp_lg;
int wp_num = tid / wp_sz;

// After optimization.
// Note: Code below would be in compiler's intermediate representation.
int wp_num = tid / 32;
// Note: A strength reduction optimization would replace divide by a right shift.
```

(*e*) Described below are three tuning levels for C++ programs, two of which are SPECcpu tuning levels the other was just made up. Identify the one which is not a SPEC tuning level, and explain why it isn't.

Level A: *Each C++ program is compiled without optimization.*

☑ Is it a SPEC level? No ☑ If so, name it: _____

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Not appropriate for anyone, because almost everyone runs programs with some optimization.

Level B: *Each C++ program can have its own set of optimization flags.*

☑ Is it a SPEC level? Yes ☑ If so, name it: Peak

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Peak. For those who plan to have highly motivated experts tune their code.

Level C: *All C++ programs must be compiled with the same set of optimization flags.*

☑ Is it a SPEC level? Yes ☑ If so, name it: Base

☑ Indicate the users that will benefit from this level, or why no one would benefit.

Base. For those who plan to run code prepared with a typical level of effort.

Name Solution

Computer Architecture
LSU EE 4720
Final Examination

1 May 2019, 12:30–14:30 CDT

Problem 1		(22 pts)
Problem 2		(22 pts)
Problem 3		(21 pts)
Problem 4		(10 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Before

Good Luck!

Problem 1: (22 pts) Notice that the execution of the code fragment below suffers two stalls when executing on our 2-way superscalar MIPS implementation. The `add` stalls due to a dependence with `or` and the `sw` stalls due to a dependence with `add`. The MIPS implementation has three unconnected logic blocks that may be useful. Each must be connected to the opcode and func of the instruction in the appropriate slot. The output of the =or is 1 if the instruction is an `or`. The output of uses rs is 1 if the instruction uses the `rs` register as a source, likewise for uses rt.

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID ----> EX ME WB
 addi r6, r6, 8      IF -> ID ----> EX ME WB
```

(*a*) In the execution below the `sw` no longer stalls for `r3`. Add a bypass path that can be used by `sw` to get the `r3` value **in the execution below** but not for other cases.

☑ Add bypass path for `r3` so `sw` executes as shown. ☑ Label the path "Part a", and **do not add** unneeded bypass paths.

Solution appears in blue (several pages ahead). The `sw` needs the value of `r3` written by the `add`. A bypass path was added to the `rtv` mux in the `EX` stage, the path connects to the slot 1 instruction in the `ME` stage. As can be seen from the execution below the `add` is in slot 1 and when the `sw` is in `EX` (in cycle 4) the `add` is in `ME`, and so it can use the added bypass path.

```
# Cycle           0  1  2  3  4  5  6  7  8
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID -> EX ME WB
 sw r3, 4(r6)        IF -> ID EX ME WB
 addi r6, r6, 8      IF -> ID EX ME WB
```

(*b*) The `add` stalls due to the dependence with `or` carried by `r1`. Add control logic that detects such a dependence and connect it to the Stall ID OR gate at the lower right. The output of the logic should be `1` for any true dependence between two instructions in a group.

☑ Provide a stall signal when there is a dependence between the two instructions in `ID`.

Solution appears in green on the diagram a page or two ahead. The =' comparison units check whether the destination of the instruction in slot 0 matches the `rs` or `rt` register of the instruction in slot 1. The uses logic checks whether the `rs` and `rt` fields of the slot-1 instruction specify sources. (In many classroom examples it is assumed that every instruction uses `rs` as a source, here we are being more careful.)

(*c*) Notice that because the second operand is `r0`, the `or` just copies the value in `r2` to `r1`. Therefore the `add` could have used `r2` instead of `r1` and avoided the stall. Design hardware to perform such *substitutions*. The hardware, including control logic, should detect when an `or` is used as a copy (as above) and if so avoid the stall and deliver the correct source operand to the slot-1 instruction.

```
# Cycle           0  1  2  3  4  5
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID EX ME WB
 sw r3, 4(r6)        IF ID EX ME WB
 addi r6, r6, 8      IF ID EX ME WB
```

☑ Detect the substitution opportunity and ☑ suppress the Stall ID signal (from the previous part).

☑ Make sure the slot-1 instruction uses the correct value ☑ and that both instructions execute correctly.

☑ Of course, pay attention to cost. Nothing added for this problem should touch 32 bits.

Solution appears a page or two ahead in purple. The logic at the lower left checks for a slot-0 `or` using `r0` as the `rt` operand. If such a copy is found and if the `rs` source of the slot-1 instruction uses `or` destination the stall is suppressed and the `rs` register *number* of the slot 1 instruction is substituted. The register number is 5 bits, while the value is 32 bits, so it is far less expensive to substitute the number than the value.

(*d*) *The following is a bonus question that did not appear on the original exam. Bonus for whom you ask? Definitely a bonus for those who took the class in the Spring 2019 semester and took a look at the posted exam. Those (you) will have an opportunity to make connections between concepts learned in the class and that will provide a deeper understanding and longer retention.* Yes, the substitution hardware eliminates a stall. Suppose that `r2` had to be copied into `r1`. Provide an argument that substitution hardware is a waste of resources, illustrate with an example. Provide another argument—also with an example—that substitution hardware eliminates a stall that cannot be eliminated in another way. Whether substitution is a good idea will depend on whether the example illustrating its utility is representative of realistically compiled actual code.

☑ Argument against substitution hardware. ☑ Code example.

The substitution hardware is not needed if the compiler can use the source register of the substitution instruction. For the code sample above the compiler would emit `add r3, r2, r4`, eliminating the dependence. See the code below. (It is always better to avoid a stall using a compiler optimization than by hardware changes.)

```
# Cycle          0  1  2  3  4  5   # SOLUTION. No stall, no costly hardware needed!
 or r1, r2, r0   IF ID EX ME WB
 add r3, r2, r4  IF ID EX ME WB     # Replacement for add r3, r1, r4.
 sw r3, 4(r6)       IF ID EX ME WB
 addi r6, r6, 8     IF ID EX ME WB
```

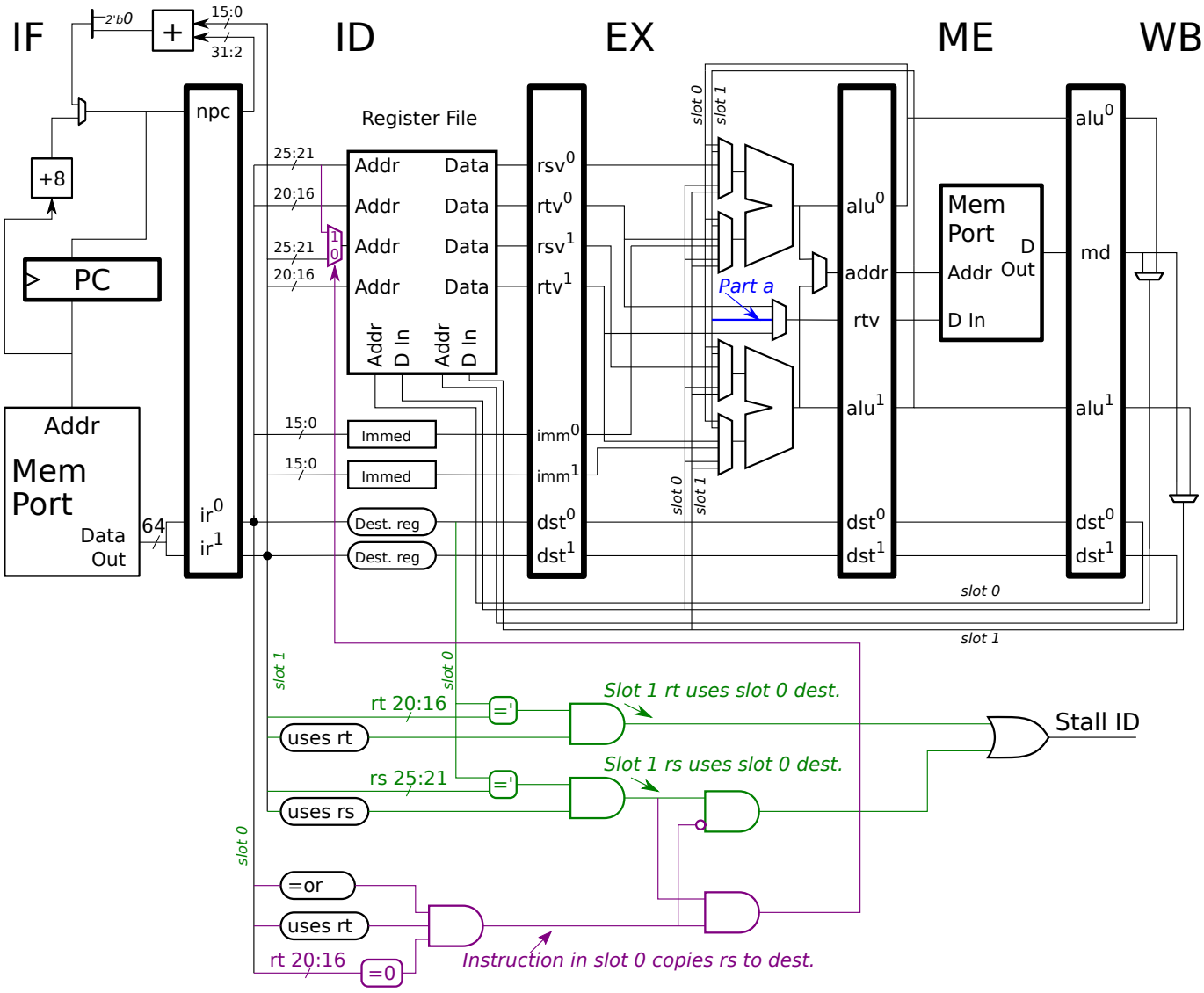
☑ Argument for substitution hardware. ☑ Code example.

The compiler cannot easily perform substitution when the two instructions are in different basic blocks. For example, in the code below the compiler cannot just replace `add r3, r1, r4` with `add r3, r2, r4` because that would not be correct if the `bne` were not taken. This is a weak example for the substitution hardware because the compiler could prepare two code sequences, both containing the `add`, `sw`, and `addi` instructions.

```
 # SOLUTION
 bne r8, r9 TARG  # If branch taken add uses r1, if not taken r2.
 nop
 or r1, r2, r0
TARG:
 add r3, r1, r4
 sw r3, 4(r6)
 addi r6, r6, 8
```

(A basic block is a sequence of instruction that can only have a control transfer [such as a branch] at the end, and which can only have a branch label [labels `TARG` and `LOOP` are frequently used in class] at the beginning. A basic block always starts with the first instruction and always ends with the last instruction.)

Solution to hardware questions appears below. A discussion of the solution appears on prior pages.




```
# SOLUTION:       Use this execution to help in understanding the solution.
# Cycle           0  1  2  3  4  5
 or r1, r2, r0    IF ID EX ME WB
 add r3, r1, r4   IF ID EX ME WB
 sw r3, 4(r6)        IF ID EX ME WB
 addi r6, r6, 8      IF ID EX ME WB
# Cycle           0  1  2  3  4  5
```

Problem 2: (22 pts) Appearing below is our MIPS FP pipeline with the comparison units added.

(*a*) Show the execution of the following fragment on this hardware.

☑ Show execution up to second fetch of `lwc1`. ☑ Pay attention to dependencies, including the FP condition.

Solution appears below. The second fetch of `lwc1` is shown using static instruction order. Note that `c.lt.s` depends on `add.s f2` (carried by `f2`) and that `bc1t` depends on `c.lt.s` (carried by `fcc`).

```
#                SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
LOOP:
 lwc1 f1, 0(r1)  IF ID EX ME WF                    IF
 add.s f2, f1, f2   IF ID -> A1 A2 A3 A4 WF
 add.s f4, f1, f3      IF -> ID A1 A2 A3 A4 WF
 c.lt.s f2, f6            IF ID ----> C1 C2 WF
 bc1t LOOP                   IF ----> ID ----> EX ME WB
 addi r1, r1, 4                       IF ----> ID EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 swc1 f2, 4(r1)
 and r1, r1, r9
```

(*b*) Notice that there are two circled letters (in blue) in the lower part of the diagram. For each letter provide a code fragment that causes the labeled wire to go to logic 1.

☑ Code fragment that makes A logic 1.

☑ Show its execution and ☑ indicate cycle at which A is 1.

Solution appears below. Wire A will be 1 in cycle 3, that's when the `lwc1` is in `ID` while the `add` is in `A2`. The `lwc1` must stall one cycle to avoid a `WF` structural hazard in cycle 6.

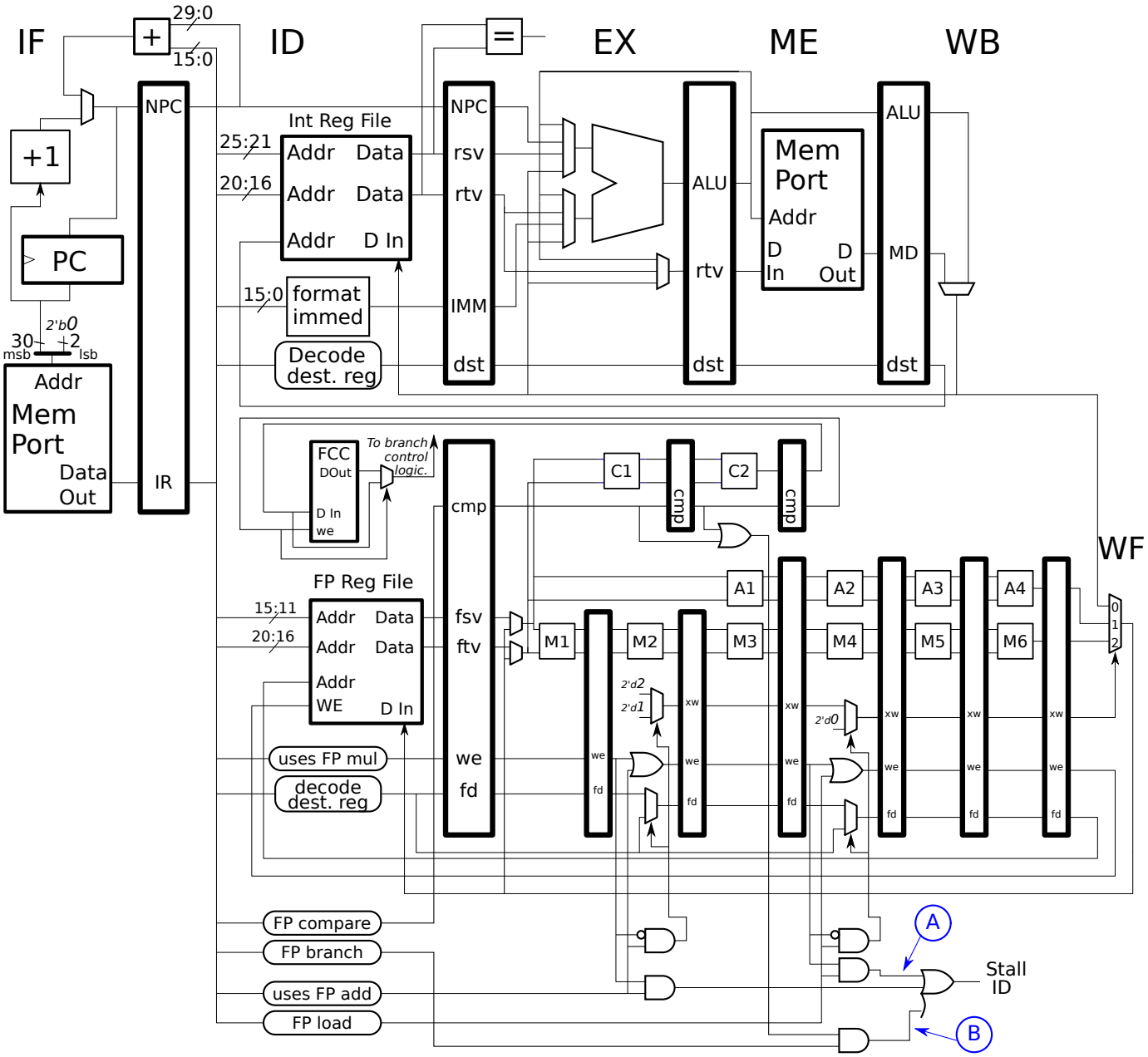
```
#                SOLUTION
# Cycle          0  1  2  3  4  5  6  7
 add.s f1, f2, f3  IF ID A1 A2 A3 A4 WF
 xor r1, r2, r3       IF ID EX ME WB
 lwc1 f4, 0(r4)          IF ID -> EW ME WF
# Cycle          0  1  2  3  4  5  6  7
```

☑ Code fragment that makes B logic 1.

☑ Show its execution and ☑ indicate cycle at which B is 1.

Solution appears below. Wire B will be 1 in cycles 2 and 3, that's when `bc1t` has to wait for the comparison result to reach the FCC.

```
#              SOLUTION
# Cycle        0  1  2  3  4  5  6  7
 c.lt.s f0, f1  IF ID C1 C2 WF
 bc1t TARG         IF ID ----> EX ME WB
```


IF
ID
EX
ME
WB
WF
+
29:0
15:0
=
NPC
+1
PC
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
IR
Int Reg File
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
FCC
DOut
D In
we
To branch
control
logic.
cmp
C1
C2
cmp
cmp
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr
WE D In
fsv
ftv
we
fd
uses FP mul
decode
dest. reg
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
FP compare
FP branch
uses FP add
FP load
A
B
Stall
ID

Problem 3: (21 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B1` has a repeating pattern, two repetitions are shown. Branch `B2` repeatedly and randomly emits three sequences, `a`, `b`, and `c`. Sequence `a` is `NT`, sequence `b` is `NNNTT` (five outcomes), and sequence `c` is `NNNN NTTT` (eight outcomes). After finishing one sequence, a new one is started. Sequence `a` is chosen with probability .4, sequence `b` with probability .5, and `c` with probability .1.

Here are some examples of `B2` outcomes, with spaces placed between the sequences for clarity. Example 1: `NT NNNTT NT NNNNNTTT` (that's `a`, `b`, `a`, `c`). Example 2: `NNNNNTTT NT NT NNNTT NNNTT` (that's `c`, `a`, `a`, `b`, `b`).

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   T  N  N  N  N  T  T    T  N  N  N  N  T  T

B2:    (a, p=.4): NT    (b, p=.5): NNNT T    (c, p=.1): NNNN NTTT
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

```
    SOLUTION WORK
    0   1  0  0  0  0  1   2   3  2  1  0  0  1  2  <- 2-bit counter values
B1:   T  N  N  N  N  T  T    T  N  N  N  N  T  T
      x              x  x       x  x        x  x   <- Prediction outcomes
                        ----------------------  <- First occurrence repeating pattern.
```

Short answer: $\frac{3}{7}$, see work above.

Explanation: The line just below `SOLUTION WORK` (above) shows 2-bit counter values for `B1`, assuming that the counter starts at zero. The prediction outcomes are shown below the branch outcomes. To compute the prediction ratio we need to use a repeating pattern. Such a pattern is underlined. Note that the counter value at the start and end is the same and that the underlined part corresponds to a set of repeating branch outcomes. The prediction accuracy is $\frac{(7-4)}{7} = \frac{3}{7}$.

☑ What is the accuracy of the local predictor on branch `B1`?

The pattern can easily fit the 8-outcome local history, so the accuracy is 100%.

☑ What is the accuracy of the local predictor on branch `B2`?

First, lets number the outcomes on each pattern:

```
a: 01
   NT

b: 0123 4
   NNNT T

c: 0123 4567
   NNNN NTTT
```

Start by considering the prediction of the first outcome of each of the three patterns. Then because one of the three patterns has just finished the local history at this point will be either `*****TNT`, `**TNNNTT`, or `NNNNNTTT`, where the `*` can be either `N` or `T`. The first outcome of all three are `N` and so the `PHT` entries for all the local histories will be zero (after warmup) and so the first outcome of all three patterns will be predicted correctly (assuming that the local histories only occur at the beginning, which is easy to verify). After the first outcome is predicted the local histories will be one of `****TNTN`, `*TNNNTTN`, or `NNNNTTTN`. If pattern `a` is occurring the next outcome is `T`, otherwise `N`. Pattern `a` occurs with probability .4 and so it is more likely that an `N` occurs. The `PHT` entries for the local histories are more likely to be 0 or 1, but it's not certain because it is possible that there are two consecutive occurrences of `a`. So for pattern `a` the first outcome is predicted with 100% accuracy and the second outcome at a lower accuracy, which can be approximated as 50% and which can be solved exactly using a Markov chain.

If outcome 1, the second outcome using the number above, is `N` then outcome 2 will be predicted with 100% accuracy because it is always `N`. That is the local histories `***TNTNN`, `TNNNTTNN`, `NNNTTTNN` only appear when the next outcome is `N` and so the `PHT` entries will reach zero after warmup. Outcome 3 will be predicted `T` nearly 100% of the time because `b` is much more likely than `c`. Outcomes 4 and later will be predicted perfectly because they are unambiguous.

Here is the number of correct predictions for each pattern: `a`, $1 + 0.5 = 1.5$; `b` $1 + 0.5 + 1 + 1 + 1 = 4.5$, and `c`$=1 + 0.5 + 1 + 0 + 1 + 1 + 1 + 1 = 6.5$. The accuracy during `a` is $\frac{1.5}{2} = .75$, during `b` is $\frac{4.5}{5} = .9$, and during `c` is $\frac{6.5}{8} = .8125$. To compute the overall prediction accuracy we need to weight these by how frequently they occur. That is:

$$\frac{.4 \times 1.5 + .5 \times 4.5 + .1 \times 6.5}{.4 \times 2 + .5 \times 5 + .1 \times 8} = .853659$$

.

☑ What is the accuracy of the bimodal predictor on branch `B2`?

The problem is solved by first considering the effect that each pattern has on the 2-bit counter. Consider pattern `b`. It starts with three `N`'s guaranteeing that the counter will be zero when the first `T` is predicted, and that the counter will be 2 after the second `T`. Similarly, the counter will be 3 after sequence `c` completes. Unless the counter is zero, pattern `a` does not change the counter. Therefore, at the start of a sequence the counter will be either 2 or 3. It will be 2 with probability $P(k = 2) = \frac{.5}{.5+.1} = \frac{5}{6}$ and 3 with probability $P(k = 3) = \frac{.1}{.5+.1} = \frac{1}{6}$, where $P(k = x)$ is the probability that the 2-bit counter is x. Let $P(a|k)$ denote the prediction accuracy during pattern `a` given that the 2-bit counter value is k and remember that $k \in \{\, 2, 3 \,\}$. Then $P(a|2) = 0$, $P(a|3) = \frac{1}{2}$, $P(b|2) = \frac{2}{5}$, $P(b|3) = \frac{1}{5}$, $P(c|2) = \frac{5}{8}$, and $P(c|3) = \frac{4}{8}$. The probabilities need to be properly combined. First,

$$P(\checkmark|a) = P(a|2)P(k = 2) + P(a|3)P(k = 3) = 0 \times \frac{5}{6} + \frac{1}{2}\frac{1}{6},$$

where $P(\checkmark|a)$ is the prediction accuracy during pattern `a`. Similarly, $P(\checkmark|b) = \frac{2}{5}\frac{5}{6} + \frac{1}{5}\frac{1}{6} = \frac{11}{30}$ and $P(\checkmark|c) = \frac{5}{8}\frac{5}{6} + \frac{4}{8}\frac{1}{6} = \frac{29}{48}$.

Each pattern's probability needs to be weighted by how often the pattern occurs. Pattern `a` has a respectable probability of .4 but only two outcomes, so its relative contribution is $.4 \times 2$. The relative contribution of patterns `b` and `c` are $.5 \times 5$ and $.1 \times 8$. If we arrived at some random time and waited for the next `B2` execution the probability of `B2` being in pattern `a` would be $\frac{.4 \times 2}{.4 \times 2 + .5 \times 5 + .1 \times 8} = \frac{.8}{4.1} \approx .195$.

So the overall branch prediction ratio for `B2` is

$$\begin{aligned} P(\checkmark|\texttt{B2}) &= \frac{P(\checkmark|a).4 \times 2 + P(\checkmark|b).5 \times 5 + P(\checkmark|c).1 \times 8}{.4 \times 2 + .5 \times 5 + .1 \times 8} \\ &= \frac{44}{123} \approx .357724 \end{aligned}$$

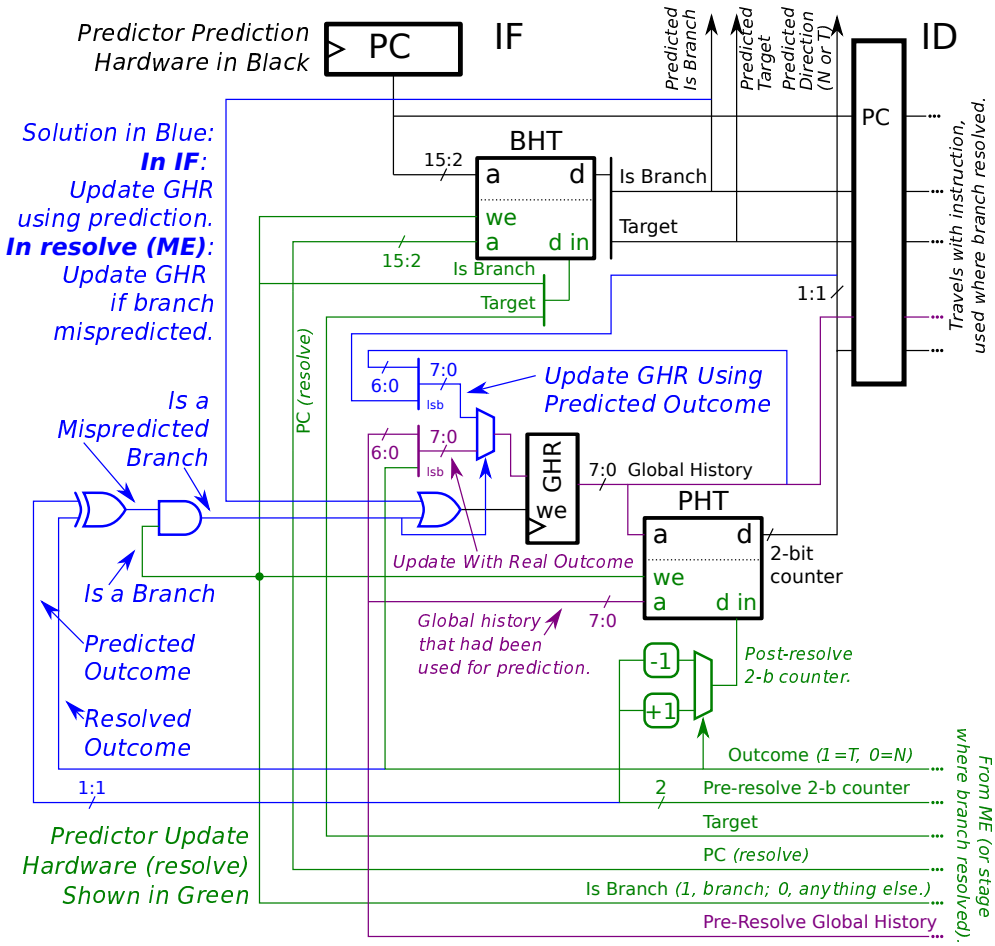
(*b*) Appearing below is a diagram of our global predictor. Notice that the GHR is not updated until the branch resolves. Modify the predictor so that the GHR is updated when the branch is being predicted (in IF) using the *predicted* outcome. When the branch resolves check whether the prediction was correct, and if not (if it was mispredicted) write the correct history into the GHR.

The following is interesting background material omitted from the original exam. The importance of updating the GHR using the predicted outcome increases with the number of post-branch instructions that are in the pipeline at the time a branch resolves. Consider our five-stage pipeline with branches resolving in ME. In that case there are just three post-branch instructions. For an 8-way superscalar pipeline there would be $3 \times 8 = 24$ instructions. One or more of those 24 instructions could itself be a branch. In the unmodified design below those branches would have been predicted with a GHR that lacked the outcome of the resolving branch and those that followed. The problem is much greater in dynamically scheduled systems where over 100 instructions can be *in flight*. For that reason global-like predictors in dynamically scheduled systems use designs like the one requested for this problem.

☑ Add hardware to detect whether the resolving branch has been mispredicted.

☑ During prediction write GHR based on prediction, ☑ during resolve apply corrected GHR if branch mispredicted.

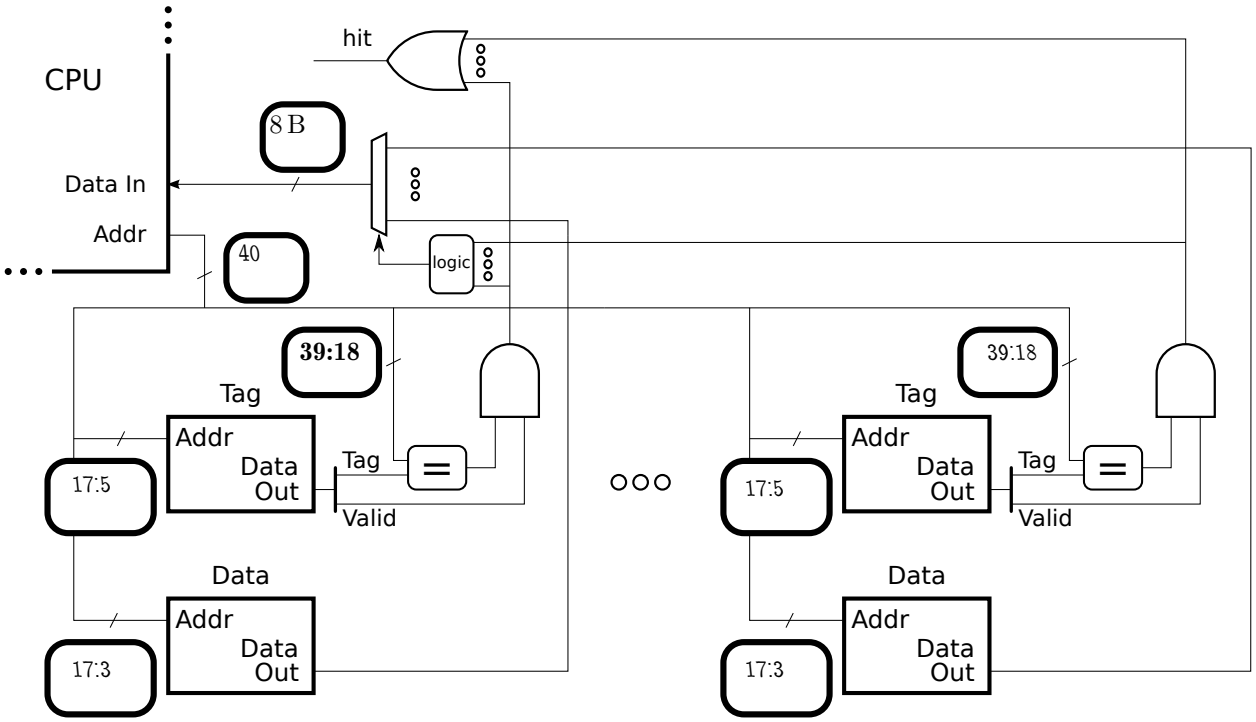
Solution appears below in blue. The mux at the input to the GHR selects either an updated global history for the branch in IF (which we hope to be the frequent case) or a corrected global history for the branch in ME (also identified as *resolve*). The latter case is only used if the branch is mispredicted, which is determined by XORing the predicted and resolved outcomes (if they are different the result is 1) and also making sure that the instruction in ME is a branch.



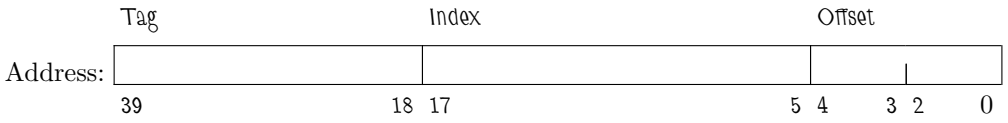
Problem 4: (10 pts) The diagram below is for a 4 MiB set-associative cache with a line size of 32 B. The character size is the usual 8 bits. Other information about the cache can be deduced using hints in the diagram. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $32 = 2^5$.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)



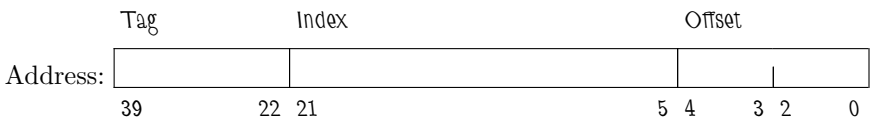
☑ Associativity:

The associativity is 16. The associativity is determined based on the given cache capacity, 2^{22} bytes, and the capacity of an individual data store, 2^{18} bytes. Since the cache capacity is the sum of the data store sizes, the associativity must be $\frac{2^{22}}{2^{18}} = 2^{22-18} = 16$.

☑ Memory Needed to Implement ☑ Indicate Unit!!:

It's the cache capacity, 4 MiB plus $16 \times 2^{18-5}\,(40 - 18 + 1)$ b $= 4\,\text{MiB} + 3014656\,\text{b}$.

☑ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.



The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 B (which is 2^6 B). The code fragment starts with the cache empty; consider only accesses to the array.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int ILIMIT = 1 << 11;    // = 2^11

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^6 = 64$ bytes is given. The size of an array element, which is of type int, is $4 = 2^2$ B, and so there are $2^6/2^2 = 2^{6-2} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the hit ratio is $\frac{15}{16}$.

Problem 5: (25 pts) Answer each question below.

(*a*) Appearing below are simple C routines and corresponding MIPS assembler code. C variable names match the MIPS registers to which they were assigned. Register `v0` is used for the return value. The first C routine, `proc1`, operates on 32-bit signed integers. Further below are two similar C routines, `proc2` and `proc3`, each followed by the MIPS routine written for `proc1`—which is wrong because the MIPS routine is only correct for `proc1`. Rewrite those MIPS routines for `proc2` and `proc3`. Note that `int16_t` is a signed 16-bit integer and `uint8_t` is an unsigned 8-bit integer.

```
int32_t proc1(int32_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Code below is correct for proc1.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

☑ Modify MIPS code for `proc2`. Pay attention to ☑ size and ☑ sign. ☑ Eliminate any unneeded instructions.

```
int16_t proc2(int16_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc2.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

SOLUTION

```
sll $t0,   $a1,   1    # Element size is 16 bits which is 2 chars, so mult by 2.
add $t0,   $t0,   $a0
lh $v0,   0($t0)       # Because element is two bytes change lw to lh ..
lh $t1,   2($t0)       # .. and change offset to 2.
jr $ra
add $v0,   $v0,   $t1
```

☑ Modify MIPS code for `proc2`. Pay attention to ☑ size and ☑ sign. ☑ Eliminate any unneeded instructions.

```
uint8_t proc3(uint8_t *a0, int a1) { return a0[a1] + a0[a1+1]; }
# Modify MIPS code to be correct for proc3.
sll $t0,   $a1,   2
add $t0,   $t0,   $a0
lw $v0,   0($t0)
lw $t1,   4($t0)
jr $ra
add $v0,   $v0,   $t1
```

SOLUTION

```
#                          Element size is 8 bits, or one byte.
add $t0,   $a1,   $a0    # No need to scale a1 (mult by elt size) before adding it to a0.
lbu $v0,   0($t0)        # Element size is one byte and unsigned ..
lbu $t1,   1($t0)        # .. so load using lbu and use offset of 1.
jr $ra
add $v0,   $v0,   $t1
```

(*b*) The statement below is based on a lack of understanding of how compilers work. Explain the misunderstanding and otherwise correct the statement.

> *It takes a great deal of effort to write a correct and effective compiler optimizer. Therefore optimizers are written for popular high-level languages such as C++11 but not for less popular languages such as COBOL.*

☑ The misunderstanding about compilers is:

...that optimization is performed on high-level code. In fact, optimization is mostly performed on an *intermediate representation* which typically is the same for all high-level languages the compiler can handle. So effort on optimizing the intermediate representation would benefit all those high-level languages.

It is the compiler *front end* that translates high-level languages into the intermediate representation.

☑ How does that change the conclusion about which languages get better optimization?

The conclusion should be that improvements to a compiler's optimizer benefit all high-level languages that the compiler supports.

(*c*) Chip A has five 4-way superscalar cores. Chip B has 20 scalar cores. The cores are similar to our pipelined MIPS implementations. All cores use a 1 GHz clock.

☑ Compute the peak execution rate in units of instructions per second of ☑ Chip A and ☑ Chip B.

Each chip can execute 20 instructions per cycle or 20×10^9 insn/s.

☑ Why would Chip A run faster on simple code, such as the routines used in the homework assignments?

It's reasonable to assume that "simple" code is single-threaded (not parallelized) and so it will run on only one core. A Chip A core executes at up to 4 IPC, ideally four times faster than a Chip B core which runs at just 1 IPC. Though in typical circumstances Chip A won't be 4 times faster, it will still be better than Chip B for single-thread code.

☑ Which chip might be less expensive? Explain.

Chip B. The number of ALU bypass paths (multiplexor inputs) for a w-way, five-stage implementation might be $4w^2$. The total number of bypass paths for Chip A is $5 \times 4 \times 4^2 = 320$ and the total number for Chip B is $20 \times 4 \times 1^2 = 80$, which is a lot less. What would you rather do, find the money to buy Chip A or parallelize your code so that it can run faster on Chip B? Of course, the answer depends on the situation.

(*d*) Answer the following questions about ISAs.

☑ Implementations of VLIW ISAs are supposed to be less costly and have higher performance than superscalar implementations of conventional ISAs. Is Intel Itanium a good example of that? Explain

No. Intel loaded Itanium with costly features, such as a large number of registers with rotating windows (for use in software pipelining), so there was no apparent cost benefit. The performance was not spectacular either, some blamed contemporary compilers for not being able to properly exploit those costly features.

☑ What important concept came out of the development IBM System/360?

That of an "Instruction Set Architecture (ISA)" (spoken using air-quotes) which would describe what the hardware would do, but not how it would do it. The ISA was intended for implementations across a product line (low- to high-end) and for implementations in the future, perhaps as long as 15 years.

They also made use of computers to prepare documentation. [I'm tempted to use my air quotes again for the phrase "word processing" but I'm not sure it originated with the 360 project, and if I start looking into it now who knows when I'd get back to real work.]

☑ VAX is a good example of which ISA type?

CISC.

☑ True or false: IA-32 (a.k.a. x86) was widely adopted because of its elegant design? Explain.

False! The original ISA was not designed for a long life. It had many compromise features such as requiring a pair of registers to specify a 32-bit memory address.

(*e*) The SPECcpu benchmarks can be run at two tuning levels, *base* and *peak*. Base scores are useful to those running software developed using typical practices.

☑ What kind of computer buyers should use peak scores?

Buyers who intend to develop or buy code highly tuned for the machine it will run on. Such buyers don't mind spending lots of money or effort to achieve 5% higher performance than those other guys get using standard development practices.

☑ How do the SPEC rules for preparing base and peak runs differ?

The base rules dictate that all benchmarks using the same language shall use the same optimization flags. This might reflect the practice of an experienced programmer having a good set of flags (and for most, that would be -O3) that would be used from project to project. The peak rules allow each benchmark to use different optimization flags. That might reflect the practice of a programmer obsessively tweaking flags to get the best performance. (Most programmers in that situation should look to their own code first to find opportunities for improving performance.)

48 Spring 2018 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 19 March 2018, 9:30–10:20 CDT

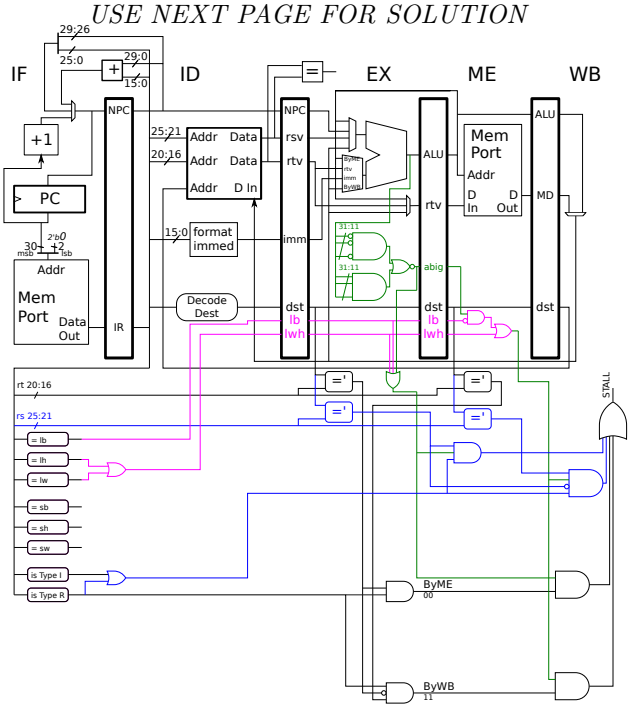
Problem 1		(25 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias or r4, 25, r6

Good Luck!

Problem 1: [25 pts] Appearing below is the solution to Homework 4, in which additional control logic is added for the 12-bit bypass paths. The illustrated hardware generates stall signals for a load/use dependence and for cases in which the value that needs to be bypassed is unknown or too wide for the 12-bit bypass paths.

USE NEXT PAGE FOR SOLUTION



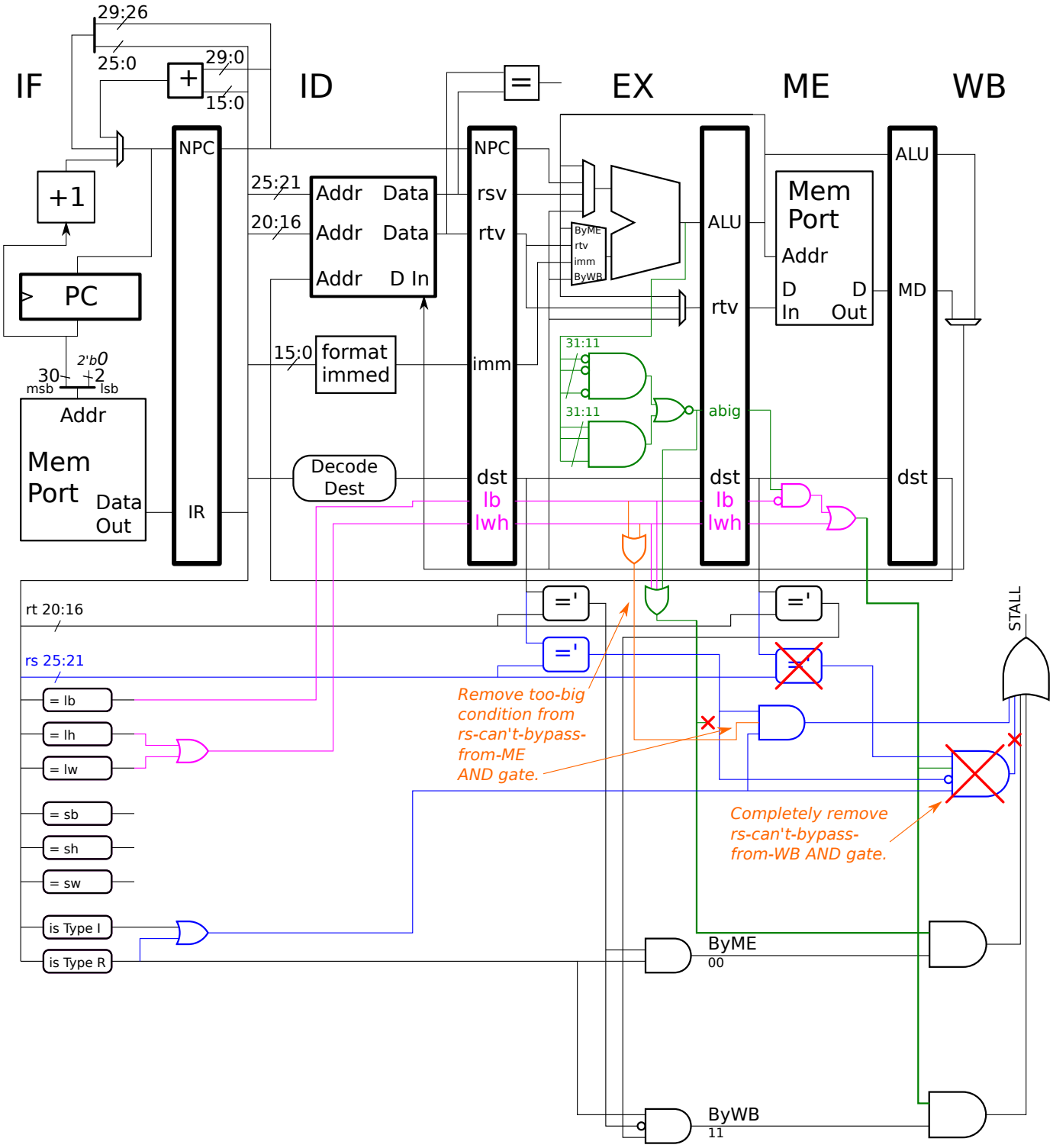
USE NEXT PAGE FOR SOLUTION

(*a*) Suppose, on further consideration, it was decided that full-sized, 32-bit bypass paths to the upper ALU mux were needed. Elsewhere 12-bit bypass paths would be retained. Modify the hardware so that a stall signal would no longer be generated for such too-big values to the upper ALU mux. Do so **without** affecting stalls for the 12-bit bypass paths to the lower ALU mux and without affecting stalls for load/use dependencies.

```
# Should not stall anymore, regardless of size of r1.
add $r1, $r2, $r3
lw $r4, 0($r1)

# Should still stall.
lw $r5, 0($r6)
addi $r7, $r5, 9

# Should still stall if r1 too big for 12-bit bypasses.
add $r1, $r2, $r3
sub $r5, $r6, $r1
```

☑ Remove hardware generating stall due to values too large for upper ALU bypass paths.

☑ **Do not** change stalls for load/use cases and bypasses to lower ALU mux.



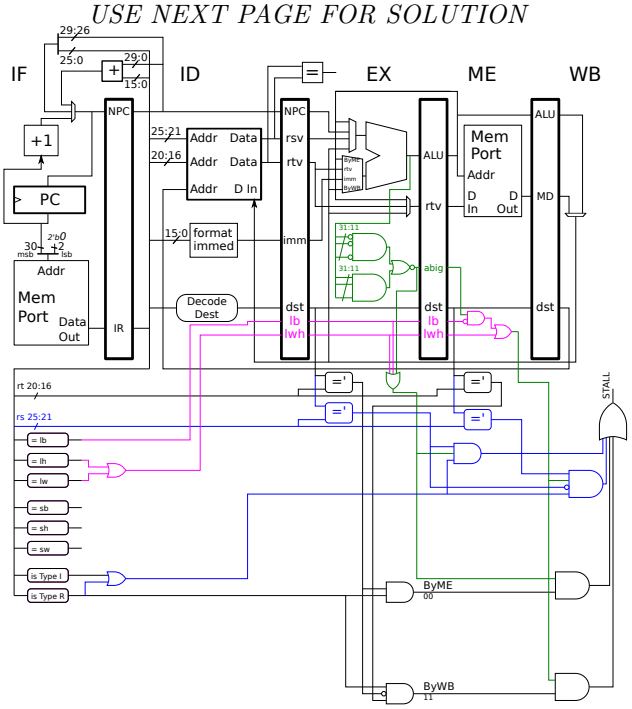
Solution appears above in orange.

Problem 1, continued:

(*b*) Suppose that the bypass paths to the `EX`-stage `rtv` mux were also just 12 bits wide. Modify the control logic on the next page so that a stall signal would be generated when appropriate for store instructions.

Note that a `lb` produces a value small enough for the 12-bit bypass paths and that the store value needed by a `sb` is always small enough for the 12-bit bypass paths. See the examples below.

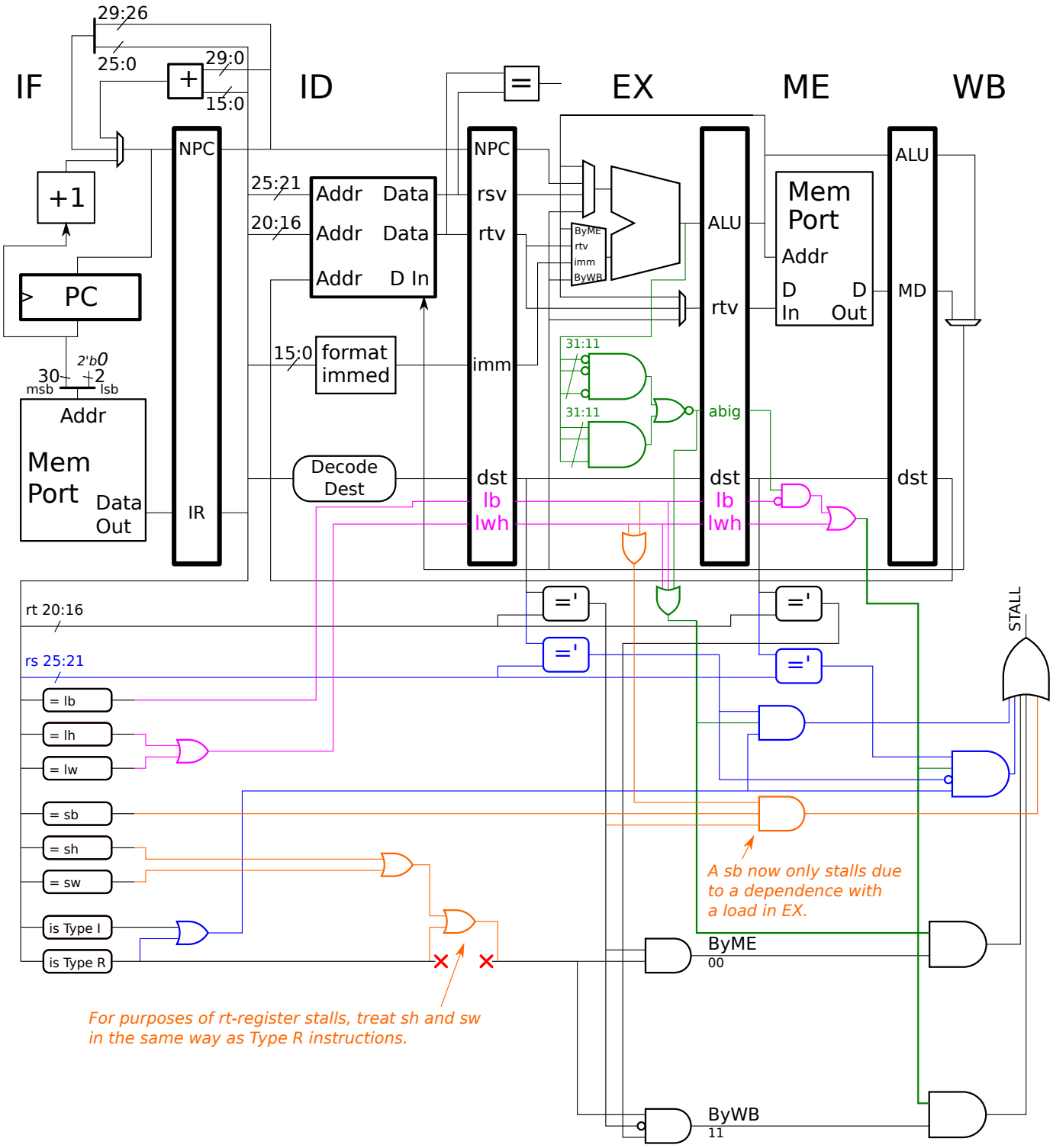
USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

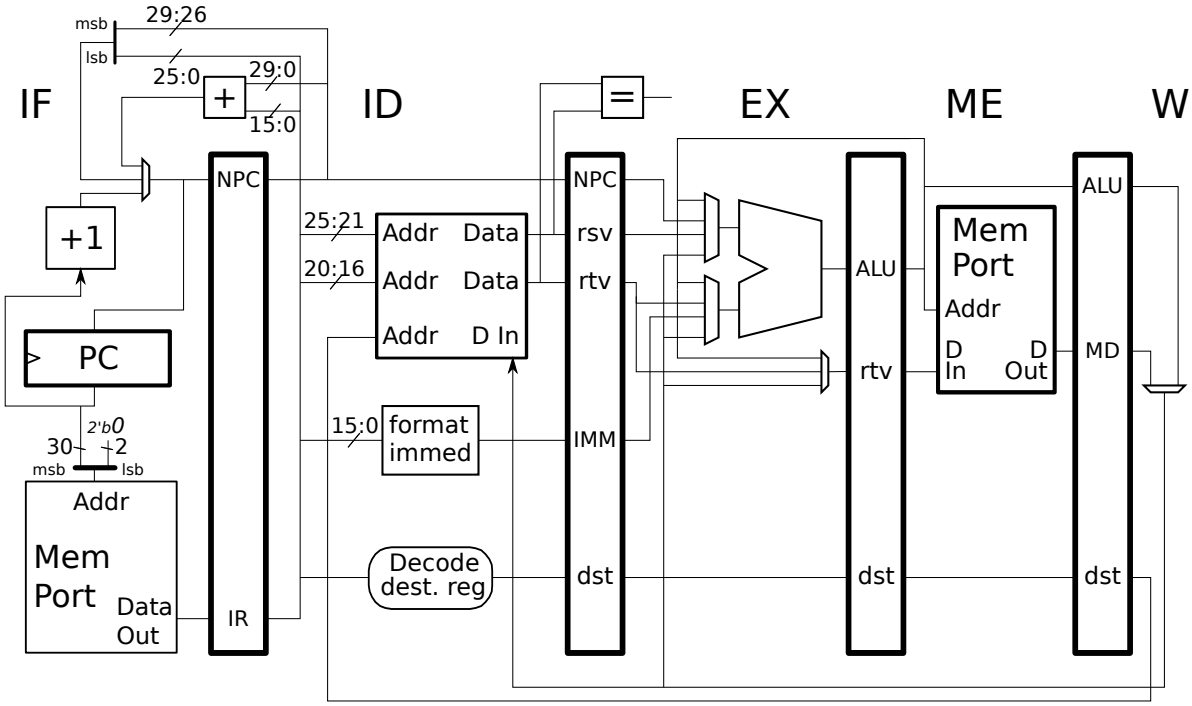
```
# sb should not stall for this dependency.
add $r1, $r2, $r3  IF ID EX ME WB
sb $r1, 0($r4)        IF ID EX ME WB

# sw should stall for one cycle.
lb $r1, 0($r5)     IF ID EX ME WB
sw $r1, 0($r4)        IF ID -> EX ME WB

# sh should stall only if r1 is too large.
add $r1, $r2, $r3  IF ID EX ME WB
sh $r1, 0($r4)        IF ID ----> EX ME WB
```

☑ Generate stall for unbypassable value from prior instructions to store value (not store address). ☑ Consider store size and any load size. (See examples on previous page.)



Solution appears above in orange.

Problem 2: [20 pts] Answer the following questions about two versions of our bypassed MIPS implementation.



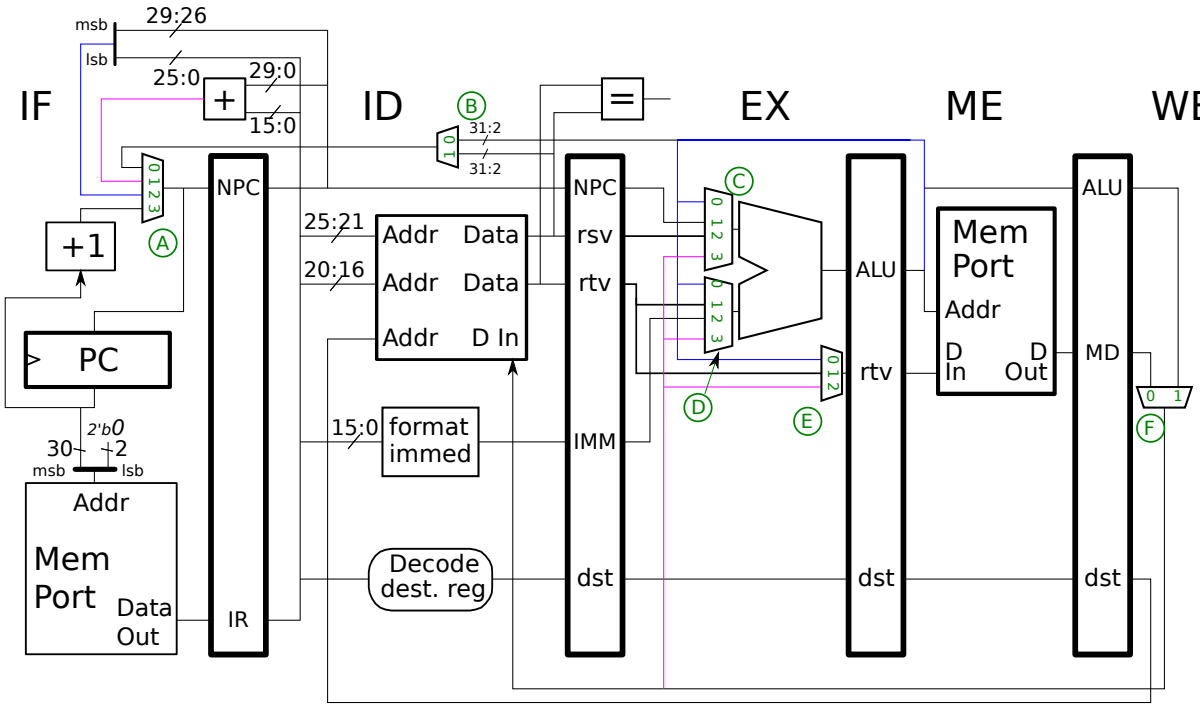
(*a*) Show the execution of the code below on the implementation illustrated above when the branch is taken.

☑ Show Execution.

☑ Check the code for dependencies.

```
# SOLUTION
# Cycles          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
lw r2, 0(r4)      IF ID EX ME WB
addi r1, r2, 3       IF ID -> EX ME WB
bne  r1, r3  TARG       IF -> ID ----> EX ME WB
ori r1, r9, 10                IF ----> ID EX ME WB
andi r11, r1, 14
# Cycles          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
TARG:
xor r20, r11, r1                       IF ID EX ME WB
```

(*b*) Each mux in the implementation below is labeled with a circled letter, and mux inputs are numbered. Some wires are colored to make them easier to follow. Write code sequences that use the mux inputs as requested below. Some code sequences may consist of a single instruction.



☑ Use C0 to carry `r1`, elsewhere `r1` not used.

```
# Cycle          0  1  2  3  4  5   SAMPLE SOLUTION
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☑ Use D3 to carry `r1`, elsewhere `r1` not used.

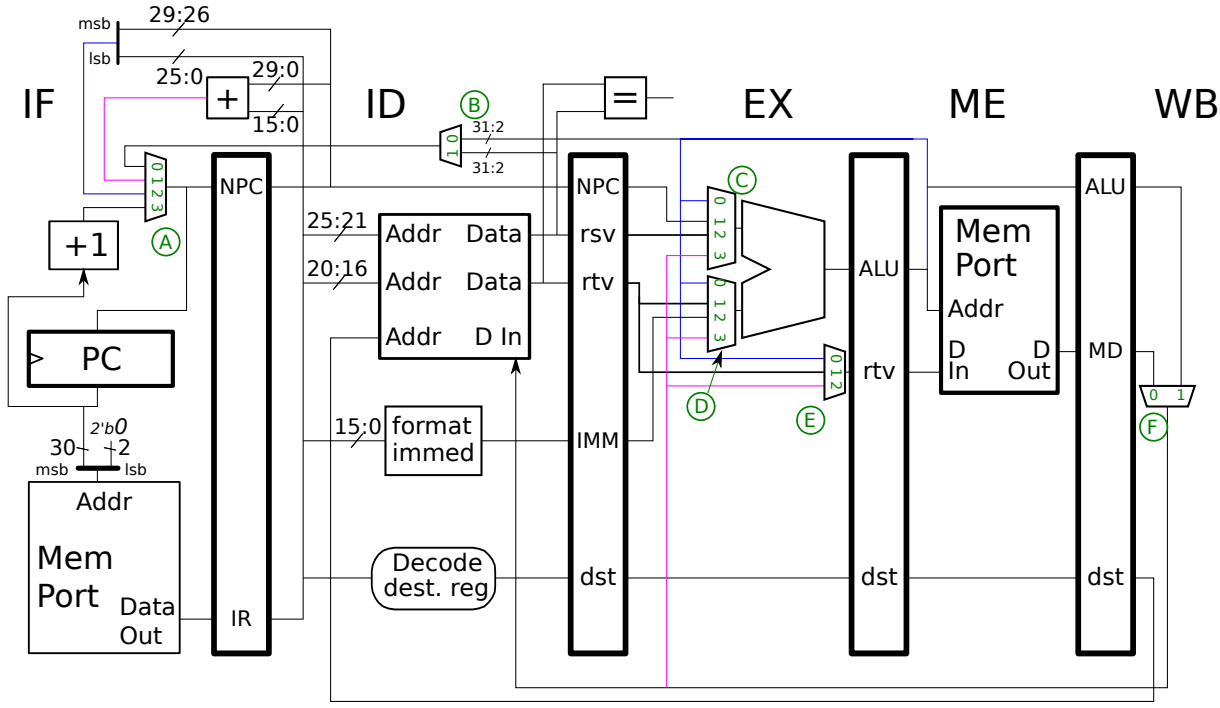
Solution appears below. Note that the D mux is used to bypass an `rt` value and so `r1` had to be the `rt` source of the `or` instruction. In other words `or r7, r1, r8` WOULD BE WRONG.

```
# Cycle          0  1  2  3  4  5  6    SOLUTION - D3 used in cycle 4.
add r1, r2, r3   IF ID EX ME WB
sub r4, r5, r6      IF ID EX ME WB
or  r7, r8, r1         IF ID EX ME WB
```

☑ Use E0 to carry `r1`, elsewhere `r1` not used.

Solution appears below. Note that the E mux is only used by store instructions, and that `rtv` is the store value, not the address.

```
# Cycle          0  1  2  3  4  5       SOLUTION - E0 used in cycle 3.
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r2)        IF ID EX ME WB
```



☑ Use A1.

Solution appears below. Only a branch uses the ID-stage adder and so the instruction must be some kind of a branch.

```
# Cycle            0  1  2  3  4       SOLUTION - A1 used in cycle 1.
beq r0, r0, TARG   IF ID EX ME WB
```

☑ Use B0.

Solution appears below. B1 carries the `rsv` to the PC, and so this could only be used by a `jr` or `jalr` instruction. B0 carries a value from the ALU to the PC, which could only reasonably be a bypassed value needed by a `jr` or `jalr` instruction. The existing connections to the ALU would allow it to compute the branch target, but there's already an ID-stage adder and so there is no need to do so.

```
# Cycle          0  1  2  3  4  5  6    SOLUTION - B0 used in cycle 3.
addi r1, r1, 4   IF ID EX ME WB
sub r2, r3, r4      IF ID EX ME WB
jr r1                  IF ID EX ME WB
```

☑ Use C1.

Solution appears below. The only instructions that would save the NPC would be a `jal` or a `jalr`. (There is already an ID-stage adder for the branch, so a branch would not use NPC.)

```
# Cycle          0  1  2  3  4       SOLUTION - C1 used in cycle 2.
jal SOMEWHERE    IF ID EX ME WB
```

Problem 3: [15 pts] The floating point code fragment below computes `f2 = f3 * f0 + f1`, where `f3` is a call argument and `f0` and `f1` are loaded from a table. A total of eight instructions are used to load the constants into `f0` and `f1`. Re-write the code so that fewer instructions are used. It is possible to load both registers using a total of two instructions.

☑ Load constants into `f0` and `f1` using two or three instructions.

☑ The constants must be loaded from the table.

```
        .data
famous_constants: # 0x10010300
        .float 2.718282818
        .float 3.141592654
        .text
pie:
        # Load f0 with first element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x300
        lw $t1, 0($t0)
        mtc1 $t1, $f0

        # Load f1 with second element of table.
        lui $t0, 0x1001
        ori $t0, $t0, 0x304
        lw $t1, 0($t0)
        mtc1 $t1, $f1

        # Don't modify the code below this line.
        mul.s $f2, $f3, $f0
        add.s $f2, $f2, $f1
```

Solution on next page.

Two solutions appear below, A and B. In both solutions `mtc1` instructions are avoided by loading directly into floating-point registers. The `ori` instructions are avoided by using the load offset to add on the lower 16 bits of the famous constants table address. In Solution A a `ldc1` instruction is used to load both of the floating-point registers with just one instruction. Solution B uses two instructions.

Grading Note: The idea of using a `ldc1` to load both registers was from a student's solution, my best solution was the three-instruction version.

```
        .data
famous_constants: # 0x10010300
        .float 2.718282818
        .float 3.141592654
        .text
pie:
        # SOLUTION A -- Two Instructions
        lui $t0, 0x1001
        ldc1 $f0, 0x300($t0)   # Note: loads both f0 and f1.


        # SOLUTION B -- Three Instructions
        lui $t0, 0x1001
        lwc1 $f0, 0x300($t0)
        lwc1 $f1, 0x304($t0)


        # Don't modify the code below this line.
        mul.s $f2, $f3, $f0
        add.s $f2, $f2, $f1
```

Problem 4: [40 pts] Answer each question below.

(*a*) In class we said that MIPS-I lacks an instruction like `bgt` (branch greater-than) because the magnitude comparison would take a little too long. MIPS-I does have `beq` and `bne` instructions that compare two registers. However, SPARC v8 has a `bgt` instruction, but it is done in such a way that here is no risk of critical path impact.

☑ How is the actual SPARC `bgt` different than a hypothetical MIPS `bgt`?

The SPARC `bgt` uses the condition code register to determine if the branch should be taken, whereas the MIPS `bgt` would compare the contents of two registers.

☑ How does that difference avoid critical path impact in resolve-in-ID implementations?

Short Answer: The branch condition is computed in SPARC using a 4-bit condition code register rather than retrieving and then examining $2 \times 32 = 64$ bits of the two registers, which would take longer.

Explanation: The integer condition code register, `icc`, is only four bits and so checking for a particular bit configuration won't take much time. For `bgt` the hardware would check that the `N` (negative) and `Z` (zero) bits are both zero (meaning that the last `cc` operation produced a value that was positive). Further, there is no need to retrieve `icc` from anywhere. In contrast, for an instruction like `bgt r1, r2, TARG` the contents of registers `r1` and `r2` would first have to be retrieved, which would take some time, and then a magnitude comparison would have to be made.

☑ Explain why a `bgt r1, r2, TARG` would not have a big critical path impact if it were resolved in EX.

Because the `r1` and `r2` values would be available at the beginning of the clock cycle, rather than the middle as they would be in `ID`.

(*b*) In the MIPS `add` instruction the `sa` field must have a value of zero. Consider a future version of MIPS in which the `sa` field would hold a scale factor, `s`. The result of the add would be `rsv + rtv * s`. Suppose that analysis of users' programs found that such an instruction would be **very useful** and that it could **easily be implemented in hardware**. Should the `add` be extended in that way? If not, suggest another way of providing the scaled add.

☑ Should `add` be extended to compute `rsv + rtv*s`?

No. No! Noooo!!!!!

☑ Explain a possible objection and suggest an alternative way of including the instruction.

Since the `sa` field must be zero in MIPS-I instructions `add` instructions written for MIPS-I would produce the wrong answer on implementations with the scaled add. An alternative would be to define the instruction result as `rsv + rtv * ( sa + 1 )`, so that when `sa` is zero the instruction behaves like the original `add`. The scaling feature is still available (and can reach 32 instead of 31). The assembly language format for the instruction could be defined something like `add rd, rs, rt, s` with `sa = s - 1` .

Grading Note: Many solutions argued that the scaled add would be cumbersome to implement. Ordinarily, that would be a good point, but the problem said to suppose that the hardware would be easy to implement. In the original exam boldface was not used for emphasis.

(*c*) The MIPS-I assembly instruction below is invalid. Explain why and replace it with one that correctly adds two double-precision values.

```
add.d $f0, $f1, $f2
```

Double precision instructions must use even numbered registers. A correct version appears below.

```
add.d $f0, $f10, $f2  # SOLUTION
```

(*d*) What is the difference between a dependency and a hazard?

☑ The difference between a dependency and a hazard is:

A dependency describes a relationship between instructions in a program, while a hazard describes a potential problem an implementation can have executing instructions.

(*e*) Identify the type of dependence between each pair of instructions below, and indicate the corresponding hazard.

☑ The type of dependence is: Output . The corresponding hazard is: WAW

```
add r1, r2, r3
sub r1, r5, r6
```

☑ The type of dependence is: True . The corresponding hazard is: RAW

```
add r1, r2, r3
sub r4, r1, r6
```

Note: The dependency above is known by three names: true, data, and flow. Full credit would be given for any of those names.

☑ The type of dependence is: Anti . The corresponding hazard is: WAR

```
add r1, r4, r3
sub r4, r2, r6
```

(*f*) In class we said that the lifetime of an ISA can be decades and so it must be carefully designed to take into account current and future implementation technologies. Is IA-32 (80x86) a good example of this rule? Explain.

☑ IA-32 ◯ *is* ⊗ *is not* a good example of this rule because ....

Short Answer: ... because despite being designed to fit on small chips of the time, it has been implemented in successively larger chips, and done so in a way that overcomes shortcomings (such as a limited number of registers and segmented addressing) to the point where IA-32 implementations were faster than implementations of much well-designed RISC ISAs. If the rule were always correct IA-32 implementations would be slower.

Details: The earliest ISA that could be called IA-32, implemented by the Intel 8086 and less costly 8088, was limited by the number of transistors that could fit on chips of the time. The ISA was never designed to last decades and to host a robust operating system. It was IBM's second choice for CPU in their initial entry into the personal computer market, the IBM PC. Their first choice was the Motorola 68000, but Motorola could not make enough of them for IBM's initial order and so the 8088 it was. The IBM PC was spectacularly successful, spawning a large market for IBM-compatible (and so 8088-compatible) software. Switching ISAs in the IBM PC, say to 68000 (which had problems of its own), would risk alienating customers, so IBM continued using the 8088 and its descendants. Intel deftly made improved implementations of IA-32 (or pre IA-32), overcoming its various weaknesses. Early RISC implementations would easily outperform IA-32 implementations, but some time in the aughts IA-32 implementations took the top spot, at least in the SPECcpu benchmarks. The ISA too was extended from the original 8088, to the first "real" IA-32, implemented in the 80386, up until the present. Currently IA-32 is sort of a subset of Intel 64. Adding features to an ISA does not change what's already there and that just made implementations that much harder. Many Intel 64 implementations work by *cracking* Intel 64 (or IA-32) instructions into *micro-ops*. Micro-ops are kind of like RISC instructions, and they are executed by something kind of like a RISC implementation. RISC implementations don't have to crack instructions, making implementations less expensive and easier to design.

IA-32 was not designed to last decades. According to the rule it should not have been a long-term success because future implementations would be hobbled by shortsighted ISA features. But that didn't happen and so IA-32 is not a good example of the rule. (Future implementations weren't hobbled because Intel could afford to put a large number of engineers on the design team to work around the ISA shortcomings.)

(*g*) Indicate the most appropriate ISA family for each ISA below.

☑ Intel 64 is considered ◯ *RISC* ⊗ *CISC* ◯ *VLIW*

☑ Itanium is considered ◯ *RISC* ◯ *CISC* ⊗ *VLIW*

☑ MIPS is considered ⊗ *RISC* ◯ *CISC* ◯ *VLIW*

☑ SPARC is considered ⊗ *RISC* ◯ *CISC* ◯ *VLIW*

☑ VAX is considered ◯ *RISC* ⊗ *CISC* ◯ *VLIW*

Name Solution

Computer Architecture

EE 4720

Final Examination

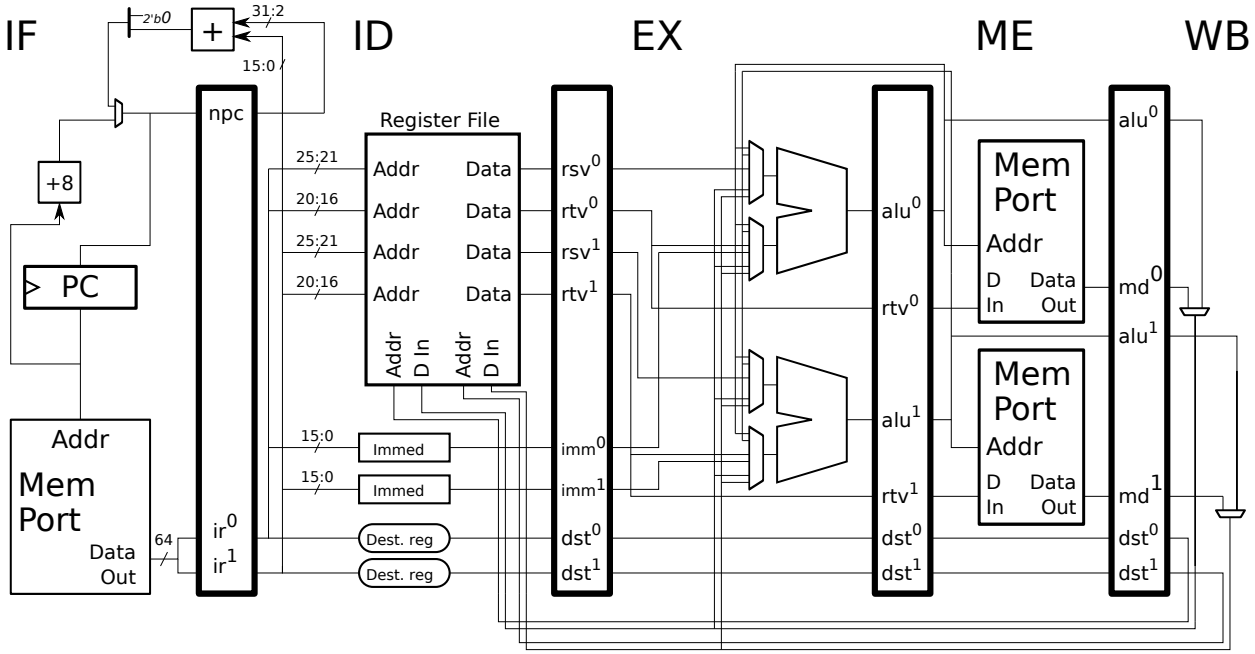
2 May 2018, 15:00–17:00 CDT

Problem 1		(15 pts)
Problem 2		(10 pts)
Problem 3		(20 pts)
Problem 4		(16 pts)
Problem 5		(10 pts)
Problem 6		(8 pts)
Problem 7		(21 pts)
Exam Total		(100 pts)

Alias [redacted]acted

Good Luck!

Problem 1: (15 pts) Appearing below is the 2-way superscalar implementation used in class. As we usually assume, fetch groups are aligned and stalls must keep instructions within a stage in order.



(*a*) Show the execution of the code below on the implementation above.

☑ Show execution. ☑ Check for dependencies!!

Solution appears below. The `lw r3` stalls two cycles because of the dependency with the `lw r1`. The `sw r5` stalls two cycles because of the dependency with the `lw r5` and because there is no way to bypass a value to the path to the memory port `D In` connection in the illustrated implementation. (In some of the implementations used in class there are bypass paths leading to `ME.rtv`.) A common mistake was to overlook the fact that there is no bypass path for the store value.

```
LINE1: # Address of the first lw insn below is 0x1000
#                       SOLUTION
# Cycle                 0  1  2  3  4  5  6  7  8  9  10
0x1000: lw r1, 0(r2)    IF ID EX ME WB
0x1004: lw r3, 0(r1)    IF ID ----> EX ME WB
0x1008: lw r4, 4(r1)       IF ----> ID EX ME WB
0x100c: lw r5, 8(r1)       IF ----> ID EX ME WB
0x1010: sw r5, 12(r1)                 IF ID ----> EX ME WB
0x1014: sw r4, 16(r1)                 IF ID ----> EX ME WB
# Cycle                 0  1  2  3  4  5  6  7  8  9  10
```

(*b*) Show the execution of the code fragment below on the illustrated implementation.

☑ Show execution and ☑ check for dependencies here too.

☑ Don't overlook the fact that the branch is taken.

☑ Pay attention to fetch groups and the aligned fetch restriction.

Solution appears below. The `lw` is fetched in the cycle after the `beq` because it is not in the same fetch group as the `beq`. (The first instruction in a fetch group on a 2-way superscalar has and address that's a multiple of 8, and so the least significant hex digit must be 0 or 8.) The `sw` is the second instruction in the fetch group and so gets fetched along with the `lw`. Since the branch is taken the `sw` is squashed. Fortunately, the branch target, `0x2008`, is a multiple of 8 and so we fetch two good instructions (the two `andi` instructions). But dependencies stall execution.

```
#        Address of beq is 0x1004                SOLUTION
#        Cycle               0  1  2  3  4  5  6  7  8  9
0x1004:  beq r1, r1, SKIP1   IF ID EX ME WB
0x1008:  lw r2, 0(r3)           IF ID EX ME WB
0x100c:  sw r4, 0(r3)           IFx
0x1010:  addi r3, r3, 4
#        Cycle               0  1  2  3  4  5  6  7  8  9
SKIP1: # Address of andi is 0x2008
0x2008:  andi r2, r2, 0xfff        IF ID -> EX ME WB
0x200c:  andi r6, r2, 0xff0        IF ID ----> EX ME WB
0x2010:  add r7, r2, r6               IF ----> ID EX ME WB
0x2014:  sub r8, r2, r6               IF ----> ID EX ME WB
#        Cycle               0  1  2  3  4  5  6  7  8  9
```

(*c*) Appearing below is again our 2-way superscalar MIPS. Notice that the branch hardware shown can only provide the target for a branch in slot 1. Add hardware for providing the branch target of a branch in slot 0. **Do not** add hardware for checking the branch condition. **Do not** add control logic.

☑ Add hardware for a slot-0 branch.

☑ **Pay attention to cost**.

☑ Be sure the hardware computes the correct target address.

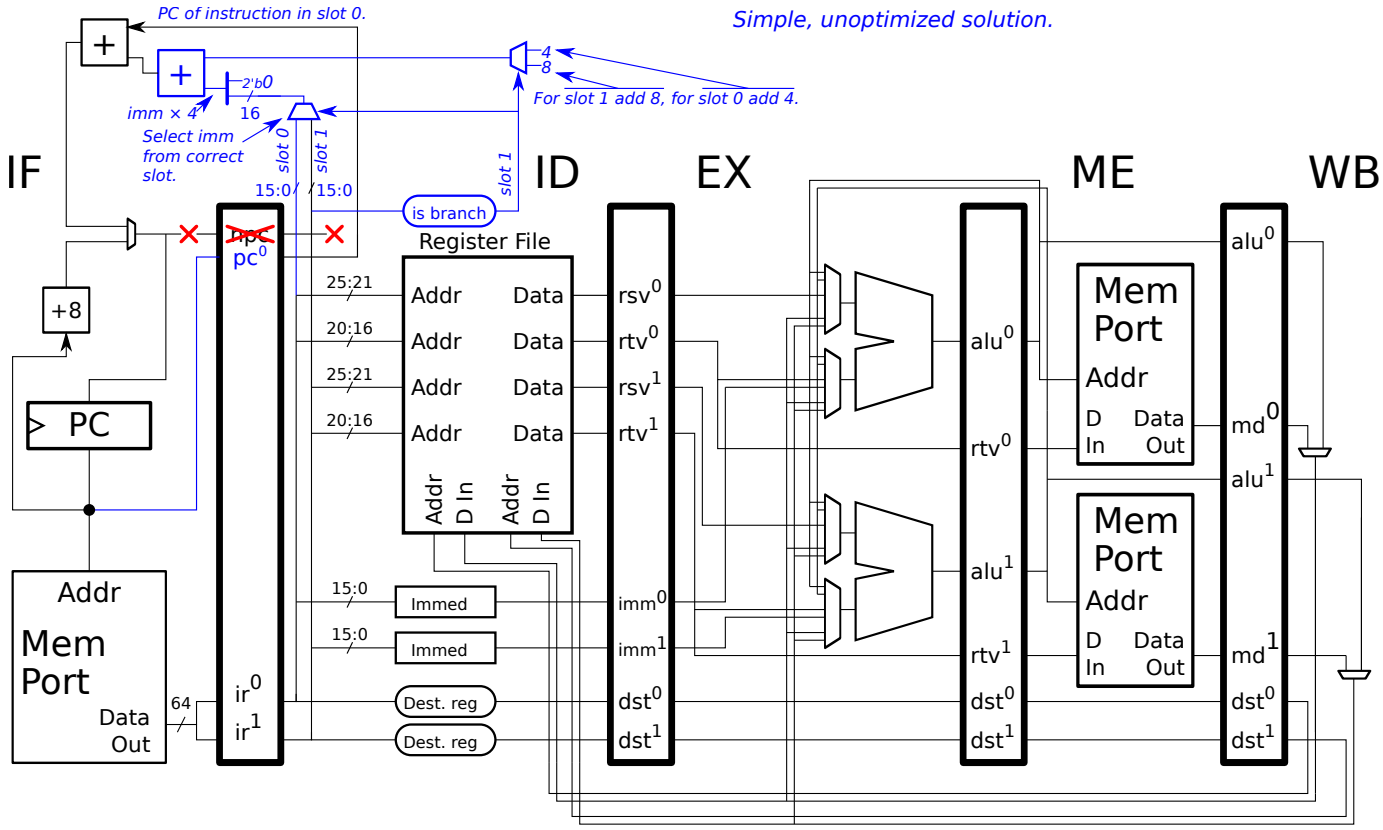
Two solutions appear below (on the following pages). The first one is correct, and would receive full credit, but it's more expensive than it needs to be because of the additional adder. The second one does not use an additional adder.

In both solutions the PC, rather than the NPC (next program counter) is passed to ID. The PC is the address of the slot-0 instruction so it is labeled pc0 in the diagram. If the branch is in slot 0 then the branch target is computed in the usual way, `pc0 + 4 + imm*4`. If the branch is in slot 1 then 8 rather than 4 is added: `TARG = pc0 + 8 + imm*4`. The two code executions below show the target address computed this way. In the first example the branch is in slot 0, in the second it is in slot 1.

```
 # Example: Branch in slot 0. Target is TARG = slot_0_pc + 4 + imm0 * 4 = 0x1814
 # Cycle                   0  1  2  3  4  5  6
 0x1000: beq r1, r2, TARG  IF ID EX ME WB       # slot 0 pc = 0x1000
 0x1004: add r4, r5, r6    IF ID EX ME WB       # imm = ( 0x1814 - 0x1004 ) / 4
 0x1008: sub r7, r8, r9       IFx               #     = 0x810 / 4 = 0x204
 0x100c: or  r10, r11, r12    IFx               # TARG = 0x1000 + imm*4 + 4
# Cycle                    0  1  2  3  4  5  6  #      = 0x1000 + 0x204*4 + 4
TARG:                                           #      = 0x1000 + 0x810 + 4
 0x1814: lw r10, 0(r11)          IF ID EX ME WB #      = 0x1814

 # Example: Branch in slot 1. Target is TARG = slot_0_pc + 8 + imm1 * 4 = 0x1814
 # Cycle                   0  1  2  3  4  5  6
 0x1000: xori r14, r14, 5  IF ID EX ME WB       # slot 0 pc = 0x1000
 0x1004: beq r1, r2, TARG  IF ID EX ME WB       # imm = ( 0x1814 - 0x1008 ) / 4
 0x1008: add r4, r5, r6       IF ID EX ME WB    #     = 0x80c / 4 = 0x203
 0x1008: sub r7, r8, r9       IFx               # TARG = 0x1000 + imm*4 + 8
 # Cycle                   0  1  2  3  4  5  6  #      = 0x1000 + 0x203*4 + 8
TARG:                                           #      = 0x1000 + 0x80c + 8
 0x1814: lw r10, 0(r11)          IF ID EX ME WB #      = 0x1814
```

Solution continued on next page.

A simple version of the solution appears below. The is branch hardware checks whether there is a branch in slot 1. If there is not a branch in slot 1 the hardware will assume that there is a branch in slot 0. That's okay, because if neither slot has a branch then the branch target is ignored. The is branch signal selects the appropriate immediate and also the constant to add to the immediate, 4 or 8. The selected immediate is multiplied by four by prepending two zeros to the least-significant side.

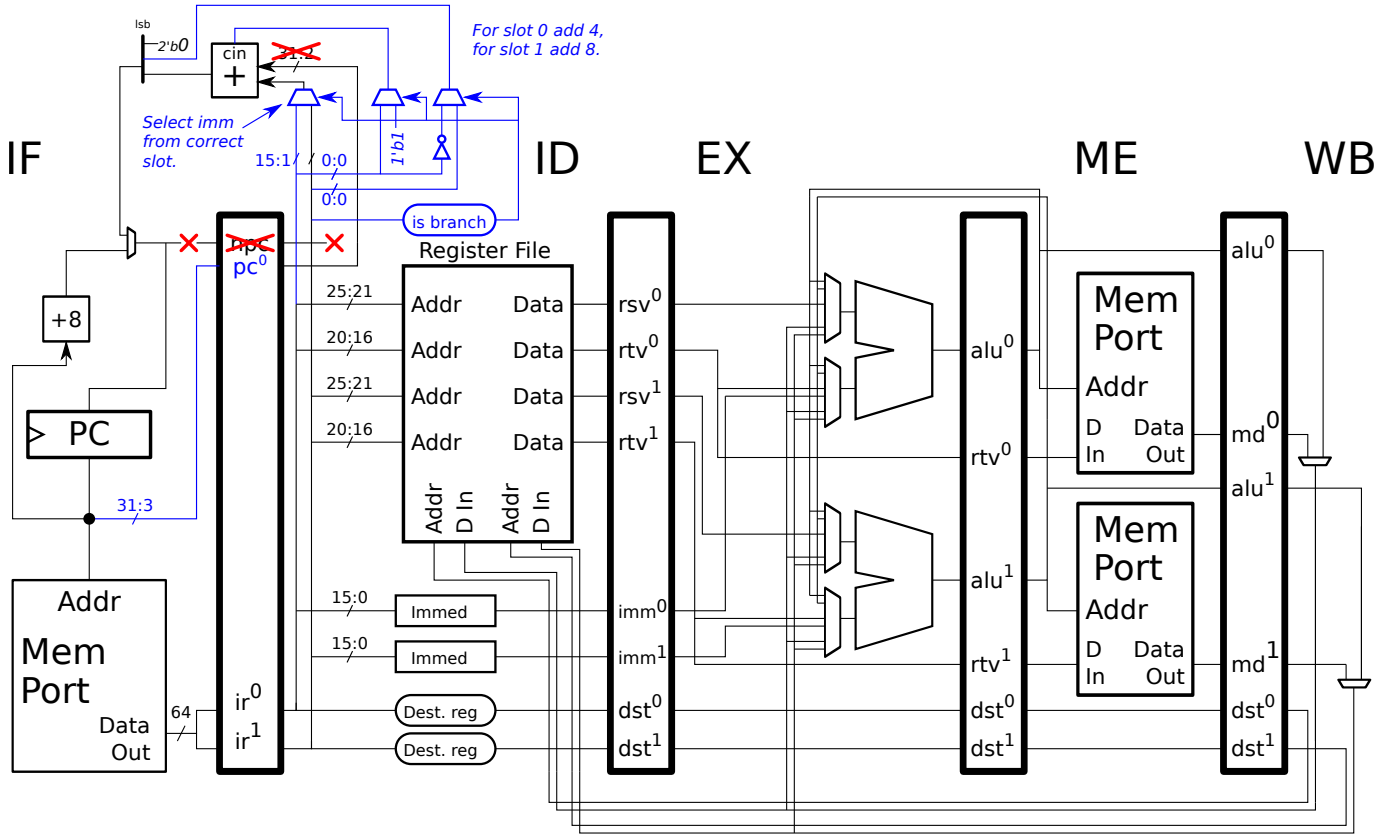


The cost of the solution above can be reduced by eliminating the adder above the `imm` $\times 4$ label. Notice that the upper input to this adder is either a 4 or an 8. Also notice that `pc0` must be a multiple of 8. Based on these we can eliminate the adder, and instead use the carry-in input to the remaining adder to perform the $+4$ or $+8$.

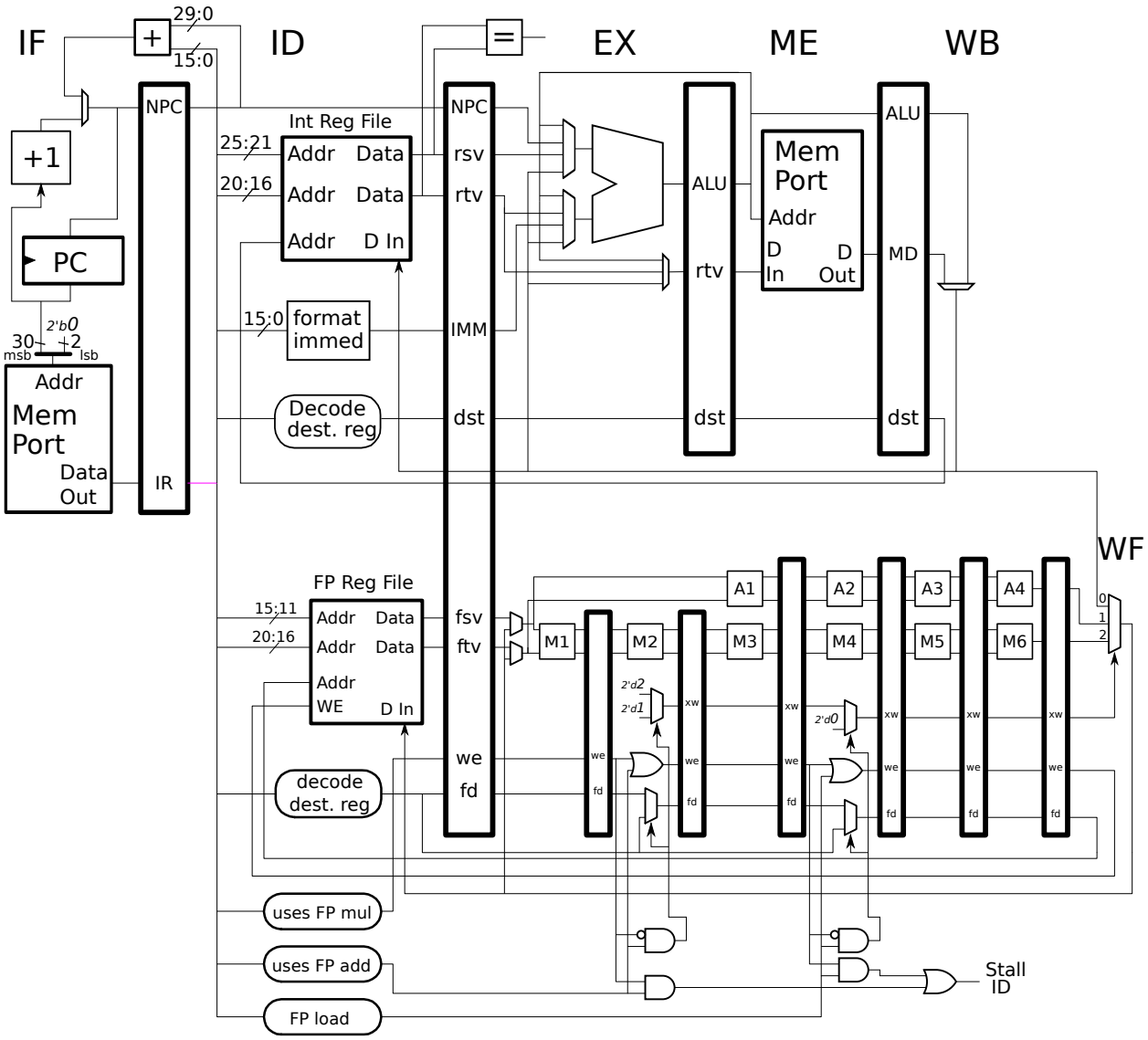
Solution continued on next page.

The more efficient solution appears below, with changes in blue. The solution below is more efficient because it uses one less adder than the one above. The adder is eliminated by using separate hardware to compute bit 2 of the target. (Bits 0 and 1 of the target are always zero.) Bit 2 is not computed using `pc0` because `pc0` is a multiple of 8.

Common mistake: not accounting for the fact that `ID.NPC` is for the instruction in slot 1.



Problem 2: (10 pts) Appearing below is the execution of a bit more than two iterations of a loop on the illustrated MIPS implementation. The execution shows the use of a two-stage FP compare unit, `C1-C2`, by the `c.lt.s` instruction, but the unit isn't shown.




```
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2  IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3        IF ID -------------> C1 C2 WF
bc1f LOOP               IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                          IF ----> ID A1 A2 A3 A4 WF
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2                             IF ID -------> M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3                                   IF -------> ID -------------> C1 C2 WF
bc1f LOOP                                                   IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                                                              IF ----> ID A1 A2 A3 A4 WF
LOOP:  #          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
mul.s f1, f1, f2                                                                    IF ID -------> M1 M2 M3 M4 M5 M6 WF

# SOLUTION:                                  ! <--- 2nd Iter ------------------------> ! <-- 3rd Iter ..
```

(*a*) Compute the CPI of the execution of the loop above for a large number of iterations.

☑ Compute the CPI.

☑ Clearly show how the time for an iteration was determined, perhaps using the pipeline diagram.

The CPI is $\frac{25-11}{4} = \frac{14}{4} = 3.5$.

The time for the second iteration is shown on the diagram, as well as the start of the third. The second iteration takes $25-11 = 14$ cycles. We expect the third and subsequent iterations to also take 14 cycles each because the state of the pipeline at the start of the 2nd and 3rd iterations is identical: `mul.s` in `IF`, `add.s` in `ID`, and `bc1f` in `EX`.

(*b*) Reschedule the instructions to reduce the time needed to execute a large number of iterations of the loop. Add a `nop` if that helps. A correct solution will still have many stalls.

☑ Re-schedule to improve performance.

☑ Don't change what the loop is computing.

```
# SOLUTION

LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
mul.s f1, f1, f2  IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3        IF ID -------------> C1 C2 WF
add.s f1, f1, f4        IF -------------> ID A1 A2 A3 A4 WF
bc1f LOOP                                 IF ID -> EX ME WB
nop                                          IF -> ID EX ME WB
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
mul.s f1, f1, f2                                    IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3                                          IF ID -------------> C1 C2 WF
add.s f1, f1, f4                                          IF -------------> ID A1 A2 A3 A4 WF
bc1f LOOP                                                                   IF ID -> EX ME WB
nop                                                                            IF -> ID EX ME WB
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
mul.s f1, f1, f2                                                                        IF ID M1 M2 M3 M4 M5 M6 WF

# ALT SOLUTION (Student)

LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
c.lt.s f1, f3     IF ID C1 C2 WF
add.s f1, f1, f4     IF ID A1 A2 A3 A4 WF
bc1f LOOP               IF ID -> EX ME WB
mul.s f1, f1, f2           IF -> ID -> M1 M2 M3 M4 M5 M6 WF
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
c.lt.s f1, f3                  IF -> ID -------------> C1 C2 WF
add.s f1, f1, f4                     IF -------------> ID A1 A2 A3 A4 WF
bc1f LOOP                                              IF ID -> EX ME WB
mul.s f1, f1, f2                                          IF -> ID -> M1 M2 M3 M4 M5 M6 WF
LOOP:             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
c.lt.s f1, f3                                                    IF -> ID -------------> C1 C2 WF
```

Problem 3: (20 pts) The MIPS implementation on the next page shows the two stages of the comparison units, C1 and C2, but they are not connected to anything. The illustration also shows an FCC register that will hold the floating-point condition code value computed by compare instructions such as `c.lt.s`. Connect the comparison units and the FCC register so that they operate correctly and as described by the check items below. Notice that logic to detect FP branch instructions and FP compare instructions has been added to the ID stage near the bottom.

```
LOOP:     # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
mul.s f1, f1, f2   IF ID M1 M2 M3 M4 M5 M6 WF
c.lt.s f1, f3         IF ID -------------> C1 C2 WF
bc1f LOOP                IF -------------> ID ----> EX ME WB
add.s f1, f1, f4                           IF ----> ID A1 A2 A3 A4 WF
          # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

☑ Provide connections to C1, C2, and the two FCC inputs so that the code above executes as shown.

☑ Modify the control logic so that a compare does not arrive in WF in the same cycle as other FP instructions. (This is despite the fact that compares do not write the FP register file.)

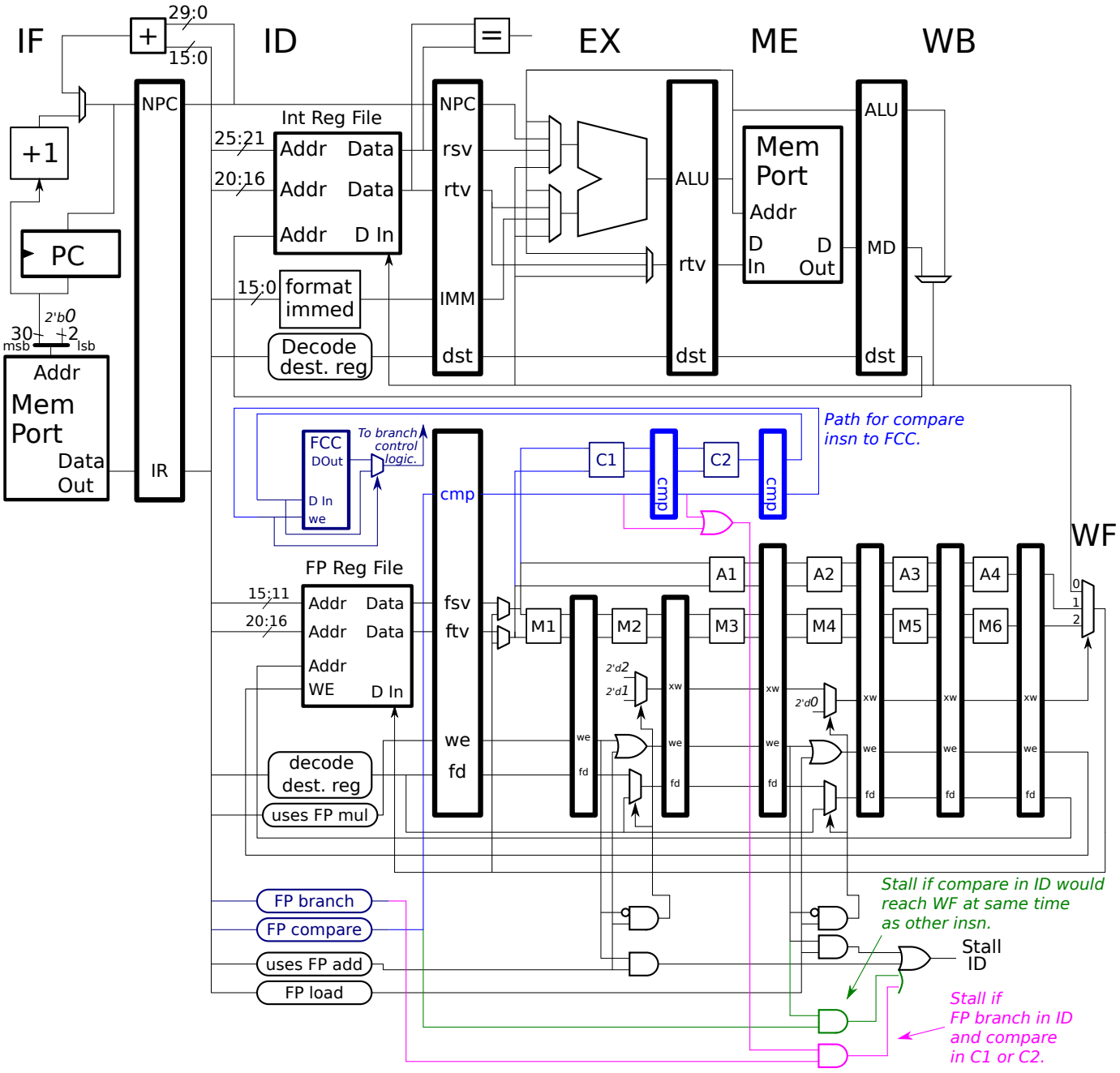
☑ Modify the control logic so that the `Stall ID` signal is asserted for dependencies from compare to branches, such as occurs above with the `bc1f`.

☑ As always, pay attention to cost and performance and ☑ don't break existing functionality.

Solution appears below. The connections from the C1 and C2 units to the FCC are shown in blue. Of course, pipeline latches are added so that each unit has the better part of a cycle to compute its result. The logic preventing a compare from arriving in WF at the same cycle as another instruction appears in green. Note that unlike other instructions that enter the pipeline at A2/M4, compare instructions do not inject `we` nor `fd` signals into the pipeline because compare instructions don't write the FP register file. Finally, logic to stall a branch dependent on a compare appears in purple.

Common Mistakes: Using the `we` (write enable) signal for the FP register file as the `we` for FCC. That won't work because we don't want instructions like `add.s`, which write the FP register file, to change the FCC and we don't want compare instructions to change the FP register file.

Another common mistake is putting logic in the wrong stage. Control logic that checks for things like stall conditions must connect to those stages holding the instructions that are involved. Use a pipeline execution diagram to determine those stages. For example, consider the `c.lt.s/bc1f` stall from the example above. The stall occurs when the `bc1f` is in ID and the `c.lt.s` is in stages C1 or C2. A common mistake was forgetting about C2.


IF
ID
EX
ME
WB
WF
29:0
15:0
+
=
+1
PC
NPC
IR
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
Int Reg File
25:21
20:16
Addr Data
Addr Data
Addr D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst
FCC
DOut
D In
we
To branch
control
logic.
Path for compare
insn to FCC.
cmp
C1
C2
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr
WE D In
fsv
ftv
we
fd
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
decode
dest. reg
uses FP mul
FP branch
FP compare
uses FP add
FP load
Stall if compare in ID would
reach WF at same time
as other insn.
Stall
ID
Stall if
FP branch in ID
and compare
in C1 or C2.

Problem 4: (16 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on two systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor and one system has a local predictor with an 8-outcome local history.

Branch `B2` starts with a random outcome, then repeats that same outcome two more times, followed by another random outcome followed by two more repeats of that. The random outcome is `T` with probability .3 and is independent of other outcomes. The following are possible `B2` outcome sequences: `TTT NNN NNN TTT TTT`. Note that the number of consecutive `T`'s or `N`'s must be a multiple of 3 and so the following **is not** a possible sequence of outcomes for `B2`: `TT NN T NNNN T`.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   T  T  N  T  T  T  N  N    T  T  N  T  T  T  N  N  ...

B2:    r  r  r  q  q  q  s  s  s ...
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

Based on the work shown below, the accuracy is $\frac{4}{8}$. The accuracy is based on the second repeat of the pattern because the 2-bit counter was the same value at the beginning and end, 1. In contrast, the counter was 0 at the start of the first occurrence of the pattern and 1 at the end, so whatever happened for those first 8 outcomes can't be used to predict accuracy.

```
     0 1  2  1  2  3  3  2  1    2  3  2  3  3  3  2  1  <-- 2-bit counter
B1:   T  T  N  T  T  T  N  N    T  T  N  T  T  T  N  N  ...
      x  x  x  x        x  x    x     x           x  x   <-- Pred Outcome
                               ------------------------  <-- repeating
```

☑ What is the accuracy of the bimodal predictor on branch `B2`?

There are two ways to start a 3-outcome sequence, `T` or `N` of course. To determine the number of correct predictions we need to know what preceded the sequence, which could be 3 `T`'s or 3 `N`'s. The four cases are shown in the table below. The `Freq` column value indicates how frequently the case occurs. The `Correct` column value indicates the number of correct predictions *in the last 3 outcomes*. Note that we can't determine the number of predictions in the first three outcomes because we don't know what preceded them. `Acc` is the accuracy, and `Weighted` is the accuracy weighted by frequency. Totaling the last column gives the prediction accuracy, .72 or 72%.

```
          Freq  Correct  Acc  Weighted
 TTT.TTT: .09    3       1    .09
 NNN.TTT: .21    1       1/3  .07
 NNN.NNN: .49    3       1    .49
 TTT.NNN: .21    1       1/3  .07
 --------------------------------
                             .72
```

☑ What is the accuracy of the local predictor on branch `B1`?

Since the pattern for branch `B1` is less than the local history and it repeats perfectly and because we are assuming no collisions in the predictor, the local predictor accuracy is 100%.

☑ What is the accuracy of the local predictor on branch B2?

Because there are always length-three sequences and because we are assuming no interference, the local predictor will always predict the second and third occurrence with 100% accuracy. We expect that the 2-bit counter for the first occurrence will predict N since that is the more common outcome. Assuming that it always predicts N the prediction accuracy for the first occurrence is 70%. Combining these yields $\frac{.7+2}{3} = 90\%$.

```
     Freq   Corr  Wht
T.T: .09    2     .18
N.T: .21    2     .42
N.N: .49    3    1.47
T.N: .21    3     .63
--------------------
                2.70 / 3
```

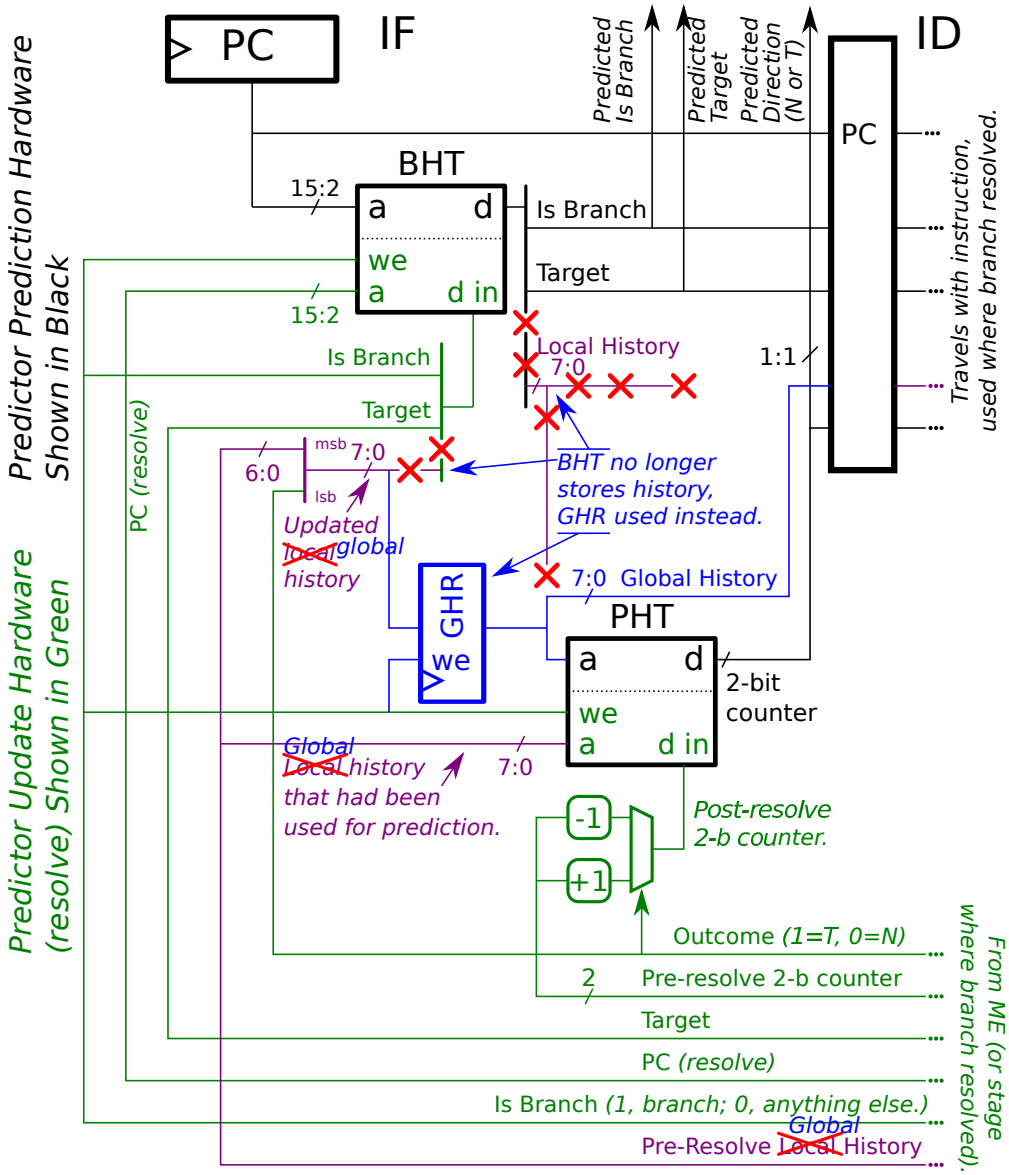
(*b*) Appearing below is a diagram of a local predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a global predictor with an 8-outcome global history.

☑ Modify so that it is a global predictor.

☑ Remove hardware that's no longer needed.

☑ Be sure to show the GHR (global history register).

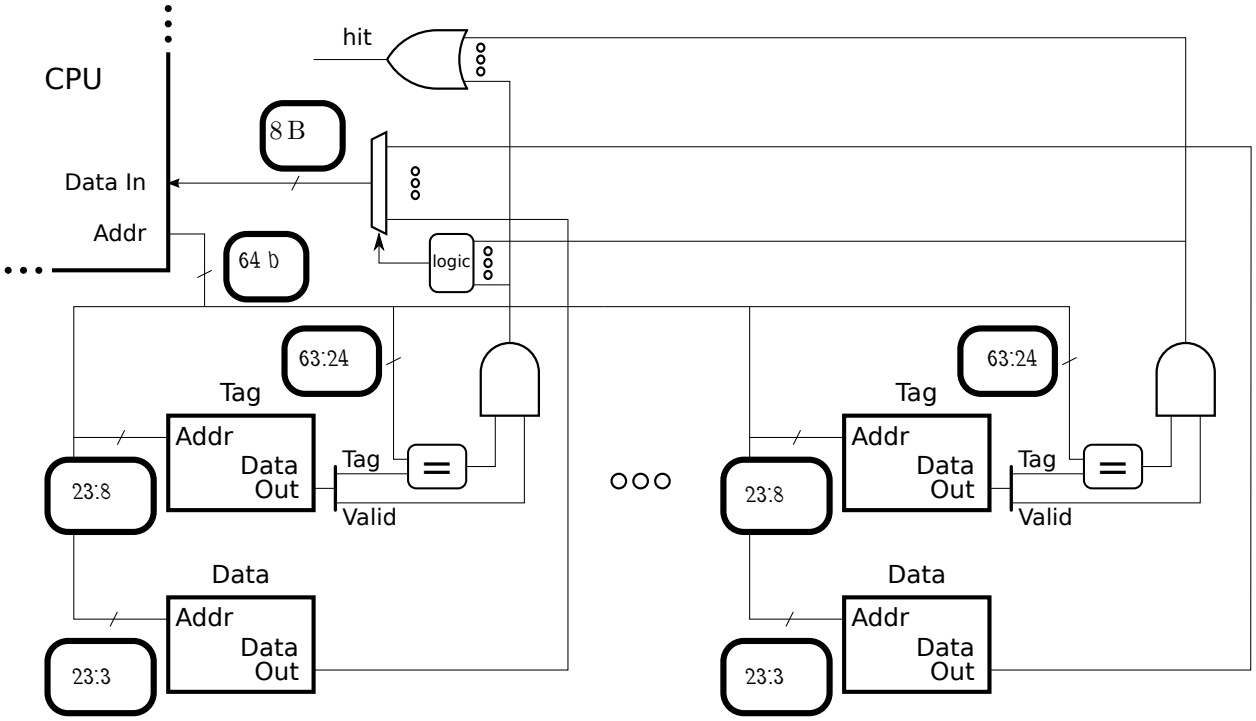
Solution appears below, added hardware appears in blue and red exes mark removed hardware, cut wires, etc. The BHT no longer stores local branch history (nor any other kind of history), instead the GHR is used to store global branch history. In this solution the GHR is not updated until the branch resolves. (In dynamically scheduled systems the GHR might be updated in `IF` using the predicted direction, when the branch resolves the GHR is updated again only when the prediction turns out to be wrong. This is important in dynamically scheduled systems because there can be many branches in flight.)



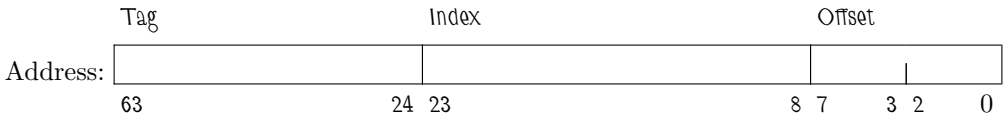
Problem 5: (10 pts) The diagram below is for a 64 MiB, 4-way set-associative cache with a line size of 256 B, a bus width (w) of 8 B, for a 64 b address space. Helpful facts: $64\,\text{MiB} = 64 \times 2^{20}\,\text{B} = 2^{26}\,\text{B}$ and $256 = 2^8$.

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

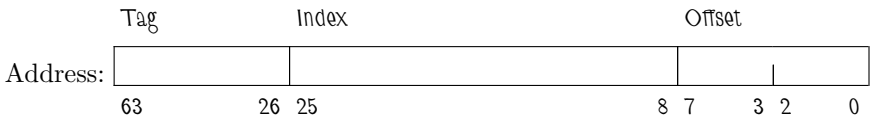


☑ Memory Needed to Implement ☑ Indicate Unit!!:

It's the cache capacity, 64 MiB, plus $4 \times 2^{24-8}\,(64 - 24 + 1)$ b $= 10747904$ b.

☑ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

The cache above is 64 MiB and 4-way set associative. In a direct mapped cache there is just one way with four times the storage of a way in the cache above. To get four times the number of entries the number of index bits is increased by two, and so the index bits will start at position 26 instead of 24. The other bit positions remain the same.



The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 256 B (which is 2^8 B). Each code fragment starts with the cache empty; consider only accesses to the arrays.

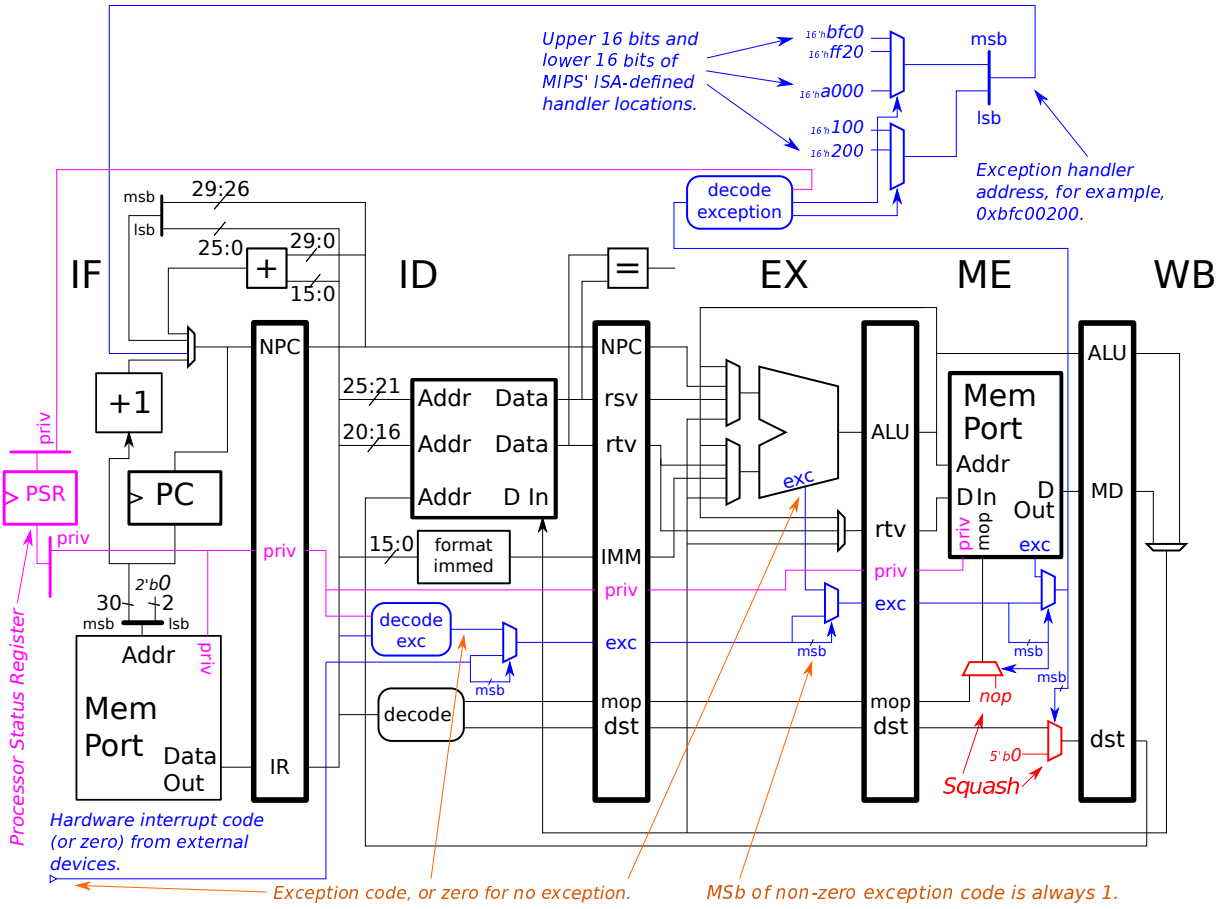
(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
short *a = 0x2000000; // sizeof(short) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^8 = 256$ bytes is given. The size of an array element, which is of type short, is $2 = 2^1$ B, and so there are $2^8/2^1 = 2^{8-1} = 2^7 = 128$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^7 elements, and so the next $2^7 - 1 = 127$ accesses will be to data on the line, hits. The access at `i=128` will miss and the process will repeat. Therefore the hit ratio is $\frac{127}{128}$.

Problem 6: (8 pts) Appearing below is a MIPS implementation that includes hardware for interrupts (hardware interrupts, exceptions, and traps). An exception code, `exc`, is collected and passed down the pipeline. Its value indicates the type of hardware interrupt, exception, or trap that has been encountered, a value of 0 indicates no interrupt of any kind.



(*a*) Notice that the decode exc logic in the ID stage examines the opcode of the instruction in ID as well as the value of ID.priv. *Hint:* priv *is an abbreviation for privileged.* Some opcode values raise exceptions only when ID.priv is zero, others raise exceptions whether or not ID.priv is zero.

☑ Describe an instruction that raises an exception in ID only if ID.priv is zero.

Any privileged instruction, for example **mtc0**, move to co-processor zero.

☑ Why is it important that such an instruction raise an exception?

It is important that such instructions raise exceptions so that restrictions can be placed on user-mode code (for which ID.priv is zero), for example, preventing user-mode code from unfettered access to I/O devices. Privileged instructions can control access to such devices.

Common Mistake: An incorrect answer would be *a load instruction that accesses a privileged address*. Yes, such an instruction would raise an exception, but it would do so in the ME stage. The ID stage is too early because the address would not have been computed, and the hardware for looking up permissions is in the memory port.

☑ Describe an instruction that raises an exception in ID whether or not ID.priv is zero.

An instruction with an unrecognized opcode. This might be a nonexistent opcode. In classroom examples we use the mnemonic **ant** for such instructions. An unrecognized opcode might be an opcode defined by the ISA but not implemented in the hardware. Many ISAs allow implementations to not implement certain instructions with the expectation that the OS would emulate these in software when they raise exceptions. For example, SPARC has quad-precision floating point instructions which are rarely implemented. When an instruction like **faddq f0, f4, f8** tries to execute it raises an exception and the exception handler would compute the quad-precision result, place it in registers **f0–f3**, and then return execution to the instruction after **faddq**.

Common Mistake: An incorrect answer would be *a divide instruction with a zero divisor*. Yes, such an instruction raises an exception regardless of privilege level, but it does so in the FP pipeline, not in ID.

(*b*) The illustrated hardware squashes the faulting instruction in ME, but no hardware is shown to squash any instructions that may be in the stages before ME nor for the stage after ME. That hardware may have been omitted for simplicity (the same reason that control logic is omitted) or because it is not needed.

☑ To implement precise exceptions should the instructions in the stages before ME be squashed? ☑ Explain in terms of the handler and what would go wrong if instructions were not treated the right way.

Yes, they should be squashed because they come after the faulting instruction and for the exception to be precise no instruction after the faulting instruction should complete execution (write a register value, change a memory location, etc). Suppose the instruction in EX writes **r2** and was allowed to complete, and suppose the faulting instruction uses **r2** as a source. Then the handler would not be able to re-try the faulting instruction because **r2** has the wrong value.

A common mistake was to describe the stages before ME as carrying instructions before (should be after) the instruction in ME and WB as carrying the instruction after (should be before) the one in ME. For those prone to confusion in rushed situations, just remember that those before you on line at the sandwich shop arrived after you did.

☑ To implement precise exceptions should the instructions in the stage after ME be squashed? ☑ Explain in terms of the handler and what would go wrong if instructions were not treated the right way.

No. For the exception to be precise all instructions before the faulting instruction must execute normally. Suppose the instruction in WB is to write **r3** but was squashed before it could do so. Further, suppose the faulting instruction uses **r3** as a source. If the handler tries to re-execute the faulting instruction it won't execute correctly because **r3** was not written.

Problem 7: (21 pts) Answer each question below.

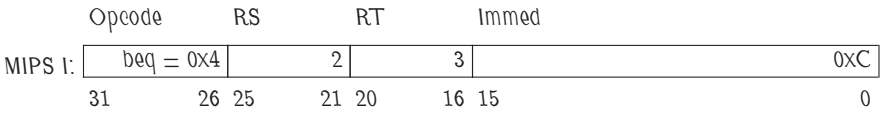
(*a*) Show the encoding of the following MIPS instructions. Write the instruction name in opcode or func field values that cannot be determined.

```
0x1000:  beq r10, r11,  TARG
0x1004:  add r7, r8, r9

TARG:
0x1034:  lw r12, 14(r15)
```

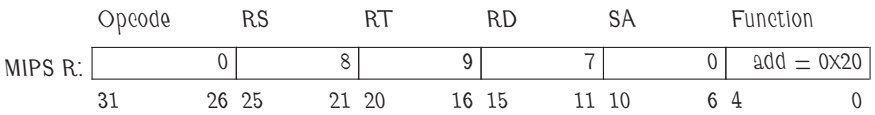
☑ Encoding for `beq` from code fragment above. ☑ Pay attention to the branch target.

The solution appears below. The immediate field is encoded with the displacement from the delay slot to the target in units of instructions. That is, the immediate value is: $\frac{1034_{16}-1004_{16}}{4} = \frac{30_{16}}{4} = C_{16}$.

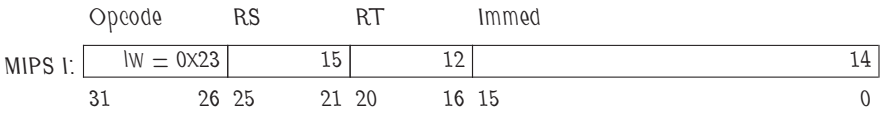
A common mistake was to forget to divide by four.



☑ Encoding for `add` from code fragment above:



☑ Encoding for `lw` from code fragment above:



(*b*) MIPS has one kind of memory addressing for all load and store instructions, such as in `lw r1, 2(r3)` where the immediate, 2, is added to the value in `r3`. A CISC ISA might have two versions of the load, `lw r1, (r3)`, which lacks an immediate (the immediate would be zero in MIPS), and `lwi r1, 2(r3)` for when an immediate is needed.

☑ What would be the benefit for the CISC ISA of having the no-immediate version of the `lw`?

The instruction would be shorter, saving instruction cache space and fetch bandwidth. If the CISC implementation were not pipelined it might be able to execute the no-immediate instruction in one less cycle than the immediate version because it would not need a cycle to add the immediate to the base. Note that modern CISC implementations operate by translating CISC instructions into RISC instructions, so the execution time benefit would not be realized.

☑ Why would MIPS and other RISC ISAs not realize the same benefit?

Because all instructions are the same size and because all integer instructions, including loads, go through the same pipeline stages.

(*c*) A design team is considering removing a bypass connection to the ALU and adding a bypass connection to the branch resolve unit. This won't change the cost, they hope it will improve performance. "Simulation of this design change shows that performance drops by 5%," a sad-faced engineer announces. "We forgot to talk to the compiler people!," another excitedly points out, splashing hope and excitement around the room.

☑ What should they ask the compiler people?

They should ask the compiler people to prepare a version of the compiler that will optimize for the new bypass locations. In particular, instructions should be scheduled to avoid needing the removed bypass path and instructions writing a register needed by a branch can be moved closer to the branch than they would in the current implementation.

Grading Note: Many students did not see the compiler as something that could and should be modified to fit the implementation.

(*d*) Someone preparing the SPECcpu benchmarks for their company's next product (currently under development) decides to replace one of the benchmark programs with an improved version, one which better reflects that company's customers. *Note: the emphasis below was added after the original exam, as was the phrase "not to market."*

☑ Should the company use the SPECcpu benchmarks in this way **to develop (but not to market)** their product?

Yes. If they are using it internally SPEC rules don't apply. As the problem states, the substituted benchmark improves the relevance of the results.

With the substituted benchmark program the SPEC scores are higher (better). The company decides to release these higher SPEC results without mentioning the substitution.

☑ How does the design of SPECcpu make it likely that they will get caught?

Short Answer: They must disclose a config file, which competitors will surely use to reproduce their result, and they won't stay quiet when the results don't match.

They must provide and make public a config file that can be used to automatically run the benchmarks. Others attempting to run SPECcpu with their config file (and using their computer) will quickly discover that the results don't match because those others will be running the original SPEC benchmarks, not the substituted one.

(*e*) Compiler optimization is more important for a supercalar implementation than a scalar implementation.

☑ Optimization is more important for superscalar than scalar because:

☑ The important optimization is:

It's more important because there are more situations where results can't be bypassed. For example, dependent instructions in the same fetch group. A stall also has a larger relative impact on performance. The important optimization is instruction scheduling, that's needed to avoid stalls.

Grading Note: Many students used the term *dependency* where *hazard* should be used. This is the correct usage: because there are multiple instructions in a stage there are more data hazards.

49 Spring 2017 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

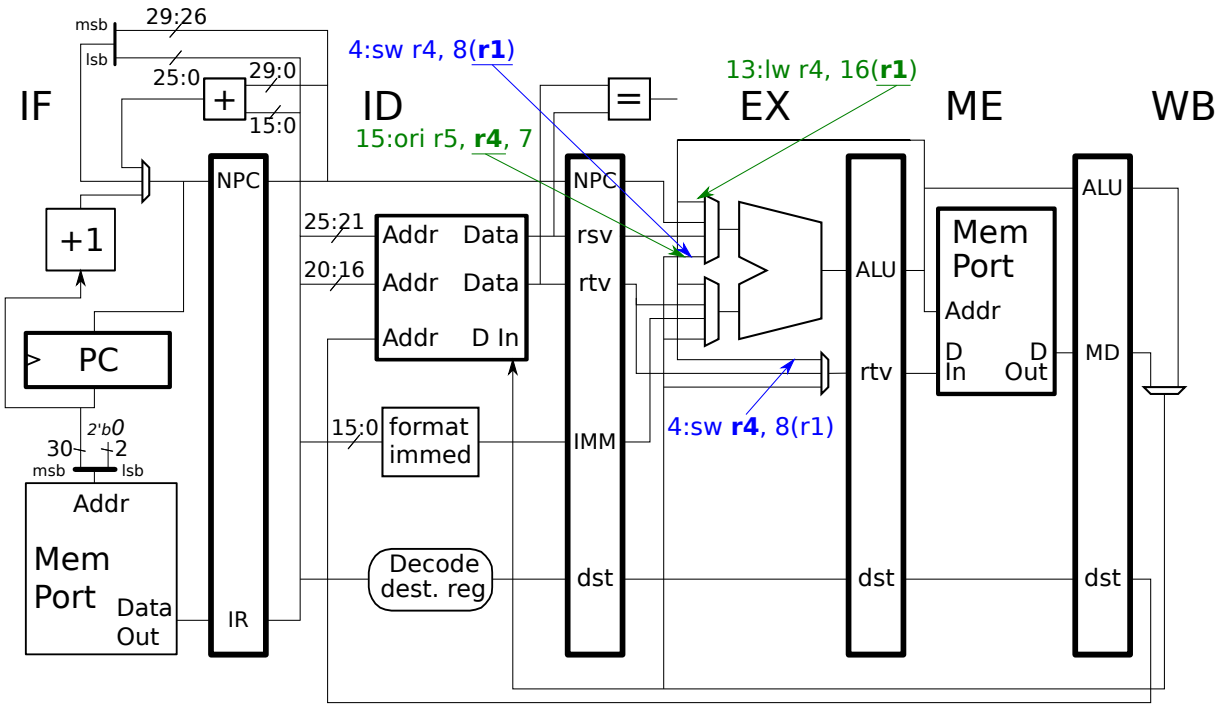
Wednesday, 22 March 2017, 9:30–10:20 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias MIPS-a-brazo

Good Luck!

Problem 1: [20 pts] Appearing below is our familiar MIPS implementation.



(*a*) Show pipeline execution diagrams for the code fragments below executing on the illustrated implementation and label as indicated.

☑ Show pipeline diagram. ☑ Doublecheck dependencies.

☑ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

The pipeline diagram appears below and the labeled diagram appears above with the solution to this part in blue. The labels shown above include the complete instruction, not just the instruction name which would have been sufficient for full credit.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EX ME WB
sub r4, r5, r6      IF ID EX ME WB
sw r4, 8(r1)           IF ID EX ME WB
```

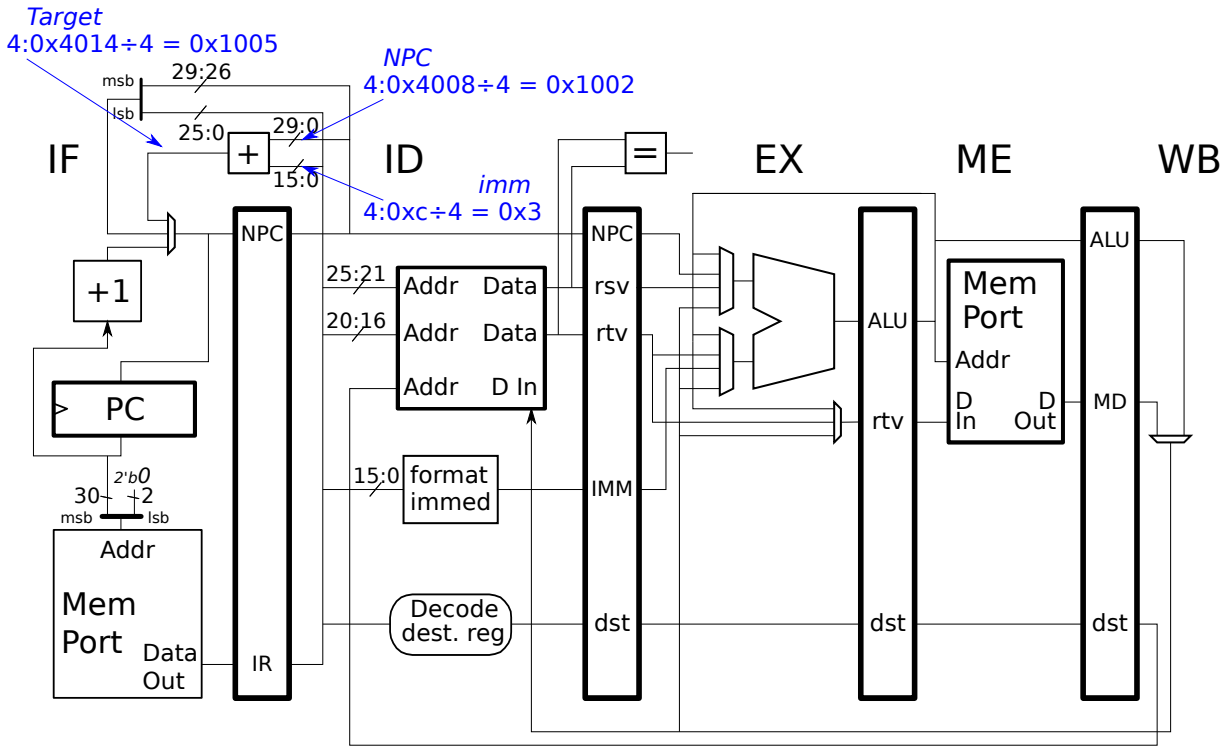
☑ Show pipeline diagram. ☑ Doublecheck dependencies.

☑ Label bypass paths used **at mux inputs** with $C : I$, where C is the cycle number (such as 2) and I is the instruction **consuming** the bypassed value. Be sure to check for dependencies.

The pipeline diagram appears below and the labeled diagram appears above with the solution to this part in green. The cycle numbers start at 10 to avoid confusion with part a.

```
# Cycle          10 11 12 13 14 15 16 17
and r1, r2, r3   IF ID EX ME WB
lw r4, 16(r1)       IF ID EX ME WB
ori r5, r4, 7          IF ID -> EX ME WB
```

Problem 1, continued:



(*b*) Show a pipeline execution diagram for an execution of the code fragment below when the branch is taken. Label the ID-stage unit as indicated.

☑ Show pipeline diagram. ☑ Pay close attention to the branch.

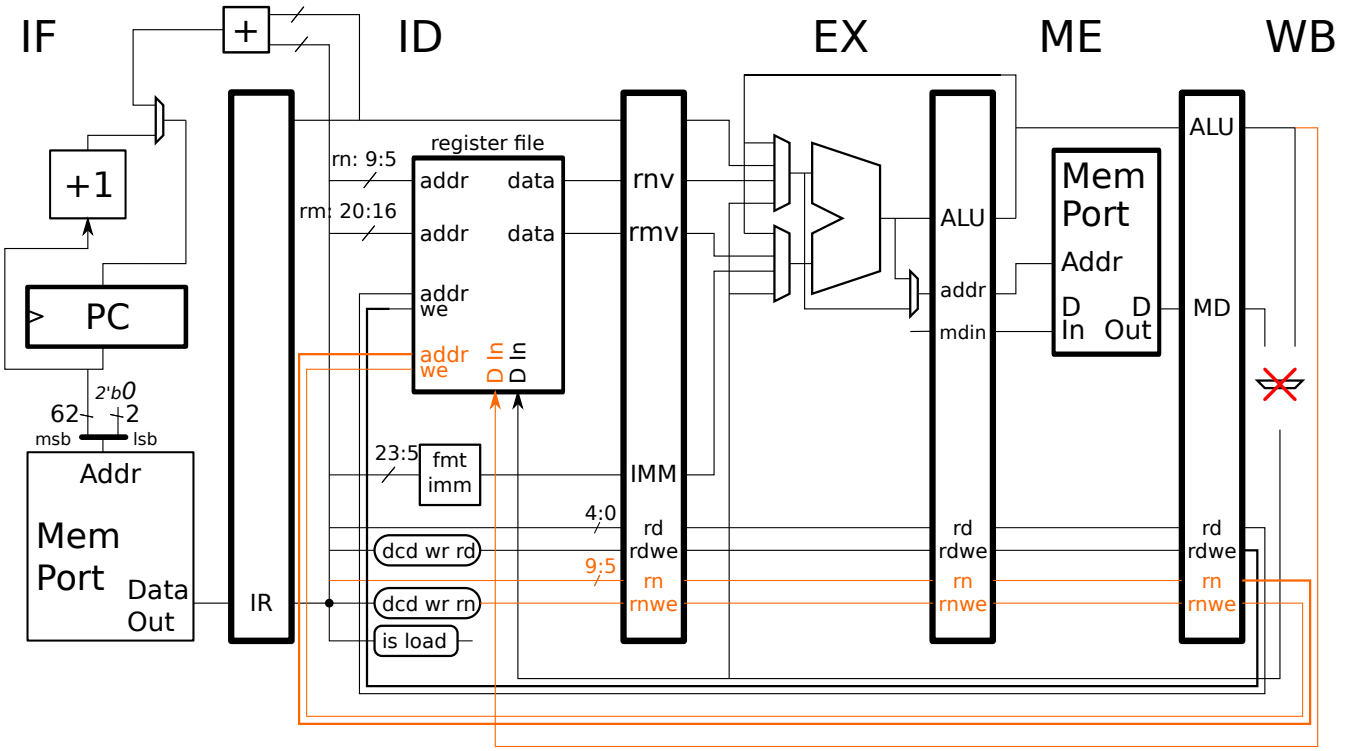
Solution appears below. Note that the branch stalls due to a dependency carried by `r1`. Because there are no bypass paths to the ID-stage comparison unit the `bne` must stall until `addi` reaches WB.

☑ Label the **inputs and outputs** of the ID-stage unit that computes the branch target. Label with $c : v$, where c is the cycle number and v is the value on the input or output.

Solution appears above in blue. Note that `IF.PC`, `ID.NPC`, and `EX.NPC` are all 30 bits and hold bits 31:2 of an instruction address. There's no need to store bits 1:0 since they are zero due to the instruction alignment requirement imposed by MIPS. (That is, the address of a MIPS instruction must be a multiple of 4.)

```
#       SOLUTION
#       Cycle             0  1  2  3  4  5  6  7  8  9  10
0x4000: addi r1, r2, 3    IF ID EX ME WB
0x4004: bne r1, r2, TARG     IF ID ----> EX ME WB
0x4008: lw r4, 0(r5)            IF ----> ID EX ME WB
0x400c: addi r5, r5, 4
0x4010: xor r8, r9, r10
TARG:
0x4014: add r5, r4, r4                      IF ID -> EX ME WB
#       Cycle             0  1  2  3  4  5  6  7  8  9  10
```

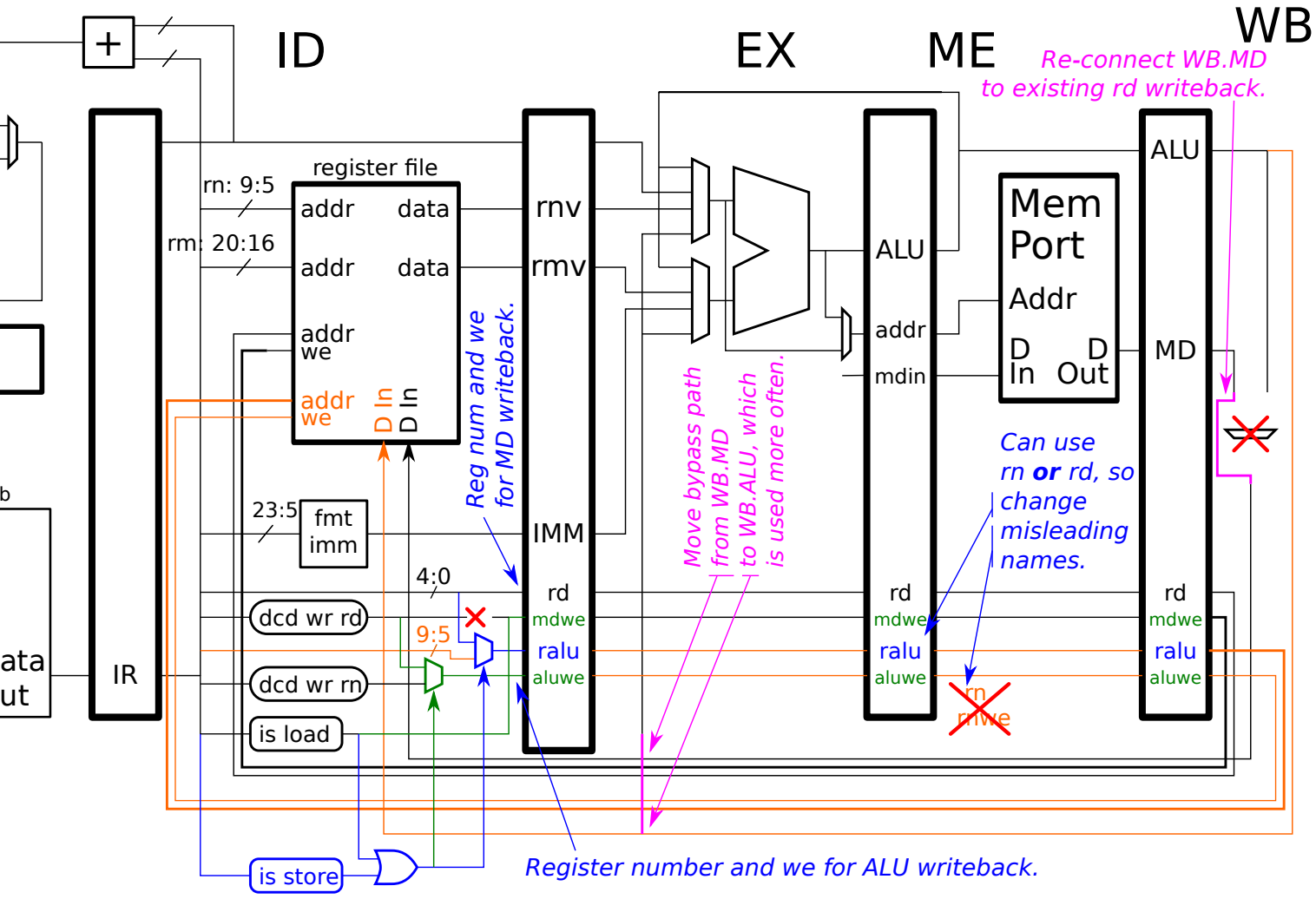
Problem 2: [20 pts] Appearing below is a partial implementation of ARM A64 taken from the solution to Homework 4. The WB-stage mux is crossed out because it's wasteful to use a 64-bit mux when the same functionality can be realized using less expensive logic in the ID stage. For reference, some A64 instructions are shown below, the comments show which field registers are encoded in.




```
@   rd   rn         : Writes rd and rn
ldr x1, [x2], #8    @ x1 = Mem[x2];  x2 = x2 + 8

@   rd, rn, rm      : Writes rd
add x3, x4, x1      @ x3 = x4 + x1

@   rd, rn          : Writes rd
and x1, x2, #34     @ x1 = x2 & 34;
```

Complete the changes so that instructions such as the ones above can write back their results. Assume that only the load instructions write the rn-field register.

Solution on next page.



☑ In WB consider re-connecting wires broken by the removal of the mux.

The WB.MD latch is re-connected to the register file D In (which was connected to the mux output before it was removed), shown in purple. The other input to the mux, WB.ALU, already has a path to the register file and because of the changes in ID that existing path suffices.

☑ In ID make changes so that instruction results can be written back to the correct registers.

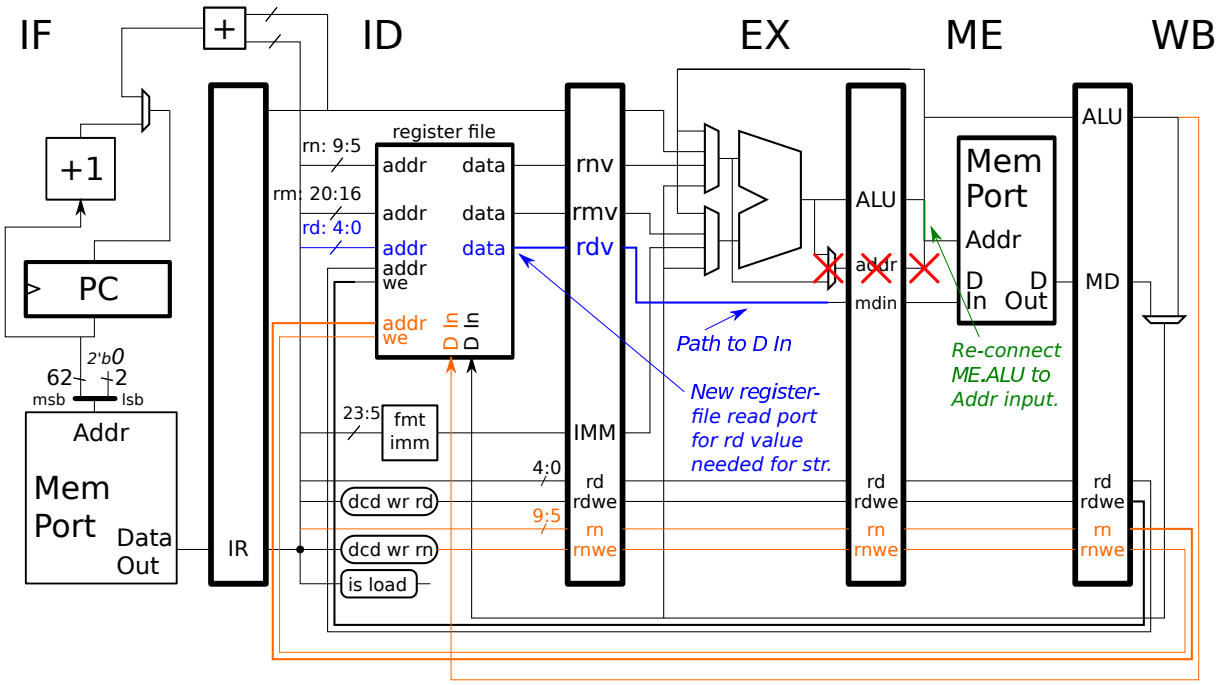
Previously, the black inputs to the register file were used for destinations specified in the instruction rd field (called rt in load instructions), and the orange inputs were used for destinations specified in the rn field, which according to the problem, could only be pre- and post-index memory instructions. Now, the black register file inputs will be used exclusively for WB.MD, which means load instructions, and the orange inputs will be used for WB.ALU, which is used by a variety of instructions. The WB.MD can only be written to rd (also called rt), so the pipeline latches carrying the register number, labeled rd, are unchanged. Because the black register file input is only used by loads, the write enable signal is set to is load rather than dcd wr rd (the logic is near the red × in ID) and the pipeline latch labels have been changed from rdwe to mdwe.

The orange register file inputs can now be used to the register specified in the rd **or** rn field. The only instructions that can use the rn field as a destination are load and store instructions. If there is a load or store instruction in ID, detected by the OR gate, the rn field and dcd wr rn logic are used for the orange writeback signals, otherwise rd and dcd wr rd.

☑ Make the best use of the existing from-WB bypass path.

We would expect that there are more arithmetic instructions than load instructions, so the bypass path has been moved from the WB.MD latch to the WB.ALU latch.

Problem 3: [20 pts] Appearing below again is the partial implementation of ARM A64 taken from the solution to Homework 4.




```
@   rd   rn         : Writes rd and rn.  Post-index
ldr x1, [x2], #8    @ x1 = Mem[x2];  x2 = x2 + 8

@   rd   rn         : Writes rd and rn.  Pre-index
ldr x1, [x2, #8]!   @ x2 = x2 + 8;  x1 = Mem[x2];

@   rd   rn  rm
str x1, [x2, x3]    @ Mem[x2+x3] = x1
```

(*a*) Make changes needed to implement the store instruction, see the example above. Just show datapath, not control logic.

☑ Changes for the store instruction.

Changes are in blue. The store instruction writes the value from the register specified in the `rd` (also called `rt`) field and so we need to add a new read port to the register file. We can't share the port currently used for `rn` or `rm` because some store instructions need both of those values.

(*b*) Do we really need *both* pre-index and post-index addressing for loads and stores? Eliminating either of them will reduce cost, but one's elimination would reduce cost by more than the other's. Indicate which saves more and show the hardware that can be removed and other needed changes. *Note: The phrase* and other needed changes *was not in the original exam.*

☑ Greater cost reduction by eliminating: ◯ *pre-index* ☑ *post-index* (check exactly one).

☑ Show the hardware that's not needed ☑ and other needed changes.

See the red × above. The removed hardware disconnects the Mem Port Addr input, so that's reconnected to `ME.ALU`.

Problem 4: [20 pts] When MIPS routine `coursei` is called register `a0` will hold an entry number, referring to the table at label `courses`. Complete the routine so that when it returns register `v0` will have the integer representation of entry number `a0` in the table. Note that the table itself holds floats. For example, when called with `a0=0` it should return with `v0=2740`, when called with `a0=2` it should return with `v0=3755`, etc.

☑ Complete so `v0` is integer representation of `a0`th table entry.

☑ Read the table as it is, don't modify it or read a different table.

Solution appears below.

```
        .data
courses:
        .float 2740
        .float 3750
        .float 3755
        .float 4755
        .float 4720
        .float 7722
        .float 7725
        .text
        # CALL VALUE: $a0: Entry in table to look up.
        # RETURN:     $v0: Table entry #$a0 represented as an integer.
coursei:
        la $t0, courses


        ## SOLUTION
        #
        sll $t1, $a0, 2    # Multiply by size of float, four bytes.
        add $t2, $t0, $t1  # Compute address of table entry.
        lwc1 $f0, 0($t2)   # Load entry into a FP regiser.
        cvt.w.s $f1, $f0   # Convert to an integer.
        jr $ra
        mfc1 $v0, $f1      # Move integer version of entry to result register.
```

Problem 5: [20 pts] Answer each question below.

(*a*) What kind of implementations were RISC ISAs designed to simplify?

☑ Kinds of implementations that RISC designed to simplify:

Pipelined implementations.

(*b*) Describe how the features below simplify RISC ISA implementations.

☑ Fixed-size instructions.

Short Answer: Because the hardware for fetching the next instruction does not need to check the size of the current instruction nor does it need shift and mask logic to extract the next instruction from the memory fetched in the current and previous cycles.

Detailed Answer: The address of the next instruction is always a constant distance (in many cases 4 bytes) from the address of the current instruction, and so we can compute the next instruction address with an adder connected to the program counter. In contrast, if instruction sizes varied the exact address of the next instruction could not be obtained until the current instruction was decoded. An implementation would either have to wait for the exact address before fetching or it would have to conservatively guess (adding only a small value to the address of the current instruction) and perhaps fetch a larger chunk of data than it would need, and then in a subsequent stage, when the current instruction is decoded, it would have to extract next instruction from the chunk of memory that was fetched.

☑ Avoiding arithmetic instructions that access memory.

Short Answer: There is no need for extra memory stages before the **EX** stage.

More Details: In RISC implementations instructions access the data memory port in a particular stage, **ME** in our five-stage MIPS implementation. If the arguments to arithmetic instruction could come from memory then instructions might have to access a data memory port both before and after they reach the **EX** stage. That would either require additional memory ports, which are costly, or logic that would allow a single memory port to be accessed from multiple stages, which would add stalls and complicate control logic.

(*c*) The SPECcpu package contains the source code for the SPEC benchmarks and scripts to compile and run them, but it does not come with compilers. The tester provides his or her own. Consider a *SPECcpu+* package that comes with compilers, and the requirement that those compilers be used. Why would that make SPECcpu+ less useful to computer engineers?

☑ SPECcpu+ less useful because:

Short Answer: SPECcpu+ cannot be used for the development of a new ISA since there is no way an existing compiler could target the new ISA (otherwise it wouldn't be a new ISA). For the same reason, it could not optimize for a new implementation.

More Details: If we are designing a brand new ISA then it could not possibly be one of the targets of SPEC's compiler and so we would have no way to run SPECcpu+ on our brand new system. If we are designing a new implementation of an existing ISA that SPEC's compiler can generate code for, then the optimizations performed by the compiler might not be right for our brand new implementation. These are not problems if we provide our own compiler.

(*d*) In class we described some optimizations as high-level, and some as low-level, performed by the back end. What distinguishes high- and low-level optimizations? Provide an example of a low-level optimization that could only be performed by the compiler back end.

☑ Difference between high- and low-level optimization.

A high-level optimization can be made without knowing the ISA, but low-level optimizations are made based on the ISA and also the implementation.

☑ Example of an optimization that must be low-level (that can only be done in the back end).

Register assignment.

☑ Briefly explain why.

Because registers can't be assigned if you don't know how many there are, or for that matter, if you don't know the exact instructions that will be accessing those registers.

(*e*) MIPS I has instruction `bgtz r1, TARG` in which the branch is taken if `r1>0` but it lacks an instruction like `bgt r1, r2, TARG` that would branch if `r1>r2`. Why?

☑ Why does MIPS lack `bgt r1, r2, TARG`?

Because the magnitude comparison will take just a bit too long.

☑ What would be the impact on performance of including `bgt r1, r2, TARG` in MIPS on "our" five-stage implementation?

It would lower the clock frequency slowing down all instructions.

The Good News: By including a `bgt` instruction we can eliminate `slt` (set less than) instructions, reducing the instruction count for many programs by 2%.

The Bad News: By lengthening the critical path our clock frequency has been reduced by 5%, which affects **all** instructions.

Name Solution

Computer Architecture
EE 4720
Final Examination

1 May 2017, 10:00–12:00 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(15 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

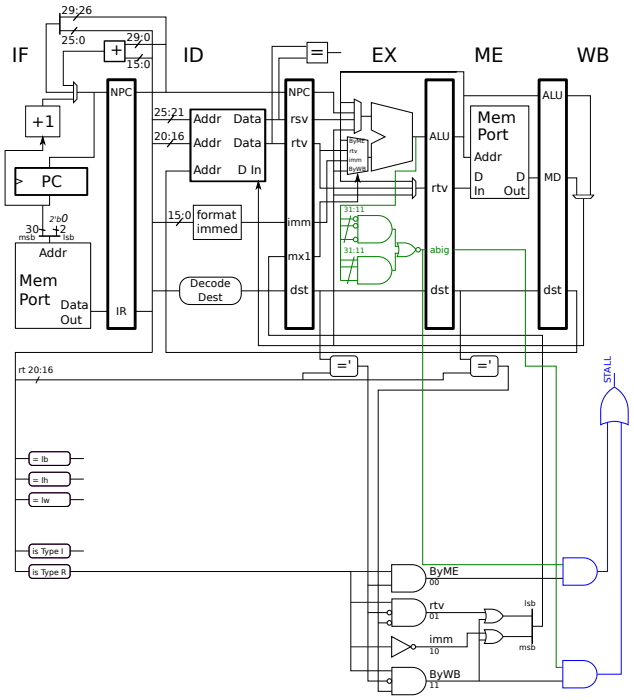
Alias My Alias Placeholder

Good Luck!

Problem 1: (20 pts) The diagram below, based on the solution to Homework 5, shows control logic that generates a stall signal when the value to be bypassed is too large for 12-bit bypass paths. The logic only works when the dependency is with the `rt` register of the consuming instruction and when the producing instruction is not a load. Modify the control logic so that it will generate a stall signal for a dependency to an `rs` register (first example below) and dependencies with loads. Pay attention to the load sizes.

```
# Dependency through rs register (r1 in the sub).
add r1, r2, r3
sub r4, r1, r5

# Producing instruction is a load word.
lw r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load half.
lh r1, 2(r3)
and r4, r5, r1

# Producing instruction is a load byte.
lb r1, 2(r3)
and r4, r5, r1
```

Use next page for solution.



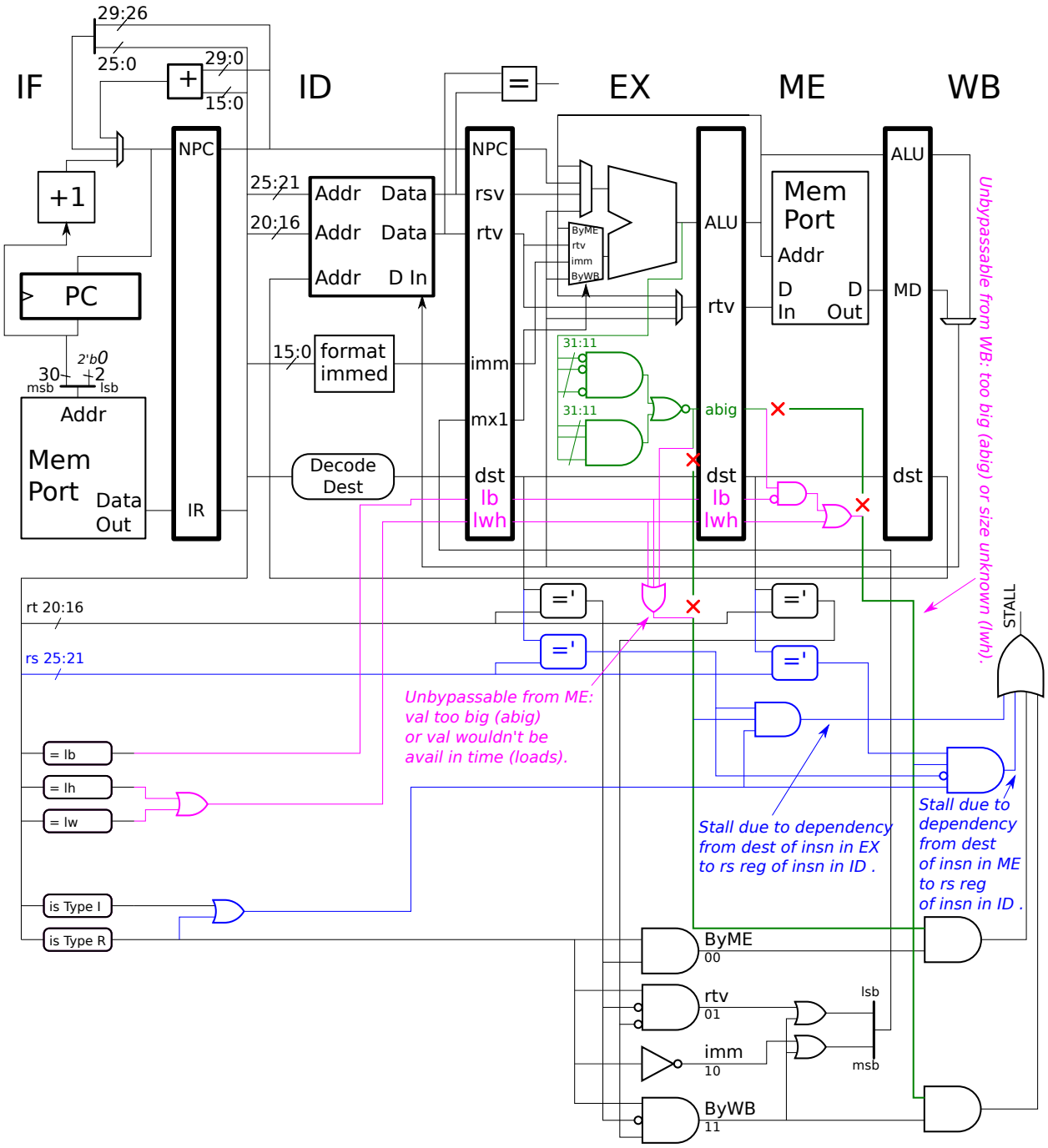
Use next page for solution.

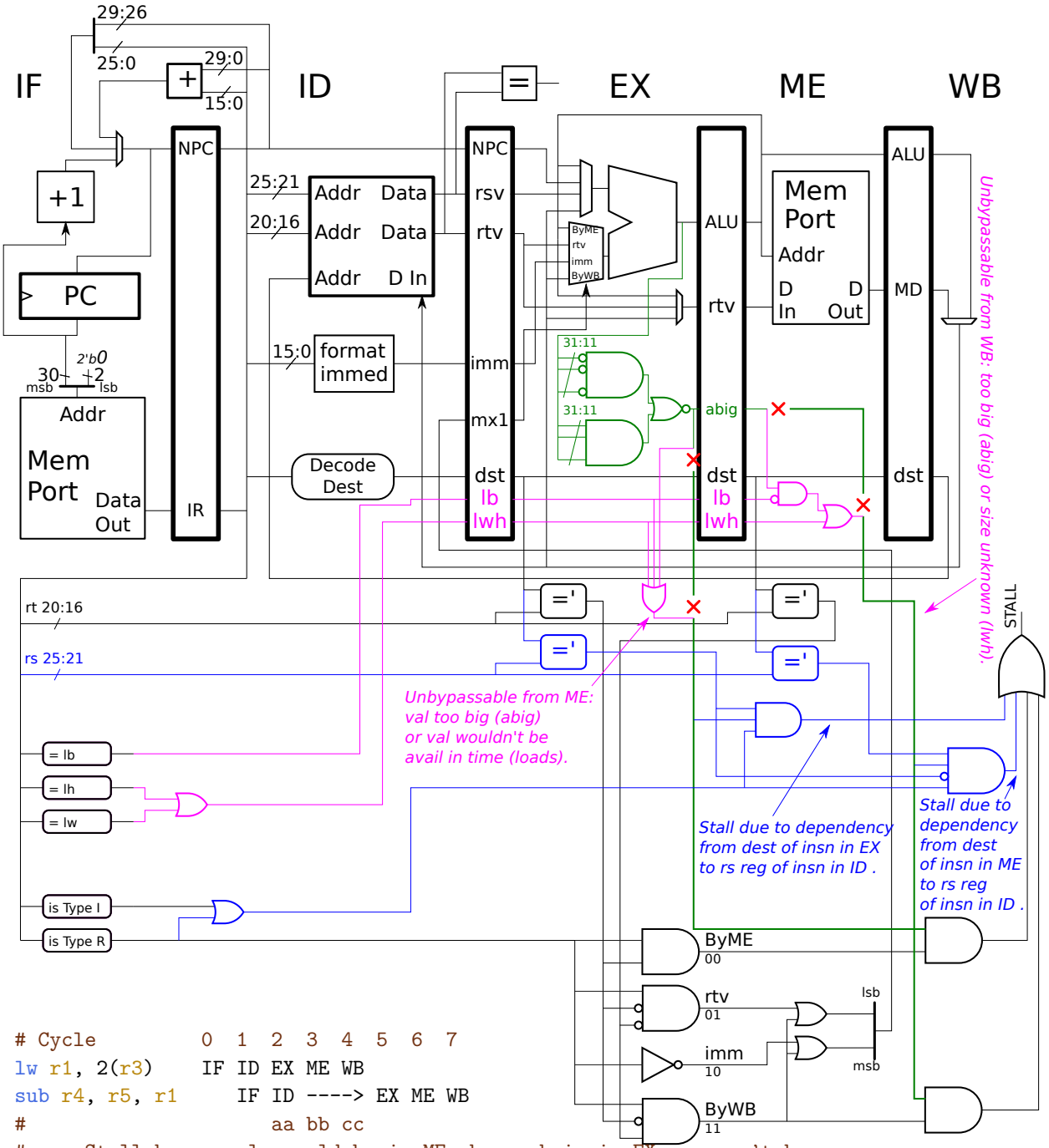
☑ Modify the control logic so that it also generates the stall signal for dependencies through the `rs` register that can't use 12-bit bypasses.

☑ Modify the stall control logic for when loads `lb`, `lh`, and `lw` produce the value to bypass, ☑ take into account whether value can use the 12-bit bypasses and ☑ whether the instructions are too close to bypass.

☑ Do not break existing control logic. As always ☑ consider cost and performance.

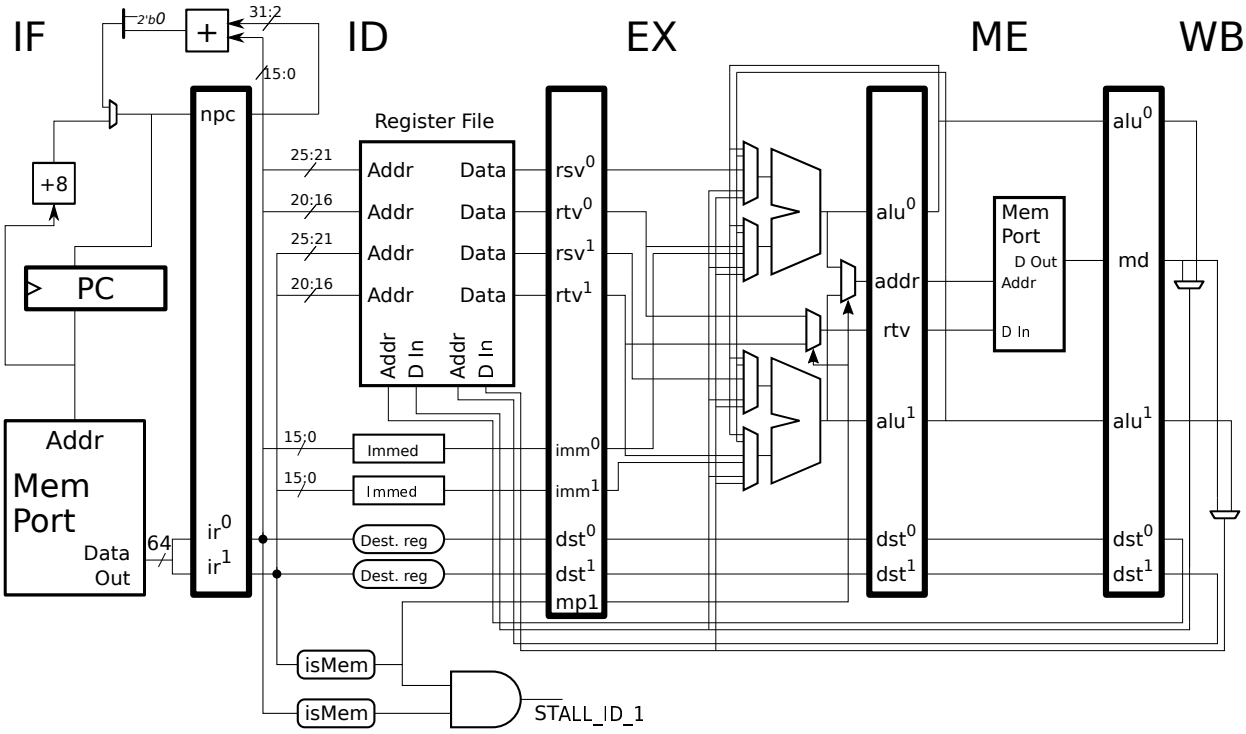
Solution appears below. The logic for stalling due to dependencies through the `rs` register appears in blue, and the logic for dependencies related to loads appears in purple.






```
# Cycle          0  1  2  3  4  5  6  7
lw r1, 2(r3)     IF ID EX ME WB
sub r4, r5, r1      IF ID ----> EX ME WB
#                      aa bb cc
# aa: Stall because lw would be in ME when sub is in EX, so can't bypass.
# bb: Stall because don't know if loaded value will fit in 12-bit bypass paths.
# cc: Don't stall, loaded value now is available from register file.

# Cycle          0  1  2  3  4  5  6  7
lb r1, 2(r3)     IF ID EX ME WB
sub r4, r5, r1      IF ID -> EX ME WB
#                      aa bb
# aa: Stall because lb would be in ME when sub is in EX, so can't bypass.
# bb: Don't stall, can bypass WB->EX and lb-loaded value can fit.
```

Problem 2: (15 pts) Illustrated below is a superscalar implementation taken from the solution to last year's final exam and the subject of this semester's Homework 7. Show the execution of the code sequences below on the illustrated superscalar MIPS implementation. Don't forget to check for dependencies.



(*a*) Show the execution of the code below on this implementation. Note that the address of the first instruction is `0x1000`.

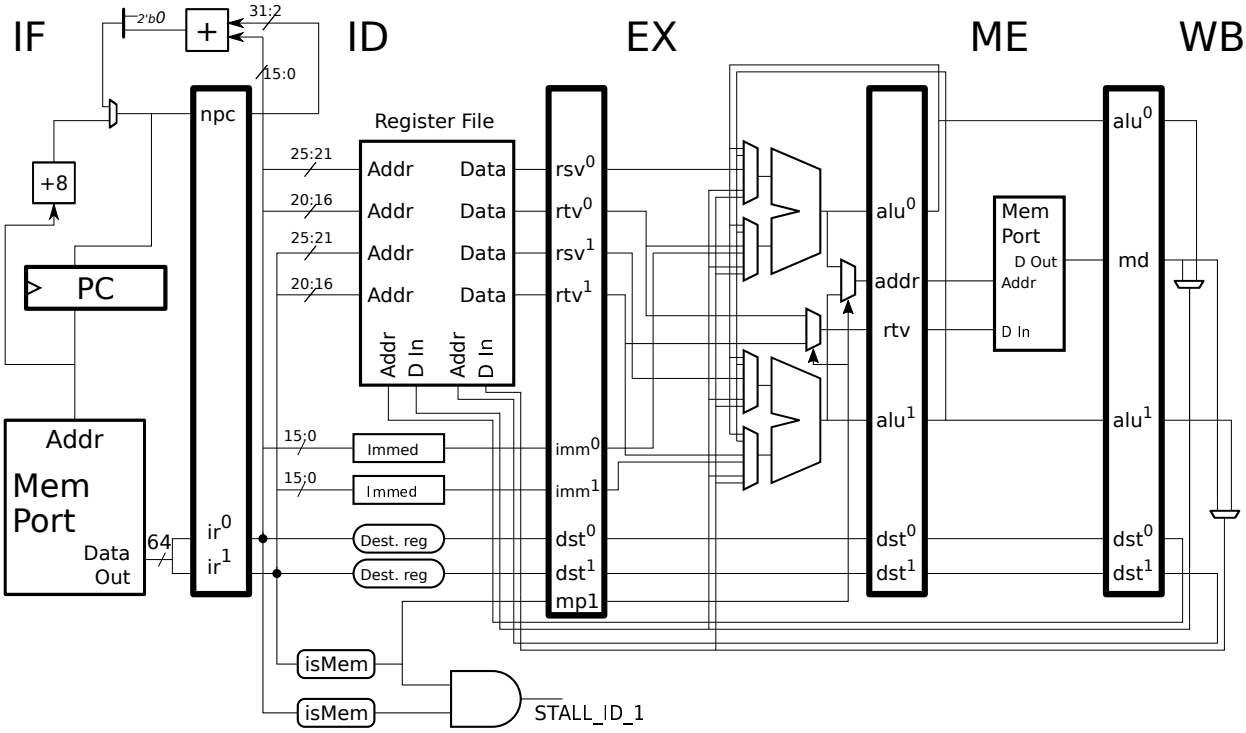
☑ Show execution of the following code sequence. ☑ Pay attention to ME in the diagram.

☑ Check for dependencies.

The solution appears below. The `lw r3` stalls because the ME stage can only accommodate one memory instruction. The last `add` stalls due to a dependence.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID EX ME WB
lw r3, 4(r2)     IF ID -> EX ME WB
lw r4, 8(r2)        IF -> ID EX ME WB
add r5, r1, r5      IF -> ID EX ME WB
add r5, r3, r5            IF ID EX ME WB
add r5, r4, r5            IF ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

Problem 2, continued: The illustration below is the same as the one on the previous page.



(*b*) Show the execution of the code below on the illustrated implementation when the branch is taken. Use the classroom default assumption: fetches are aligned.

☑ Show execution of the following code sequence. ☑ Check for dependencies.

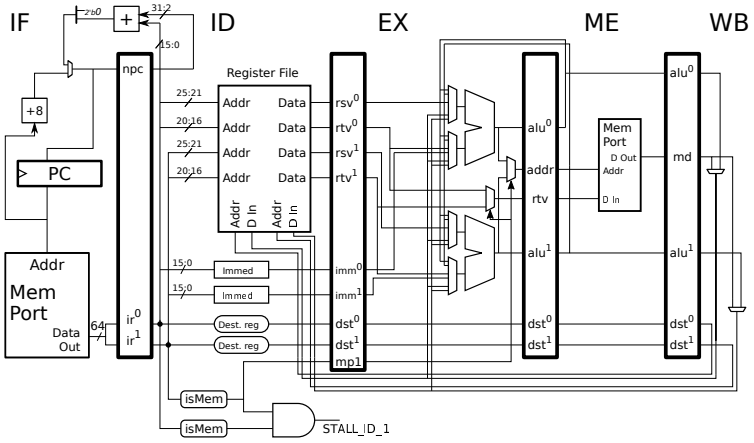
☑ Show all instructions that enter the pipeline, even those that are squashed in IF or later.

☑ Pay attention to instruction addresses, such as `0x1000`.

Solution appears below. Because the branch is resolved in **ID** the branch target is not fetched until the branch reaches **EX**, in cycle 2, and therefore two wrong-path instructions are fetched in cycle 1. Because the memory port in **IF** can only fetch aligned groups (meaning that the address of the instruction in slot 0 must be a multiple of 8) the **andi** instruction is fetched and then squashed as soon as it arrives. The **sb** stalls because of a dependency with **or**. Because there is no bypass into **EX/ME.rtv** the **sb** must stall in **ID** until **or** reaches **WB**. (The implementation used in Problem 1 has a bypass that would eliminate this stall.)

```
# SOLUTION
# Branch is taken.      Cycle  0  1  2  3  4  5  6  7  8  9
0x1000: bne  r1, r4  TARG      IF ID EX ME WB
0x1004: sub  r5, r2, r7        IF ID EX ME WB
0x1008: xor  r10, r11, r12        IFx
0x100c: lbu  r9, 0(r5)            IFx
0x1010: andi r8, r9, 12              IFx
# Cycle                        0  1  2  3  4  5  6  7  8  9
TARG:
0x1014: or   r11, r5, r12            IF ID EX ME WB
0x1018: sb   r11, 0(r5)                 IF ID ----> EX ME WB
# Cycle                        0  1  2  3  4  5  6  7  8  9
```

Problem 2, continued: The illustration below is the same as on the previous page.



(*c*) Appearing below is an execution of MIPS code on the illustrated superscalar implementation shown for the first two iterations. Compute the CPI for a large number of iterations. If necessary extend the execution diagram.

```
 lw r1, 0(r2)     IF ID EX ME WB
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4     IF ID -> EX ME WB                    # First Iteration
 lw r1, 0(r2)       IF ID -> EX ME WB
 bne  r2, r3  LOOP     IF -> ID EX ME WB
 addi r2, r2, 4        IF -> ID EX ME WB
 ???                         IFx                        # Fallthrough insn.
 ???                         IFx                        # Fallthrough insn.
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4                IF ID EX ME WB           # Second Iteration
 lw r1, 0(r2)                  IF ID EX ME WB
 bne  r2, r3  LOOP                IF ID EX ME WB
 addi r2, r2, 4                   IF ID EX ME WB
 ???                                 IFx                # Fallthrough insn.
 ???                                 IFx                # Fallthrough insn.
LOOP: #  Cycle   0  1  2  3  4  5  6  7  8  9  10
 add r1, r1, r4                           IF ID EX ME WB  # Third Iteration
#        Cycle   0  1  2  3  4  5  6  7  8  9  10 11 12
                 ! 1st Itr   ! 2nd Itr! 3rd Itr!
                 ! 4 cyc     ! 3 cyc  ! 3 cyc  !
```

☑ CPI for a large number of iterations.

To determine the number of cycles in an iteration we need to find a repeating pattern. The state of the pipeline at the start of the first iteration, in cycle 1, is clearly different than the state at the start of the second iteration, in cycle 5. Therefore in the solution above the start of a third iteration has been added. We can see that the third iteration starts in cycle 8. In both cycles 5 and 8 the pipeline contains: `add` in `IF0`, `addi` in `EX1`, `bne` in `EX0`, `lw` in `ME1`, and `add` in `ME0`. Since the states are identical we can expect the third iteration to take the same time as the second.

The duration of iteration i is the time from when the first instruction of iteration i enters `IF`, to the time when the first instruction of iteration $i+1$ enters `IF`. That time is highlighted above for the second iteration, which has a duration of $8-5=3$ cycles.

Therefore the CPI is $\boxed{\frac{8-5}{4} = \frac{3}{4} = 0.75\,\text{CPI}}$.

Problem 3: (20 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{12} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 8-outcome local history, and one system has a global predictor with a 8-outcome global history. Branch `B2` consists of a repeating pattern that starts with `TNTT` and is either followed by three not-taken outcomes, `nnn`, or four taken outcomes, `tttt`. (They are shown in lower case for clarity.) The `nnn` sequence occurs with probability .4, and is not correlated with anything.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
                                 ↓
B1:   T  N  T  T  N          T  N  T  T  N           T  N  T  T  N
B2:    T  N  T  T  n  n  n    T  N  T  T  t  t  t  t
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

The diagram below shows the counter values starting from an initial value of 0. A repeating pattern starts at the third group because the counter value is the same, 2, at the beginning and end of the group. So based on that five-outcome group the accuracy is $\boxed{\frac{3}{5}}$.

```
     0  1  0  1  2  1          2  1  2  3  2           3  2  3  3  2  <-- Counter
B1:   T  N  T  T  N          T  N  T  T  N           T  N  T  T  N
      x     x  x  x          x  x  x     x              x        x    <-- Pred. Outcome
```

☑ What is the accuracy of the bimodal predictor on branch B2? ☑ Account for the variable pattern length.

This is best analyzed by considering the four possible cases of the way the random sequence can occur before and after the fixed sequence (TNTT). These are shown in the table below. For each case the number of mispredictions is computed starting at the fixed sequence and continuing into the second random sequence. What makes this easy (relatively) is that when the fixed sequence starts the counter will be either 0 or 3. Therefore we can compute an exact prediction ratio for each of the four cases. These are shown under the Pred column. The Prob column is the probability that the fixed sequence will be surrounded with the particular random outcomes. The numbers under the weight column give something like the space taken up by the particular case. These are used to weight the prediction accuracies. In particular the value under Weight is the product of the value under Pred and the value under Weight. The sums are shown at the bottom. The prediction accuracy is the weighted value divided by the weight: $\boxed{\frac{4.92}{7.6} = .647368}$.

											Pred	Prob	Weight	Weighted
			0	1	0	1	2	1	0					
n	n	n	T	N	T	T	n	n	n					
			x		x	x	x				3/7	.4 * .4	.4 * .4 * 7	.4 * .4 * 3
			3	3	2	3	3	2	1					
t	t	t	t	T	N	T	T	n	n	n				
				x			x	x			3/7	.6 * .4	.6 * .4 * 7	.6 * .4 * 3
			0	1	0	1	2	3	3	3				
n	n	n	T	N	T	T	t	t	t	t				
			x		x	x					5/8	.4 * .6	.4 * .6 * 8	.4 * .6 * 5
			3	3	2	3	3	3	3	3				
t	t	t	t	T	N	T	T	t	t	t	t			
				x							7/8	.6 * .6	.6 * .6 * 8	.6 * .6 * 7
											---	--------	----------	----------
												1	7.6	4.92

☑ What is the accuracy of the local predictor on branch B2? ☑ Account for the variable pattern length.

Short Answer: Assuming that it always predicts t for the outcome after TNTT, the accuracy will be

$$\frac{7 \times .4 \times \frac{6}{7} + 8 \times .6 \times \frac{8}{8}}{7 \times .4 + 8 \times .6} = .947368$$

where the prediction accuracy for TNTTnnn, $\frac{6}{7}$, and for TNTTtttt, $\frac{8}{8}$, have been weighted by the probability that a B2 outcome is part of TNTTnnn, $\frac{7 \times .4}{7 \times .4 + 8 \times .6}$, or part of TNTTtttt, $\frac{8 \times .6}{7 \times .4 + 8 \times .6}$.

Long Answer: Because the local history length, 8, is long enough to identify the position within the pattern, the only outcome that can't be predicted with 100% accuracy is the first branch after the fixed sequence, TNTT. For example, consider TNTTnnn. It will correctly predict the fixed-sequence outcomes, TNTT and it will correctly predict the last two ns because once it sees the first of the three ns it will recognize that there will be two more. Or, to put it more precisely, when the local history contains tttTNTTn or nnnTNTTn the corresponding PHT entries will hold a zero because each time either of the two local histories was encountered in the past the B2 outcome would be n (that's the second n) and so the PHT entry would be decremented. By the same logic the third n would always be correctly predicted (after warmup) as would the second, third, and fourth t. When predicting the first outcome after the fixed sequence the local history will be either TnnnTNTT or ttttTNTT. We know that 60% of the time the outcome is t. As an approximation we can assume that the PHT entry would be 2 or 3 since 60% of the time it is incremented and 40% of the time it is decremented. It is possible to compute an exact probability distribution for the counter values by constructing a four-state Markov chain and solving the balanced flow equations $ap_i = (1-a)p_{i+1}$ for $0 \geq i \leq 2$, where a is the probability that the branch is taken, $a = .6$ here. Solving these yields $p_0 = \frac{\frac{a}{1-a} - 1}{\left(\frac{a}{1-a}\right)^4 - 1}$ and $p_i = \left(\frac{a}{1-a}\right)^i p_0$. From this we get $p_0 = .123077$ and

the probability of a taken prediction $p_2 + p_3 = .692308$ and a not taken prediction is $p_0 + p_1 = .307692$. We can use these numbers to compute an overall prediction accuracy

$$\frac{7 \times .4 \times \frac{6.307692}{7} + 8 \times .6 \times \frac{7.692308}{8}}{7 \times .4 + 8 \times .6} = .939271,$$

which is only slightly lower than the estimated accuracy.

☑ What is the minimum local history size for which branch `B1` and `B2` will not interfere with each other?
☑ Explain.

Seven outcomes. With seven outcomes the `B2` local history must contain either three consecutive `t`s or two consecutive `n`s, which never occur in a `B1` local history. This means that `B1` and `B2` will never use the same `PHT` entries and so won't interfere with each other with a seven-outcome local history. Now consider six outcomes. Local history `nTNTTn` could be for `B1` and `B2`, and so they would both use the same `PHT` entry. For `B1` the next outcome would be `T`, but for `B2` the next outcome would be `n`, and so the shared `PHT` entry could not predict both branches accurately. (Remember that there's no difference between `n` and `N` and no difference between `t` and `T`, so a local history of `nTNTTn` is exactly the same as `NTNTTN`. Upper and lower case are only being used to show which branch outcomes belong to the fixed part (upper case) and which belong to the repeating part (lower case).

☑ Note that an arrow (↓) points at an execution of `B1`. Show the value of the GHR at the time that that execution is being predicted.

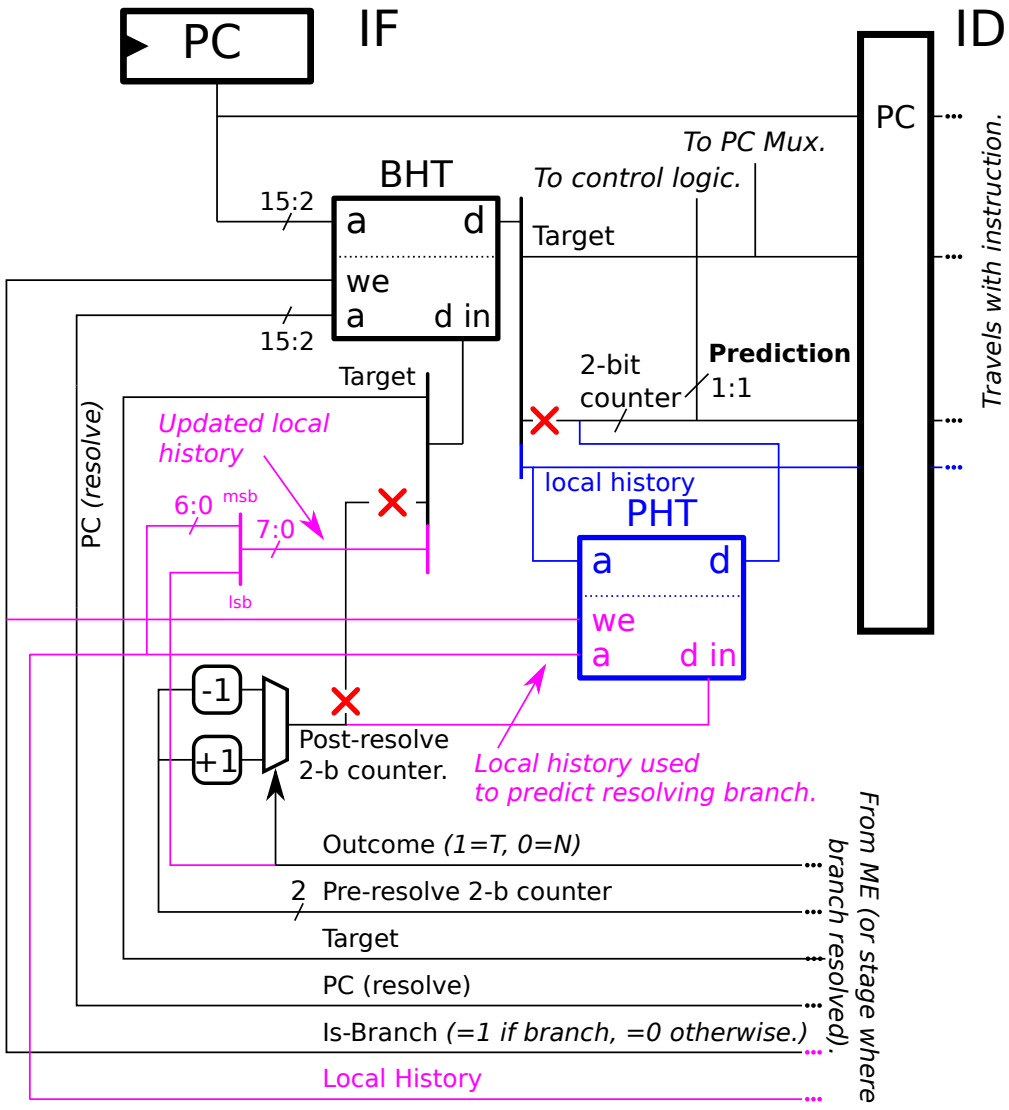
The local history will contain `TTTTNnnn`.

Problem 3, continued:

(*b*) Appearing below is a diagram of a bimodal predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a local predictor with an 8-outcome local history.

☑ Show the PHT, and connections for ☑ prediction and ☑ update.

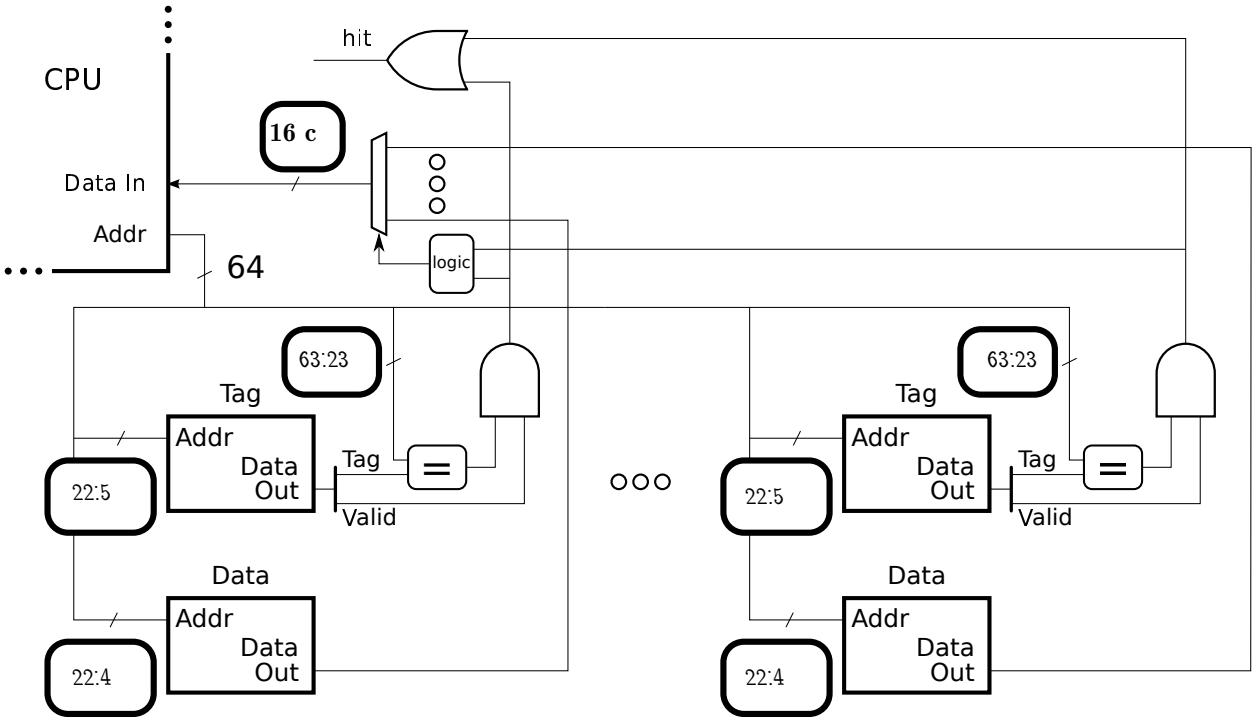
Solution appears below. The changes needed to predict the branch appear in blue. The BHT now stores local history, and that is connected to a newly added PHT. The prediction comes from the 2-bit counter in the PHT rather than the BHT. Update hardware appears in purple. Note that for update the original local history is used to index the PHT, but the updated local history is written into the BHT.



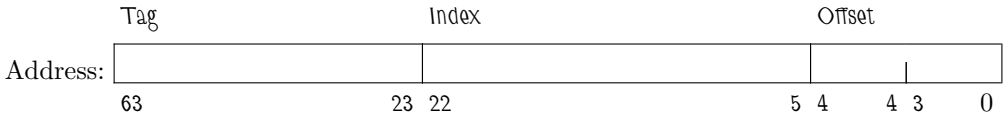
Problem 4: (15 pts) The diagram below is for a 32 MiB (2^{25} B) four-way set-associative cache with a line size of 32 B.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)



☑ Memory Needed to Implement ☑ Indicate Unit!!:

It's the cache capacity, 32 MiB, plus $4 \times 2^{23-5}\,(64 - 23 + 1)$ bits.

☑ Show the bit categorization for a direct-mapped cache with the same line size and capacity as the cache above.

The cache above is 32 MiB and 4-way set associative. In a direct mapped cache there is just one way with four times the storage of a way in the cache above. To get four times the number of entries the number of index bits is increased by two, and so the index bits will start at position 24 instead of 22. The other bit positions remain the same.

Problem 4, continued: The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 1024 B (2^{10} B). Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
int *a = 0x2000000; // sizeof(int) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^{10} = 1024$ bytes is given. The size of an array element, which is of type int, is $4 = 2^2$ B, and so there are $2^{10}/2^2 = 2^{10-2} = 2^8 = 256$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^8 elements, and so the next $2^8 - 1 = 255$ accesses will be to data on the line, hits. The access at `i=256` will miss and the process will repeat. Therefore the hit ratio is $\frac{255}{256}$.

Problem 5: (30 pts) Answer each question below.

(*a*) Consider a 4-way superscalar system and a scalar system with a 4-lane vector unit. Both can compute arithmetic at a rate of 4 operations per cycle. The vector system is cheaper but the superscalar system is more flexible.

☑ Why is the vector system less costly?

The vector system only needs to fetch and decode one instruction per cycle, unlike four for the superscalar system and so the superscalar system uses four times as much decode hardware. In a 4-way superscalar processor we expect there to be floating-point bypass paths from each of 4 slots in WF to the two functional unit (A1/M1) inputs for each slot. That's a total of $4 \times 4 \times 2 = 32$ bypass multiplexor inputs. But in a system with a vector unit we don't expect cross-lane bypasses. That is, there's no bypass from the lane-2 value of a result in WF to, say, the lane-1 input, though we do expect a bypass from the lane-2 value in WF to the lane-2 unit input. Each lane requires 2 multiplexor inputs for bypass (one input to each mux), therefore the number of bypass paths is just $2 \times 4 = 8$.

☑ Show something the superscalar system can do that the vector system cannot.

☑ Explain why vector system can't execute equivalent vector code as efficiently.

Vector instructions must apply the same operation to each lane of its operands. A 4-way superscalar system could execute four different operations, for example, the set of operations below.

```
# SOLUTION
 add.d f0, f2, f4
 sub.d f6, f8, f10
 mul.d f12, f14, f16
 add.s f18, f19, f20
```

(*b*) Unlike MIPS, ARM A64 has pre-index and post-index load and store instructions. Show two code examples, one in A64 that uses a post-index load, and one in MIPS that does the same thing (but without a post-index load). The exact syntax of the ARM instructions is not important, use comments to clarify instructions.

☑ ARM code and ☑ equivalent MIPS code.

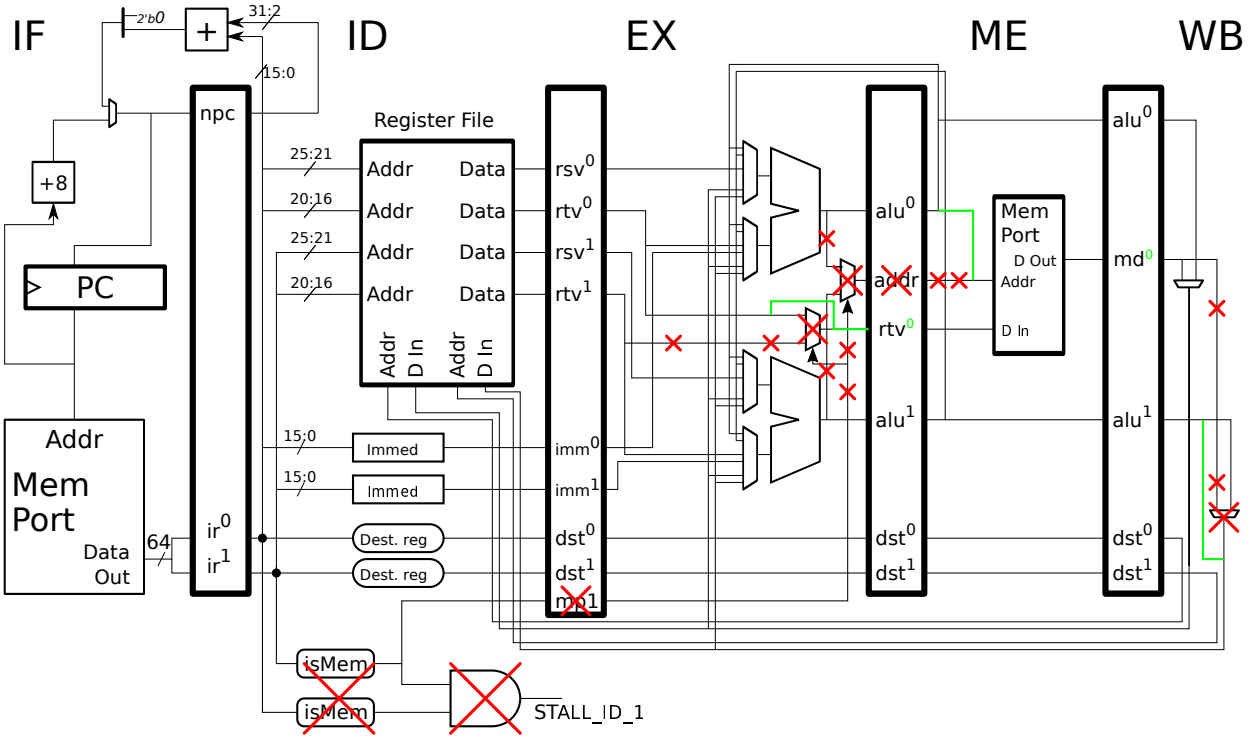
```
@ SOLUTION

@ ARM A64
 ldr x1, [x2], #8     @  x1 = Mem[x2];  x2 = x2 + 8

# MIPS
 lw r1, 0(r2)         #  r1 = Mem[r2]
 addi r2, r2, 4       #  r2 = r2 + 4
```

(*c*) What substantial additional hardware is needed to implement ARM A64 pre- and post-index loads when starting with something like our five-stage MIPS implementation. (Think about Homework 4.) *Note: The words "substantial" and "costly" were not included in the original exam.*

☑ Costly additional hardware for pre- and post-index loads.

Both the pre- and post-index loads need a second write port to the register file. The post-index load might require an additional pipeline latch so that both the unincremented value and incremented value can be passed from EX to ME, or a separate adder in the ME stage to do the post-increment, or some other costly addition.

(*d*) VLIW ISAs are supposed to do for superscalar implementations what RISC ISAs did for pipelined implementations. The diagram below shows our 2-way superscalar MIPS. Show how a 2-slot-bundle VLIW ISA (perhaps one a lot like MIPS) could simplify hardware in this implementation related to the sharing in ME.



☑ Modify the hardware above.

☑ Explain the bundle slot restrictions based on modified hardware.

Solution appears above. Unnecessary hardware is crossed out with red exes and where necessary wires reconnected in green.

A 2-slot bundle VLIW ISA designed for implementations like the one above in which there was one ME-stage memory port would require that a memory (load or store) instruction be placed only one slot (slot 0 in the solution above). That is, a memory instruction in slot 1 would be invalid and raise an illegal instruction exception. With that restriction the memory port D In input only needs a connection to slot 0, and so the EX-stage mux providing a path from slot 1 can be removed. Similarly, other connections to or from the memory port to slot 1 have been removed.

☑ Explain why the control logic driving STALL_ID_1 would no longer be needed.

The bundle slot restrictions forbid two memory instructions in a bundle, and so there's no need to check for it. (In fact, there can't be a memory instruction in slot 1 even without a memory instruction in slot 0.)

(*e*) When an exception occurs (or a trap instruction is executed) the processor switches from user mode into privileged mode (also called system mode). Explain how privileged mode affects instruction execution, including loads, compared to user mode.

☑ Effect of privileged mode on instruction execution including ☑ effect on load instruction execution.

In privileged mode *all* instructions can be executed, but in user mode only a subset of instructions can be executed. Similarly, in privileged mode a load or store can access any valid memory address, in user mode loads and stores can only accesses addresses to which they have been granted access.

(*f*) It's hard to choose a line size that makes everyone happy. Explain how a long line size might slow down some programs in a small cache in comparison to the right line size (for those programs).

☑ With a small cache large lines can slow some programs because:

Short Answer: because the S bytes of data (say) that some program needs cached won't fit in a $4S$ byte cache because the program only uses $\frac{1}{8}$ of the data in a line. It would take an $8S$-byte cache to hold the S bytes in such a case.

For example, consider a $64\,\mathrm{kiB}$ cache with a $1024\,\mathrm{B}$ cache. Such a cache can hold 64 lines. Consider a program that frequently needs to access 100 bytes of data where the address of byte i is $2000i$, for $0 \le i < 100$. Each line holds just one byte of the needed data plus 1023 unneeded bytes. Now consider a cache with 32-byte lines, but also of $64\,\mathrm{kiB}$. This cache can hold $\frac{64\times 2^{10}}{32} = 2048$ lines. That's more than enough for the program.

☑ Describe the characteristics of code that works well with long lines.

Code that accesses data sequentially. For example, `for (i=0; i<1000000; i++ ) sum += a[i];` . Here, the access to data is sequential.

50 Spring 2016 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 30 March 2016, 9:30–10:20 CDT

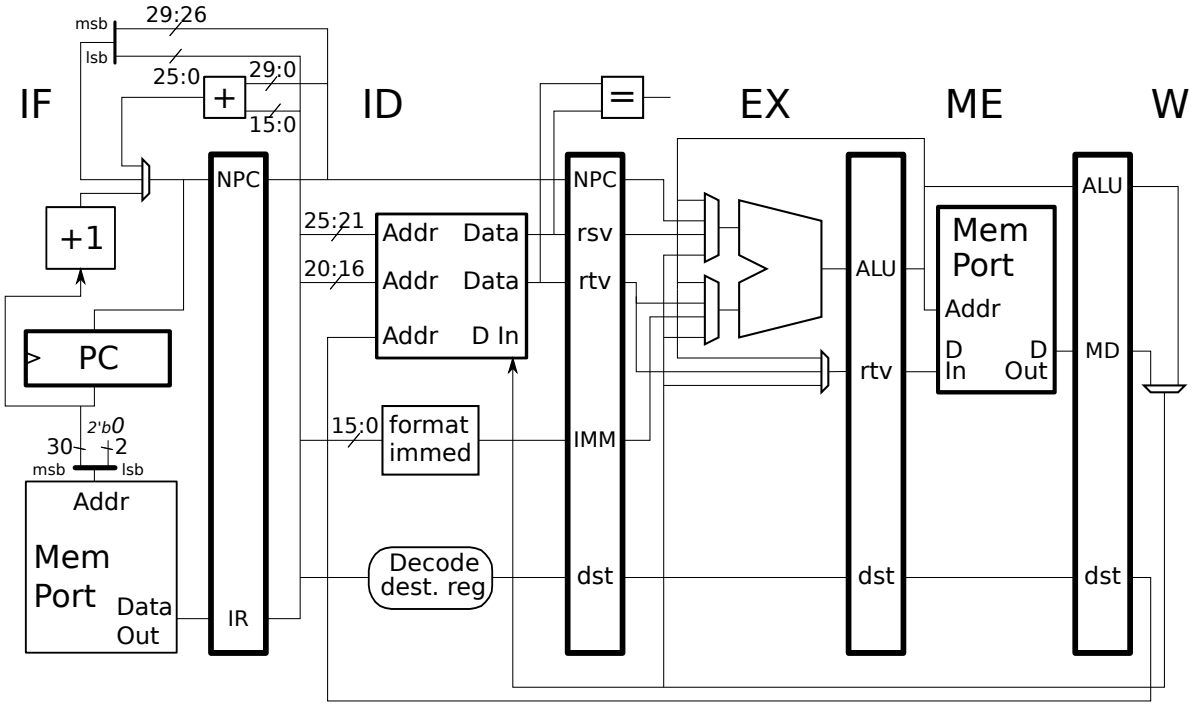
Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(12 pts)
Problem 4		(28 pts)
Problem 5		(10 pts)
Exam Total		(100 pts)

Alias Apple/Mate?

Good Luck!

Problem 1: [25 pts] Appearing below are what are supposed to be pipeline execution diagrams (PEDs) of code fragments executing on the illustrated implementation. The PEDs are incorrect.

(*a*) Correct the PEDs.



☑ Correct the PED below.

```
add r1, r2, r3   IF ID EX ME WB
lw r3, 0(r1)        IF ID -> EX ME WB

# SOLUTION
add r1, r2, r3   IF ID EX ME WB
lw r3, 0(r1)        IF ID EX ME WB
```

Solution appears above. Note that the dependency can be bypassed and so there is no need to stall.

☑ Correct the PED below.

```
lw r3, 0(r1)     IF ID EX ME WB
add r4, r3, r5      IF ID -> EX ME WB
sub r6, r7, r8         IF ID EX ME WB

# SOLUTION
# Cycle          0  1  2  3  4  5  6  7
lw r3, 0(r1)     IF ID EX ME WB
add r4, r3, r5      IF ID -> EX ME WB
sub r6, r7, r8         IF -> ID EX ME WB
```

The `sub` has to stall in cycle 3 since `ID` is occupied by the `add`.

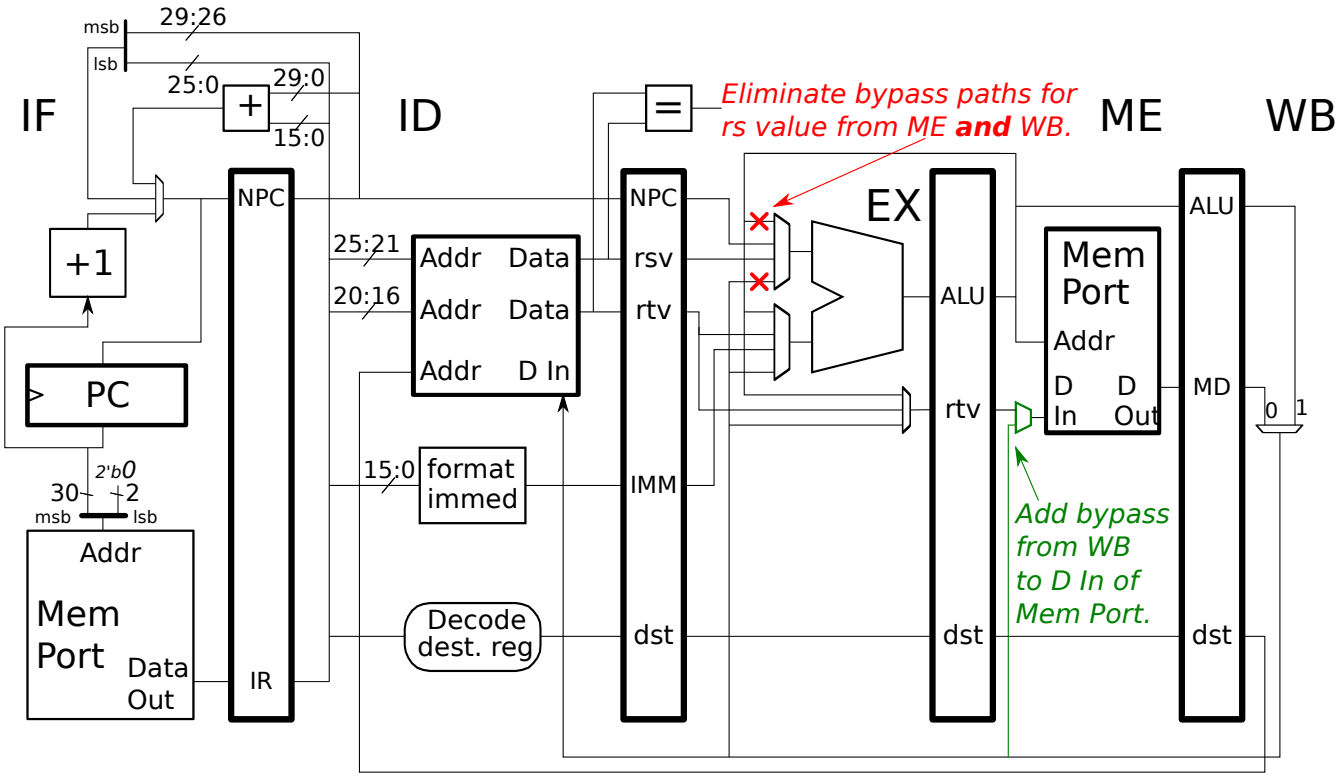
☑ Correct the PED below.

```
# Cycle            0  1  2  3  4  5  6  7
 beq r1, r1  TARG  IF ID EX ME WB   # Branch is taken.
 xor r5, r6, r7       IF IDx
 add r8, r9, r10         IFx
TARG:
 sub r2, r3, r4             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7

 # SOLUTION
 # Cycle            0  1  2  3  4  5  6  7
 beq r1, r1  TARG  IF ID EX ME WB   # Branch is taken.
 xor r5, r6, r7       IF ID EX ME WB
 add r8, r9, r10
TARG:
 sub r2, r3, r4          IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7
```

The delay-slot instruction should be executed, and the branch is resolved in ID so the target must be fetched when the branch is in EX.

(*b*) Appearing below are more PEDs which are not correct for the illustrated implementation. This time **modify the implementation** so that the executions are correct. Only make necessary changes.

- Delete a bypass path by showing an × at the **mux input** where it ends.
- Do not delete or add more hardware than is necessary.



☑ Modify the implementation so that the execution below is correct.

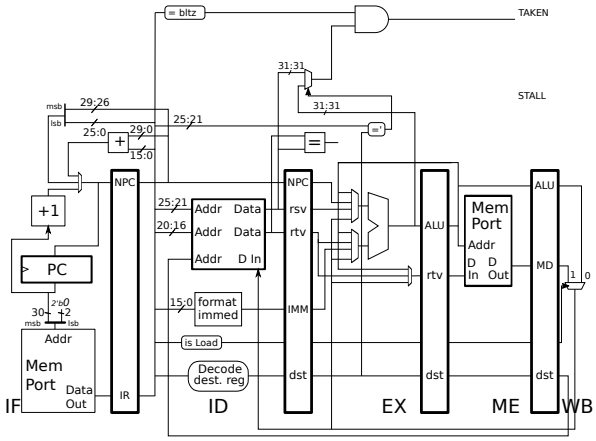
```
add r1, r2, r3   IF ID EX ME WB
sub r3, r1, r5      IF ID ----> EX ME WB
```

☑ Modify the implementation so that the execution below is correct.

```
lw r1, 0(r2)   IF ID EX ME WB
sw r1, 0(r3)      IF ID EX ME WB
```

Problem 2: [25 pts] The implementation below is based on the solution to Homework 2 Problem 2 in which a bypass was added for `bltz` instructions.

Use Next Page for Solution



Use Next Page for Solution

(*a*) The implementation can only bypass values in `EX` to a `bltz`. Modify the implementation on the next page so that values can be bypassed from both `EX` and `ME`. With these changes the two fragments below should run without a stall and of course bypass the correct value.

```
# Example 1
add r1, r2, r3
sub r4, r5, r6
bltz r1, TARG

# Example 2.
add r1, r2, r3
sub r1, r1, r6
bltz r1, TARG
```

(*b*) A bypass from EX isn't possible for the code fragment below, and a bypass from `ME` is problematic too. On the next page add logic to generate a stall signal for these situations (load/`bltz` dependencies) and connect it to the word `STALL` in the upper-right of the diagram. Notice that there is an is Load logic block in `ID`.

```
lw r1, 0(r2)
bltz r1, TARG
```

(*c*) Explain why it would not be a good idea to bypass the load value to the `bltz` when the load is in `ME`.

☑ Bypassing load from `ME` not a good idea because:

Because data is available at the `D Out` output of the memory port very late in the clock cycle and so it can't be used for anything without increasing the clock period. That is, `D Out` is on the critical path.

Problem 2, continued:

☑ Modify implementation so `bltz` can bypass from `EX` and `ME`.

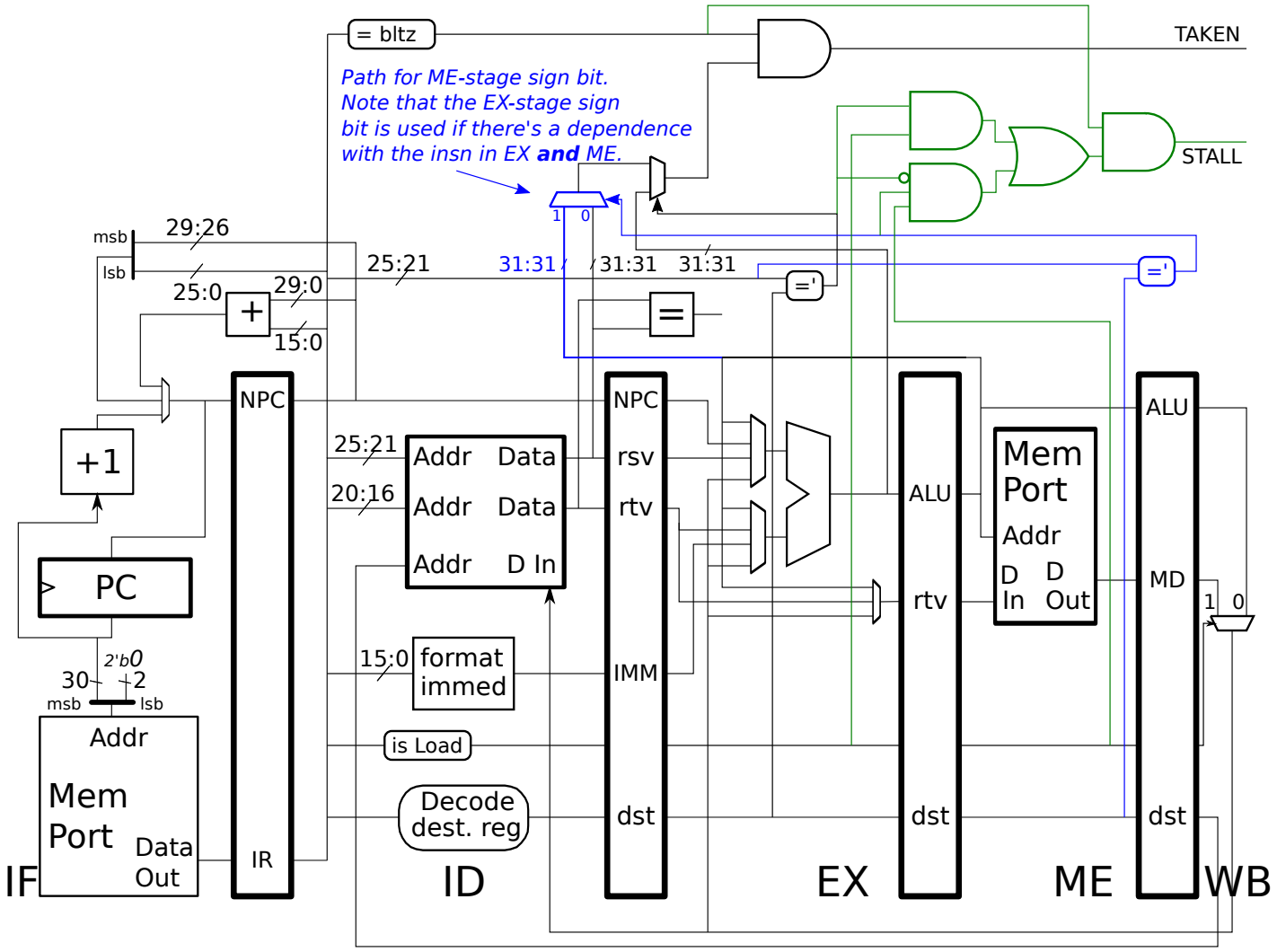
The solution appears below in blue. Note that the multiplexor is positioned such that if there is a dependence both on the instruction in `EX` and `ME`, the sign bit from the `EX` stage will be used. See Example 2 on the previous page.

☑ Logic to generate stall signal for `bltz` dependent on load.

Solution appears in green. Note that the stall is generated if the respective instructions are in the respective stages and if there's a dependence. A common mistake was to also check whether the "sign bit" was 1 (whether the branch is taken). That doesn't make sense because the sign bit isn't really available, if it was we wouldn't need to stall.

☑ Answer part c.

I remembered, but thanks for the reminder.



Problem 3: [12 pts] Answer each question below.

(*a*) Each code fragment below writes register `f30` with the sum `f2 + 4720`.

```
# Plan A
        addi $t0, $0, 4720
        mtc1 $t0, $f17
        cvt.s.w $f16, $f17
        add.s $f30, $f2, $f16


# Plan B
        lui $t0, 0x4593
        ori $t0, $t0, 0x8000
        mtc1 $t0, $f16
        add.s $f30, $f2, $f16
```

☑ What is the difference between `mtc1` and `cvt`?

The `mtc1` instruction moves a value from an integer register to a floating-point (co-processor 1) register. The `cvt` instruction converts a value from one format to another, in the example above from integer to single-precision floating point. So the difference is, `mtc1` moves its operand from an integer to FP register, while `cvt` changes the value of its operand from an integer to FP value.

☑ Why doesn't Plan B need a `cvt`?

Because the register contents is already in a FP format.

(*b*) All MIPS integer instructions have their source register numbers in the `rs` and, if needed, `rt` fields. But the destination register number can be found in either the `rt` or `rd` fields.

☑ How does limiting integer sources to `rs` and `rt` reduce cost and improve performance?

It reduces cost because no mux is needed at the register file `Addr` inputs. It improves performance for the same reason, there is no delay that there would be with a mux, which includes the logic generating the mux's control signal.

☑ Why isn't performance hurt by having the destination in either `rt` or `rd`?

Because the `dst` value is not needed until the end of `ID` (for the pipeline latch) and so the mux and its control logic are not on the critical path.

Problem 4: [28 pts] Answer each question below.

(*a*) The statement below omits an important reason why customers can be kept by companies that manage an ISA and implementation as two different things.

By separating the ISA from the implementation we can keep our customers by offering them a faster implementation when they are ready to buy a new system.

☑ What is the important reason that has been omitted?

Short Answer: . . ., that runs the software **that they already have**.

More details: Software compatibility. The newer, faster computer must be an implementation of the **same** ISA or a superset of it.

(*b*) To use profiling to improve performance a program is compiled twice.

☑ What is done between the first and second compilation?

The program is run using typical input data, the run is called a *training run*. (Sorry about using the word *run* three times in one sentence.)

☑ Why does the program need to be compiled a second time?

So that the compiler can read the results of the training run and use that to make better optimization decisions.

☑ Suppose that taken branches have a penalty. Show how profiling helps.

Suppose that in the first code fragment below the branch is mostly taken, meaning that the ELSEPART is frequently executed. When the compiler learns this by reading the output of the training run it will rearrange code so that the branch is mostly not taken. It will also move the less-frequently executed `IF_PART` out of the way. In the optimized code zero control transfers are needed for the frequent case.

```
 # SOLUTION

 # Before optimization. Either branch or jump always taken.
 # Assume that branch is mostly taken.
 add r1, r2, r3
 beq r4, r5, ELSEPART
 xor r6, r7, r8
IF_PART:
 lw r8, 0(r9)
 ...
 j ENDIF
 add r10, r11, r12
ELSEPART:
 lw r8, 0(r20)
 ...
ENDIF:
 sw r21, 0(r22)

 # After optimization. Now branch is mostly not taken.
 add r1, r2, r3
 bne r4, r5, IFPART
 xor r6, r7, r8
ELSEPART:
 lw r8, 0(r20)
 ...
ENDIF:
 sw r21, 0(r22)
 ...

IF_PART:
 lw r8, 0(r9)
 ...
 j ENDIF
 add r10, r11, r12
```

Problem 4, continued:

(*c*) Consider an instruction such as `add (r1), r2, 4(r3)`. What about it makes it unsuitable for a RISC ISA? Explain why it would be difficult to implement in our pipelined design.

☑ `add (r1), r2, 4(r3)` unsuitable for RISC because:

☑ It would be difficult to implement because:

The instruction is unsuitable for RISC because it both performs arithmetic and accesses memory, a RISC no-no. This makes it hard to implement in a pipelined microarchitecture because the instruction would need to access memory twice, once for the source, `4(r3)` and once to write the result, `(r1)`, which would require extra memory ports, and that's costly, or it would require some way of using the memory port from different stages, which would greatly complicate the design.

(*d*) When we compared the un-optimized and optimized versions of the π program we found that the optimized version had many fewer load and store instructions. Why?

☑ The optimized π program had fewer loads and stores because:

Without optimization, the compiler will emit code to load the value of a variable into a register each time it is used and to write the value to memory each time it is changed. But if a variable is updated and used many times (say, in a loop body) then the value can be loaded just once, into a register, before the loop and stored just once, after the loop. That was the case with the π program where all loads and stores were eliminated from the loop body.

(*e*) A tester preparing a run of the SPECcpu suite is responsible for compiling the benchmarks. Why does that make SPECcpu results interesting to computer engineers?

☑ Tester compilation makes SPECcpu interesting to computer engineers because:

Computer engineers can evaluate the performance of new microarchitectural features, such as bypass paths, by running the SPECcpu benchmarks with a compiler back-end designed to take advantage of the new features. They also can run the benchmarks on new ISAs. Note that the tester is the computer engineer.

Problem 5: [10 pts] Answer the following questions about bypass paths.

(*a*) Consider the two statements below about bypasses in implementations like our five-stage MIPS **running typical programs**.

A: Compiler scheduling makes bypass paths unnecessary.

☑ Explain why the statement above is wrong.

In typical programs the compiler cannot always schedule instructions to avoid stalls because it can't always find enough instructions to put between a dependent pair. The problem is exacerbated without bypass paths because dependent pairs must be farther apart.

B: Bypass paths make compiler scheduling unnecessary.

☑ Explain why the statement above is wrong.

Bypass paths don't exist for every possible dependence, such as a `lw` followed by an `add`.

(*b*) Consider the two above statements (about bypass paths) again as it applies to our MIPS implementation, but this time **running a special set of programs**. We plan to design an implementation for this set of programs. For these programs the two statements are true! *Note: The original exam did not mention the new implementation, and it had an "or both" option below.*

☑ For such programs should we eliminate bypass paths or should we eliminate compiler scheduling?

☑ Explain.

Bypass paths, because that would save money. Note that just because both statements are true does not mean that bypass paths and compiler scheduling can be eliminated at the same time. Statement A implies that compiler scheduling is eliminating stalls without the help of bypass paths. But if scheduling is also eliminated there can be stalls.

Name Solution

Computer Architecture
EE 4720
Final Examination

4 May 2016, 15:00–17:00 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias Leaked

Good Luck!

Problem 1: (20 pts) Appearing below is our two-way superscalar MIPS implementation with a single, unconnected memory port in the `ME` stage. Since there is only one memory port there will have to be a slot-1 `ID`-stage stall whenever `ID` contains two memory instructions, for example:

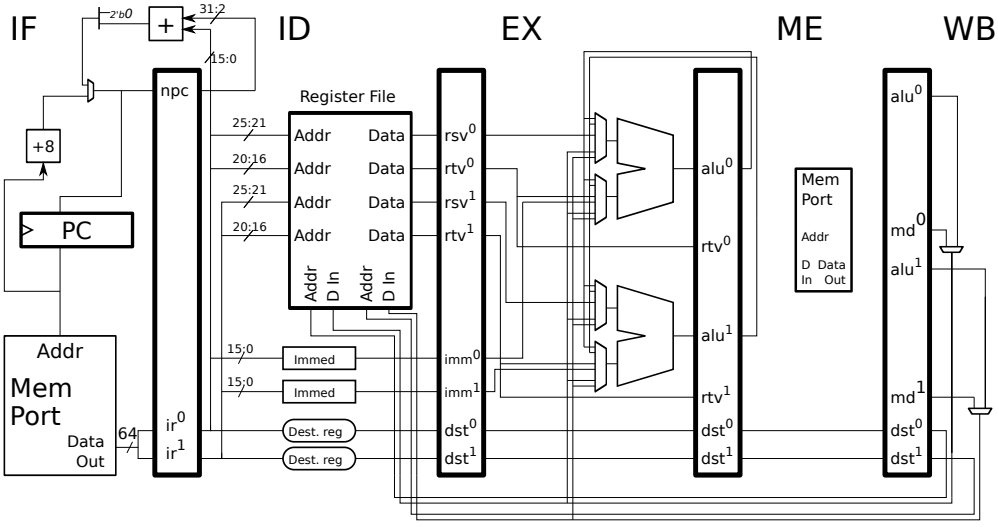
```
# Cycle        0  1  2  3  4  5
lw r1, 0(r2)   IF ID EX ME WB
lw r3, 0(r4)   IF ID -> EX ME WB
```

(*a*) On the next page add datapath to the implementation so that the memory port can be used by a load or store instruction in either slot. Pay attention to the cost of pipeline latches.

(*b*) On the next page add control logic needed for the datapath changes. The control logic should be for any multiplexors that you've added, don't include control logic for the memory port itself.

(*c*) On the next page add control logic to generate a `STALL_ID_1` signal when there are two memory instructions in `ID`, as occurs in cycle 1 to the `lw r3` in the example above.

Use next page for solution.



Use next page for solution.

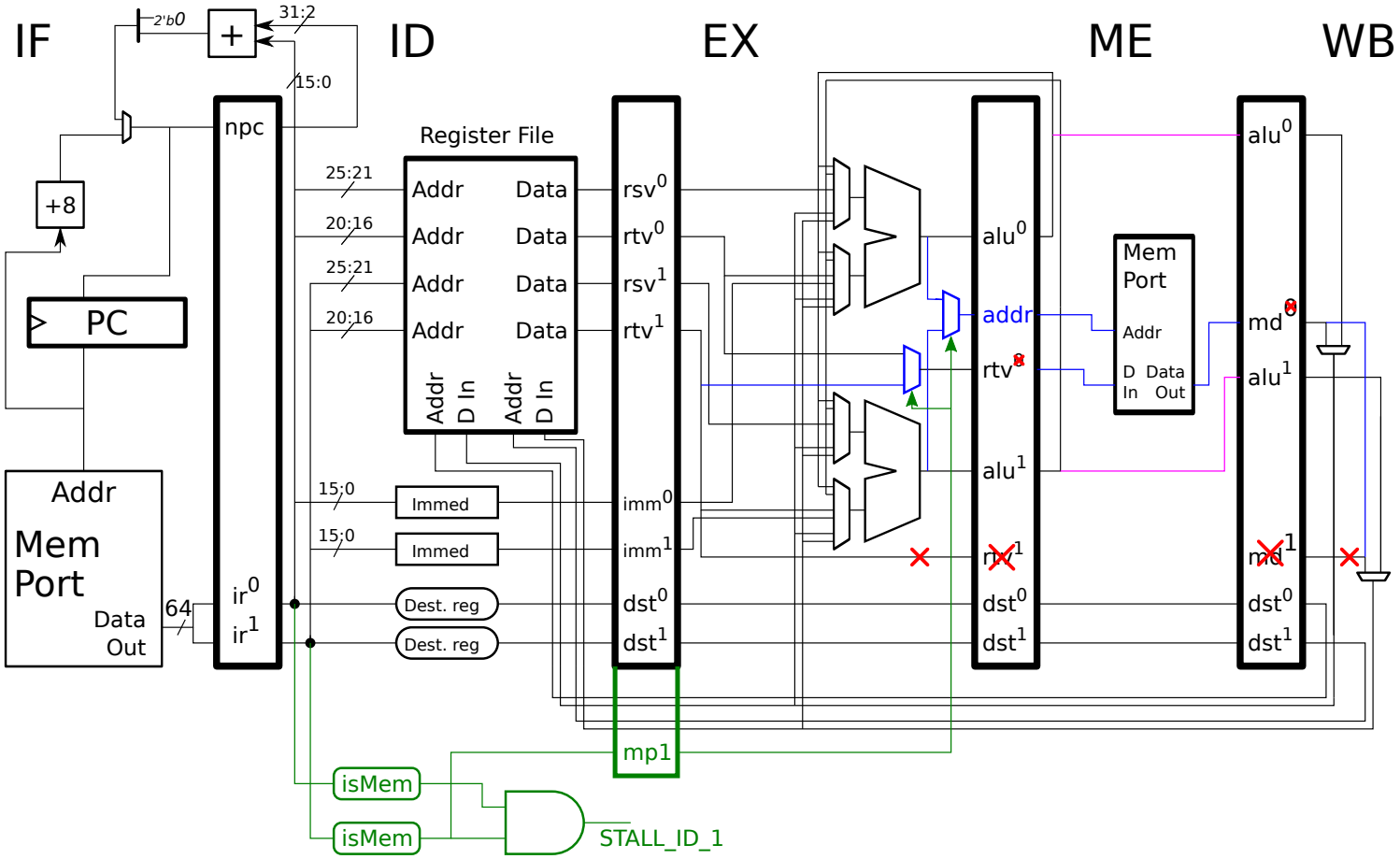
Problem 1, continued:

☑ Datapath so that memory port can be used by a ☑ load or ☑ store in either slot.

☑ Control logic for multiplexors that you've added. ☑ Control logic to generate `STALL_ID_1`.

☑ Pay attention to ☑ **pipeline latch cost** and ☑ the critical path.

Solution appears below. The added datapath is in blue, reconnected existing datapath is in purple, and control logic is in green. In `EX` multiplexors have been added to select the address and store value from the appropriate slot. By putting the store value multiplexor in `EX` we can eliminate the `ME.rtv1` pipeline latch, shown exed out in red. A multiplexor *and* a new pipeline latch were added for the store address in `EX`; putting the address multiplexor in `ME` was out of the question because of critical path and since we don't know whether the memory instruction will be in slot 0 or slot 1, we couldn't just connect the mux to `ME.alu0` (as we did with the store value, to `ME.rtv0`). The memory port output is connected only to `md0`, the connection to slot 1 is made in the `WB` stage, saving another pipeline latch. The non-memory-related `alu0` and `alu1` connections are reconnected, that's shown in purple. Control logic appears in green. The output of isMem is logic 1 if the instruction is any kind of load or store.



(*d*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Only show `ID`-stage changes. Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values.

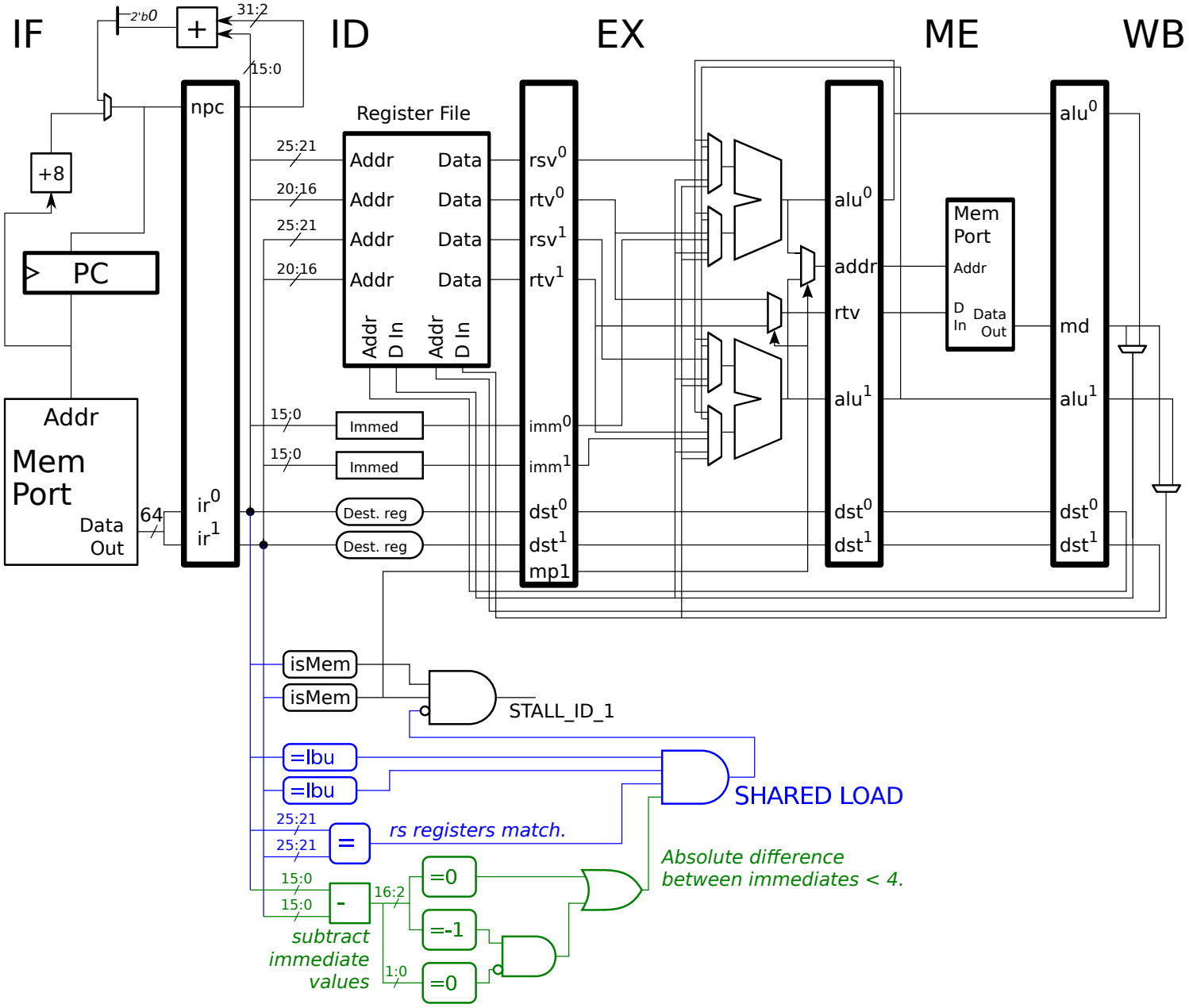
☑ Control logic to detect possible shared loads.

Solution appears below. The logic in blue checks that both instructions are `lbu` and that the `rs` register numbers of both instructions are the same. The logic in green checks that the absolute value of the difference between the immediate values is less than four. This is shown with a subtraction followed by simple gates. Full credit would have been given for a box labeled with something like `abs < 4`.

The logic from Parts a, b, and c are shown in black. The `SHARED_LOAD` signal is used to suppress the two-loads stall.

☑ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.

Due to alignment restrictions both addresses must be identical in bits 31:2. If we assume that bits 1:0 of the base register (`r2` in the example) are 0 then all we need to do is check whether bits 15:2 of the two immediates are identical. But suppose bits 1:0 of the register are 3. Then the two addresses in the example will not be in the same four-byte aligned chunk.


IF
ID
EX
ME
WB
2'b0
+
31:2
15:0
npc
+8
PC
Addr
Mem
Port
Data
Out
64
ir0
ir1
Register File
25:21
20:16
25:21
20:16
Addr
Data
Addr
D In
rsv0
rtv0
rsv1
rtv1
15:0
Immed
15:0
Immed
imm0
imm1
Dest. reg
Dest. reg
dst0
dst1
mp1
alu0
addr
rtv
alu1
dst0
dst1
Mem
Port
Addr
D In
Data
Out
alu0
md
alu1
dst0
dst1
isMem
isMem
STALL_ID_1
=lbu
=lbu
SHARED LOAD
25:21
25:21
=
rs registers match.
15:0
15:0
-
16:2
=0
=-1
=0
1:0
subtract
immediate
values
Absolute difference
between immediates < 4.

The following part was NOT on the final exam as given. It will be assigned as homework in 2017.

(*e*) Notice that the two instructions in the code below load from adjacent addresses. The superscalar implementation from the previous page would stall the `lbu r3`. But that might not be necessary because the memory port retrieves 32 b of data. A properly designed alignment network could provide data for both instructions, in some cases. Let's call such nice situations, *shared loads*.

```
lbu r1, 0(r2)
lbu r3, 1(r2)
```

Here's the plan: In `ID` detect whether a shared load is possible. In `EX` check addresses. In `ME` have memory port perform only one kind of load, 32 b (stores are unaffected). In `WB` have a separate alignment network for each slot.

Show control logic to detect possible shared loads in `ID`. Generate a signal named `SHARED_LOAD` which is `1` if the instructions in `ID` could form a shared load, based on register numbers and immediate values. In `EX` generate a `STALL-EX-1-SL` signal if the shared load cannot be performed. This will stall the pipeline allowing the slot-0 load to proceed normally in the current cycle and the slot-1 load to be executed in the next cycle. Show connections for the alignment units in the appropriate places.

☑ Control logic to detect possible shared loads.

Solution appears below in green and blue. The only difference with the changes in `ID` below and those from Part d is that below the low three bits of the difference between the immediates is put in a pipeline latch, `EX.di`, shown in purple. Signal `di` (delta immediates) is used in `EX` to check whether a shared load can actually be done.

☑ Explain why alignment and critical path make it impossible to know for sure in `ID` that a shared load is possible.

See solution to Part d.

☑ `EX`-stage hardware to generate `STALL-EX-1-SL` if can't do shared load based on addresses.

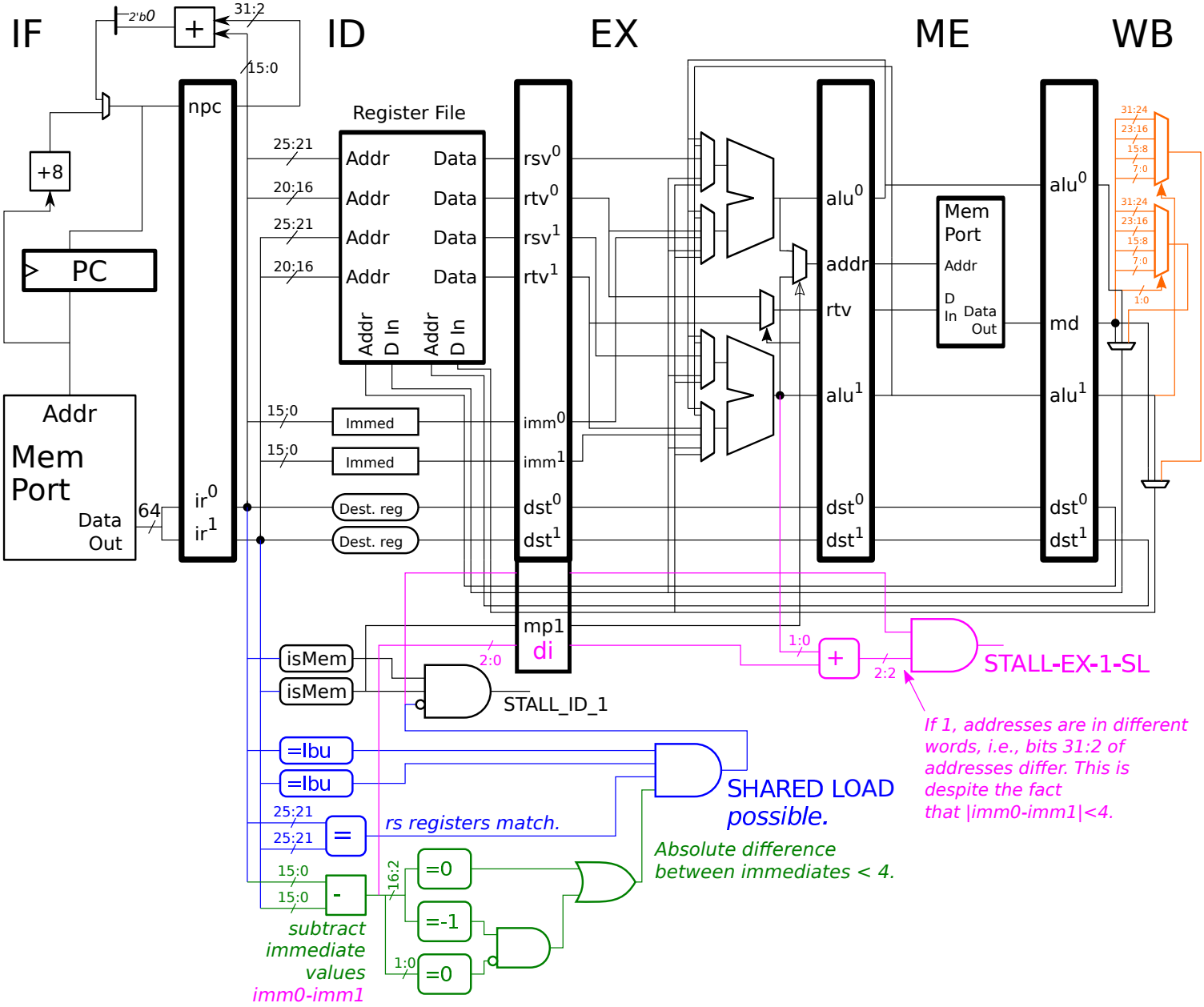
The difference `imm0-imm1` is added to the low two bits of the address of the slot-1 load. If that sum is > 3 or < 0 then bits 31:2 of the two addresses must differ. That logic is shown in purple. The `STALL-EX-1-SL` signal is generated when the sum is out of range and when there was possible shared load when the instruction was in `ID`.

For example, consider the two loads `lbu r1, 0(r2) lbu r3, 1(r2)`. For these loads `imm0-imm1` is -1 or 111_2 as a 2's complement 3-bit number. Suppose `r2` (`rsv`) is `0x1000`. The slot-1 address will be `0x1001`, the two least significant bits are 01_2. Adding these two yields: $111_2 + 01_2 = 000_2$. Bit 2 is zero so the shared load can go ahead. Next, suppose that `r2` is `0x1003`. Now, the address for the slot-1 load is `0x1003+1=0x1004`. The two least significant bits are 00_2. In the sum $111_2 + 00_2 = 111_2$ the bit at position 2 (lsb is position 0) is 1, and so a shared load is not possible.

Note that the stall determination is made by logic that examines just $3 + 2$ bits. Checking whether the high 30 bits of the two ALU outputs were the same would be correct but would stretch the critical path.

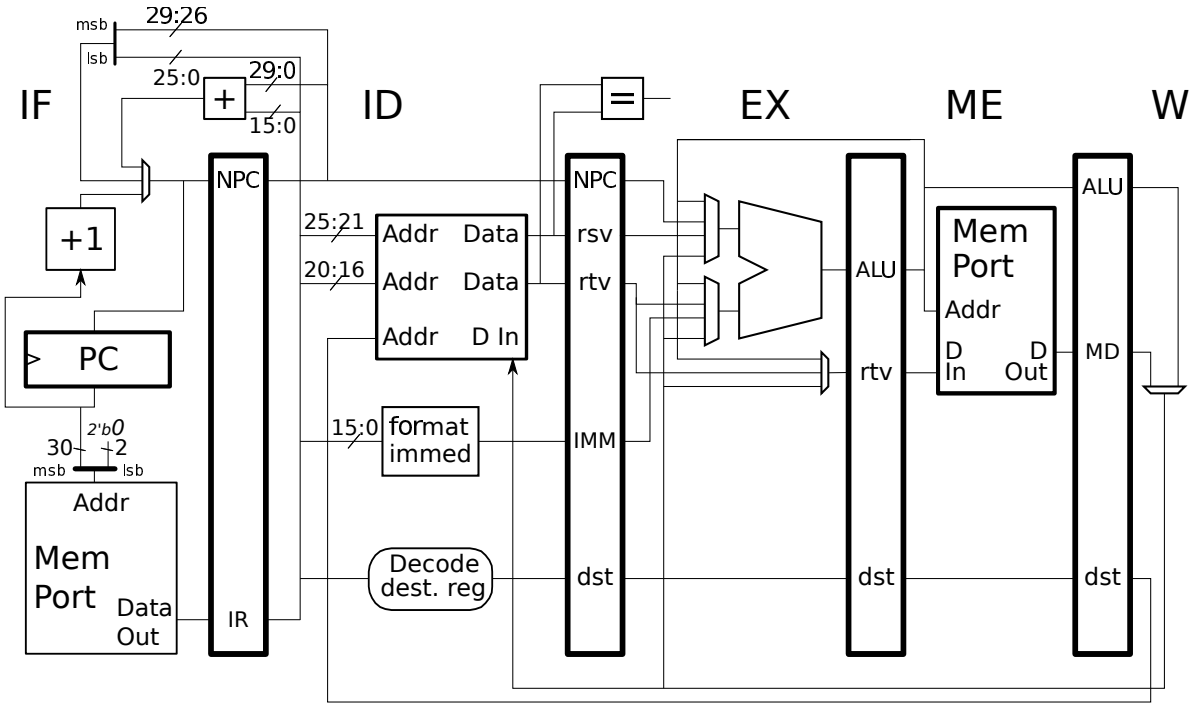
☑ Add remaining `ME`- and `WB`-stage hardware.

The solution appears below in orange. Multiplexors select the appropriate 8 bits from the memory port output based on the two least significant bits of the address.


IF
ID
EX
ME
WB
2'b0
+
31:2
15:0
npc
+8
PC
Addr
Mem
Port
Data
Out
64
ir0
ir1
Register File
Addr
Data
25:21
20:16
Addr D In
rsv0
rtv0
rsv1
rtv1
15:0
Immed
imm0
imm1
Dest. reg
dst0
dst1
mp1
di
2:0
alu0
addr
rtv
alu1
Mem
Port
Addr
D
In
Data
Out
md
31:24
23:16
15:8
7:0
1:0
isMem
STALL_ID_1
=lbu
=
rs registers match.
SHARED LOAD
possible.
Absolute difference
between immediates < 4.
-
subtract
immediate
values
imm0-imm1
16:2
=0
=-1
STALL-EX-1-SL
If 1, addresses are in different
words, i.e., bits 31:2 of
addresses differ. This is
despite the fact
that |imm0-imm1|<4.

Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken. Don't forget to check for dependencies.

(*a*) Show executions.



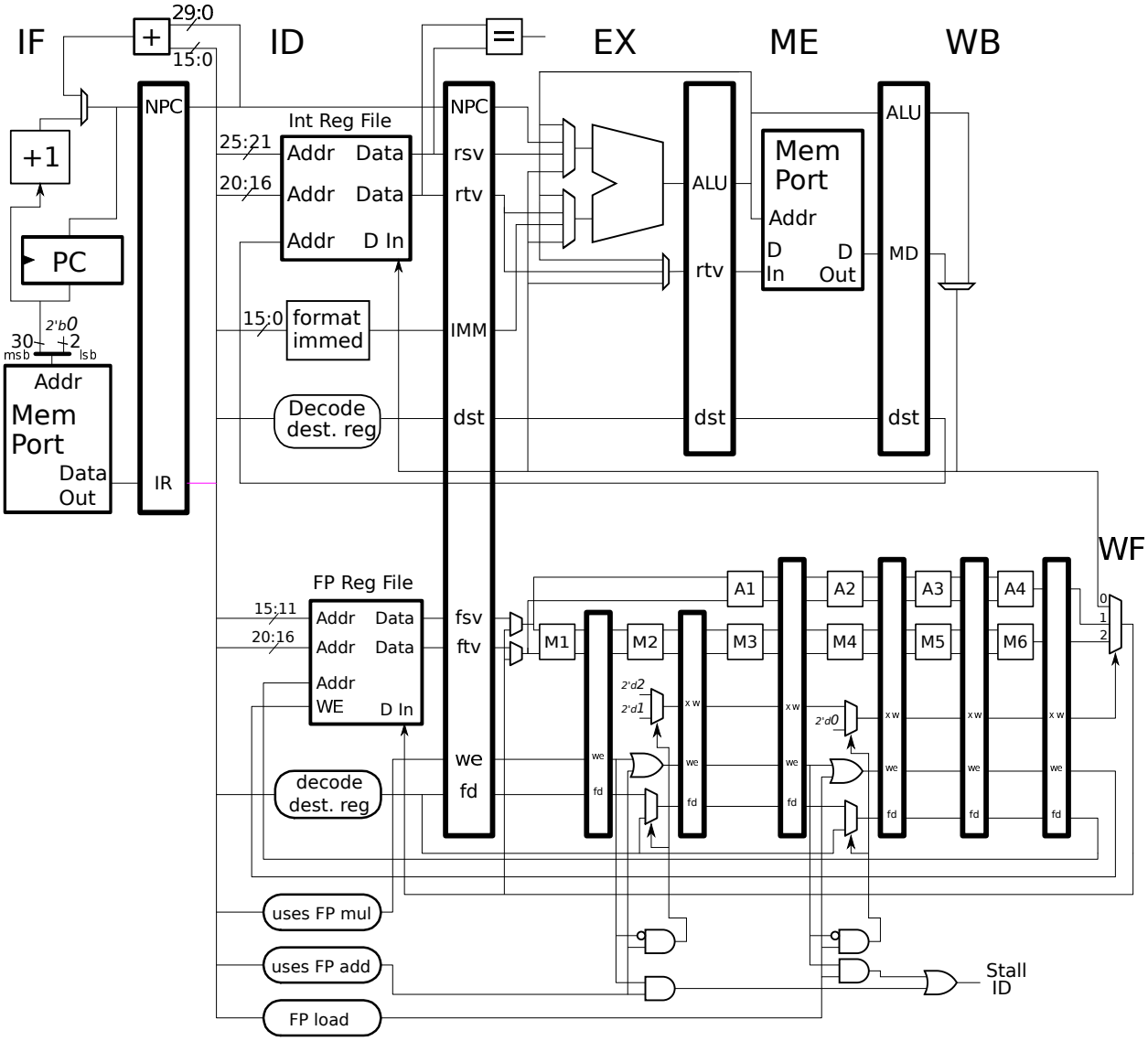
☑ Show execution of this simple code sequence.

```
# SOLUTION
# Cycle          0  1  2  3  4  5
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☑ Show execution of the following code sequence.

```
# SOLUTION
# Cycle             0  1  2  3  4  5  6
beq r1, r1 TARG     IF ID EX ME WB
or  r2, r3, r4         IF ID EX ME WB
sub r5, r6, r7
xor r8, r9, r10
TARG:
lw r10, 0(r11)            IF ID EX ME WB
# Cycle             0  1  2  3  4  5  6
```

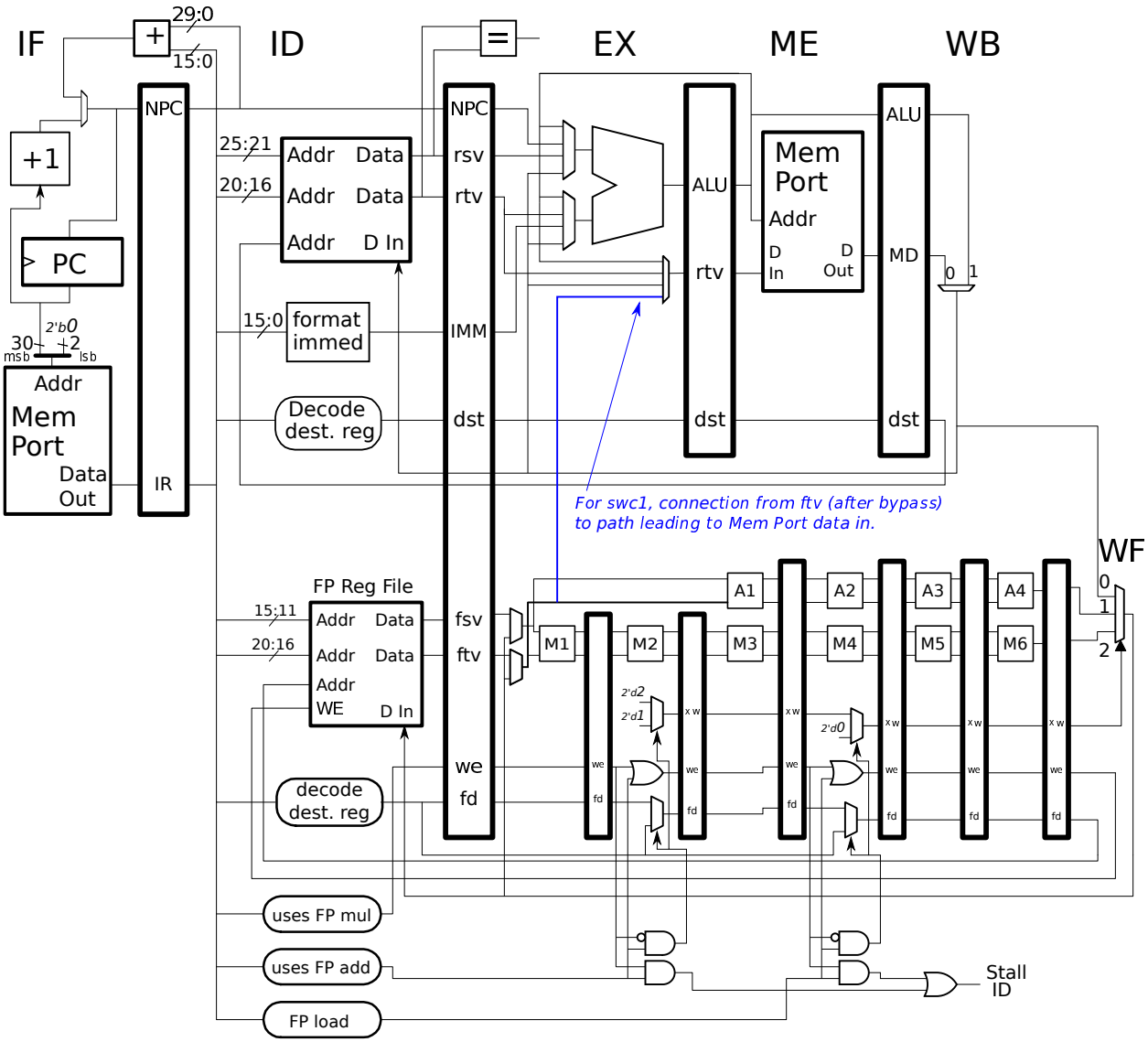
(*b*) Show the execution of the code sequences below on the illustrated MIPS implementation. Don't forget to check for dependencies.



☑ Show execution of the following code sequence.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12
mul.s f5, f6, f7   IF ID M1 M2 M3 M4 M5 M6 WF
add.s f1, f2, f3      IF ID A1 A2 A3 A4 WF
sub.s f4, f8, f5         IF ID ----------> A1 A2 A3 A4 WF
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12
```

(*c*) Add any datapath hardware needed to execute the code sequence below, and show its execution. (Control logic is not needed.) Don't forget to check for dependencies.



☑ Add hardware needed by, and ☑ show execution of, the following code sequence.

☑ Consider both stores and, as always, avoid unnecessary costs.

Added hardware appears above in blue, and the execution appears below. The added hardware consists of a connection from the M1/A1 stage to the mux leading to the memory port data in. For `swc1 f1` the connection is used in cycle 3, carrying data from the IF/ID.ftv pipeline latch. For `swc1 f2` the connection is used in cycle 6, where it is carrying data bypassed from WF.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8
add.s f2, f3, f4   IF ID A1 A2 A3 A4 WF
swc1 f1, 0(r7)        IF ID EX ME WB
swc1 f2, 4(r8)           IF ID ----> EX ME WB
```

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 10-outcome local history, and one system has a global predictor with a 10-outcome global history.

(*a*) Branch behavior is shown below. Two repetitions of a repeating sequence are shown for branches `B1` and `B2`. Branch `B2` generates the following repeating sequence: the eight outcomes `TNTNTNTN` followed by either `NN`, probability .2, or `TT`, probability .8. These last two outcomes are shown below as `r r` and `q q`.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:   N   N   T   T   T   N   T    N   N   T   T   T   N   T ···

B2:    T   N   T   N   T   N   T    N   r   r   T   N   T   N T N T N q q ···
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

Branch `B1` repeats every 7 outcomes. Based on the analysis below there are 4 mispredicts and 3 correct predictions, for an accuracy of $\frac{3}{7} = .4286$.

```
 SOLUTION:  Analysis of counter values and outcomes for branch B1.
                                ! <-------------------------!<--- matching ctr
    0   0   0   1   2   3   2    3    2   1   2   3   3   2   3  <- 2 bit counter
B1:   N   N   T   T   T   N   T     N   N   T   T   T   N   T    ···
                                  x   x   x           x        <- prediction outcome

B2:    T   N   T   N   T   N   T    N   r   r   T   N   T   N T N T N q q ···
```

☑ What is the accuracy of the bimodal predictor on branch B2?

Short Answer: The counter must be either 0 or ≥ 2 at the start of TNTNTNTN yielding 4 correct predictions and ending with a value of 0 (prob .2) or 2 (prob .8). Summing the weighted number of correct predictions of the next two outcomes yields $.2 \times .2 \times 2 + .2 \times .8 \times 0 + .8 \times .2 \times 1 + .8 \times .8 \times 2 = 1.52$ correct predictions. That's $4 + 1.52$ correct predictions for 10 outcomes, or 5.52% prediction accuracy.

Long Answer: This problem is easy to solve by considering a twelve outcome sequence rrTNTNTNTNqq and computing the accuracy based on the last 10 of these outcomes, TNTNTNTNqq. There are four cases to consider: rr=NN,qq=NN, rr=NN,qq=TT, rr=TT,qq=NN, and rr=TT,qq=TT. First suppose rr=NN. The counter value after the second r must be zero (since there was an N before the first r). With a counter value of zero the TNTNTNTN sequence yields four correct predictions and ends with a counter value still of 0. If the qq sequence is NN there will be two more correct predictions, for a total of 6. The accuracy is 60% and the probability of the twelve outcome sequence NN TNTNTNTN NN is $.2 \times .2 = .04$. This analysis is shown in the first row of the table below. The C columns show counter values. In the first column the counter values are shown as a range, from 0 to 2 (3 is omitted because it cannot occur at that point because the preceding outcome is an N). The last column shows the prediction accuracy weighted by the probability. The remaining three rows show the other three cases, and bottom line shows a sum of the weighted probabilities, for an overall prediction accuracy of .552.

C		C		C		C	Acc	Prob	Weighted Acc
0-2	NN	0	TNTNTNTN x x x x	0	NN	0	60%	.2 * .2 = .04	.6 * .04 = .024
0-2	NN	0	TNTNTNTN x x x x	0	TT xx	2	40%	.2 * .8 = .16	.4 * .16 = .064
0-2	TT	2-3	TNTNTNTN x x x x	2	NN x	0	50%	.8 * .2 = .16	.5 * .16 = .08
0-2	TT	2-3	TNTNTNTN x x x x	2	TT	3	60%	.8 * .8 = .64	.6 * .64 = .384
								1.0	= .552

Grading Note: Most looked at cases starting with the TNTNTNTN *part and just assumed without justification a single counter value at the start this sequence.*

☑ What is the accuracy of the local predictor on B2?

The local history size is ten outcomes, and so the local predictor can see where it is in this repeating sequence. The only outcome it can't predict is the first of the two random outcomes (the second always matches the first). Since that's biased taken 80% of the time we can estimate that it will be correctly predicted 80% of the time. The other 9 outcomes will be predicted every time for an overall prediction accuracy of $\frac{9+.8}{10} = .98 = 98\%$.

☑ What is the minimum local history size needed to predict B1 with 100% accuracy?

The minimum local history size is three outcomes. Two outcomes would be too few. For example, if the local history were TT then the next outcome could either be T or N. With three outcomes we'd have TTT which can only be followed by N in the pattern above. The minimum local history size can be found methodically by constructing a tree in which an edge represents an outcome. The root is a zero length history. Edges are labeled with N or T and the possible positions (see below) at which the N or T can occur. The tree is constructed until all leaf nodes are connected to edges labeled with one position. The minimum local history size is the maximum number of edges from the root to a leaf (the tree height).

```
0   1   2   3   4   5   6  Numbering of positions.
N   N   T   T   T   N   T

N     @ 0,1,5   All length-1 histories N.  (Three positions.)
  N   @ 1       All length-2 histories NN  (One position.)  Leaf
  T   @ 2,6     All length-2 histories NT  (Two positions [of the T].)
    N @ 0       All length-3 histories NTN (One position.)  Leaf
    T @ 3
T     @ 2,3,4,6
  N   @ 5,0
    N @ 1
    T @ 6
  T   @ 3,4
    N @ 5
    T @ 4
```

☑ What is the minimum local history size needed to predict B2 with maximum accuracy?

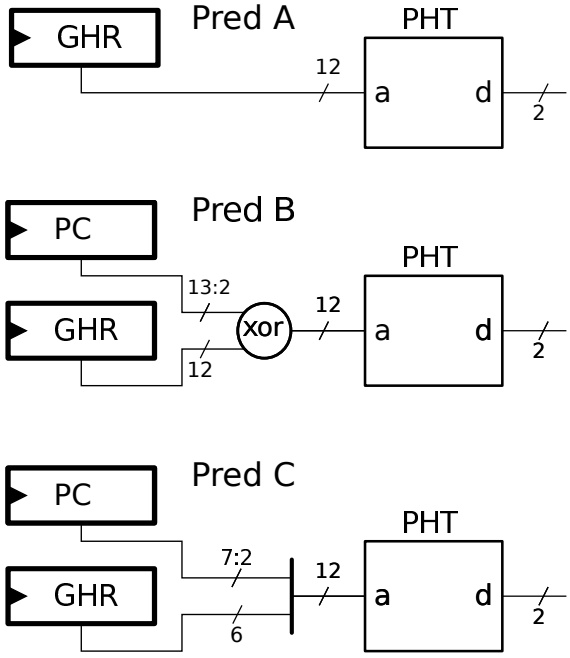
```
   B2:   TNTNTNTNrrTNTNTNTNqq
                     --------    Eight outcomes not enough. Look at q=T
                  --------       and r=N.
```

The local predictor needs to know where it is, in particular to identify the first and second random outcomes. That would require a local history size of 9. See the diagram above.

Problem 3, continued:

(*b*) The diagram below shows four branches. Branches `B1` and `B3` are loop branches, each in a 30-iteration loop. Branches `B2` and `B4` are perfectly biased branches (`B2` always taken, `B4` never taken). Consider their execution on the three variations of the global predictor shown below.

```
B1:   TT ··· TN                  TT ··· TN                  TT ··· TN
B2:           T                          T                          T
B3:               TT ··· TN                  TT ··· TN
B4:                       N                          N
```



☑ Which variation (A , B, or C) will predict `B2` or `B4` poorly?

☑ Why does the variation do poorly on one or both of these simple to predict branches?

In A the PHT is indexed only by the GHR. The GHR value present when predicting B2 will be TTTTTTTTTTTN, the GHR value when predicting B4 will also be TTTTTTTTTTTN, and so they will share a PHT entry. Since they have a different bias the branch will be mispredicted.

☑ Which variation (A, B , or C) is best for `B2` and `B4` and also for branches with **repeating patterns**?

☑ Show an example that one variation predicts accurately that the other can not.

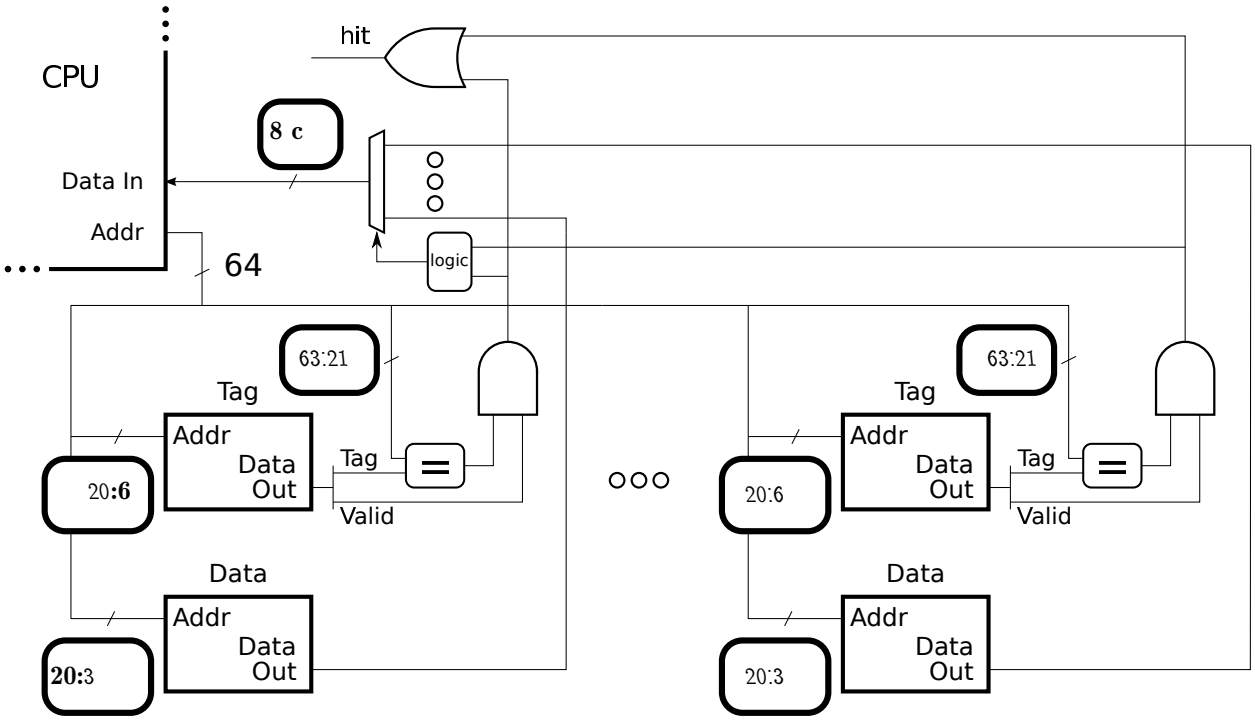
If a branch has repeating patterns then to be predicted accurately the history needs to be large enough to hold the branch's outcomes. Predictor B uses 12 bits of history, whereas Predictor C uses only 6 bits, and so Predictor B is better. Based on the first answer we know that Predictor A will not work well on B2 and B4.

Note: Predictor A is the global predictor, Predictor B is the gshare predictor, and Predictor C is called the *gselect* predictor.

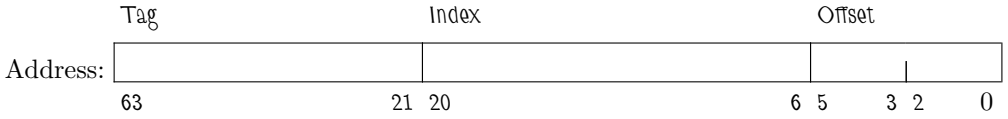
Problem 4: (20 pts) The diagram below is for a three-way set-associative cache. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)



☑ Cache Capacity ☑ Indicate Unit!!:

The cache capacity can be determined from the following pieces of information: The lowest tag bit position is 21, which means that the size of the combined index and offset is 21 bits, and so each data store holds 2^{21} characters. Nothing was said about the character size and so we can assume that it is 8 bits, which everyone alive today calls a byte. We were told that the cache is 3-way set associative, and so there are 3 data stores, and so the cache capacity is $3 \times 2^{21} = 6\,\text{MiB}$.

☑ Memory Needed to Implement ☑ Indicate Unit!!:

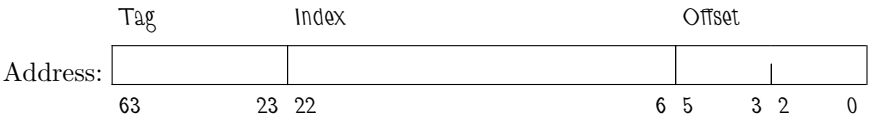
It's the cache capacity, 6 MiB, plus $3 \times 2^{21-6}\,(64 - 21 + 1)$ bits.

☑ Line Size ☑ Indicate Unit!!:

Lower bit position of the address going into the tag store gives the line size, $2^6 = 64$ characters.

☑ Show the bit categorization for the **smallest** direct mapped cache with the same line size and which can hold at least as much data as the cache above.

The cache above is $6\,\mathrm{MiB}$, which is not a power of two. Since a direct-mapped cache must be a power of two we'll make it $8\,\mathrm{MiB} = 2^{23}\,\mathrm{B}$. To achieve that in a direct-mapped cache of the same line size the low tag bit position must be 23, the other bit positions remain the same.



Problem 4, continued: The problems on the following pages are **not** based on the cache from Part a. The code in the problems below run on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
long double sum = 0;
long double *a = 0x2000000; // sizeof(long double) == 16
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^8 = 256$ bytes is given. The size of an array element, which is of type long double, is $16 = 2^4$ B, and so there are $2^8/2^4 = 2^{8-4} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the hit ratio is $\frac{15}{16}$.

Problem 4, continued: The code in the problem below runs on a 16 MiB (2^{24} byte) two-way set-associative cache with a line size of $2^8 = 256$ bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*c*) The code below scans through an array (or arrays) twice, once in the `First Loop` and a second time in the `Second Loop`. Three different variations are shown, Plan A, Plan C, and Plan D. For each Plan, find the largest value of `BSIZE` for which the second loop will have a 100% hit ratio.

```
// Note:  sizeof(double) == 8

// Plan A
struct Some_Struct   { double val, norm_val;   double stuff[14];   };

// Plan C
struct Some_Struct   { double val, norm_val; };
struct Some_Struct_2 { double stuff[14];};

// Plan D
double *vals, *norm_vals;
struct Some_Struct_2 { double stuff[14]; };

// Plan A and Plan C
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second Loop.

// Plan D
  for ( int i = 0;  i < BSIZE;  i++ ) sum += vals[i] + norm_vals[i];   // First Loop.
  for ( int i = 0;  i < BSIZE;  i++ ) norm_vals[i] = vals[i] / sum;   // Second Loop.
```

☑ Largest `BSIZE` for Plan A. ☑ Show work or explain.

Each iteration loads one struct, size is $(1+1+14) \times 8 = 2^7$ B. There are consecutive accesses to array `b` and so the cache used efficiently (except for the fact that only 16 bytes of the 128-byte structure are used). Because the structure size is smaller than a cache line the cache can hold $2^{24}/2^7 = 2^{17}$ elements, which is what `BSIZE` should be.

☑ Largest `BSIZE` for Plan C. ☑ Show work or explain.

Size of struct (the one we need) is now $(1+1) \times 8 = 2^4$ B. So $\boxed{\texttt{BSIZE} \text{ is } 2^{24}/2^4 = 2^{20}}$. `BSIZE` is much larger because the cache holds only what we need, `val` and `norm_val`.

☑ Largest `BSIZE` for Plan D: ☑ Show work or explain.

Each iteration loads one element from `val` and one from `norm_val`. Total size is 2^4 B. Since the cache is two-way set-associative we don't need to worry about conflicts. So `BSIZE` is $2^{24}/2^4 = 2^{20}$, the same as for Plan C.

(*d*) What's wrong with the statement below:

> *In the first iteration of the First Loop under Plan C there will be one miss but with Plan D there will be two misses. Therefore Plan C is better, at least for the First Loop.*

☑ What's wrong with the statement?

What's wrong with the statement is that it is basing a conclusion about the entire First Loop on an analysis of only the first iteration of the First Loop. Yes, with C there is one miss while with D there are two. But with C there will be hits in the next 15 iterations, (line size divided by struct size) while with D there will be hits for the next 31 iterations. Overall, the number of misses is the same.

Problem 5: (20 pts) Answer each question below.

(*a*) Suppose that a new ISA is being designed. Rather than requiring implementations to include control logic to detect dependencies the ISA will require that dependent instructions be separated by at least six instructions. As a result, less hardware will be used in the first implementation.

☑ Explain why this is considered the wrong approach for most ISAs.

Short Answer: Because ISA features should be tied as little as possible to implementation features. The separation requirement in the new ISA is based on a particular implementation, one with five or six stages and that lacks dependency checking.

Long Answer: An ISA should be designed to enable good implementations over a range of cost and performance goals. It is reasonable to assume that low hardware cost was a goal of the first implementation and, as stated in the problem, the ISA's separation requirement helped achieve that goal. So the separation requirement would be a good idea **if there was only going to be one implementation or if cost constraints and technology wouldn't change**. However, ISAs are usually designed for a wide product line and a long lifetime. The separation requirement would become a burden if in the future a higher-performance, higher-cost implementation was needed.

☑ What is the disadvantage of imposing this separation requirement?

Short Answer: It would make a higher-performance implementation that included bypass paths pointless since the compiler would be forced to separate dependent instructions, possibly by inserting `nop`s, and so the bypass paths would never be used.

Long Answer: Call the ISA with the separation requirement, ISA S (strict), and one that lacked the separation requirement but was otherwise identical, ISA N (normal). Consider a low-cost implementation of each ISA. Neither implementation would include bypass paths but the implementation of ISA N would need to check for dependencies and stall if any were found, making the cost slightly higher than that of the implementation if ISA S. Now consider a code fragment compiled for both ISAs in which the version for ISA S requires `nop`s to meet the separation requirement. (The version for ISA N has fewer instructions.) Now consider their execution on their respective implementations. The execution times should be the same because for each `nop` in the ISA S implementation there will be a stall in the ISA N implementation.

So the performance of the low-cost implementations of the ISAs are about the same, but the cost of the ISA N version is slightly higher. (Note that the hardware for checking dependencies is of relatively low cost since it operates on register numbers, 5 bits each in many ISAs. That is in contrast to the cost of the hardware for bypasses, which are 32 or 64 bits wide and include multiplexors.)

Next, consider high-performance implementations. For ISA N we can add bypass paths, as we've done in class. As a result, some instructions that would stall in the low-cost implementation would not stall in the high-performance implementation, and so code would run faster. In contrast, code for ISA S would still have `nop` instructions since the ISA itself imposes the instruction separation requirement. That would make it much harder to design higher performance implementations.

Therefore, the disadvantage of the instruction separation requirement is that it leads to much higher-cost high-performance implementations with only a small cost benefit for the low-cost implementation.

Grading Note: Several students incorrectly answered that a disadvantage of the separation requirement is that it would be tedious and error prone for hand (human) coding, and that it would require that the compiler schedule instructions rather than having the hardware check for dependencies and stall when necessary.

It is 100% true that hand-coding assembly language with such a requirement would be tedious, especially considering branch targets. However, an ISA is designed to facilitate efficient implementations, not to improve assembly language programmers' productivity. (An assembler can still be helpful by pointing out likely separation issues.) Also, if something can be done equally well in the compiler and in hardware, it should be done in the compiler because the cost of compiling is borne beforehand, by the developer. In contrast, the higher cost of the hardware is payed by every customer and the energy of execution is expended each time the program is run.

(*b*) In the execution below the `add.s` instruction encounters an arithmetic overflow when in the `A4` stage and as a result raises an exception in cycle 6.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
add.s  f1, f2, f3  IF ID A1 A2 A3 A* WFx
lwc1 f3, 0(r2)        IF ID EX ME WB
lw r4, 0(r5)             IF ID EX ME WBx
add r6, r7, r8              IF ID EX MEx

HANDLER:
sw                                     IF ID ..
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```

☑ Explain why this exception can't be precise.

Because the `lwc1` wasn't squashed and it wrote a register. To be precise nothing after the faulting instruction (`add.s`) can modify registers or memory.

☑ Describe something the handler could do if the exception were precise.

If the exception were precise then after the handler was finished it could return to the `add.s` giving it a second chance to execute. It can't return to the `add.s` in the imprecise execution shown above because the value of `f3` was overwritten by the `lwc1`.

(*c*) Chip A has 60 2-way superscalar cores and Chip B has 20 4-way superscalar cores. Both chips run at 3 GHz, and so Chip A has a higher peak instruction throughput. The cost and power consumption of the two chips are identical. Why might someone prefer Chip B over Chip A?

☑ Chip B preferred, despite lower peak throughput, because:

Because their workload consists of one single-threaded program. (Or maybe a program that has no more than 20 threads.) It will run faster on Chip B.

(*d*) The SPECcpu suite is used to test new chips that might have cost hundreds of millions of dollars to develop, and test results are closely watched.

☑ Why might we trust the SPECcpu selection of benchmarks and rules for building and running the benchmarks?

Because major chip makers are members of SPEC. Company X would not be a member of SPEC if Company Y twisted the benchmark selection and rules in favor of Company Y's products.

☑ Why might we trust the SPECcpu scores that Company X publishes?

Because any published score is part of a *disclosure* that includes a config file. With the config file anyone can run the test in exactly the same way as Company X does. Anything used in the test must be publicly available.

51 Spring 2015 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 20 March 2015, 9:30–10:20 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(10 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias The Implementation Game

Good Luck!

Problem 1: [25 pts] Appearing on the next page is the *HW 3 Implementation*, one of the solutions to Homework 3, Problem 2, in which the `bgt` instruction resolves in `EX`. In the HW 3 Implementation there is a 1-cycle branch penalty when `bgt` is taken. (The branch penalty is the number of squashed instructions.) Suppose that based on benchmark analyses we find that `bgt` is mostly taken. For that we would like a New Implementation [tm] in which there is no penalty when `bgt` is taken, and a 1-cycle penalty when it's not taken. The PEDs below show execution examples for the HW3 and New implementations.

```
 # Cycle          0  1  2  3  4  5  6  7   HW 3 Impl, bgt taken, 1-cyc penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF IDx
TARG:
 and r9, r10, r11           IF ID EX ME WB
```

```
 # Cycle          0  1  2  3  4  5  6  7   HW 3 Impl, bgt not taken, no penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF ID EX ME WB
TARG:
 and r9, r10, r11
```

```
 # Cycle          0  1  2  3  4  5  6  7   New Impl, bgt taken, no penalty
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8
TARG:
 and r9, r10, r11        IF ID EX ME WB
```

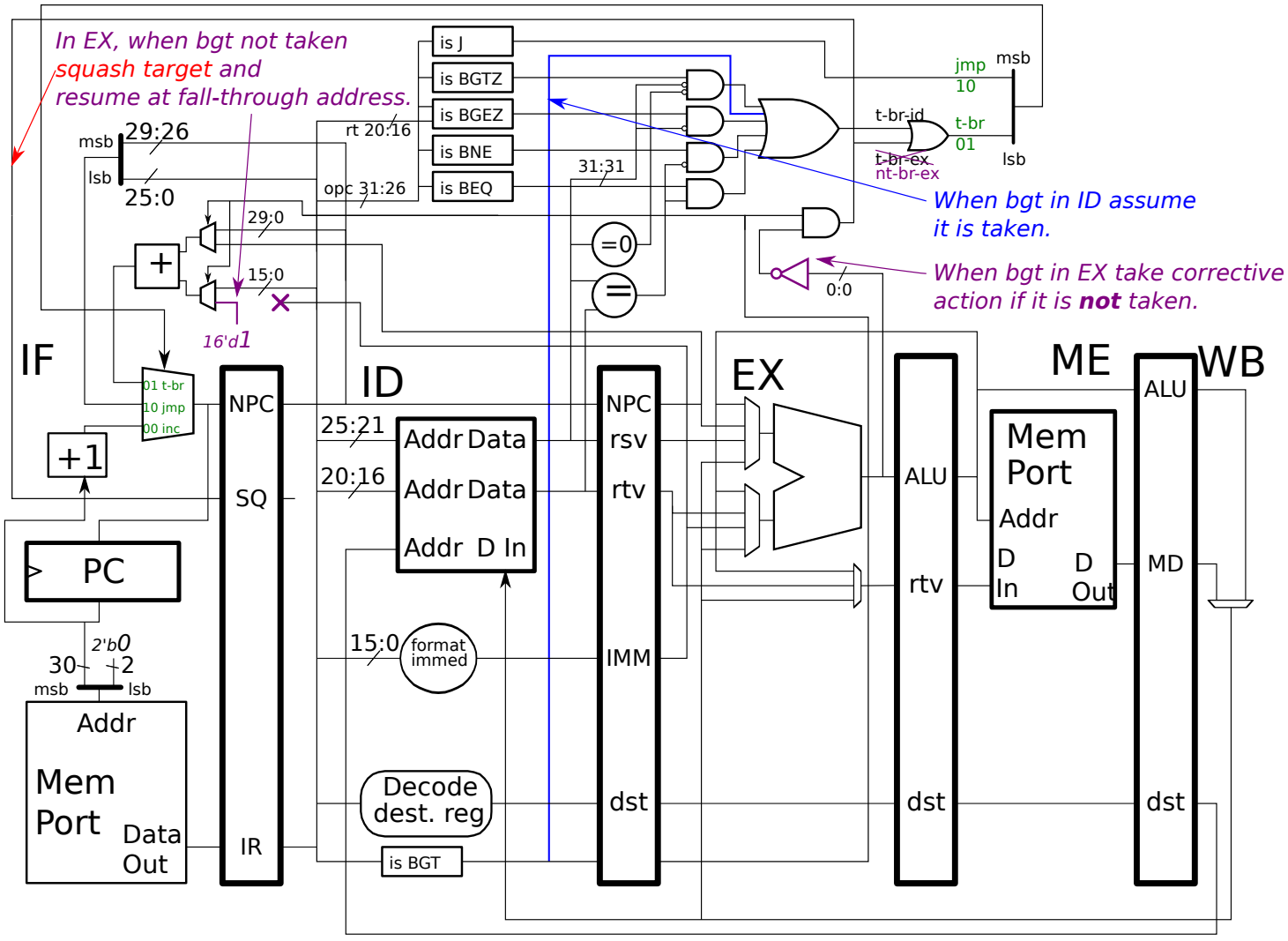
```
 # Cycle          0  1  2  3  4  5  6  7   New Impl, bgt not taken, 1-cyc penalty.
 bgt r1, r2  TARG  IF ID EX ME WB
 xor r3, r4, r5       IF ID EX ME WB
 or r6, r7, r8           IF ID EX ME WB
TARG:
 and r9, r10, r11        IF IDx
 # Cycle          0  1  2  3  4  5  6  7
```

Problem 1, continued: Convert the HW 3 Implementation below into the New Implementation. *Hint: Calm down! A correct solution only requires three minor changes, one of those changes is substituting a mux input with a constant.*

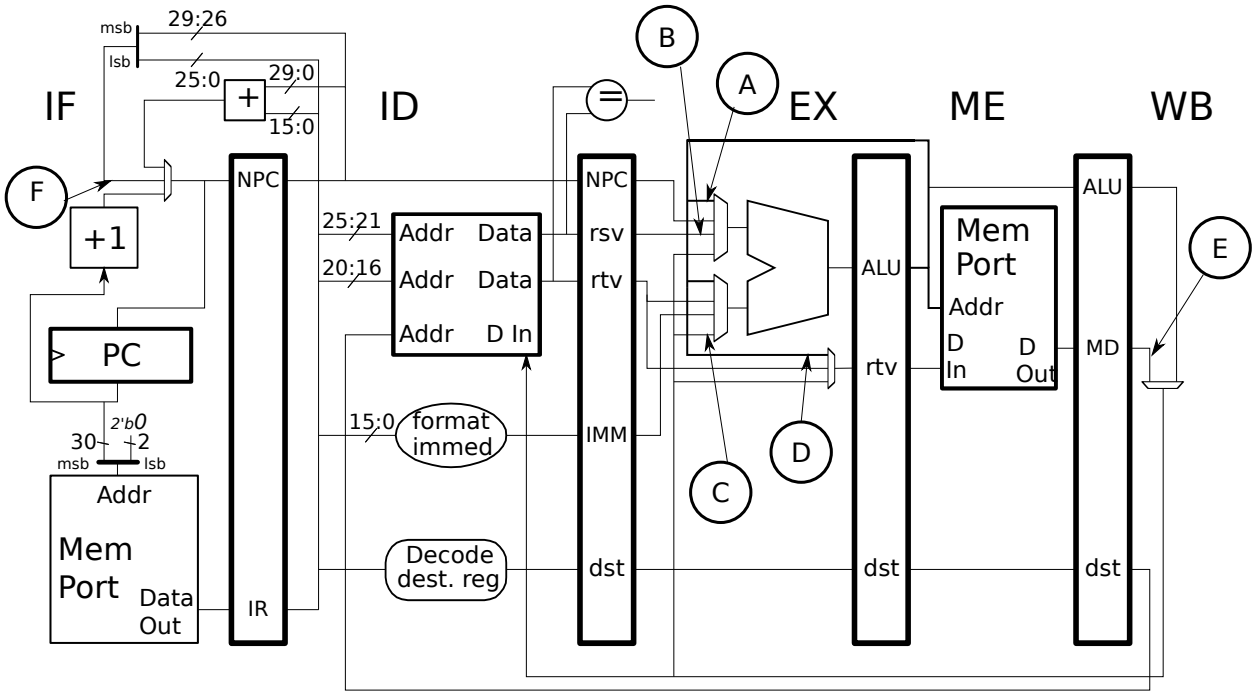
☑ `ID` changes: `bgt` acts like it's always taken.

☑ `EX` changes: if `bgt` resolves not taken, fetch insn after delay slot insn.

Solution appears below.



Problem 2: [25 pts] In the implementation below several multiplexor inputs are labeled. For each labeled input write a program that uses it. A sample solution is provided for **A**.



☑ Write a code fragment for mux input **A**. ☑ Mux input used in cycle __3__, ☑ for register __r1__.

```
# SAMPLE SOLUTION -- Mux input A.
# Cycle          0  1  2  3  4  5
add r1, r2, r3   IF ID EX ME WB
sub r4, r1, r5      IF ID EX ME WB
```

☑ Write a code fragment for mux input **B**. ☑ Mux input used in cycle __2__, ☑ for register __r2__.

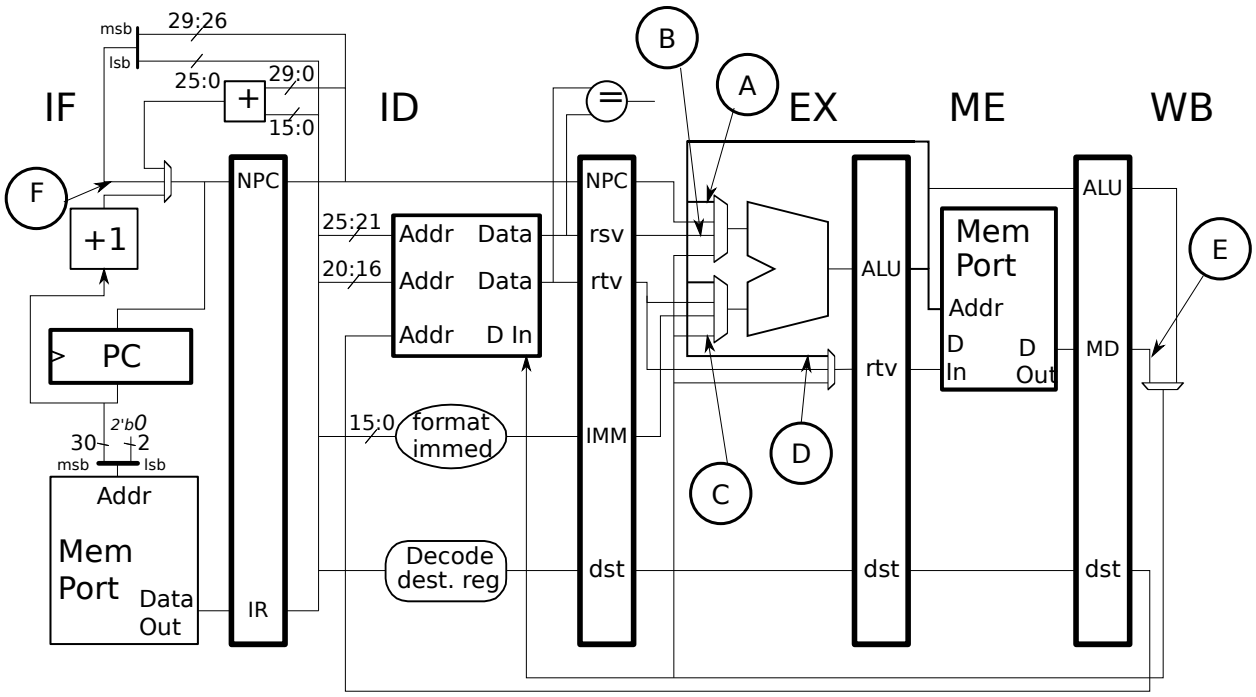
```
# SOLUTION
# Cycle          0  1  2  3  4
add r1, r2, r3   IF ID EX ME WB
```

☑ Write a code fragment for mux input **C**. ☑ Mux input used in cycle __4__, ☑ for register __r1__.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EX ME WB
sub r4, r5, r6      IF ID EX ME WB
or  r7, r8, r1         IF ID EX ME WB
```

Note: To be correct, the `r1` register must be the second source operand of the `or` instruction.

Problem 2, continued:



☑ Write a code fragment for mux input **D**. ☑ Mux input used in cycle __3__, ☑ for register __r1__.

```
# SOLUTION
# Cycle          0  1  2  3  4  5
add r1, r2, r3   IF ID EX ME WB
sw r1, 0(r4)        IF ID EX ME WB
```

☑ Write a code fragment for mux input **E**. ☑ Mux input used in cycle __4__, ☑ for register __r1__.

```
# SOLUTION
# Cycle          0  1  2  3  4
lw r1, 0(r2)     IF ID EX ME WB
```

☑ Write a code fragment for mux input **F**. ☑ Mux input used in cycle __1__.

```
# SOLUTION
# Cycle          0  1  2  3  4
j FORJOY         IF ID EX ME WB
```

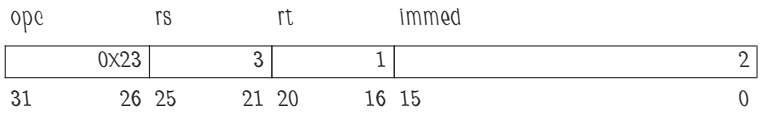
Problem 3: [15 pts] Answer each question below.

(*a*) Show the encoding of the MIPS instructions below. The opcode for `beq` is `0x4` and the opcode for `lw` is `0x23`. *Hint: For the fields' bit positions see the implementation diagrams in previous problems.*

```
TARG:
 lw r1, 2(r3)
 beq r4, r5, TARG
```

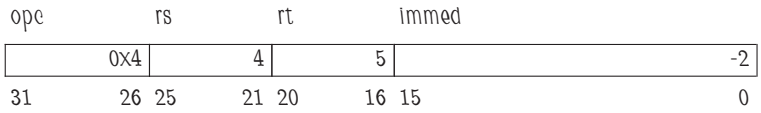
☑ Encoding of `lw r1, 2(r3)`:

Solution:



☑ Encoding of the `beq` above ☑ with the correct immediate field value.

The solution appears below. The immediate field contains the number of instructions to skip, starting at the delay slot instruction. To jump up to the `lw` we need to skip -2 instructions.



(*b*) Many MIPS Format-R instructions, such as `add` and `sub`, have an opcode value of 0. Explain why they don't have their own opcode field values, such as `0x11` for `add` and `0x12` for `sub`.

☑ R-format instructions have opcode 0 because ...

... they have enough space for an extension of the opcode field, called the `func` field. Assigning opcode 0 to type R instructions leaves lots of opcodes for format-I and -J instructions.

☑ If R-format instructions each had their own opcodes that would be a problem because ...

... there would not be enough opcodes available for format-I and -J instructions. These instructions have immediate fields, the larger the immediate the better, and so it's better to omit any kind of opcode extension field.

Problem 4: [10 pts] Answer each question below.

(*a*) Describe what the dead-code elimination optimization is and provide an example.

☑ Description of dead-code elimination.

In dead code elimination statements (or instructions) are eliminated if they write variables (or registers) that are never used.

☑ Example.

In the example below line `L1` is eliminated because the value of `x` that it writes is never used, we know that because it is overwritten by `L2`.

```
// SOLUTION EXAMPLE
L1: x = a + b;  // This line is eliminated.
L2: x = c + d;
L3: my_procedure(x);
```

Grading Note: Many examples were much longer than they had to be, for example, showing the π program with π not actually begin printed.

(*b*) For instruction scheduling optimizations is it necessary or just helpful for the the compiler to know the implementation?

☑ Implementation: Necessary, important, not needed. ☑ Circle one ☑ and explain.

Instruction scheduling is rearranging instructions in order to avoid stalls. The amount by which to separate two true-dependent instructions depends on the bypass paths that are available and (which was not quite covered before the test) the latency of the functional unit. A compiler can make assumptions about which instructions have longer latency and which bypass paths are available. If the assumptions are reasonable then scheduling based on them will be better than not scheduling at all.

Problem 5: [10 pts] Answer each question below.

(*a*) A CISC ISA might have an instruction like `add (r1), r2, 8(r3)`, in which the source and destination come from memory.

☑ Explain why such an instruction is not suitable for a RISC ISA, ☑ refer to RISC goals in your answer.

A goal of a RISC ISA is to enable simple, low-cost pipelined implementations. The instruction above accesses memory twice, once to fetch an operand and once to write the result. It also does arithmetic twice, once to compute the second source operand address, and to add the two operands together. A pipelined implementation would need to adders and worse two data memory ports to execute that instruction, which is too costly for RISC goals. Alternatively an implementation can use a single data memory twice, but such an implementation would not be pipelined. Either way it's not RISC.

(*b*) Describe a feature and goal of VLIW ISAs.

☑ VLIW feature:

Instructions managed in groups called *bundles*. Dependency info placed in *template* fields.

☑ VLIW goal:

Low-cost implementations that can execute more than one instruction per cycle.

Problem 6: [15 pts] Answer each question below.

(*a*) SPECcpu benchmark results have two levels of tuning, *peak* and *base*. In one of these tuning levels the same optimization flags must be used for all benchmarks of the same language.

☑ Which tuning level is this for?

Base.

☑ What is the purpose for requiring the same flags?

The base tuning level is supposed to reflect the amount of tuning performed by a conscientious and skilled programmer for whom performance is one of many things to do. In other words, the programmer also needs to add features and fix bugs, and so cannot waste time getting an extra 0.001% performance. It is assumed that such a programmer will come up with a good set of flags and then use that same set consistently on the different programs he or she compiles.

Note: In class we discussed what might be a better base rule: just have one optimization flag, such as `-O3` or `-fast`.

(*b*) Explain how the following corruptions of SPECcpu would be (we hope) prevented.

☑ *The suite excludes benchmarks that perform poorly on Evil Company's products.* This won't happen because ...

... Evil Company's competitors are also members of SPEC, and they won't sit idly while Evil Company stacks the deck against them. Either they will veto Evil Company's efforts, or else they will quit SPEC and make sure that their names are removed from the list of SPEC members.

Grading Note: Many students misunderstood the question. The question is asking about the design of the SPECcpu suite, it is not asking about a tester running the suite.

☑ *The SPECcpu results for Evil Company's System X are just made-up numbers.* This won't happen because ...

... the SPEC results disclosure must include a config file that would allow anyone to duplicate the test. If the numbers are made up they will quickly be caught and few will believe anything they say in the future.

Name Solution

Computer Architecture
EE 4720
Final Examination

4 May 2015, 10:00–12:00 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias I feel like this is a good alias.

Good Luck!

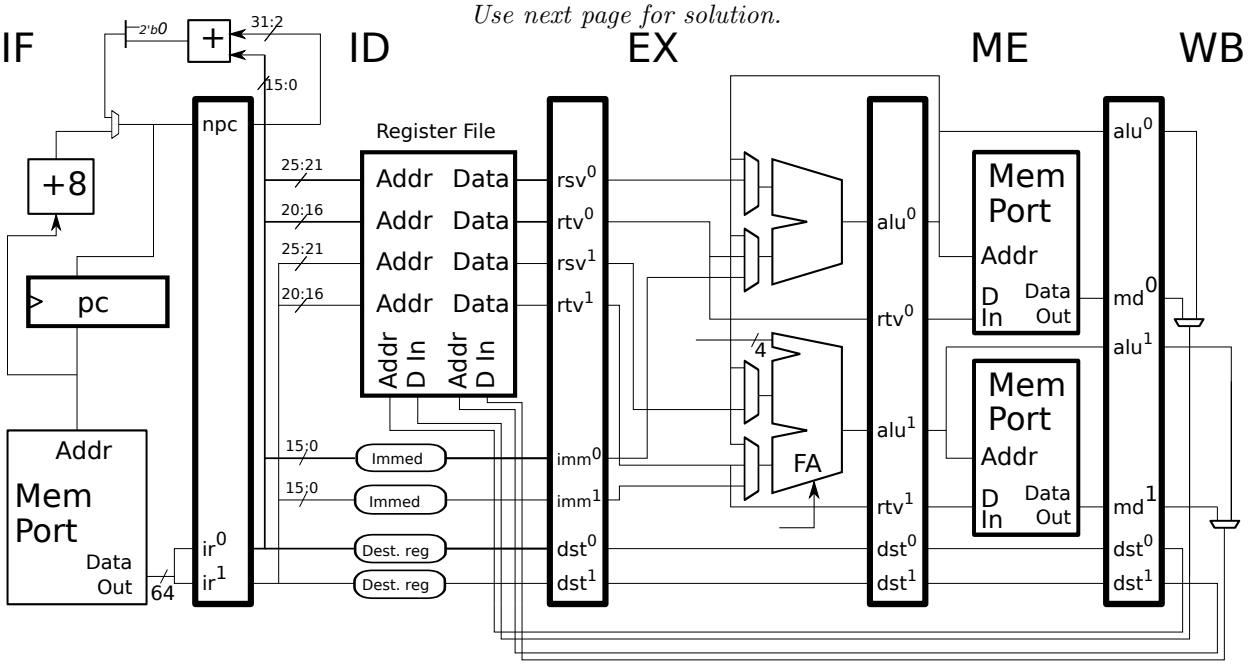
Problem 1: (20 pts) The instruction sequence below performs the calculation `r2+3+r5` and requires two MIPS instructions to do so. If these two instructions were in the same fetch group in a two-way superscalar processor then the second one would stall, this is shown in the execution below.

```
addi r1, r2, 3   IF ID EX ME WB
add  r4, r1, r5  IF ID -> EX ME WB
```

An *FA-ALU* has two regular inputs, and a third input reserved for small quantities, here for values < 16, and it can produce a result in the same time as an ordinary two-input ALU. An FA-ALU can be used to execute instruction pairs like the one above without a stall, call such an execution a *fused add*.

Illustrated below is a 2-way superscalar implementation with an FA-ALU. Most bypass paths have been removed to provide space for the solution to this problem. In addition to a 4-bit input for the third operand, the FA-ALU has a control input, `FA`, which should be set to `1` when the third input is to be used.

Use next page for solution.



Use next page for solution.

(*a*) Add datapath connections to this implementation so that the FA-ALU can be used for fused adds. *Hint: The datapath changes are not just for that new FA-ALU input.*

☑ Add datapath so that the correct operands are delivered to the FA-ALU for fused adds.

☑ Non-fusable sequences must still execute correctly. (Don't break existing functionality.)

☑ Make reasonable cost and clock frequency tradeoffs.

(*b*) Add control logic.

☑ Design logic to generate the `FA` signal for the FA-ALU and deliver it in the correct cycle.

☑ Consider ☑ instruction type, ☑ dependencies, and ☑ operand size.

☑ Add control signals for the datapath added in the previous part.

☑ Make reasonable cost and clock frequency tradeoffs.

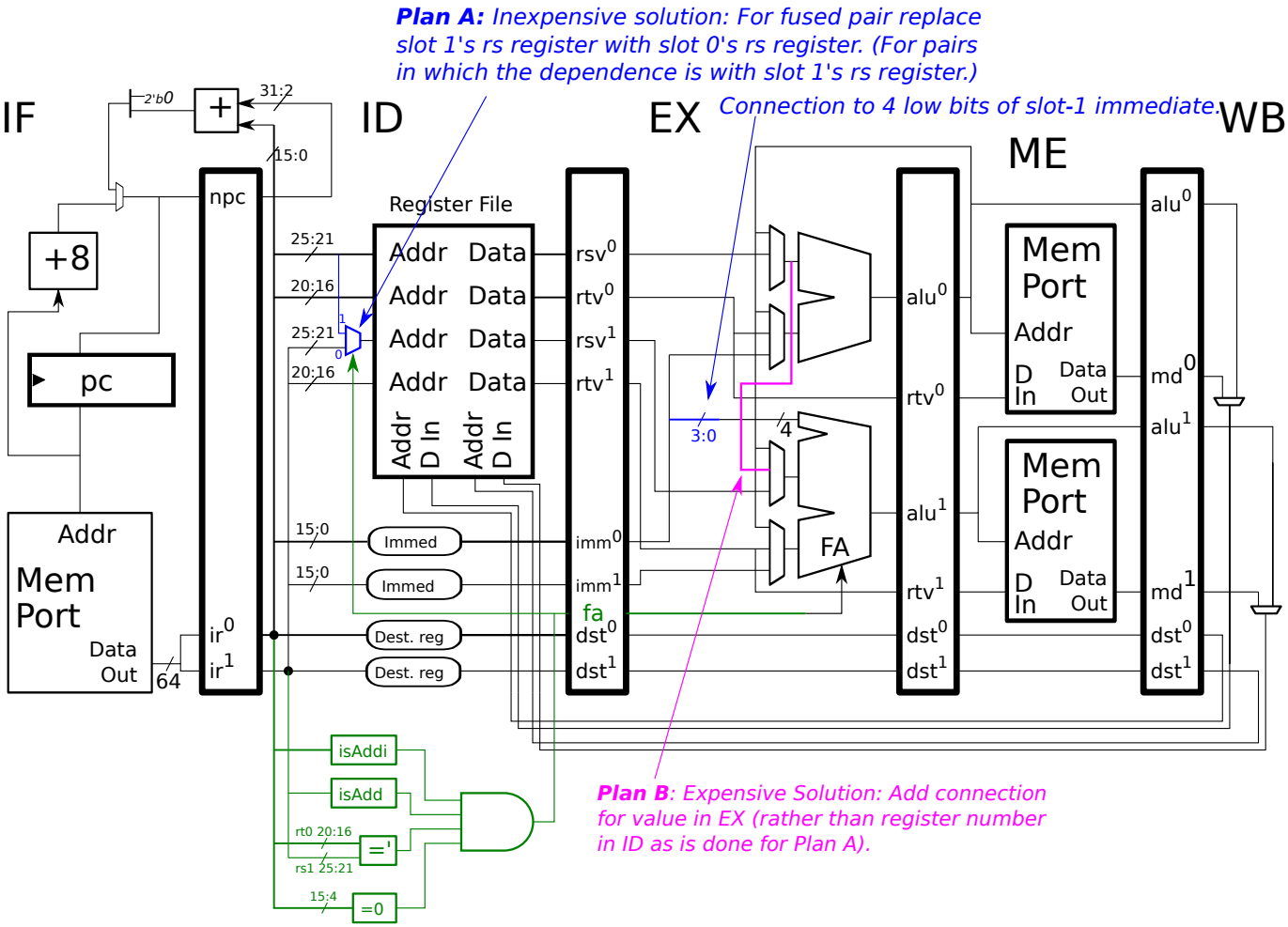
Problem 1, continued: You can use logic blocks such as isADD and isADDI in your solution.

```
addi r1, r2, 3    # Sample fusable sequence.
add  r4, r1, r5
```

Solution appears below. The datapath is in blue and control logic is in green, and an alternative solution appears in purple.

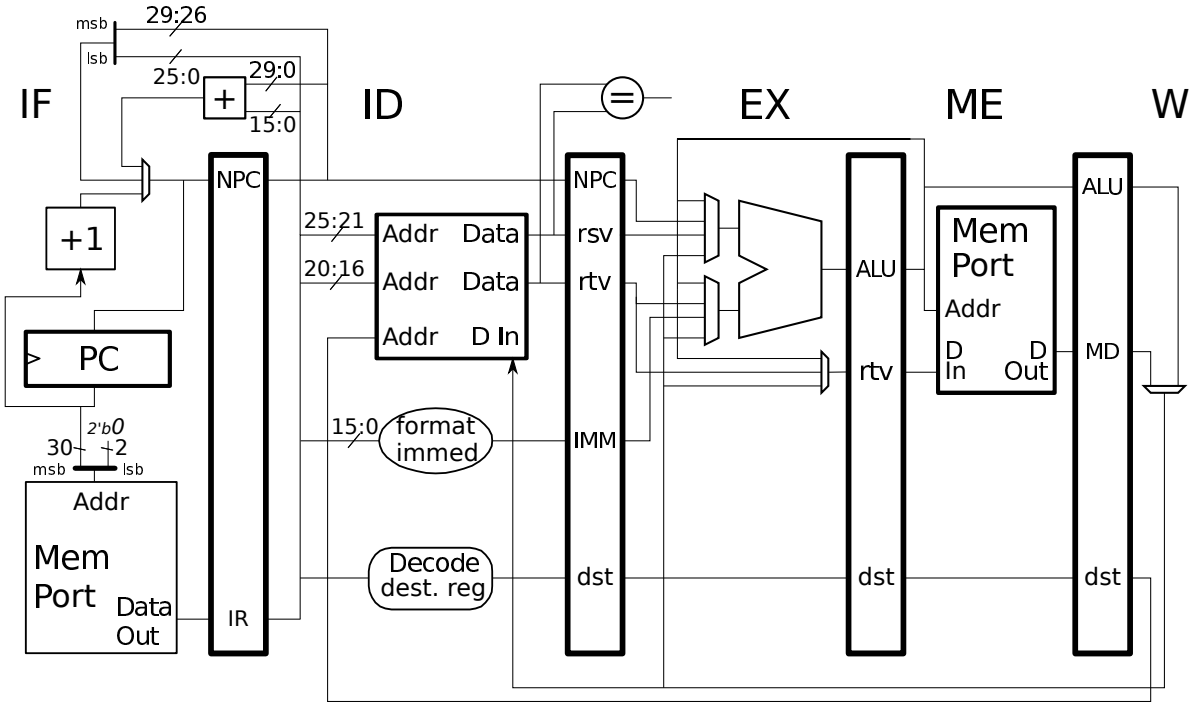
The datapath needs to be changed so that the immediate and `rs` register value from the slot 0 (`addi`) instruction are fed to the slot 1 FA-ALU. Connecting the immediate is straightforward. Two solutions are shown for the `rs` register value. In the Plan A solution the slot 1 `rs` register number is switched to the slot 0 `rs` register number. That requires just a 5-bit mux. In the not-as-good Plan B solution the value is moved, require an additional 32-bit mux input. Notice that the value is taken at the output of the upper ALU mux, this way a bypassed value can be used. That's possible in the Plan A solution is the output of the ID-stage mux is used for the bypass control logic (which isn't shown).

The control logic, in green, needs to check for three things: the presence of the two instructions, whether there is a dependence, and whether the immediate will fit in four bits. The solution only works for the arrangement of instructions shown above, and that's all that was needed for full credit. In real life one might want this trick to work for similar sequences, for example if the dependence were carried into the `rt` rather than the `rs` register.



Problem 2: (20 pts) Show the execution of the code fragments below on the illustrated MIPS implementations. All branches are taken.

(*a*) Show the execution of the code below on the following implementation. The branch is taken.



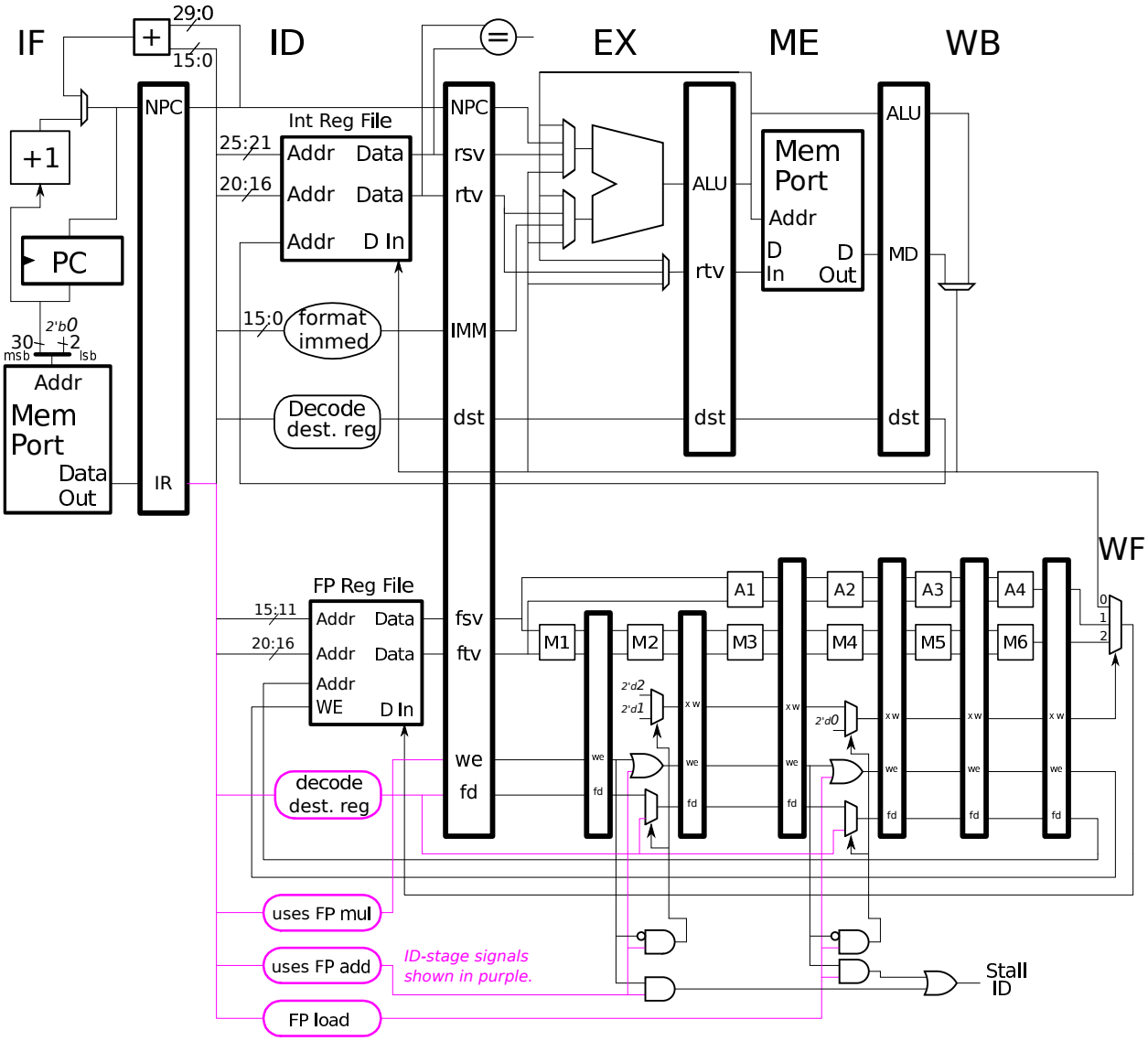
☑ Show execution. ☑ Doublecheck for dependencies.

Solution appears below.

Common mistakes: squashing the delay slot instruction (the `and`); not noticing that the branch depends on the `sub`.

```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
 add r4, r2, r3      IF ID EX ME WB
 lw r6, 8(r4)           IF ID EX ME WB
 sub r1, r6, r5            IF ID -> EX ME WB
 beq r7, r1 TARG              IF -> ID ----> EX ME WB
 and r8, r7, r10                    IF ----> ID EX ME WB
 or  r11, r12, r13
 xor r14, r11, r8
TARG:
 sw r1, 0(r2)                                  IF ID EX ME WB
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12
```

(*b*) Show the execution of the MIPS FP code below on the following implementation. Assume that **there is** a bypass from `WF` to `M1` and `A1`.

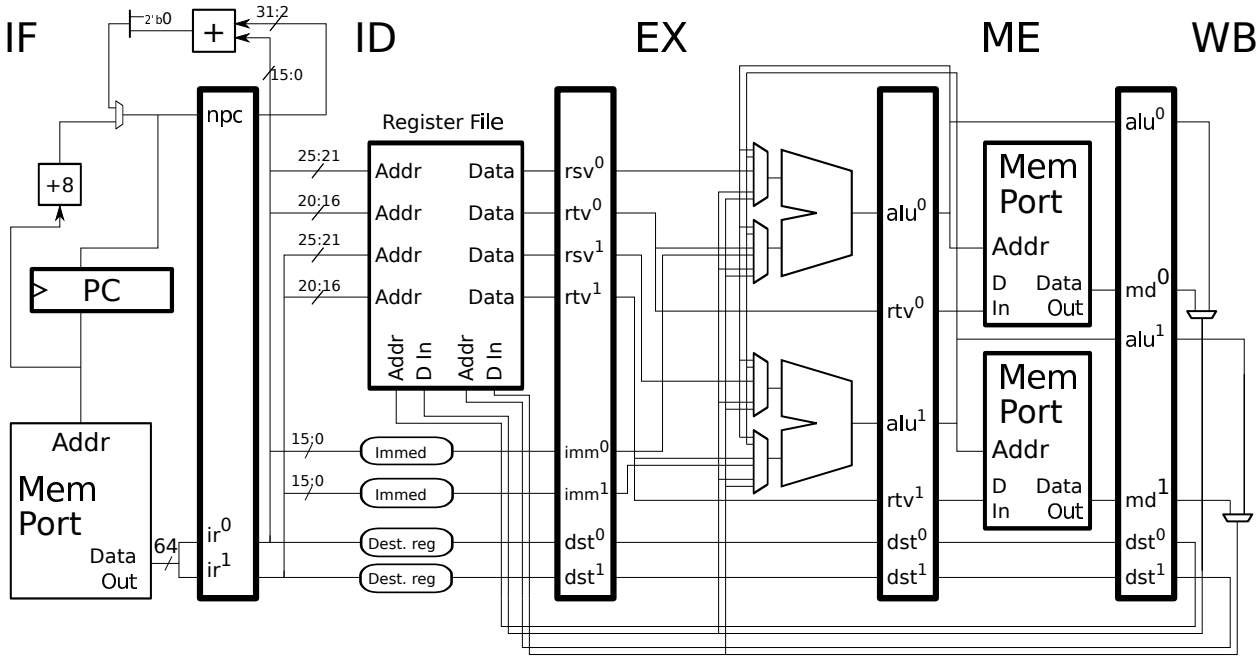


☑ Show code execution. ☑ Doublecheck for dependencies.

Solution appears below. Common mistakes: Not realizing that the `lwc1` uses `WF` (not `WB`) and therefore it must stall to avoid the structural hazard. Another common mistake is stalling in a stage other than `ID`.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
add.s  f6, f2, f3  IF ID A1 A2 A3 A4 WF
add.s  f1, f7, f8     IF ID A1 A2 A3 A4 WF
lwc1   f9, 0(r10)        IF ID ----> EX ME WF
mul.s  f4, f1, f9           IF ----> ID -> M1 M2 M3 M4 M5 M6 WF
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 n12 13 14
```

(*c*) Show the execution of the MIPS code below on the following 2-way superscalar implementation. The branch is taken. Fetch groups in this implementation must be 8-byte-aligned.



☑ Show code execution. ☑ Show squashed instructions with an `x`.

☑ Doublecheck for dependencies. ☑ Account for aligned fetch groups.

Solution appears below. Note that the `sll` is fetched only because the fetch unit can only fetch an address that's a multiple of 8, but it is squashed as soon as it arrives.

```
START: Instruction address of add is 0x1000
 # Cycle             0  1  2  3  4  5  6  7  8  9
 add r4, r2, r3      IF ID EX ME WB
 lw r6, 8(r4)        IF ID -> EX ME WB
 sub r1, r6, r5         IF -> ID -> EX ME WB
 beq r7, r15 TARG       IF -> ID -> EX ME WB
 and r8, r7, r10              IF -> ID EX ME WB
 or  r11, r12, r13            IFx
 xor r14, r11, r8

 sll r16, r17, 8                    IFx
TARG: Instruction address of sw is 0x2004
 sw r1, 0(r2)                       IF ID -> EX ME WB
 lui r15, 0x1234                       IF -> ID EX ME
 # Cycle             0  1  2  3  4  5  6  7  8  9  10
```

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{30} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(*a*) Branch behavior is shown below. Branch B2 consists of a pattern in which there will be either 2, 4, or 6 T outcomes followed by exactly four N outcomes. After the fourth N the probability of exactly two Ts is $\frac{1}{3}$ and the probability of exactly four Ts is $\frac{1}{3}$.

Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  N    T    T    N    N    N      N    T    T    N    N    N   ...
B2:   T{2,4,6}  N    N    N    N      T{2,4,6}  N    N    N    N  ...
B3:    T    T    T    T    T    T      T    T    T    T    T    T ...
```

☑ What is the accuracy of the bimodal predictor on branch B1?

The accuracy is $\boxed{\frac{3}{6}}$. The work to compute this result is shown below, in which the counter was assumed to start at 3. (Any assumed start value is correct.) The prediction accuracy is based on the second pattern repetition because the counter value at the start of the pattern and the end of that pattern is the same, 0. (In contrast, the counter value at the start of the first repetition is 3 and at the end it is 0, so we can't use the first six outcomes of B1 to compute an accuracy, though by coincidence it would be correct.)

```
#  SOLUTION Work
   3   2    3    3    2    1    0      0    1    2    1    0    0 <- Counter values.
B1:  N    T    T    N    N    N      N    T    T    N    N    N   ...
     x              x    x              x    x    x              <- Mispred location.
                                      ----------------------------  <- Repeating pattern.
```

☑ What is the accuracy of the bimodal predictor on branch B2? ☑ Explain.

To answer this question consider each number of Ts separately. First, TTNNNN. When this pattern starts the counter will be zero (because every repetition ends with four Ns). So the two Ts will be mispredicted, bringing the counter to 2, and the first N will be mispredicted. The number of correct predictions is $6 - 3 = 3$ and the accuracy is $\frac{1}{2}$. Next consider TTTTNNNN. Here the first two Ts will be mispredicted as will the first two Ns. The number of correct predictions is $8 - 4 = 4$ and the accuracy is also $\frac{1}{2}$. Finally, in TTTTTTNNNN the first two Ts and Ns will be mispredicted. The number of correct predictions is $10 - 4 = 6$ and the accuracy is .6.

For an overall prediction accuracy we need to combine these numbers. Since they are all equally likely we will consider one pattern of each length, 6, 8, and 10 together as a group. The correct prediction $\boxed{\text{ratio is } \frac{3+4+6}{6+8+10} = \frac{13}{24} = .541667}$. If we wanted to take a weighted average of the prediction accuracies we would have to weight by both the probability of occurrence and the length. In other words if we computed $\frac{1}{3}\frac{3}{6} + \frac{1}{3}\frac{4}{8} + \frac{1}{3}\frac{6}{10} = \frac{8}{15}$ we would have given the $\frac{6}{10}$ accuracy too low a weight. Yes, it happens one third the time, but its length, 10 outcomes, is longer than the other two so it has a larger overall impact on execution time.

☑ What is the minimum local history size needed to predict B1 with 100% accuracy?

$\boxed{\text{Four outcomes}}$ are sufficient. With three outcomes a local history of NNN could appear before an N or a T.

☑ What is the accuracy of the local predictor on branch B2, after warmup. ☑ Explain.

Because there is randomness the local predictor won't be right 100% of the time. Notice that we always have four consecutive N's followed by two T's. That means if the most-recent four outcomes in the local history are N then a T is certainly the next outcome (and so the corresponding PHT entry will have warmed up to 3). Similarly, if the most recent PHT outcomes are a T followed by one, two, or three Ns the next outcome is certainly an N. The table below shows some possible local history values, the next outcome, and how often that outcome will be encountered. Stars indicate either a T or N.

Local History		Next Outcome Frequency
**TTN	N	Every Time
**TTNN	N	Every Time
**TTNNN	N	Every Time
**TTNNNN	T	Every Time
**TTNNNNT	T	Every Time
**TTNNNNTT	T	2/3 of the time
**TTNNNNTT	N	1/3 of the time
**TTNNNNTTT	T	Every Time
**TTNNNNTTTT	T	1/2 of the time.
**TTNNNNTTTT	N	1/2 of the time.
TTNNNNTTTTTT	N	Every time.

First consider TTNNNN. The Ts will always be predicted correctly because at least two Ts always follow the four Ns. The first N can be mispredicted because $\frac{2}{3}$ of the time the run of Ts is larger than 2. First, lets assume that the N is *always* mispredicted. Later, in an advanced solution, we'll use a better misprediction probability.

The following three Ns will be correctly predicted. So for the first pattern we have 5 correct predictions and one misprediction.

Next, consider TTTTNNNN. The first four Ts will likely be predicted correctly because after the first two Ts a third T is more likely than an N. Given that we have four Ts, the probability of a fifth one is $\frac{1}{2}$. Since the bias is even the 2-bit counter can predict either direction with equal likelihood. So the first N is mispredicted half the time for this pattern, and similarly the fifth T is mispredicted half the time for the 6 T pattern. Considering a set of all three patterns, the expected total number of mispredicts is just 2, so the accuracy is $\frac{(5+7.5+9.5)}{(6+8+10)} = \frac{11}{12} = .91666$.

Advanced Solution: The solution above ignores the fact that for pattern TTNNNN there is a chance that the first N is correctly predicted. That would occur if the local predictor saw at least two consecutive TTNNNN patterns before predicting a third consecutive TTNNNN. Here *saw* means that the local histories were the name for all three patterns when predicting the first N. A simple Markov chain can be used to determine the probability of the counter value for local history *NNNNTT (the local history used when predicting the first N). Let p_0 denote the probability that the counter value is 0, p_1 denote the probability that the counter value is 1, and so on. The problem states that the T probability is $2/3$ in this case. Based on this we can write a *balanced flow* equation: $ap_0 = (1-a)p_1$, where a is the T probability. Solving we get $p_1 = \frac{a}{1-a}p_0$. Between p_1 and p_2 we can write $ap_1 = (1-a)p_2$ and get $p_2 = \left(\frac{a}{1-a}\right)^2 p_0$. Skipping a few steps we get $p_0 = \frac{\omega-1}{\omega^4-1}$ where $\omega = \frac{a}{1-a}$. Using $a = \frac{2}{3}$ we find $p_0 = \frac{1}{15}$ and $p_1 = \frac{2}{15}$ and so the probability of correctly predicting the first N for the T2 case is $\frac{3}{15} = \frac{1}{5}$. The number of correct predictions for T2 is $5 + \frac{1}{5}$ or an accuracy of $\frac{13}{15} = .8667$. Similarly, the number for T4 is $6 + \frac{4}{5} + \frac{1}{2}$ or an accuracy of $\frac{73}{80} = .9125$ and for T6 is $8 + \frac{4}{5} + \frac{1}{2}$ or an accuracy of $\frac{93}{100}$. The overall correct prediction accuracy is $\frac{109}{120} = .9083$.

☑ What is the minimum GHR size to predict B1 with about 100% accuracy. ☑ Explain, and state any assumption about B2's behavior.

Based on the answer to the minimum-local-history-size question above, the minimum local history needed to predict B1 with 100% accuracy is four outcomes. But here we are dealing with a global history, which contains the outcomes of the most recently executed branches, which in this case is B1, B2, and B3. In the illustration below the outcomes of B1 have been made lower case, and the value of a six-outcome GHR is shown at time X (see the diagram) when we are about to predict B1.

```
Time->                  X
B1:  n    t    t    n    n    n      n    t    t    n    n    n   ...
B2:   T{2,4,6}  N    N    N    N      T{2,4,6}  N    N    N    N  ...
B3:    T    T    T    T    T    T      T    T    T    T    T    T ...
              ----------X
               tNT  nNT     <--- Value of 6-outcome GHR at time X.
```

Notice that this six-outcome GHR includes two outcomes of B1. Since we need *four* outcomes of B1 to predict with 100% accuracy we would need a GHR size of $4 \times 3 = 12$ for the global predictor to predict B1 with 100% accuracy.

The analysis above assumes that there is always one occurrence of B2 between B1 and B3. The diagram makes that clear for the N outcomes of B2 but it's not clear for the T outcomes. In other words, from the diagram above one might conclude that there are 2, 4, or 6 T outcomes between B1 and B3. Since we were *invited* to make an assumption, lets make the easy one: that in all cases, N and T B2 outcomes, there is exactly one occurrence of B2 between B1 and B3.

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries.

Branch B1 below is perfectly biased not taken. It executes r_1 times over a short time interval, followed by a big time gap before its next bunch of r_1 executions. Branch B2 behaves similarly, except that it is biased taken and executes in bunches of length r_2 executions.

```
0xee4720  B1:   N..N r1                          N..N r1
# ...
          B2:             T..T r2                          T..T r2
```

(*b*) Choose an address for branch B2 that will result in a BHT collision with branch B1. (Branch B1 is at address 0xee4720.)

☑ Address for B2 that results in a collision.

Branch B1 will collide with B3 in a BHT lookup if the bit values used to index the BHT for the two branches are the same. Since the BHT has 2^{14} entries and because MIPS instructions are 4-byte aligned we will use bits 15:2 of the branch address to index (to use as a lookup address for) the BHT. The value of those bits for B1 is `0x4720 >> 2` or `0x11c8` but for simplicity think of the bits as `0x4720`. (Note that each hex digit spans four bits). For B2 to use the same entry the four least significant hex digits must match. One such address is `0xaa4720`.

(*c*) Assume that branch B1 and B2 collide in the BHT. For what positive values of r_1 and r_2 will this collision be most harmful? For what range of positive values of r_1 and r_2 will collisions be least harmful?

☑ Worst values of r_1 and r_2 for collision. ☑ Explain.

The worst values are $r_1 = 1$ and $r_2 = 1$. For those values the predictor will see one occurrence of B1 followed by one of B2, then one of B1 and so on. The 2-bit counter will not saturate at any value. In the worst case all predictions will be wrong.

☑ Favorable range of positive values of r_1 and r_2 for collision. ☑ Explain.

Large values for r_1, r_2 or both. When execution switches from one branch to the other there will be two mispredictions (assuming both $r_1 \geq 2$ and $r_2 \geq 2$). The accuracy would be $\frac{r_1-2+r_2-2}{r_1+r_2}$, so with the sum large the accuracy will be close to 100%.

(*d*) A tag can be placed in the BHT to detect collisions. Which branch will this help?

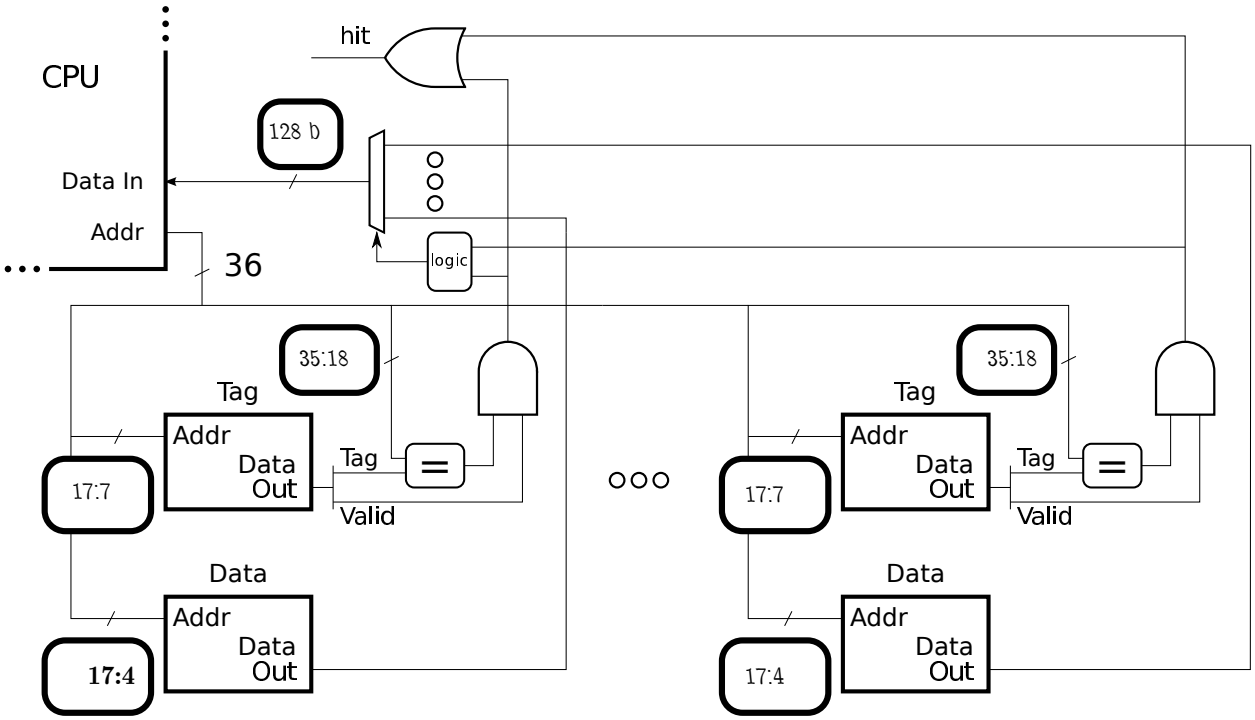
☑ Branch that would benefit from tag: B1 or B2. ☑ Explain.

Branch B1. If there is a collision when predicting B1 we will predict B1 taken (because of B2's history). A branch predictor that does not use a tag has no way of knowing that there's been a collision and so when predicting B1 it will jump to the target of B2 (not the target of B1). In the predictor with a tag the collision will be detected (by comparing the stored tag to the tag of the address of the branch being predicted). Since a collision means that the target address is wrong, the predictor will ignore the two-bit counter (which would be 3 in this example) and instead predict not taken, which has a chance of being correct. (When a collision has been detected you wouldn't predict taken because the target address is for some other branch sharing the entry, and so it won't be correct.) Predicting not taken for B2 is wrong, but that's what we'd do with or without a tag.

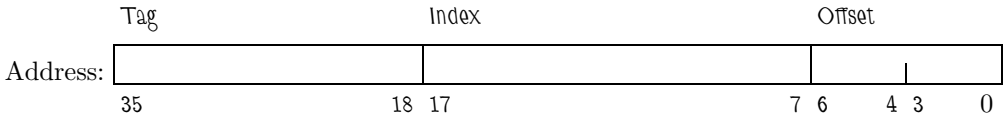
Problem 4: (20 pts) The diagram below is for a 4 MiB (2^{22} B) set-associative cache with a line size of 128 bytes. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)



☑ Associativity:

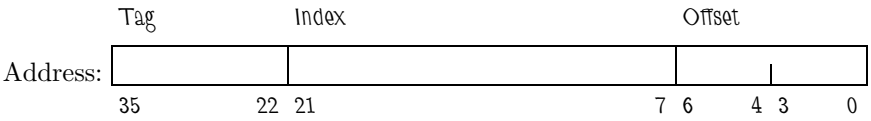
The associativity is 16. The associativity is determined based on the given cache capacity, 2^{22} bytes, and the capacity of an individual data store, 2^{18} bytes. Since the cache capacity is the sum of the data store sizes, the associativity must be $\frac{2^{22}}{2^{18}} = 2^{22-18} = 16$.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 4 MiB, plus $16 \times 2^{18-7}\,(36 - 18 + 1)$ bits.

☑ Show the bit categorization for a **direct mapped** cache with the same capacity and line size.

In a direct-mapped there is just one data store, which here must be 2^{22} bits, and so the low tag bit position must be $18 + 4 = 22$. The other bit positions remain the same.



Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 64 bytes. Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
int sum = 0;
half *a = 0x2000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^6 = 64$ bytes is given. The size of an array element, which is of type half, is $2 = 2^1$ characters, and so there are $2^6/2 = 2^{6-1} = 2^5 = 32$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^5 elements, and so the next $2^5 - 1 = 31$ accesses will be to data on the line, hits. The access at `i=32` will miss and the process will repeat. Therefore the hit ratio is $\frac{31}{32}$.

(*c*) The code appearing below is the same as the code from the 2014 Final Exam and 2015 Homework 6, except that two possibilities for `Some_Struct` are shown, Plan A, and Plan B. The caches too are identical except that here the line size is 64 bytes whereas in the prior problems it was 128 bytes. The value of `BSIZE` has been chosen to the largest value for which the hit ratio of the second loop will be 100% when using the Plan B struct. *Note: In the original exam* `norm_val` *was omitted from the first loop.*

```
struct Some_Struct { // Plan A
  double val;        // sizeof(double) = 8
  double norm_val;
  double a1[14];   };

struct Some_Struct {  // Plan B
  double val;         // sizeof(double) = 8
  double a1[7];
  double norm_val;
  double a2[7];    };

  const int BSIZE = 1 << 15;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val + b[i].norm_val;  // First loop.
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;   // Second loop.
```

☑ Explain why the Plan A struct will result in better performance on the first loop than the Plan B struct.

Short Answer: With Plan A a miss to `b[i].val` also brings in `b[i].norm_val`, with Plan B there would be two misses (per iteration) and so Plan A is faster.

Detailed Answer: Notice that the first loop accesses members `val` and `norm_val`. In the Plan A struct these members are close to each other, and so the line containing `val` also contains `norm_val`. Each iteration of the first loop misses on the access to `val` and because they are close together hits on the access to `norm_val`. The `val` member is at the beginning of the Plan B struct, but `norm_val` is $8 \times 8 = 64\,\mathrm{B}$ from the beginning. A miss that brings in `b[i].val` will not bring in `b[i].norm_val` because they both can't fit on the same 64 B line. Both the Plan A struct and Plan B struct are $16 \times 8 = 128\,\mathrm{B}$. When using the Plan A struct the first loop will encounter `BSIZE` misses, but when using the Plan B struct there will be `2*BSIZE` misses, and so better performance is achieved with the Plan A struct.

☑ Suppose that the value of `BSIZE` is to be chosen based on the Plan A struct, to a maximum value for which the second loop enjoys a 100% hit ratio. Explain why that value is the same as for Plan B.

Short Answer: Because with Plan A the least significant index bit is always zero or always one and so half the cache is used.

As explained in the previous answer, when using the Plan A struct we only need one line per iteration, whereas with the Plan B struct we need two lines per iteration. So you'd think that with Plan B we'd need to make `BSIZE` half the size to avoid cache misses. But no.

FINISH

Problem 5: (20 pts) Answer each question below.

(*a*) Which kind of implementations were RISC ISAs originally designed to simplify? Explain how a RISC ISA feature achieves this goal.

☑ Type of implementation RISC designed to simplify.

Pipelined implementations.

☑ Choose a feature and ☑ explain how it achieves the goal.

One RISC feature is that the arguments of arithmetic instructions must be either immediates or registers, they cannot come from or be written to memory. If source and destination arguments could come from memory then either there would need to be a memory port before and after the EX stage, which would be expensive, or such instructions would have to pass through the pipeline multiple times, stalling prior instructions and complicating the design.

(*b*) What is the difference between a trap instruction and an instruction that raises an exception? Give an example of how each is used in a system with bug-free code.

☑ Difference between trap and exception.

A trap always causes an interrupt, that's its purpose, whereas in an instruction that raised an exception something went wrong.

☑ Example of what trap instruction used for.

A trap instruction can be used by user code to request a service from an operating system, for example, to read data from a file. In a typical system user code cannot itself, for example, read data from a file since it is not allowed to access those memory location that control the disk drive. It is not allowed access so that filesystem access policies can be enforced, for example.

☑ Example of what instruction exception used for.

Load and store instructions raise exceptions when they attempt to accesses invalid memory addresses, which might happen in buggy code. In bug-free code load and store instructions can still raise exceptions. This would happen for cases where the OS needs to adjust the memory system. For example, a load instruction encountering a TLB miss will raise an exception. The OS (acting as the exception handler) will read the page tables, update the TLB, and re-start the load.

Problem 5, continued:

(*c*) Consider a $5n$-stage scalar implementation and an n-way superscalar implementation, both derived from our 5-stage design with the goal of improving performance by as much as $n\times$. As n increases the clock frequency of the $5n$-stage implementation increases but the frequency of the n-way superscalar implementation decreases. Explain why. Assume that both types of system are well designed.

☑ Clock increases in the $5n$-stage system because ...

... the critical path in each stage is shorter as n increases and so the clock frequency can be increased. (In the $5n-$stage system each stage in the 5-stage system is divided into n stages each taking $\frac{1}{n}$ the amount of time, ideally.) Ignoring latch setup time and assuming stages can be perfectly divided, the clock frequency would be $n\phi$, where ϕ is the clock frequency of the 5-stage system. Accounting for a latch setup time of t_L, the clock frequency would be $\frac{n\phi}{t_L\phi(n-1)+1}$. It is this increase in clock frequency that results in higher performance.

☑ Clock decreases in the n-way system because ...

The design is becoming larger and so physical distances increase. A stage can have $n\times$ as much hardware, one set of hardware for each of n slots. However signals only go through one set of hardware, so there is no need for a reduction in clock frequency by a factor of n. The problem is that physical distances are increasing, forcing paths to be longer.

(*d*) A company is preparing a run of the SPEC benchmarks for their new system. While trying to tweak compiler flags to get the best performance they discover a new optimization technique and implement it in their compiler. They re-test with that compiler and disclose the test results. When is the use of the modified compiler allowed under SPEC rules? When isn't that allowed under SPEC rules?

☑ Modified compiler allowed provided that ...

☑ This is a beneficial because ...

... provided that the compiler is seriously made available to the public. This is beneficial because the modified compiler generates better code than the old one.

☑ Modified compiler isn't allowed if ...

☑ The modified compiler is not beneficial because ...

... isn't allowed if the compiler is not made public, or not seriously marketed as a product. (For example, if the compiler has a product number but does not appear in any catalog or promotional material.) Ordinarily a company would be happy to market a better compiler. If they are not doing so it might be because the compiler generates buggy code when using the new optimizations. So it is not beneficial because the compiler, though it works on the SPEC benchmarks, would generate buggy code when compiling anything else. That means that the performance numbers obtained on SPEC code do not reflect what a user might obtain on other code.

52 Spring 2014 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

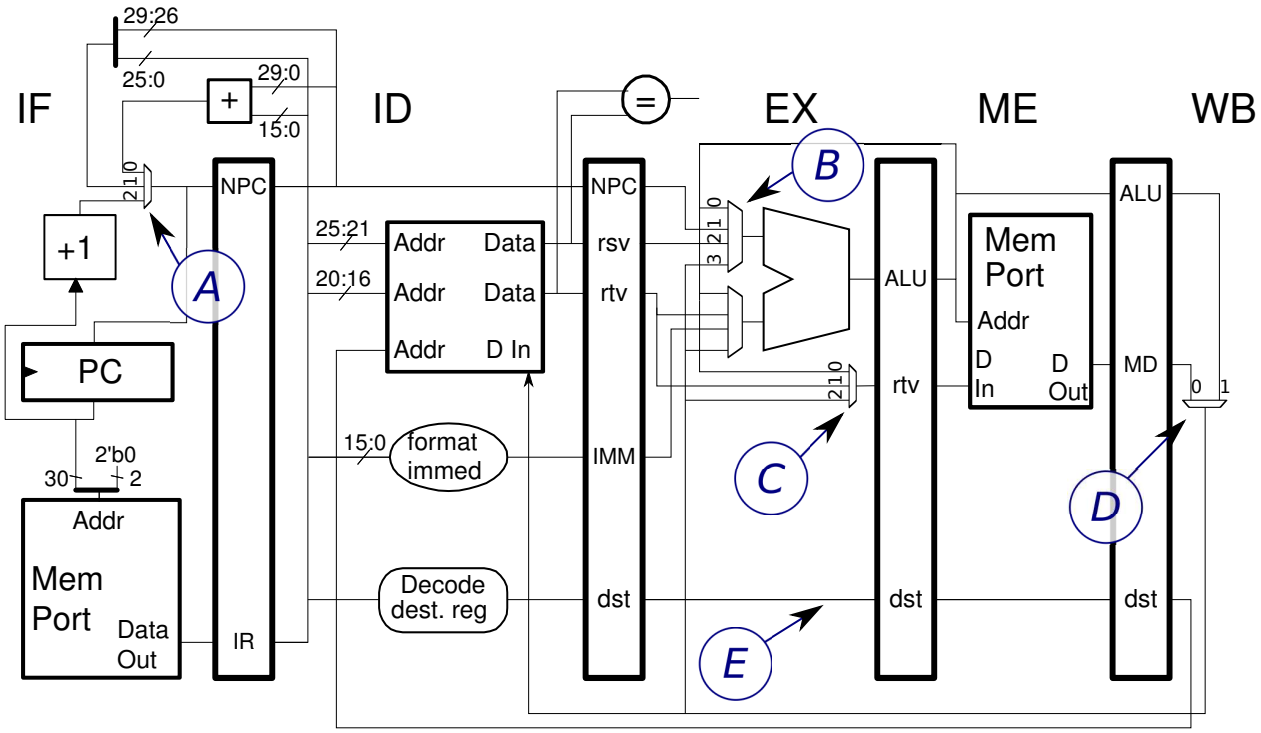
Monday, 31 March 2014, 9:30–10:20 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(18 pts)
Problem 4		(12 pts)
Problem 5		(6 pts)
Problem 6		(12 pts)
Problem 7		(12 pts)
Exam Total		(100 pts)

Alias -O11

Good Luck!

Problem 1: [20 pts] The MIPS code fragment below executes on our familiar five-stage scalar MIPS implementation.



(*a*) Show the execution of the code for enough iterations to determine CPI, and determine the CPI assuming a large number of iterations.

(*b*) There are five labels in the diagram, labels A through D point at multiplexor control inputs and label E points at the output of the `EX.dst` pipeline latch. Show the values of these signals up until the second execution of `lw r1`. For A show only values ≠ 2, for E show only values ≠ 0, for the others only show values at cycles in which the multiplexor is in use.

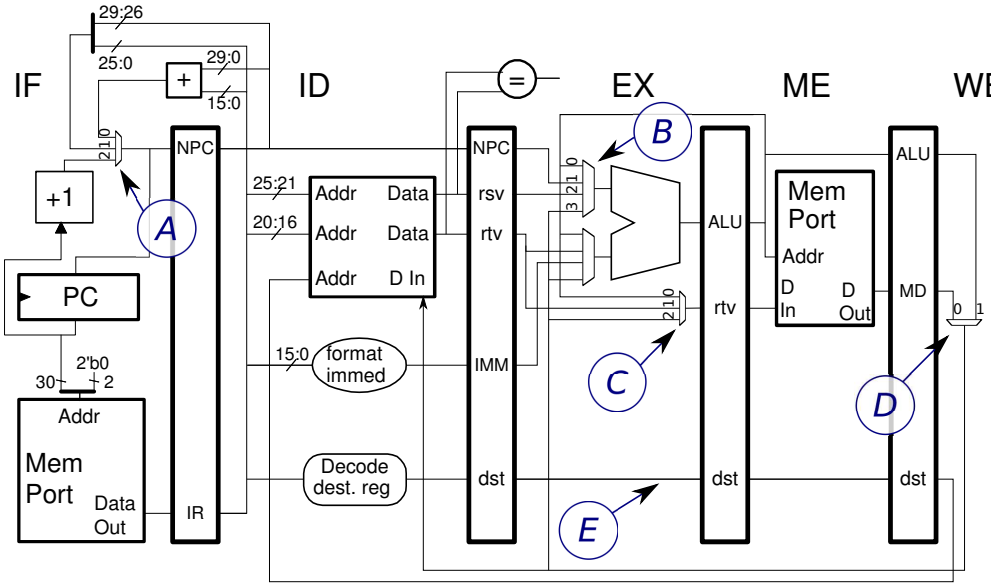
USE NEXT PAGE FOR SOLUTION

```
LOOP: Cycle        0
 A                 2
 B
 C
 D
 E                 0
 lw r2, 0(r5)      IF
LOOP:
 lw r1, 0(r2)
 sh r1, 16(r2)
 bne r1, r0, LOOP
 add r2, r2, r3
 xor r4, r5, r2
```

USE NEXT PAGE FOR SOLUTION

Problem 1, continued:

- ☑ Check code for dependencies.
- ☑ Show pipeline execution diagram for enough iterations to determine CPI.
- ☑ Determine the CPI.
- ☑ Show values for A through E.



Solution appears below. The pipeline execution diagram is straightforward. Don't forget that for the `sh r1, 16(r2)` instruction both `r1` and `r2` are source registers, and that there is no destination register. Also notice that `lw r1` only stalls in the first iteration. In the row for signal A, non-default values are present when a branch instruction is in `ID`, this is in cycles 6 and 11, the value is 0, picking up the output of the adder. Signal B is the upper ALU mux. A value of 2 indicates that the value from the register file is used (not a bypassed one). The `lw r2, 0(r5)` and `add r2,r2,r3` instructions use this value. A value of 0 indicates a bypass from `ME`, this occurs in cycle 9 when the `lw r1` picks up `r2` from the `add`. In cycle 4 the `lw` bypasses from `WB`. Signal C is for the write value of store instructions (register `r1` in this case), D picks which to write back, an ALU output or a memory output. Signal E is the destination register number of the instruction in `EX`.

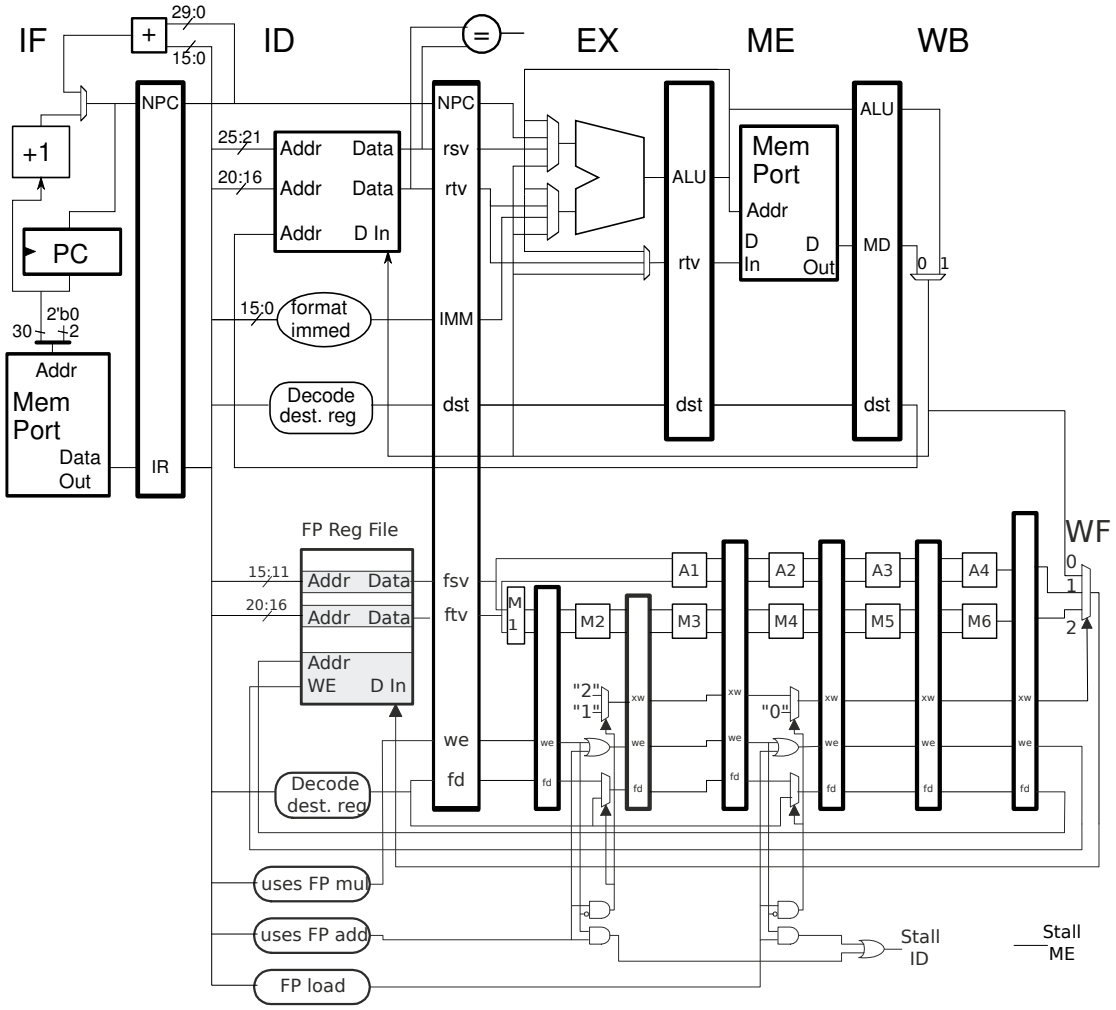
The CPI is computed using execution between cycles 12 and 7, in which the iterations start identically. The value is $\frac{12-7}{4} = 1.25$.

```
#     SOLUTION
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 A                2                 0              0           Blanks = 2
 B                   2     3     2     2  0
 C                                  2              2
 D                            0     0           1  0
 E                0     2     1           2  1           2     Blanks = 0
 lw r2, 0(r5)     IF ID EX ME WB
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)        IF ID -> EX ME WB
 sh r1, 16(r2)          IF -> ID -> EX ME WB
 bne r1, r0, LOOP             IF -> ID EX ME WB
 add r2, r2, r3                     IF ID EX ME WB
 xor r4, r5, r2
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)                          IF ID EX ME WB
 sh r1, 16(r2)                            IF ID -> EX ME WB
 bne r1, r0, LOOP                            IF -> ID EX ME WB
 add r2, r2, r3                                    IF ID EX ME WB
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 0(r2)                                         IF ID EX ME WB
```

Problem 2: [20 pts] The pipeline execution diagram below shows the *incorrect* execution of the MIPS code for the illustrated pipeline. That is, in the pipeline below the stall to avoid the structural hazard would have occurred when `lwc1` was in the `ID` stage.

```
# Cycle            0  1  2  3  4  5  6  7
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
addi r1, r1, 4        IF ID EX ME WB
lwc1  f4, 0(r1)          IF ID EX ME -> WF
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

As described below, modify the pipeline so that FP load instructions, such as `lwc1`, will stall when they are in `ME` rather than `ID` to avoid a `WF` structural hazard.

(*a*) Show logic to generate a signal `Stall ME` in the correct cycle. When `Stall ME` is logic `1` stages `ME` and earlier will stall.

(*b*) Cross out the logic generating the `Stall ID` for the `lwc1` that is no longer needed.

(*c*) Make other changes, including control logic, so that the correct destination register number, destination value, and write-enable signal are delivered to the FP register file. *Hint: Some solutions are simpler than others.*

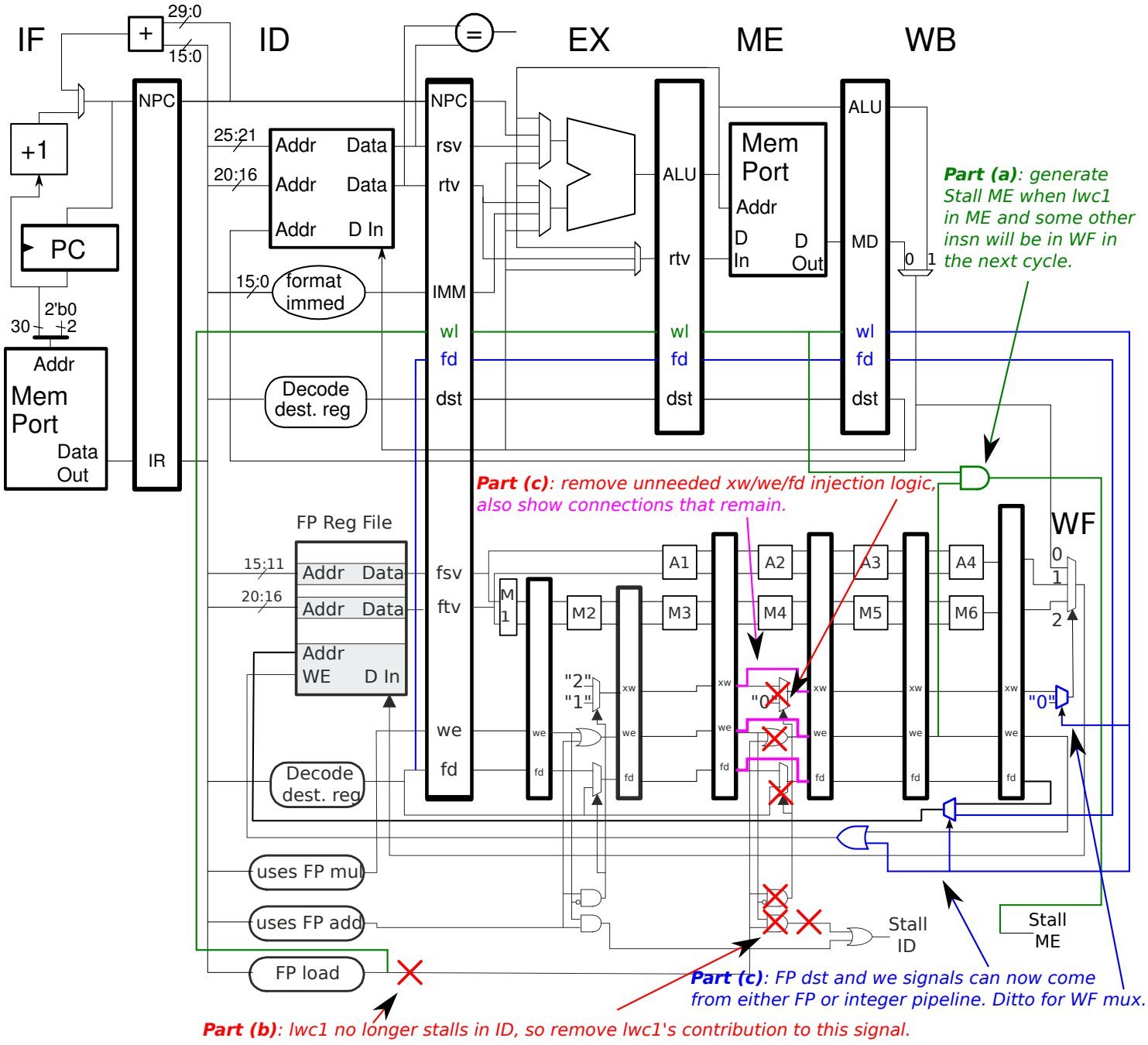
Problem 2, continued:

☑ Generate `Stall ME` to avoid `lwc1 WF` structural hazard. ☑ Make sure signal is `1` in the right cycle.

☑ Cross out `Stall ID` logic for FP loads, but leave logic for other instructions.

☑ Make sure that correct `fd`, `we`, and value delivered to FP register file.

The solution appears below, see the next page for more explanation.



The solution appears on the previous page. Looking at the code execution example, notice that in the original pipeline (see the previous page) at cycle 3 the load instruction must be placing its **we** and **fd** values in the **A2** stage, but that would replace (clobber) the corresponding values for the **add.s** instruction which are also in that stage. Since the destination for the **add.s** is lost, there is no way it can execute correctly.

The solution is to place the write enable and FP destination for FP load instructions in the integer pipeline, using new pipeline latch segments **wl** (write-enable FP load) and **fd**. This keeps them separate from those of the add or multiply instructions, and it ensures that the **Stall ME** signal will stall the **lwc1** without affecting instructions in the FP pipeline.

The logic to generate the **Stall ME** signal appears in green. We want to stall **ME** when the **lwc1** instruction is in **ME** and when there is some instruction in **A4/M6**, this is detected by the green AND gate.

For part b, the logic generating **Stall ID** for load instructions is crossed out, shown in red along the bottom.

For part c we must make sure that the write enable and destination that we put in the integer pipeline reach the FP register file, we also want to make sure the WF mux is set correctly. That's shown in blue. The **fd** values from the two pipelines enter a new mux, the control signal is just **wl** (write-enable FP load). For the write enable we just use an OR gate (we could have used a mux, but an OR gate achieves the same result). The zero value for the WF mux is inserted using a new mux.

Also for part c we must remove the old logic inserting **fd**, **we**, and **xw** for the load instruction, that's shown in red with the surviving connections shown in purple.

Problem 3: [18 pts] Answer each question below.

(*a*) Re-write the SPARC code fragment below in MIPS. Pay attention to the load instruction and the instructions related to the branch. Don't forget that in SPARC assembly language the last register is the destination.

```
!
 addcc %g1, %g2, %g3
 ld [%g5+%g3], %g4
 bg TARGET          ! Branch greater than zero.
 add %g5, 22, %g6   ! g6 = g5 + 22
```

☑ MIPS equivalent of SPARC code. ☑ Pay attention to load and branch-related insn.

The solution appears below.

Grading Note: Many students incorrectly assumed that the condition for the `bg` *instruction was taken from the* `ld` *instruction. SPARC condition codes are set explicitly: by instructions ending with* `cc`*. In this case the* `addcc` *instruction sets the condition codes.*

```
# SOLUTION
add r3, r1, r2
add r10, r3, r5
lw r4, 0(r10)
bgtz r3  TARGET
addi r6, r5, 22
```

(*b*) Show a MIPS equivalent of the ARM A32 code below. The MIPS code will use more than one instruction. (This is based on Homework 3.) *Note: A32 added in 2017 to avoid confusion with the A64 instruction set.*

```
add r1, r2, r3, LSL #4
```

☑ MIPS equivalent of ARM code above.

```
# SOLUTION
sll r1, r3, 4
add r1, r1, r2
```

(*c*) Explain why the MIPS code fragment below is certain to execute with an error.

```
lw r1, 0(r2)
lw r3, 1(r2)
```

☑ Error is certain because:

In MIPS `lw` memory addresses must be a multiple of 4, this rule is called an *alignment restriction*. The memory addresses used are `r2+0` and `r2+1`. They can't both be a multiple of 4. Note that it is possible that `r2+1` is a multiple of 4, there is no reason why the offset itself (`1` in this case) must be a multiple of 4.

Problem 4: [12 pts] Answer each question below.

(*a*) In the execution below the `ant` instruction raises an illegal instruction exception and the handler is fetched, note that the handler starts execution in cycle 4. Explain why this exception (as executed) cannot be precise and show how execution should proceed for our five-stage pipeline.

```
# Cycle             0  1  2  3  4  5  6
 add r10, r11, r12  IF ID EX ME WB
 sw r3, 4(r5)          IF ID EX x
 ant r1, r2, r5           IF*ID*x
 or  r5, r6, r7              IF x

HANDLER:
 sw                             IF ID ..
```

☑ Reason why this execution not for a precise exception.

It is not precise because the `sw` did not finish executing. In a precise exception code should execute correctly up to the instruction just before the faulting instruction, the faulting instruction and those after it cannot modify registers or memory.

☑ Show execution that could be precise when `ID` stage discovers the bad `ant` opcode.

Solution appears below. Note that the exception is not acted upon until the faulting instruction reaches the `ME` stage. We need to wait that long to make sure that the `sw` instruction does not raise an exception, if it did we'd want to handle that exception first and leave the `ant` for later.

```
# SOLUTION
# Cycle             0  1  2  3  4  5  6  7
 add r10, r11, r12  IF ID EX ME WB
 sw r3, 4(r5)          IF ID EX ME WB
 ant r1, r2, r5           IF*ID*EX MEx
 or  r5, r6, r7              IF ID EXx

HANDLER:
 sw                                IF ID ..
# Cycle             0  1  2  3  4  5  6  7
```

(*b*) An interrupt mechanism will switch the processor from user mode to privileged mode when the handler starts. What is the difference between user and privileged mode and why are they necessary to implement an operating system?

☑ Difference between user and privileged mode? ☑ Importance for OS?

In privileged mode the system can execute any instruction and access any memory location. In user mode an attempt to access a *system memory address* or execute a system instruction will result in an exception.

By preventing user programs from touching or even reading certain memory locations, the OS can implement controls over what programs do. For example, preventing a program from making arbitrary disk accesses by mapping the disk hardware control registers to memory locations in system space.

Problem 5: [6 pts] In class we described three broad families of ISAs: CISC, RISC, and VLIW.

(*a*) In which ISA family is it easy to have instructions with long (say, 32 bit) immediates? Explain how.

☑ ISA family with long immediates is:

☑ What about the ISA family enables this?

CISC ISAs have instructions with long immediates. They can accommodate them because of CISC's variable instruction size. If you need a big immediate, just use a big instruction.

(*b*) In which ISA family are dependencies between instructions explicitly given? How is the dependence information supposed to help?

☑ ISA family in which dependence information is part of program is:

☑ Reason for providing dependence information.

Dependence info is part of the program in VLIW ISAs. This information is intended to simplify the design of the hardware by providing it information a little sooner than it would otherwise obtain it. (A little sooner because it does not need to look at instructions' register operands.)

(*c*) Which ISA family tends to have the most compact programs? (That is, the size in bytes of the program is smallest.) What makes them compact?

☑ ISA family with most compact programs is:

☑ Program sizes are small because.

CISC ISAs have the most compact programs, that is because of their variable instruction sizes. All RISC instructions must be 32 bits, in some cases wasting space. Also, CISC instructions can do the work of several RISC instructions, saving space. Since VLIW ISAs bundle RISC-like instructions, they suffer the same program size issues as RISC programs.

Problem 6: [12 pts] Answer each question below.

(*a*) Consider a 2-way superscalar implementation and a 10-stage implementation, both derived from our 5-stage MIPS used in class. All implementations include reasonable bypass paths.

Explain how the instruction sequence below would always result in a stall on the 10-stage pipeline but might not result in a stall on the 2-way system. (The compiler cannot separate the two instructions.) Illustrate your answer with a pipeline execution diagram.

```
add r1, r2, r3
sub r4, r1, r5
```

☑ Reason for maybe stall on superscalar and certain stall on 10-stage.

☑ Illustrate using a pipeline diagram.

The different executions are shown below. The 10-stage implementation must stall because there is no way the sum from the `add` can be ready in the next cycle. But for the superscalar system there is no need to stall if the `add` and `sub` are in different fetch groups. In the first superscalar example below they are in the same fetch group, and there is a stall, but in the second they are in different groups and so there is no stall.

Though the superscalar system might seem to have the advantage, when it does stall it costs two instructions, whereas for the 10-stage system only one instruction execution opportunity is lost.

```
# SOLUTION  - Unavoidable stall in pipeline.
add r1, r2, r3  IF1 IF2 ID1 ID2 EX1 EX2 ME1 ME2 WB1 WB2
sub r4, r1, r5      IF1 IF2 ID1 ID2 --> EX1 EX2 ME1 ME2 WB1 WB2

# SOLUTION  Stall in 2-way Superscalar
0x1000 add r1, r2, r3  IF1 ID1 EX1 ME1 WB1
0x1004 sub r4, r1, r5  IF2 ID2 --> EX2 ME2 WB2
0x1008 xor                 IF1 --> ID1 EX1 ME1 WB1
0x100c lw                  IF2 --> ID2 EX2 ME2 WB2

# SOLUTION  No stall in 2-way Superscalar
0x1004 add r1, r2, r3  IF2 ID2 EX2 ME2 WB2
0x1008 sub r4, r1, r5      IF1 ID1 EX1 ME1 WB1
0x100c xor                 IF2 ID2 EX2 ME2 WB2
0x1010 lw                      IF1 ID1 EX1 ME1 WB1
```

(*b*) Consider two MIPS implementation, one is an n-way superscalar of MIPS-I, with n sets of FP units, the other is a scalar implementation of a hypothetical MIPS-I-vec, which includes an n-lane vector unit for the vector instructions in MIPS-I-vec. Both the superscalar and vector MIPS implementations can sustain execution at n FP operations per cycle. *Note: The phrase "In terms of n" did not appear in the original exam.*

☑ In terms of n how does the cost of bypass paths compare in the two systems?

For the scalar 5-stage implementation the value computed by the ALU is bypassed back to the **EX** stage when it is in the **ME** and **WB** stages. Since there are two ALU inputs the number of connections is $2 \times 2 = 4$. In the superscalar design there are n ALUs and so n values, these must be bypassed back to the two inputs in each of n ALUs, for a total of $2n \times 2n = 4n^2$ connections.

For the n-lane vector unit, we would expect that the value produced by an ALU in a lane would only be bypassed back to the ALU in that lane. So the number of connections is n times that in a scalar processor, or $4n$ connections.

☑ In terms of n how does the cost of registers compare in the two systems?

Short Answer: For the superscalar system the cost is constant (does not change with n). For the n-lane vector unit the cost is proportional to n.

Long Answer: A n-way statically scheduled superscalar processor has the same number registers regardless of n. (Things will be different when we consider dynamic scheduling later in the semester.) An ISA with vector instructions also has vector registers. If those instructions are for an n-lane unit the registers will have space for n operands.

Problem 7: [12 pts] Answer each question below.

(*a*) The SPECcpu benchmarks are designed to evaluate the CPU and memory system in new implementations and ISAs. To realize this testers are responsible for providing a compiler and for compiling the benchmarks.

Suppose SPEC required testers to use a compiler that they provide. How much would this impact the goal of testing new ISAs? New implementations of existing ISAs?

☑ Degree of impact of SPEC-provided compilers for testing new ISAs. ☑ Explain.

If SPEC is supplying the compiler and the ISA is new, then SPEC would need a new compiler back-end for the new ISA. That would take a lot of time and effort for SPEC to develop and so there would be a long delay before SPECcpu scores could be obtained on the new ISA, and so the degree of impact would be large. Note that existing SPEC compilers could not be used for the new ISA, since they don't know how to generate instructions for the ISA (since its new).

☑ Degree of impact of SPEC-provided compilers for testing new implementations of old ISAs. ☑ Explain.

When a new implementation is developed compiler optimizations are adjusted or developed for the new implementation. In many cases the improvements are small. For this reason, using an old compiler on a new implementation would have a small impact on the results.

Though the impact would be small, it's still not a good thing to do because in many cases the compiler back end is developed by the same people developing the implementation, and there is no reason why the two (compiler and chip) should not be tested as a unit.

(*b*) A company releases a SPEC disclosure in which they have substituted the bzip2 benchmark with another version of bzip2 which does the same thing but runs faster. Since it does the same thing it provides the correct output to the SPEC verification script. As a result they get very good scores. How will they get caught? How is it against the goal of SPECcpu?

☑ How will the benchmark substitution get caught?

Anyone disclosing a SPECcpu score must provide a "config" file with all of the settings, and any software (such as compilers) needed to build the benchmarks must be publicly available (though not necessarily free). Therefore, it is easy for a 3rd party to re-run the test and they will discover the discrepancy.

☑ How does the substitution undermine what SPECcpu is supposed to measure?

The obvious answer is that if people don't know your substituting benchmarks then they'll attribute the better scores to the computer (until the substitution is discovered).

Name Solution

Computer Architecture

EE 4720

Final Examination

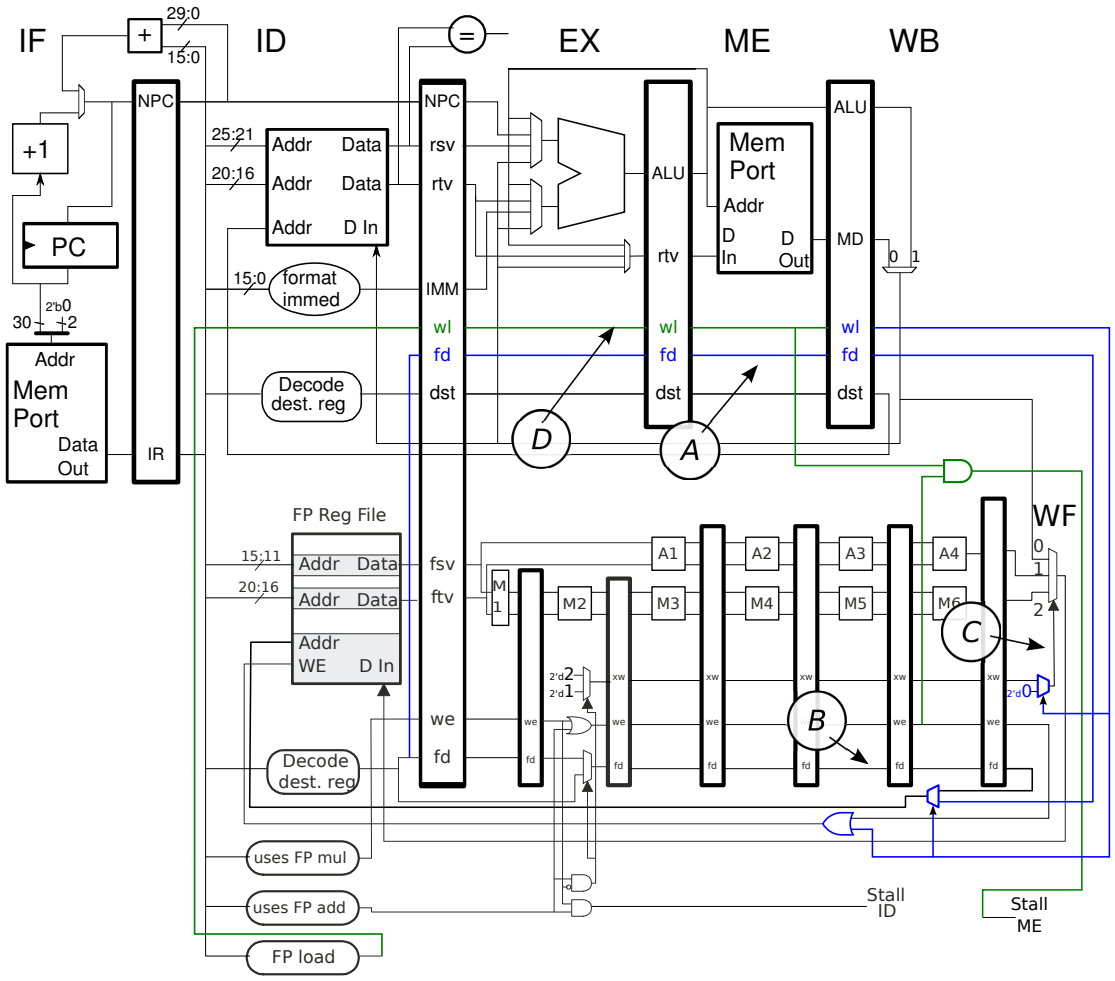
10 May 2014, 10:00–12:00 CDT

Problem 1		(15 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(5 pts)
Problem 6		(10 pts)
Problem 7		(25 pts)
Exam Total		(100 pts)

Alias Click Here

Good Luck!

Problem 1: (15 pts) Illustrated below is the **stall-in-ME** version of our MIPS implementation, taken from the solution to the midterm exam.



(*a*) Show a pipeline execution diagram of the code below on this pipeline.

(*b*) Wires in the diagram are labeled A, B, C, and D. Under your pipeline execution diagram show the values on those wires when they are in use.

☑ Show pipeline execution diagram. ☑ Show values of A, B, C, and D.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8
add.s f4,f5,f6   IF ID A1 A2 A3 A4 WF
sub.s f7,f8,f9      IF ID A1 A2 A3 A4 WF
lwc1 f1, 0(r2)         IF ID EX ME ----> WF
A                                 1  1  1
B                              4  7
C                                    1  1  0
D                              1
# Cycle          0  1  2  3  4  5  6  7  8
```

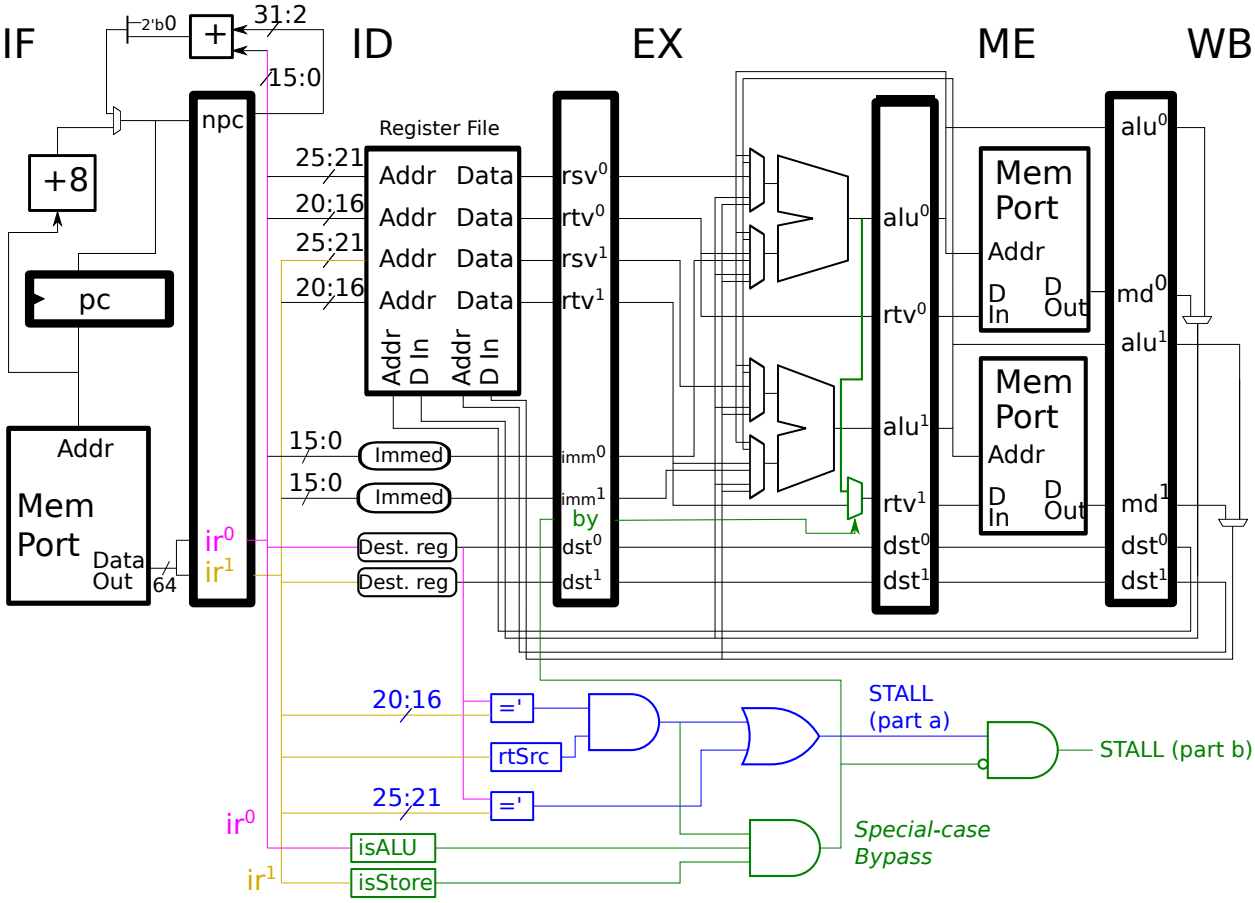
Solution appears above. Values on A are only used by the FP load instructions, in this case, `lwc1`, and so in the table above the values are only shown in cycles 5, 6, and 7. That said, in cycle 3 A would have a 4 and in cycle 4 it would hold a 7.

Values on B are used for destination of all other instructions that write the floating-point register file, for the code above the **add.s** and **sub.s** instructions. Notice that the value shown for B is for the instruction in the **A3** stage.

Values on C control which source will be written back to the FP register file.

The value on D is 1 when a FP load is in the **EX** stage.

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block isALU is 1 if the instruction's result is computed by the integer ALU. The output of logic block rtSrc is 1 if the instruction uses the `rt` register as a source. The output of logic block isStore is 1 if the instruction is a store.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☑ Control logic to detect true dependence in `ID` and assign `STALL`.

Solution appears above in blue. The stall signal for this part is the output of the OR gate. To help understand what's going on the IR (instruction register) for slot 0 is shown in purple and the IR for slot 1 is shown in gold. The logic compares the destination of the instruction in ID slot 0 with the two sources of the instruction in slot 1. If either matches, the stall signal is generated (which before part b would be the output of the OR gate). The rtSrc logic is needed because the `rt` field might hold a destination register number, and we would not want to stall for that.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular instruction pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Note: There was a similar-sounding problem in last year's final, but the solutions are different.*

```
0x1000: add r1, r2, r3
0x1004: sw r1, 0(r5)
```

☑ Add the bypass path(s) needed so that the code executes without a stall.

☑ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

Solution appears in green.

The special case here that makes a stall unnecessary is that the `sw` does not need the result of the `add` until the end of the EX stage (or the beginning of ME). Therefore the result can be bypassed from the output of the slot-0 ALU to the input of the `EX/ME.rtv1` pipeline latch. The logic for this bypass path is in the EX stage, shown in green. (It would also be correct to put the bypass in the ME stage.)

The control logic for this special-case bypass is also shown in green. The three-input AND gate checks for a dependence between the destination of the slot-0 instruction and the `rt` source of the slot-1 instruction (top input), whether the slot-0 instruction's result is produced by the ALU (middle input), and whether the slot-1 instruction is a store (bottom input). If all are true the bypass signal is set to true. The bypass signal is used for the new `ID/EX.by` pipeline latch and for the AND gate suppressing the stall signal.

Problem 3: (15 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{30} entry BHT. One system has a bimodal predictor, one system has a local predictor with a 12-outcome local history, and one system has a global predictor with a 12-outcome global history.

(*a*) Branch behavior is shown below. Notice that `B2`'s outcomes come in groups of three, such as 3 `q`s. The first outcome of each group is random and is modeled by a Bernoulli random variable with $p = .5$ (taken probability is .5). The second and third outcomes are the same as the first. For example, if the first `q` is `T` the second and third `q` will also be `T`. If the first `s` is `N`, the second and third `s` will also be `N`. Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T   T   T   N   N     T   T   T   N   N     ...
B2:   r   r   r   q   q     q   s   s   s   u     ...
B3:    T   T   T   T   T     T   T   T   T   T   ...
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

The accuracy is $\boxed{\frac{2}{5}}$. The work to compute this result is shown below. The prediction accuracy is based on the second pattern repetition because the counter value at the start of the pattern and the end of the pattern is the same, 1. (In contrast, the counter value at the start of the first repetition is 0 and at the end it is 1, so we can't use the first five outcomes of `B1` to compute an accuracy.)

```
#  SOLUTION Work
   0   1   2   3   2   1     2   3   3   2   1   <- Counter values.
B1:  T   T   T   N   N     T   T   T   N   N     ...
     x   x       x   x     x           x   x     <- Mispred location.
                         ----------------------  <- Repeating pattern.
```

☑ What is the approximate accuracy of the bimodal predictor on branch `B2`? ☑ Explain.

At the end of a group of three outcomes, call them `r1`, `r2`, and `r3`, the two-bit counter will be either 0 or 3. Consider two possible cases: in Case 1 the next three outcomes, call them `q1`, `q2`, and `q3`, are the same as the `r`; in Case 2 the next three outcomes are different. See the diagram below. The probability of a `q1` falling into Case 1 is .5. Because the branch directions agree in Case 1 the three `q`s will be predicted correctly. In Case 2, where `r` and `q` are different, `q1` and `q2` will be mispredicted, but `q3` will be correctly predicted. See counter values in the diagram below.

The overall prediction accuracy is the average of the two, $\boxed{\frac{1}{2}\left(\frac{3}{3}+\frac{1}{3}\right)=\frac{4}{6}}$.

```
Case 1: r and q agree.  Probability 0.5.
                3   3   3   3     <- Counter value
B2:   r   r   r    q   q   q  ...
B2:   T   T   T    T   T   T  ...  <- Let q = T and r = T.
                                   <- Zero mispredictions of q.

Case 2: r and q disagree.  Probability 0.5
                3   2   1   0     <- Counter value
B2:   r   r   r    q   q   q  ...
B2:   T   T   T    N   N   N       <- Let q = T and r = N.
                   x   x           <- Two mispredictions of q.
```

☑ What is the minimum local history size needed to predict `B1` with 100% accuracy?

Three outcomes are sufficient. With two outcomes a local history of `TT` could appear before the third `T` or the first `N`.

☑ What is the accuracy of the local predictor on branch B2, after warmup. ☑ Explain.

The local predictor will predict the first branch of a group of three with a 50% accuracy. (No predictor can do better or worse.) The second and third outcomes will be predicted with 100% accuracy after warmup. That's because the PHT entries corresponding to local histories ending with an N will saturate down to a zero, and those ending with T will saturate up to 3. (See the next part.) The overall accuracy is $\boxed{\frac{1}{3}\frac{1}{2} + \frac{2}{3}\frac{2}{2} = \frac{5}{6}}$.

☑ What is the best local history size for branch B2, *taking warmup into consideration.* ☑ Explain.

For any local history size greater than zero the prediction accuracy will be $\frac{5}{6}$ after warmup. The longer the local history size is the longer the warmup time is. The minimum local history size needed is just one outcome. The local predictor will predict the same outcome as the most recent occurrence.

☑ How many different GHR values will there be when predicting B3?

Let 312 312 312 312 indicate the state of the 12-bit GHR when predicting B3. Each digit indicates the branch affecting the corresponding bit position in the GHR. For example, branch B2 affects the rightmost bit and three other bits. Because B3 is perfectly biased we know that all the 3's must be T. The four 1s together can have five possible values, TTTN, TTNN, TNNT, NNTT, and NTTT. Considering just B1 and B3 there are $1 \times 5 = 5$ possible GHR values. To compute the number of possible patterns for branch B2 we need to consider three cases for the first 1 position: that it corresponds to a q1 (first of a group of three), a q2, or a q3. In each of those cases there are four distinct patterns. For example, in case 1 the first three 1's are from one group of three and the last 1 is from a different group of three. There are two possible outcomes for each group. For each case, one of the four patterns is all Ts and one is all Ns. The number of distinct patterns due to branch B2 is $3 \times 2 + 2 = 8$. The patterns due to B1 repeat every 5 occurrences of B3, while the patterns due to B2 sort of repeat every 3. Because 3 and 5 don't share a common factor > 1 the total number of patterns in the GHR is the product of the number of patterns due to B1 and B2. The total is $\boxed{5 \times 8 = 40}$.

Grading Note: For full credit one only needs to show an approach that will lead to the correct answer. The key insight that needs to appear is multiplying the five patterns of B1 *by something like* $a2^b$*, or some diagram that can be used to find the different GHR patterns.*

Problem 3, continued:

In this part consider the same predictors as on the previous page, except this time the BHT has 2^{14} entries. Also, consider the same branch patterns, they are repeated below, along with the address of branches B1 and B2. The branch predictors are part of a MIPS implementation.

```
0x1234: B1:  T   T   T   N   N     T   T   T   N   N    ...
0x1242: B2:   r   r   r   q   q     q   s   s   s   u    ...
        B3:    T   T   T   T   T     T   T   T   T   T   ...
```

(*b*) Choose an address for branch B3 that will result in a BHT collision with branch B1.

☑ Address for B3 that results in a collision.

Branch B1 will collide with B3 in a BHT lookup if the bit values used to index the BHT for the two branches are the same. Since the BHT has 2^{14} entries and because MIPS instructions are 4-byte aligned we will use bits 15:2 of the branch address to index (to use as a lookup address for) the BHT. The value of those bits for B1 is 0x1234 (note that each hex digit spans four bits). For B3 to use the same entry the four least significant hex digits should match. One such address is `0x11234`.

(*c*) How does the collision change the prediction accuracy of the bimodal predictor on the two branches?

☑ Change in B1 and ☑ Change in B3.

For this particular case the accuracy of B1 will improve because B3 will keep the 2-bit counter from falling below 2. This will improve B1s accuracy to $\frac{3}{5}$. The accuracy of B3 won't change. The Ns in B1 will never decrement the counter twice in a row (because of B3's outcome) and so the counter will never go below 2.

(*d*) (The answer to the following question does not depend on the sample branch patterns above.) Suppose we detect a BHT collision (perhaps by using tags). Why should we predict not taken?

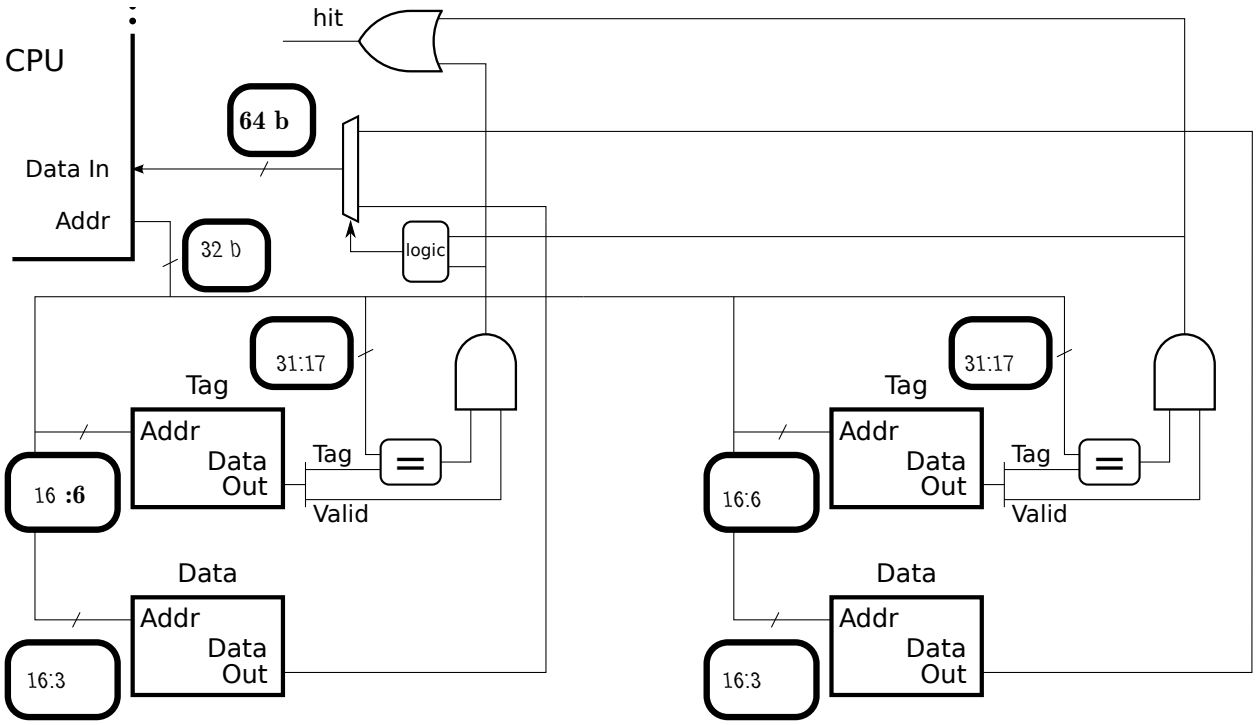
☑ Reason for predicting not-taken for a collision.

Because a collision indicates that the BHT entry we found is not for the branch we are trying to predict. Since it's for a different branch the target, which can be stored in the BHT, would very likely be wrong. So even though there's a .5 probability that a taken prediction is correct there is a much smaller chance that the target is correct.

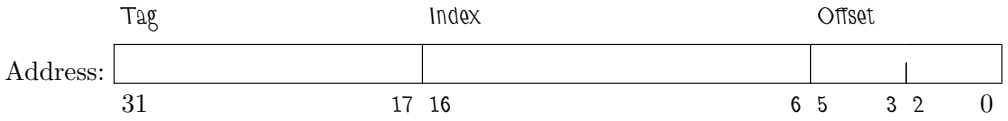
Problem 4: (15 pts) The diagram below is for a 256 kiB (2^{18} B) set-associative cache. Hints about the cache are provided in the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)



☑ Associativity:

The associativity is 2. From the diagram we can see that there are two ways. Notice that there's no ellipsis between the ways, as there would be on a diagram in which not all the ways are shown.

☑ Memory Needed to Implement (Indicate Unit!!):

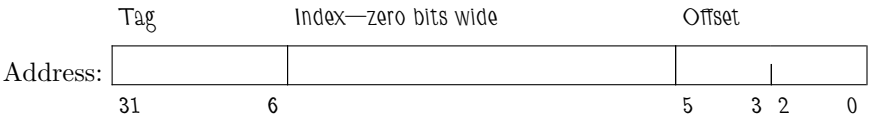
It's the cache capacity, 256 kiB bytes, plus $2 \times 2^{17-6}\,(32 - 17 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Lower bit position of the address going into the tag store gives the line size, $2^6 = 64$ characters.

☑ Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Because the cache is fully associative the number of index bits is zero. The line size doesn't change so we don't change the offset bit positions. Instead we increase the size of the tag. The bit categorization appears below.



Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 128 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
double sum = 0;
double *a = 0x2000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^7 = 128$ bytes is given. The size of an array element, which is of type double, is $8 = 2^3$ characters, and so there are $2^7/8 = 2^{7-3} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the hit ratio is $\frac{15}{16}$.

(*c*) Find the largest value for `BSIZE` for which the second `for` loop will enjoy a 100% hit ratio.

```
struct Some_Struct {
  double val;        // sizeof(double) = 8
  double norm_val;
  double a[14];
};

  const int BSIZE =     1 << 15;                    ; // <-  FILL IN
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val;
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;
```

A key fact to understand when solving this problem is that two consecutive elements, say `b[0]` and `b[1]`, will be on two consecutive lines. (That's because each element, `Some_struct`, is the size of a line. `Some_struct` contains 16 doubles [14 doubles in the array, `a`] and the size of 16 doubles is $16 \times 8 = 128\,\mathrm{B}$ which is the line size.)

The size of the cache is $4\,\mathrm{MiB}$ which works out to $\frac{4\,\mathrm{MiB}}{128\,\mathrm{B}} = 2^{22-7} = 2^{15}$ lines. A hasty student might assume the problem solved at this point, and write `BSIZE=`2^{15}, or as is shown in the solution as the equivalent C expression, `1 << 15`. In this case, haste is not being punished, that's the right answer. It's hasty because of the assumption that all 2^{15} lines could co-exist in the cache.

To determine whether they really can all be in the cache at the same time, consider the sequence of addresses broken into tag, index, and offset parts. Because the size of `Some_struct` is the same size as the cache line, we know that if the index of `&b[i].val` were x then the index of `&b[i+1].val` would be $x + 1$. Because the cache is 4-way set-associative the number of lines per way is $2^{15}/4 = 2^{13}$. That means the indices run from 0 to $2^{13} - 1 = 8191$. So when is `BSIZE=`2^{13} we will encounter each index exactly once, and may completely fill each way. If `BSIZE=`2^{14} then we would encounter each index twice, say for `i=0` and `i=32768`. Element `b[0]` might be placed in way 0 and element `b[32768]` might be placed in way 1. No problem. Since the cache is 4-way we can make `BSIZE` 2^{15} without a problem. If `BSIZE` were $2^{15} + 1$ then we would be using an index five times. The fifth time we use an index we would evict the data for element $i = 0$.

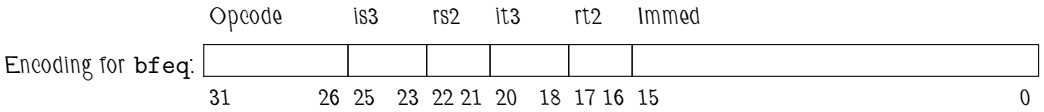
Bonus problem: What would happen if the array `a` were 30 elements? In that case the size of an element would be two lines, and if the index for element $b[i]$ were x, the index for element $b[i + 1]$ would be $x + 2$. In that case half the cache would go unused, and so `BSIZE` would have to be made half as large.

Problem 5: (5 pts) The displacement in MIPS branches is 16 bits. Consider a new MIPS branch instruction, `bfeq rsn, rtn` (branch far), where `rsn` and `rtn` are 2-bit fields that refer to registers 4-7. As with `beq`, branch `bfeq` is taken if the contents of registers `rsn` and `rtn` are equal. With six extra bits `bfeq` can branch 64 times as far.

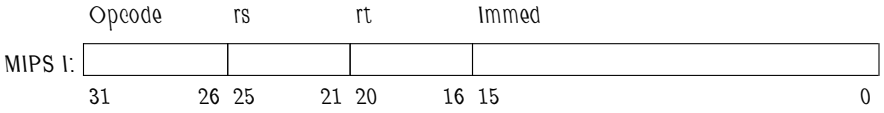
(*a*) Show an encoding for this instruction which requires as few changes to existing hardware as possible.

☑ Encoding for `bfeq`. ☑ Explain how minimizes changes.

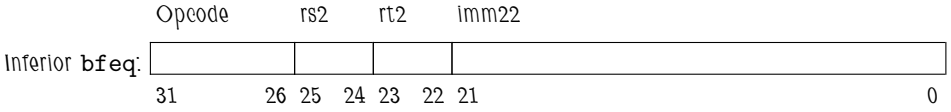
The encoding for `bfeq` is shown below. The 22-bit displacement is `is3,it3,Immed` (in Verilog notation).



In the encoding above the `rs` and `rt` register fields each have been shortened from five to two bits. (The original format I encoding is shown below.) Since the fields have been shortened but not moved the four remaining bits can be connected directly to the register file. (See the solution to the next part.)



The encoding below, which would receive partial credit, would result in more costly implementations. This inferior encoding is certainly more organized with `rs2` and `rt2` next to each other and with the immediate occupying 22 contiguous bits. However the multiplexors at the register file inputs would need to be five bits instead of three bits. Notice that it does not make a difference whether or not the immediate bits are contiguous, as they are below, or split, as they are in the solution above.

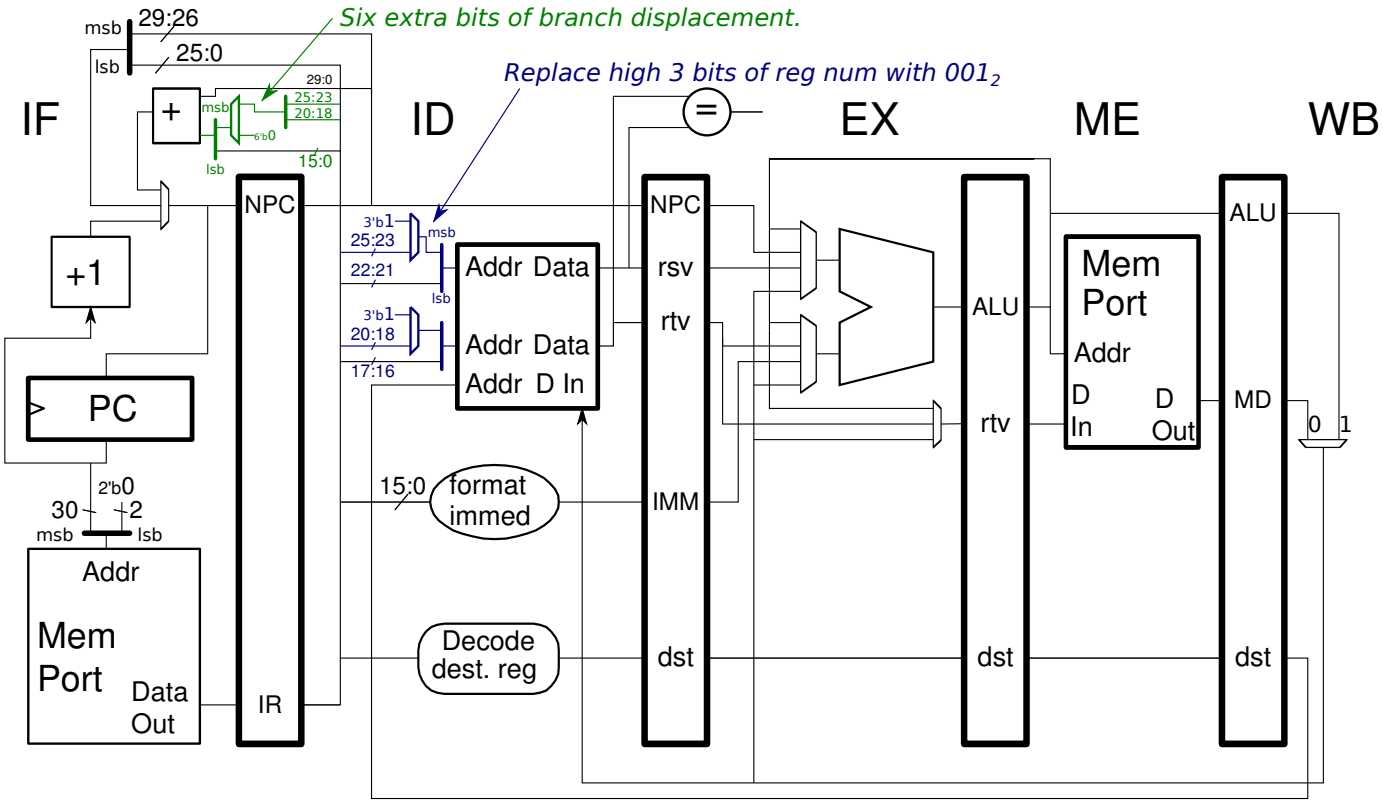


(*b*) Modify the pipeline below to implement the new instruction. Use as little hardware as possible.

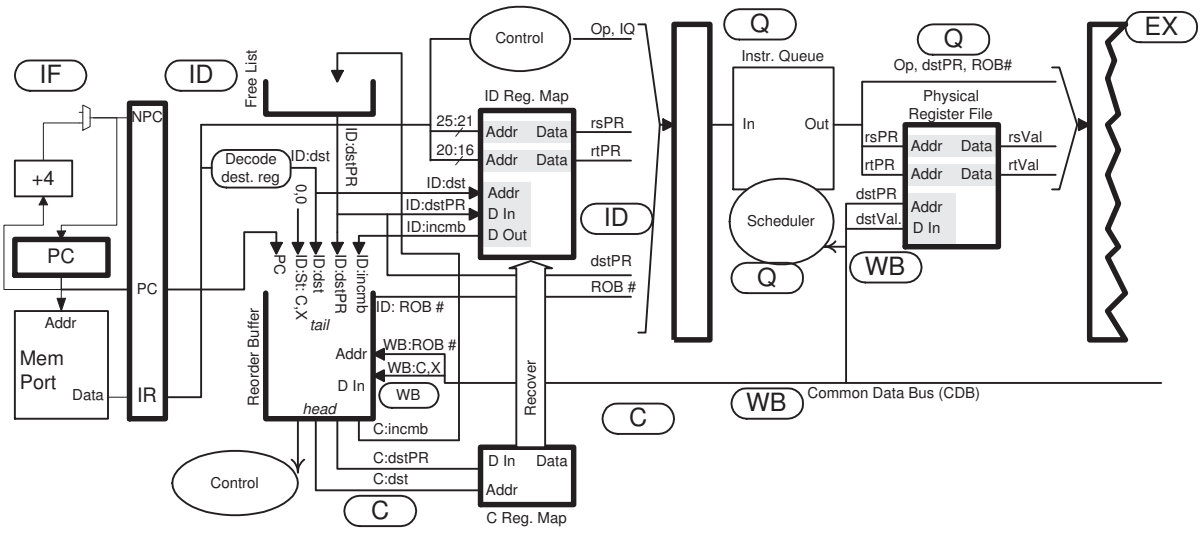
☑ Briefly show changes.

Changes appear below. At the register file address inputs, shown in blue, the high three bits of each register number is determined by a multiplexor. If a `bfeq` is present the upper inputs are used, making the upper three bits of each register number 001_2, otherwise the upper bits come from the instruction. The lower two bits are taken from the instruction regardless of whether `bfeq` is present.

If a `bfeq` is present the displacement includes the extra six bits, these changes are shown in green.



Problem 6: (10 pts) Illustrated below is a dynamically scheduled four-way superscalar MIPS implementation and the execution of code on that implementation.




```
LOOP: #   Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 lwc1 f2, 0(r1)     IF ID Q  RR EA ME WB C
 add.s f4, f4, f2   IF ID Q        RR A1 A2 A3 A4 WB C
 bne  r1, r2, LOOP  IF ID Q  RR B  WB                 C
 addi r1, r1, 4     IF ID Q  RR EX WB                 C
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 lwc1 f2, 0(r1)        IF ID Q  RR EA ME WB          C
 add.s f4, f4, f2      IF ID Q                 RR A1 A2 A3 A4 WB C
 bne  r1, r2, LOOP     IF ID Q  RR B  WB                        C
 addi r1, r1, 4        IF ID Q  RR EX WB                        C
#         Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

(*a*) On the diagram above indicate when each instruction will commit.

☑ Show commits on diagram above.

Commits shown above (they are indicated with a C). Note that commits must occur in program order and that there cannot be more than four commits per cycle. (Program order means that a commit for an instruction cannot occur before the commit for a preceding instruction.)

(*b*) What is the execution rate, IPC, for the code above for a large number of iterations assuming perfect branch prediction. Note that the system is dynamically scheduled.

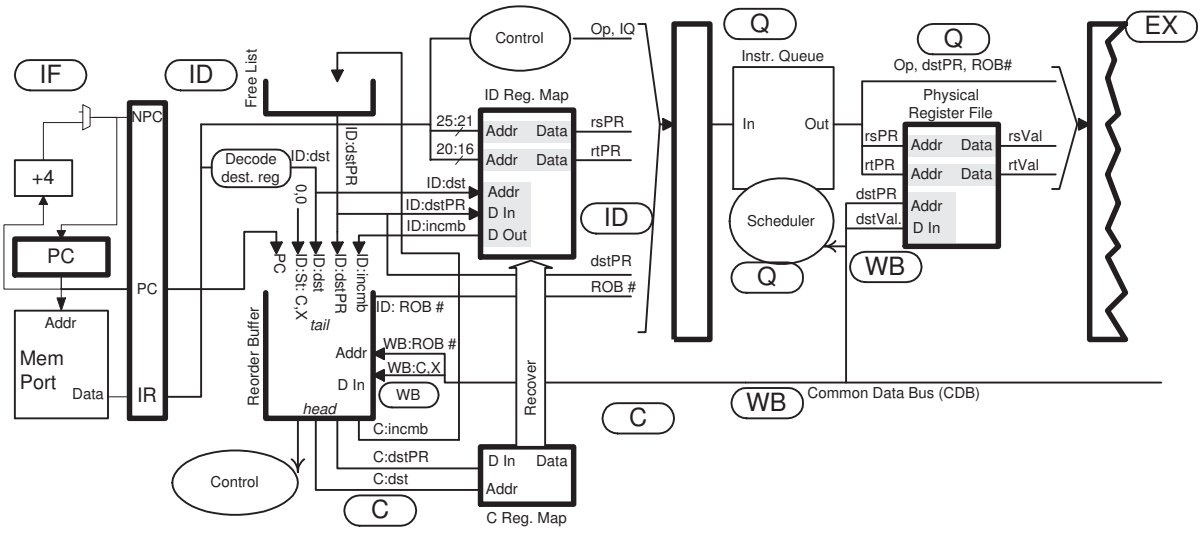
☑ IPC for code above for large number of iterations.

Short answer: the code will execute at just 1 insn/cycle due to the `add.s`.

Longer answer: Notice that one source of the `add.s` instruction, `f4`, is the result of the `add.s` instruction in the prior iteration. This means that execution can go no faster than four cycles per iteration. The sample execution (the one in the unsolved problem, not just the solution) shows that there are bypass paths so the second iteration `add.s` can start as soon as the first iteration `add.s` result is ready. This suggests that four cycles per iteration is possible. Instruction fetch and decode goes much faster, one cycle per

iteration. Eventually though the ROB will fill causing IF to periodically stall. On average each iteration will take four cycles and each iteration consists of four instructions, for an IPC of $\frac{4}{4} = 1$.

(*c*) On the next page there is a table showing the values of selected signals during the execution of the code, the signals are related to register renaming. Show values where indicated on the table. Note that `ID:incmb` is already shown in cycle 1, show its values for later cycle(s).



☑ Show values where indicated.

SOLUTION shown in blue.

```
LOOP:     Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)      IF  ID  Q   RR  EA  ME  WB  C
 add.s f4, f4, f2    IF  ID  Q           RR  A1  A2  A3  A4  WB  C
 bne  r1, r2, LOOP   IF  ID  Q   RR  B   WB                      C
 addi r1, r1, 4      IF  ID  Q   RR  EX  WB                      C
LOOP:     Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 lwc1 f2, 0(r1)          IF  ID  Q   RR  EA  ME  WB              C
 add.s f4, f4, f2        IF  ID  Q                       RR  A1  A2  A3  A4  WB  C
 bne  r1, r2, LOOP       IF  ID  Q   RR  B   WB                                  C
 addi r1, r1, 4          IF  ID  Q   RR  EX  WB                                  C
#          Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID:dstPR 0 (lwc1)          65  62          for  register  f2
ID:dstPR 1 (add.s)         97  69          for  register  f4
ID:dstPR 3 (addi)          60  79          for  register  r1
# Show values for signals below, including incmb.
#          Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID:incmb 0 (lwc1)          83  65
ID:incmb 1 (add.s)         20  97
ID:incmb 3 (addi)          67  60

ID:rsPR 0 (lwc1)           67  60

ID:rsPR 1 (add.s)          20  97

ID:rsPR 3 (addi)           67  60

ID:rtPR 1 (add.s)          65  62                                  <- rt, not rs
#          Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
WB:dstPR 0 (lwc1)                          65  62
WB:dstPR 1 (add.s)                                          97              69
WB:dstPR 3 (addi)                      60  79
#          Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

Problem 7: (25 pts) Answer each question below.

(*a*) Describe how cost and performance limit the practical largest value of width (value of n) in an n-way superscalar implementation.

☑ Cost limiter.

Bypass paths become a major element when n is large. Consider an n-way superscalar design derived from the 5-stage statically scheduled MIPS implementation used in class. The EX stage will have n 2-input ALUs, requiring a set of bypass paths for each of the $2n$ inputs. The ME and WB stage will each hold n results. The number of multiplexor inputs needed to provide bypass paths from those $2n$ results to the $2n$ inputs is $4n^2$. At 32- or 64 bits each, that will quickly dominate cost.

☑ Performance limiter.

There will be several performance limiters. One major limiter is the lack of sufficient instructions to execute in parallel. In a statically scheduled n-way superscalar design there cannot be true dependencies between the n instructions in a group to avoid stalls. If there are dependencies then the group will spend two or more cycles in ID, reducing performance. The larger n is the more frequently such dependencies will occur. Dynamic scheduling helps but does not eliminate the problem.

Another major performance limiter with large n are branches. Branches occur frequently in integer code, perhaps once every five or six instructions. In practical designs execution could only reach up to the first taken branch in a group. (In impractical [academic] designs multiple branches can be predicted per cycle and instructions can be fetched from multiple non-adjacent areas per cycle).

(*b*) What is the most important factor in determining the size of a level 1 cache?

☑ Most important factor in L1 cache size.

Clock frequency and load latency. The amount of time it takes to retrieve data from a memory is a function of its size, the larger the slower. Typically designers would set a target for the clock frequency and for the number of cycles it would take to retrieve data from the L1 cache. The largest L1 size that can meet these requirements would be chosen. For example, suppose we chose two cycles for an L1 hit (the pipeline might have two memory stages, ME1 and ME2), and suppose we chose a clock period of $0.7\,\text{ns}$. That would give us $1.4\,\text{ns}$ to retrieve data from the cache. We would choose the largest L1 size that could provide the data in $1.4\,\text{ns}$. The remaining chip area might be used for an L2 cache.

(*c*) Suppose the 16-bit offset in MIPS `lw` instructions was not large enough. Consider two alternatives. In alternative 1 the offset in the existing `lw` instruction is the immediate value times 4. So, for example, to encode instruction `lw r1, 32(r2)` the immediate would be 8. In alternative 2 the behavior of the existing `lw` is not changed but there is a new load `lws r1, 32(r2)`, in which the immediate is multiplied by 4. Note that alternative 2 requires a new opcode. Which instruction should be added to a future version of MIPS?

☑ Should choose alternative 1 or alternative 2? ☑ Explain.

Alternative 2, `lws`, should be chosen so that existing software continues to run correctly.

(*d*) The SPECcpu suite comes with the source code for the benchmark programs. How does that help with the goal of measuring new ISAs and implementations?

☑ Source code helps with testing new implementations because:

Since we have the source code we can compile the benchmarks ourselves. In fact, for SPECcpu testers are required to compile the code for themselves, using the compiler of their choosing (within reason). If we are testing a new implementation then we would want to use a compiler that can optimize for that implementation. If we are testing a new ISA then there would be no choice but to use a new compiler, since old compiler would not be able to generate code for it.

(*e*) What's the difference between a 4-way superscalar implementation (of say MIPS) and a VLIW system with a 4-slot bundle?

☑ Difference between superscalar and VLIW.

The superscalar implementation must be able to find and handle dependencies between any pair of instructions in a group (the group of 4 instructions traveling together through `ID` and beyond) and must be able to handle any instruction in any slot within a group. In contrast, the VLIW implementation need only handle a subset of possible instruction in each slot. For example, slot 0 might never have branch instruction and slot 3 might never have a memory instruction. Furthermore, dependencies between instructions in a bundle might be forbidden. The expectation is (was) that these differences would enable less expensive or higher performance VLIW implementations.

53 Spring 2013 Solutions

Name Solution

Computer Architecture
EE 4720
Final Examination

10 May 2013, 12:30–14:30 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Click Here

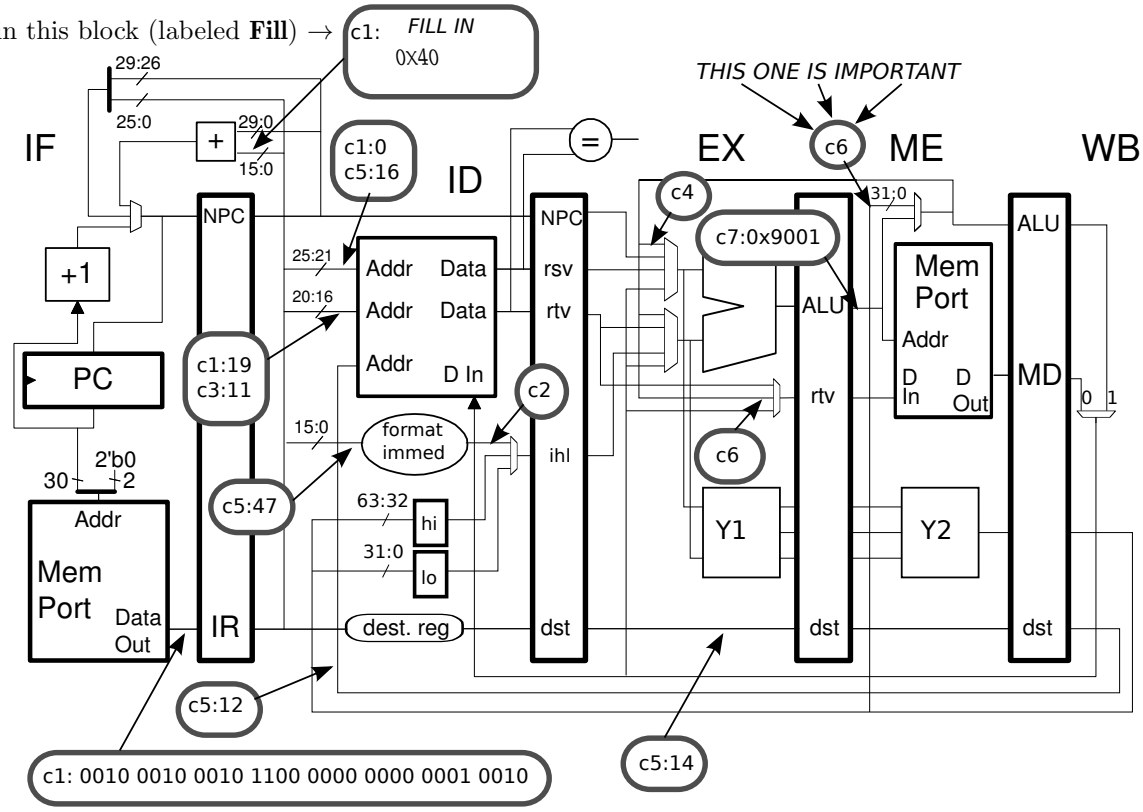
Good Luck!

Problem 1: (20 pts) The MIPS implementation below includes a two-stage integer multiply unit (`Y1` and `Y2`) for the `mult` instructions, similar to the MIPS implementation covered in Homework 4. Some wires are labeled with cycle numbers and values that will then be present. For example, c5:14 indicates that at cycle 5 the pointed-at wire will hold a 14. Other wires just have cycle numbers, indicating that they are used in that cycle. *Note that instruction addresses are provided.*

☑ Write a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Fill in this block (labeled **Fill**) →




```
# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 0x1000: beq r0, r19 TARG    IF ID EX ME WB
 0x1004: addi r12, r17, 18      IF ID EX ME WB
 0x1104: mult r12, r11             IF ID EX ME WB
 0x1108: mflo r14                     IF ID EX ME WB
 0x110c: sb   r14, 47(r16)               IF ID EX ME WB
# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Easy Source Registers: In ID the c1:0 tells us that in cycle 1 the instruction in ID (the one with address `0x1000`) uses `r0` as an `rs` source register. Similarly c5:16 tells us that the instruction at `0x110c` uses `r16` as a source. The c1:19 and c3:11 provide the `rt` registers for the instructions at `0x1000` and `0x1104`.

Easy Destination Registers: In EX the c5:14 tells us that the destination of the instruction at `0x1108` is `r14`. In WB (but appearing at the bottom of ID) the c5:12 tells us the instruction at `0x1004` has a destination of `r12`.

Easy Immediate: In ID the c5:47 tells at that the instruction at `0x110c` uses an immediate and that it is 47.

Easy Dependencies: The c4 in **EX** tells us that the first source of the instruction in **EX**, `0x1104`, uses the result of the instruction in **ME**, `0x1004`. Therefore, the first source of `0x1104` must be `r12`. Similarly the c6 in **EX** tells us that the `rt` source of `0x110c` is produced by `0x1108`; it also tells us that `0x110c` is a store instruction, the store value must come from `r14`.

Less-Common Dependency: The c6 in **ME** tells us that `0x1108` uses a result from an instruction in **WB**, `0x1004` (this bypass path can only bypass values that are to be written to the `lo` register). The `lo` register is the destination of `0x1104`, which must be a `mult` because no other instruction uses the **Y** units, and the source of `0x1108`, which must be a `mflo` because that is the only instruction that uses the `lo` register as a source. Note that the `lo` register is not shown in assembly language.

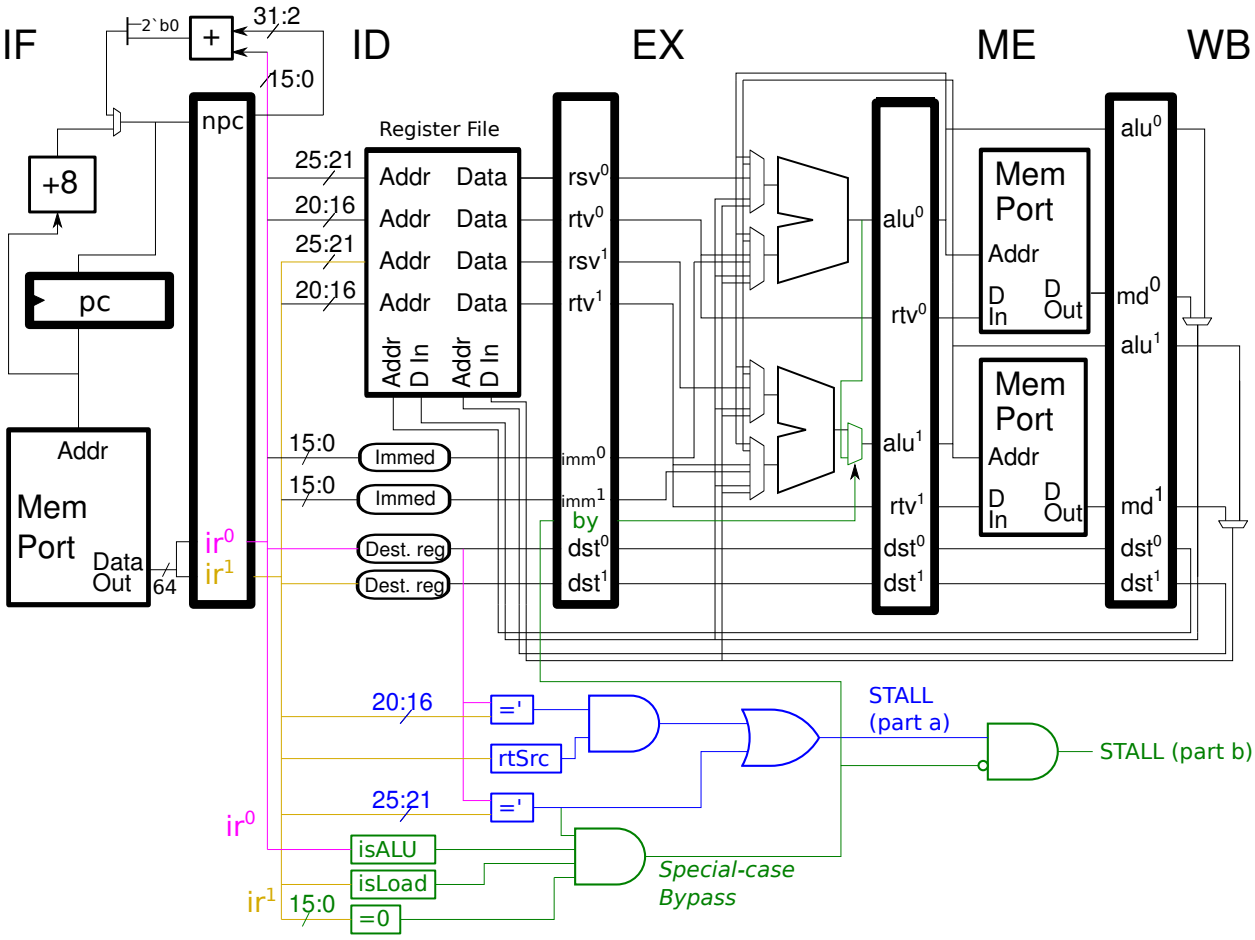
Parsing: The c2 in **ID** tells us that `0x1004` is a type I instruction. We have the encoded form of the instruction in **IF**: c1: 0010 0010 0010 1100 0000 0000 0000 0001 0010. From this we can parse out the `rs` and `rt` registers (`r17` and `r12`), as well as the immediate (18). We can also parse the opcode, 8, but one did not have to know that the opcode was for an addi instruction.

The Store Instruction: As mentioned earlier the c6 in **EX** reveals that `0x110c` is some kind of store instruction. The c7:0x9001 tells us that it cannot be a `sw` because the address of a `sw` must be a multiple of 4, similarly it can't be a `sh`. The only common store remaining is `sb` (store byte).

The `mult` and `mflo` instructions: See the less-common dependencies paragraph above.

The Branch: The pattern of instruction addresses tells us that the instruction at `0x1000` must be some kind of delayed control transfer (branch, jump, call). The c1: FILL IN in **ID** tells us it must be a branch. Let i denote the immediate field value (which is what is needed for the fill-in box. The branch target is computed by adding $4\times$ the immediate to the delay slot address, that is: `0x1104 = 0x1004 + 4i`. From that we get $i = 40_{16}$ (or `0x40`).

Problem 2: (15 pts) Illustrated below is a 2-way superscalar MIPS implementation. Design the hardware described below. You can use the following logic blocks (with appropriate inputs) in your solution: The output of logic block isALU is 1 if the instruction's result is computed by the integer ALU. The output of logic block rtSrc is 1 if the instruction uses the `rt` register as a source. The output of logic block isLoad is 1 if the instruction is a load.



(*a*) Design logic to generate a signal named `STALL`, which should be `1` when there is a true (also called data or flow) dependence between the two instructions in `ID`.

☑ Control logic to detect true dependence in `ID` and assign `STALL`.

Solution appears above in blue. The stall signal for this part is the output of the OR gate. To help understand what's going on the IR (instruction register) for slot 0 is shown in purple and the IR for slot 1 is shown in gold. The logic compares the destination of the instruction in ID slot 0 with the two sources of the instruction in slot 1. If either matches, the stall signal is generated (which before part b would be the output of the OR gate). The rtSrc logic is needed because the `rt` field might hold a destination register number, and we would not want to stall for that.

(*b*) The code fragment below should generate a stall in our two-way superscalar implementation when the two instructions are in the same fetch group. However this particular arithmetic/load pair is a special case in which the stall is not necessary when the right bypass path(s) and control logic are provided. *Hint: Something about the* `lw` *makes it a special case.*

```
0x1000: add r1, r2, r3
0x1004: lw r4, 0(r1)
```

☑ Add the bypass path(s) needed so that the code executes without a stall.

The bypass path appears at the output of the slot-1 ALU. The slot 0 ALU computes the add (for the example above) and the slot 1 ALU computes `r1 + 0`. For this special case, where the immediate is zero and we are using the destination of the slot-0 instruction, we can bypass.

☑ Add control logic to detect this special case and use it to suppress the stall signal from the first part.

The control logic appears in green. The logic checks for an ALU instruction in slot 0, a load with a zero immediate in slot 1, and a dependence between the slot-0 and -1 instructions. If all are true the special-case-bypass signal is `1`, and that is used to control the new bypass multiplexor and to suppress the stall signal.

Note that the bypass signal must go through the pipeline latch, if it were connected directly to the green EX-stage mux it would be affecting the wrong instructions.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14} entry BHT. One system has a bimodal predictor, one system has a local predictor with a *12-outcome local history*, and one system has a global predictor with a *12-outcome global history*.

(*a*) Branch behavior is shown below. The `r` outcomes for branch `B2` are random and are modeled by a Bernoulli random variable with $p = .25$ (taken probability is .25). Answer each question below, the answers should be for predictors that have already warmed up.

```
B1:  T   N   T   T   T   N    T   N   T   T   T   N    ...
B2:   T   N   T   r   T   N    T   N   T   r   T   N    ...
B3:    T   T   T   T   T   T    T   T   T   T   T   T   ...
```

☑ What is the accuracy of the bimodal predictor on branch `B1`?

By straightforward analysis (see work below) the accuarcy is $\frac{4}{6} = \frac{2}{3}$. Note that the accuracy is based on a repeating pattern (the part under the dashes). The pattern of outcomes repeat, and the counter value is the same, 2, at the beginning and the end of the repeating part.

```
SOLUTION WORK
Repeating pattern:                ------------------------------
Ctr:0  1    0   1   2   3    2   3   2   3   3   3  2
B1:  T   N   T   T   T   N    T   N   T   T   T   N    ...
Mispred:                          x                x
```

☑ What is the approximate accuracy of the bimodal predictor on branch `B2`? ☑ Explain.

Short answer: The 2-bit counter value will be 2 or 3, and so all the taken branches will be correctly predicted, for a prediction ratio of $\frac{3+0.25}{6}$.

Explanation: Notice that there never will be more than one consecutive not-taken outcome and that sometimes there are three consecutive taken outcomes (when `r` is taken). For that reason the 2-bit counter cannot be lower than 2 (after reaching a 3 during warmup). With a value of 2 or 3 it will always predict taken, that is the correct prediction for the three `T` outcomes and is correct for `r` 25% of the time. The total accuracy is then $\frac{3+0.25}{6}$.

☑ What is the minimum local history size needed to predict `B1` with 100% accuracy? Ignore branch `B2` for this question.

The minimum history size is 4 outcomes. To see why consider the six possible local histories (sorted) below. Suppose the local history size were just two outcomes, and consider the first two patterns below. If the local history were `NT` the next outcome might be `N` or `T`, and so the branch could not be predicted. Similarly three outcomes are insufficient, consider local history `TNT`. But four are sufficient, that's easy to tell in this case because every four-outcome history for this branch is different (look at the first four columns of the table below).

```
NTNTTTNT
NTTTNTNT
TNTNTTTN
TNTTTNTN
TTNTNTTT
TTTNTNTT
```

☑ What is the accuracy of the local predictor on branch `B2`, ignoring the effect of `B1`? ☑ Explain.

The local predictor will predict the non-`r` outcomes with 100% accuracy because the pattern repeats and there is enough local history. The `r` outcomes add to the number of patterns, but other than increasing warmup time they don't affect accuracy of the

non-`r` outcomes. Since `r` is biased not-taken the counter for the `r` outcomes will usually be 0 or 1. Assuming it is always 0 or 1, the accuracy or the `r` outcomes would be 75%. The overall accuracy under this assumption is $\boxed{\frac{5+0.75}{6}}$. It's easy to find the actual probability of each counter value by solving a simple Markov chain. The probability of a counter value of i is $\frac{\omega-1}{\omega^4-1}\omega^i$ where $\omega = \frac{0.25}{1-0.25}$. The probability of a zero count is .675 and a one count is .225, so the probability of predicting not-taken is .9. The probability of a correct prediction for the `r` outcome is $0.9 \times 0.75 + 0.1 \times 0.25 = .7$, slightly lower than the .75 we estimated.

☑ Describe a situation in which branch `B2` is mispredicted using the local predictor due to the effect of `B1`. The low bits of the address of `B1` and `B2` are different so there is no chance of a BHT collision. How frequently do such mispredictions occur? *Grading Note: The original exam and the Spring 2014 homework question did not include the statement about the low address bits.*

This happens when the `r` outcome is `t` two times in a row, and then is `n`. When we predict the `T` outcome (the third `r` outcome) the local history for `B2`, `tTNTNTtTNTNT`, matches a possible local history for `B1`. `B1` will have warmed up the corresponding `PHT` entry to `3`, but for branch `B2` the predicted outcome should be not taken. The chance of this happening is $\frac{1}{4}\frac{1}{4}\frac{3}{4} = \frac{3}{64}$.

Thank you to EE 4720 student Payon Huskins for providing a correction to the solution above on 4 May 2015. (The version above has the correction applied.)

The original exam and the Spring 2014 homework assignment did not have the statement about the low address bits. The following explains why a BHT collision could not cause a misprediction, meaning that the only the answer above is correct, with or without the low-address-bits statement.

If bits 15:2 of `B1` and `B2` were identical the two branches would share a `BHT` entry and so their local histories would be intermingled (in the same way local histories are intermingled in the `GHR`). The positions of the `N`'s in the local history would ensure that different `PHT` entries were used for the two branches, and so there is no chance of `B1` affecting `B2`. Even with the intermingling there would be enough local history to predict `B1` and `B2` with their respective maximum accuracies.

☑ How many possible GHR values will be present when predicting branch `B3`? It's okay to show patterns that include `r`'s, but indicate how many of each pattern there are.

Consider the following global history when predicting `B3`: `TttTtrTttTnn`. The upper-case outcomes are `B3`, the lower-case outcomes are `B1` and `B2`. Because there is an `r` this pattern represents two possible global histories. Pattern `B1` can be in six possible rotations, for four of them the `r` is in the global history, so the total number of global histories when predicting `B3` is $\boxed{2 \times 4 + 2 = 10}$.

Problem 3, continued: Recall that the local predictor uses a 12-outcome local history and a 2^{14}-entry BHT.

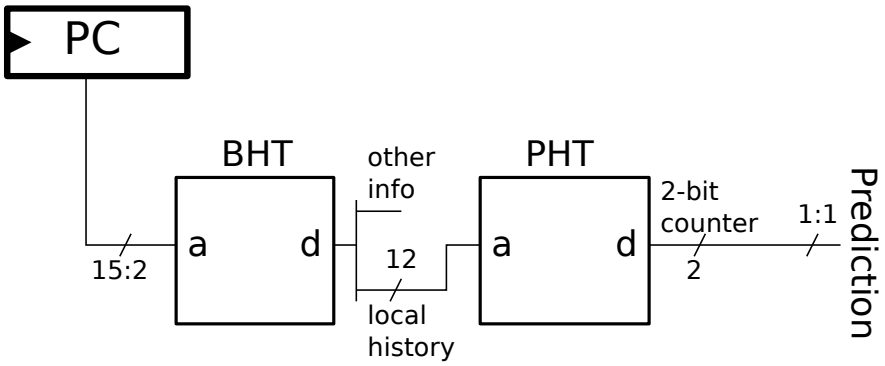
(*b*) Draw a diagram of the local predictor hardware used for making a prediction but omit the hardware for updating the predictor. The input to your hardware should be `PC`, and output should be a 1-bit quantity (`0` for not-taken, `1` for taken).

- Be sure to show the BHT and PHT.
- Assume that `PC` is the address of a branch. (That is, don't worry about detecting non-branch instructions.)
- Don't predict the target, just the direction (taken or not-taken).

☑ Local predictor hardware for predicting branch direction.

☑ Label wires with bit ranges (*e.g.* 31:26) or number of bits, as appropriate.

Solution appears below.



(*c*) How much memory is needed to implement the local predictor. Only include storage for predicting the branch direction, do not include storage for predicting the branch target.

☑ Storage needed to implement local predictor. ☑ Indicate unit.

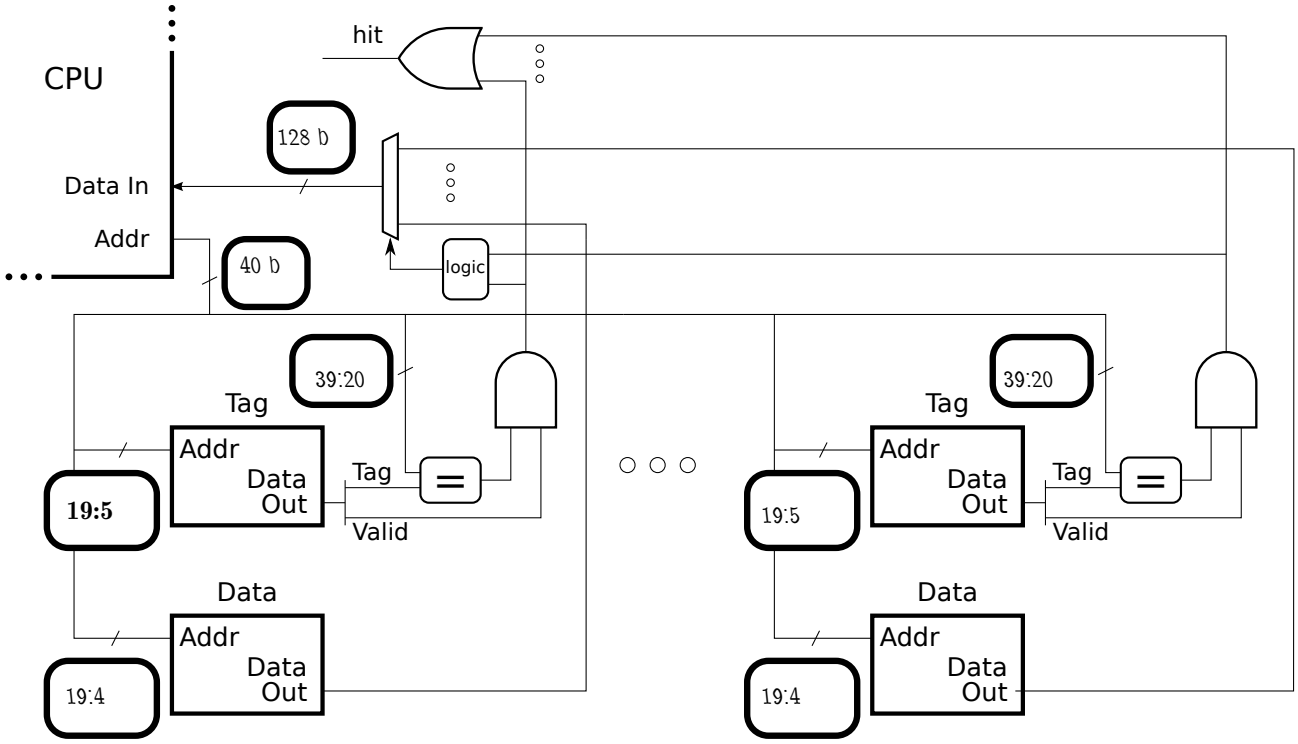
In real life [tm] a BHT entry for a local predictor might have a field for identifying the type of instruction (branch, jump, non-CTI, *etc.*), a tag, the CTI target, and of course the local history. As requested, we will only look at the local history. The size of the local history is 12 bits, and there are 2^{14} entries. The PHT has 2^{12} entries (one for each possible local history) and each entry is 2 bits.

The total storage is thus $\boxed{2^{14} \times 12 + 2^{12} \times 2 \text{ bits}}$.

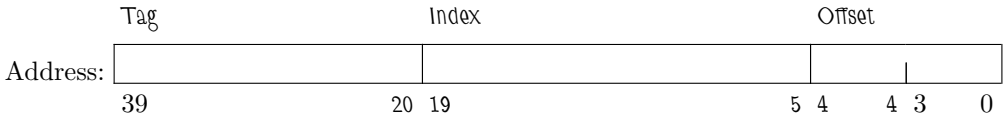
Problem 4: (20 pts) The diagram below is for an 8-way set-associative cache. Hints about the cache are provided in the diagram and also in the address bit categorization just below the diagram.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Complete the address bit categorization. Label the sections appropriately. (Index, Offset, Tag.)



☑ Cache Capacity (Indicate Unit!!):

The cache capacity can be determined from the following pieces of information: The lowest tag bit position is 20, which means that the size of the combined index and offset is 20 bits, and so each data store holds 2^{20} characters. Nothing was said about the character size and so we can assume that it is 8 bits, which we will call a byte. We were told that the cache is 8-way set associative, and so there are 8 data stores, and so the cache capacity is $8 \times 2^{20} = 8\,\text{MiB}$.

☑ Memory Needed to Implement (Indicate Unit!!):

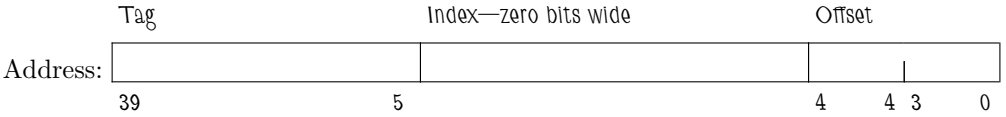
It's the cache capacity, 2^{23} bytes, plus $8 \times 2^{20-5}\,(40 - 20 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Lower bit position of the address going into the tag store gives the line size, $2^5 = 32$ characters.

☑ Show the bit categorization for a **fully associative** cache with the same capacity and line size.

Because the cache is fully associative the number of index bits is zero. The line size doesn't change so we don't change the offset bit positions, instead we increase the size of the tag. The bit categorization appears below.



Problem 4, continued: The problems on this page are **not** necessarily based on the cache from the previous page. The code in the problems below run on a 4 MiB (2^{22} byte) 4-way set-associative cache with a line size of 256 bytes.

Each code fragment starts with the cache empty; consider only accesses to the arrays.

(*b*) Find the hit ratio executing the code below.

```
Complex sum = {0,0};
Complex *a = 0x2000000; // sizeof(Complex) == 16  Each element loaded with 1 load insn.
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

☑ What is the hit ratio running the code above? Show formula and briefly justify.

The line size of $2^8 = 256$ bytes is given. The size of an array element, of type complex, is $16 = 2^4$ characters, and so there are $2^8/16 = 2^{8-4} = 2^4 = 16$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, and so the next $2^4 - 1 = 15$ accesses will be to data on the line, hits. The access at `i=16` will miss and the process will repeat. Therefore the hit ratio is $\frac{15}{16}$.

(*c*) Find the smallest value for `ASIZE` that will minimize the hit ratio (make things as bad as they can get) of the code below.

```
struct Some_Struct {
  double val;        // sizeof(double) = 8
  double norm_val;
  double a[ASIZE];  };

  const int BSIZE = 1 << 10;
  Some_Struct *b;
  for ( int i = 0;  i < BSIZE;  i++ ) sum += b[i].val;
  for ( int i = 0;  i < BSIZE;  i++ ) b[i].norm_val = b[i].val / sum;
```

☑ Smallest value of `ASIZE` to minimize hit ratio:

This problem requires some thought.

When `ASIZE` is zero the element size is 16 and a 256-byte line can hold 16 elements. The first `for` loop will enjoy a $\frac{15}{16}$ hit ratio (see the previous part). The second loop, because the cache size is 4 MiB, will have a 100% hit ratio. As we increase `ASIZE` from 0 the hit ratio first drops due to the loss of spatial locality. The drop in hit ratio will temporarily level off when `sizeof(Some_Struct)` is equal to the line size. That occurs when $(2 + s)8 = 256$, where s is `ASIZE`, the value is 30. With this value the hit ratio in the first `for` loop is zero, but the hit ratio in the second `for` loop is still 100%. This answer, `ASIZE=30`, would only get partial credit because a smaller hit ratio is possible.

Increasing `ASIZE` to, say, 40, won't reduce the hit ratio from its `ASIZE=30` value, but there is a point where it will start dropping again. If `ASIZE` were $2^{17} - 2$ then the structure size would be $(2 + 2^{17} - 2)8 = 2^{20}$ bytes. For this cache two consecutive elements of the structure would have the same index but different tags. Since the cache is 4-way it could only hold no more than four elements, which means the second `for` loop would have a hit ratio of only 50% (it's not zero because a miss to the `val` member brings in the data for the `norm_val` member). This answer, `ASIZE=` $2^{17} - 2$ would only get partial credit because it's possible to get the same hit ratio with a smaller value of `ASIZE`.

With `ASIZE` set to $2^{17} - 2$ the cache can only hold four elements (the part with `val` and `norm_val`). Suppose the cache could only hold `BSIZE/2` elements (or for that matter, `BSIZE-1` elements). Then the hit ratio of the second `for` loop would be just 50%, the minimum value. The minimum `ASIZE` is for the case where eight elements have the same tag: if more than eight elements

have the same tag (as in the previous paragraph) then `ASIZE` is larger than it has to be. If fewer than eight (four is the next smaller value) have the same tag then there will be no misses in the second `for` loop. If eight elements have the same tag then there must be `BSIZE/8=128` different indices. That means $128(s+2)8 = 2^{20}$. Solving yields $s = 2^{10} - 2 = 1022$, the full-credit value of `ASIZE`.

Problem 5: (25 pts) Answer each question below.

(*a*) Indicate whether each feature below is a feature of an ISA or the feature of an implementation.

☑ Number of stages in pipeline. *Circle One*: ISA or [Implementation]

☑ Number of integer registers. *Circle One*: [ISA] or Implementation

☑ Number of bits in an integer register. *Circle One*: [ISA] or Implementation

☑ Whether an adjacent pair of instructions, such as `add r1,r2,r3; sub r4,r1,r2` will generate a stall. *Circle One*: ISA or [Implementation]

(*b*) Describe the problem with each of the following MIPS code fragments.

☑ Problem with fragment below:

```
addi r1, r2, 0x123456
```

The immediate value is too large, it must be limited to 16 bits. Fixed code appears below:

```
# SOLUTION
lui r1, 0x12
ori r1, r1, 0x3456
add r1, r2, r1
```

☑ Problem with code execution on our FP pipeline.

```
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
sub.s f4, f1, f5      IF ID A1 A2 A3 A4 WF
```

There is a dependency carried by `f1`, the `sub.s` should have stalled. The correct execution appears below.

```
# SOLUTION
add.s f1, f2, f3   IF ID A1 A2 A3 A4 WF
sub.s f4, f1, f5      IF ID -------> A1 A2 A3 A4 WF
```

☑ Problem with code execution on our 5-stage pipeline.

```
lw r1, 2(r3)      IF ID EX ME WB
beq r1, r4 TARG      IF ID EX ME WB
xor r6, r7, r8          IF ID EX ME WB
TARG:
add r9, r10, r11           IF ID EX ME WB
```

There is a dependence between the `lw` and `beq` carried by `r1`, the branch should stall.

(*c*) Consider the following two equal-cost design options:

A dual-core chip, each core is 4-way superscalar and dynamically scheduled.

A 16-core chip, each core is 2-way superscalar and statically scheduled.

Both have the same clock frequency.

☑ What is the peak execution rate of each chip, in IPC?

For the dual core the rate is $2 \times 4 = 8$ insn/cycle.

For the 16-core chip the rate is $16 \times 2 = 32$ insn/cycle.

☑ Describe a situation in which the dual-core chip is a better choice than the 16-core chip.

If we are running code with only one thread the dual core can execute it at up to 4 insn/cycle, but the second can only reach a peak of 2 insn/cycle.

☑ Describe a situation in which the 16-core chip is a better choice than the dual-core chip.

If a program had 16 threads, and there were at most minor inefficiencies due to parallelization, the 16-core chip would be the better choice. Or, you needed to run 16 different programs at the same time. Either way, the 16-core chip is better because of its higher peak.

(*d*) Consider a $5n$-stage implementation created by splitting each stage of a 5-stage implementation into n pieces.

☑ How important is it to have a higher clock frequency in the $5n$-stage design (compared to the 5-stage design)? ☑ Explain.

It's very important. Both the original and $5n$ stage pipeline have a peak execution rate of 1 insn/cycle, so if the clock frequency isn't higher there is no gain in performance.

(*e*) When an instruction raises an exception execution will switch to a trap handler.

☑ How is the trap handler address determined for MIPS?

The MIPS ISA specifies a handler address for each category of interrupt.

☑ How is the trap handler address determined for SPARC?

SPARC has a special register called the Trap Base Register (TBR). The TBR has the address of the start of what's called a trap table. Hardware will generate a trap type, the value of the trap type is determined by the kind of exception. The trap type is combined with the value in the TBR to form the handler address.

54 Spring 2012 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

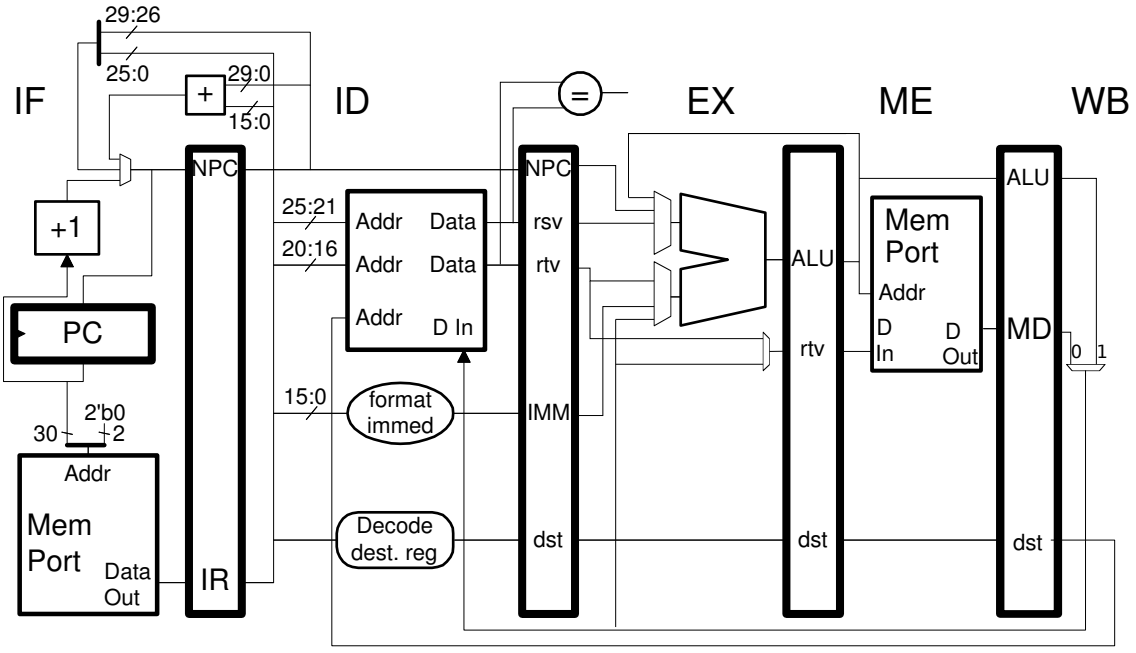
Friday, 23 March 2012, 9:40–10:30 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias A Century of Turing

Good Luck!

Problem 1: [20 pts] The MIPS code below executes on the illustrated **seemingly** familiar implementation. If you look closely you'll notice that certain bypass paths that are present in the five-stage implementation usually used in class are missing from the diagram below.



(*a*) Show the execution of the code below on the illustrated implementation for enough iterations to determine CPI. Determine the CPI.

☑ Execution diagram.

☑ Double-check for bypass paths, stall when necessary.

☑ Compute CPI.

The pipeline execution diagram appears below.

Note the following close dependencies: `lw/add` through `r2`, `add/sw` through `r1`, and `addi/lw` (in the 2nd iteration) through `r4`.

The `add` stalls two cycles (2 and 3) due to the `lw/add` dependence, so that when the `add` can read the value of `r2` when it is in `ID` (bypassed through the register file). In the bypassed implementation used in class the `add` would only stall one cycle, since the value of `r2` could be bypassed from `WB` to `EX`, but in the implementation above there is no bypass from the `WB` stage to the upper input to the ALU, where the `rs` value is needed.

Due to the `add/sw` dependence the `sw` stalls one cycle (5), because the `ME`-to-`EX` bypass to the store value multiplexor (the one connecting to `EX/ME.rtv`) is missing. With that one-cycle stall the `add` is in `WB` while the `sw` is in `EX`, where the `WB`-to-`EX` bypass to the store value multiplexor can be used. In the bypassed implementation used in the class notes there would be no stall here.

The `addi/lw` dependence does not cause a stall because there is a bypass from `ME` to `EX` going to the upper ALU mux.

Common Mistakes: Often beginners forget that the store value, `r1` in this case, is a source value. In this case it doesn't matter, but another common mistake is forgetting that the branch operands are sources, and that these sources cannot be bypassed to in most of the implementations used in class. People in a hurry sometimes mistake the first branch operand as an output.

To compute the CPI, divide the iteration time by the number of instructions. The first iteration starts at cycle 0 (the iteration starts when the first instruction of the loop body is in `IF`), the second at cycle 8, and so the first iteration takes 8 cycles. Since the first

and second iterations will have the same stalls we can conclude that all subsequent iterations will take 8 cycles. An iteration has five instructions, so the CPI is $\frac{8}{5} = 1.6\,\mathrm{CPI}$.

```
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lw  r2, 0(r4)     IF ID EX ME WB
 add r1, r2, r3       IF ID ----> EX ME WB
 sw r1, 4(r4)            IF ----> ID -> EX ME WB
 bne r4, r5  LOOP                IF -> ID EX ME WB
 addi r4, r4, 8                        IF ID EX ME WB
LOOP: # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 lw  r2, 0(r4)                            IF ID EX ME WB
```

Problem 1, continued:

(*b*) The code above should have encountered at least one of the missing bypass paths. Choose one of those missing bypass paths and design control logic for it. The `ID`-stage control logic should generate a 1-bit signal `STALL` that will be logic `1` when the instruction in `ID` will have to stall because of the missing bypass path.

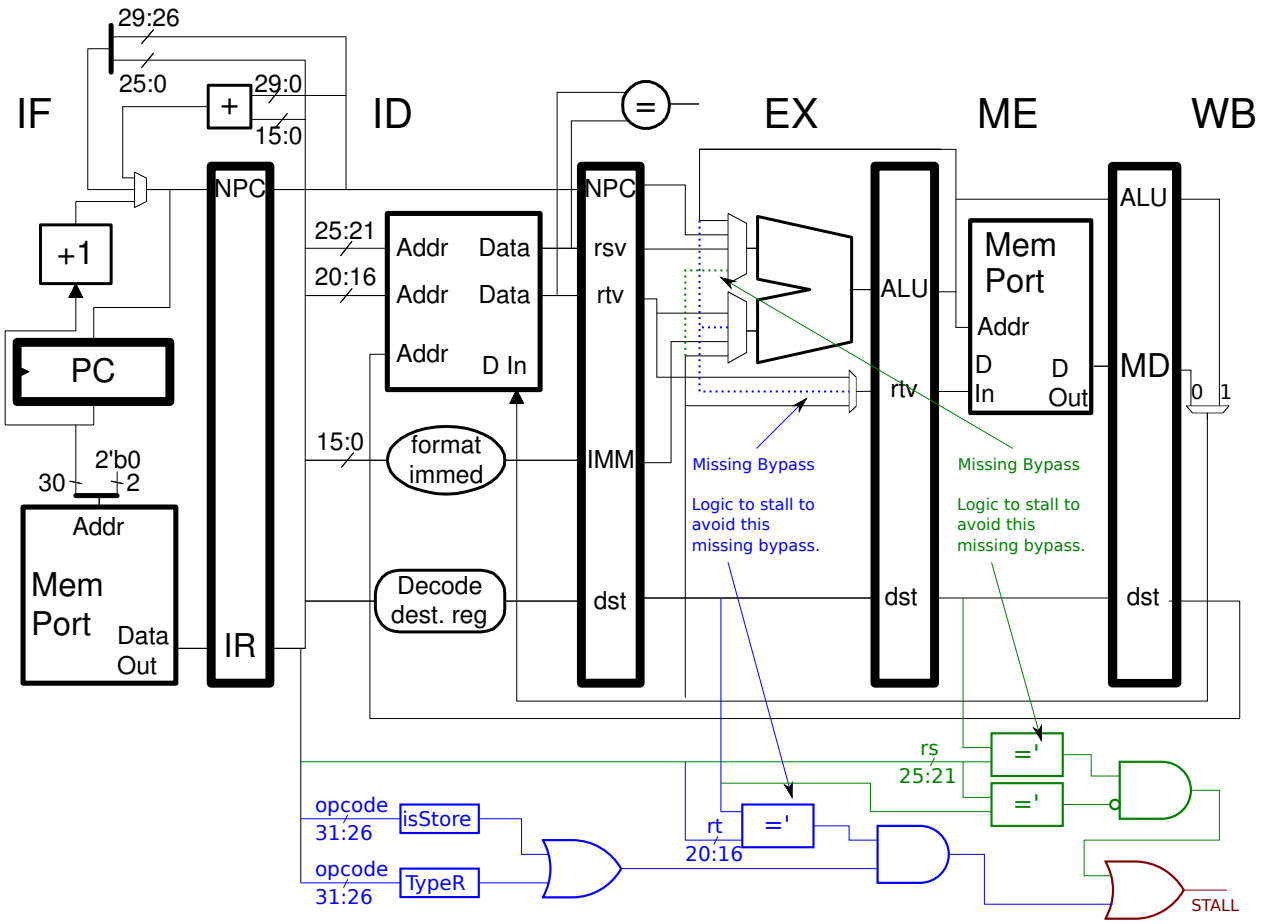
☑ Show which missing bypass path the control logic is for.

☑ Control logic for missing bypass.

Solution appears below, and includes logic for both bypass paths. The logic for the `lw/add` dependence appears in green, and logic for the `add/sw` dependence appears in blue. The stall signal appears in maroon.

The logic for the `lw/add` dependence will stall for any instruction that needs a value bypassed from the `WB` stage to the upper ALU input. For the `lw/add` case, this logic will generate the stall signal in cycle 3, but not in cycle 2. We assume that some other logic generates the cycle-2 stall. This logic assumes that every instruction uses an `rs` source (and most do).

The logic for the `add/sw` dependence will stall for any instruction that needs a value bypassed from the `ME` stage for the lower ALU input. The logic detects an instruction that uses the `rs` value as a source, assuming any Type R instruction and store instructions. The stall is generated if such an instruction is present in `ID` and if the instruction in `EX` writes the same register as the `ID` instruction's `rt` source.



Problem 2: [15 pts] Consider a MIPS implementation in which the memory port is split into two stages, `Ma` and `Mb`. With this change the clock frequency can increase from 1 GHz to 1.2 GHz; with this added stage our implementation is now six stages. A sample execution appears below.

```
add r2, r3, r4 IF ID EX Ma Mb WB
lw r1, 4(r2)      IF ID EX Ma Mb WB
```

(*a*) Ignoring the cost of `Ma` and `Mb`, how is the cost of the pipeline changed in this design? Consider bypass paths.

☑ Cost change, ignore `Ma` and `Mb`, consider bypass, etc.

There is another pipeline latch, which includes an **ALU** register (32 bits) and a **dst** register, 5 bits, plus control bits. The **ALU** value needs to be bypassed to the **EX** stage, the cost of that is 32 bits times three mux inputs (the two at the ALU inputs, and the one for the store value).

(*b*) Provide an example of a code fragment which will execute more slowly with the six-stage pipeline. Assume that all reasonable bypass paths are provided.

☑ Code fragment that will run more slowly.

The code fragment below runs more slowly, that's because of the dependence of the **add** on the **lw**, called a *load/use* pair.

```
# SOLUTION
# Cycle           0  1  2  3  4  5  6  7  8
lw r1, 0(r2)      IF ID EX Ma Mb WB
add r3, r1, r3       IF ID ----> EX Ma Mb WB
```

(*c*) Provide an argument that this change is good.

☑ This change is good because:

The change will add one stall cycle to a load/use pair, as in the example above. Suppose 1 out of every 3 instructions was a load followed by an instruction that uses the loaded value, also suppose that there are no other stalls. The ideal execution time for 300 million instructions is 300 million cycles. If 1 out of 3 is a load/use, then there are 100 million load/uses which will add 100 million stall cycles on the five-stage implementation and 200 million stall cycles on the six-stage implementation. Then the execution time for 300 million instructions would increase from 400 million cycles $400 \times 10^6 \frac{1}{1\,\text{GHz}} = 400\,\text{ms}$ to 500 million cycles which is $500 \times 10^6 \frac{1}{1.2\,\text{GHz}} = 416\,\text{ms}$, which is slower.

But, it's likely that far fewer than 1 out of 3 instructions will be the annoying load/use case, suppose it is 1 out of 10. Then with the ordinary **ME** stage execution is $330 \times 10^6 \frac{1}{1\,\text{GHz}} = 330\,\text{ms}$ and with the split stage $360 \times 10^6 \frac{1}{1.2\,\text{GHz}} = 300\,\text{ms}$ which is faster.

Note that the extra stage does not slow things down when there is no stall, as in the example below.

```
# Cycle           0  1  2  3  4  5  6  7  8
lw r1, 0(r2)      IF ID EX Ma Mb WB
addi r2, r2, 4       IF ID EX Ma Mb WB
xor r5, r6, r7          IF ID EX Ma Mb WB
add r3, r1, r3             IF ID EX Ma Mb WB
```

Problem 3: [15 pts] Answer the following questions about a `blt` instruction.

(*a*) MIPS lacks an instruction such as `blt r1, r2 TARG` (branch if `r1` less than `r2`). For the questions below, which ask about the suitability of such an instruction, consider implementations similar to the five-stage pipeline covered in class.

☑ Explain why adding such an instruction would slow such implementations even though `beq r1,r2 TARG` is okay.

Because of the time needed to do a magnitude comparison, which starts only after register values are retrieved.

☑ Using a PED, explain why there would be no problem adding a `blt` instruction if the ISA had two delay slots.

Because then the branch could be resolved in EX without penalty, that would give plenty of time for the magnitude comparison. For example, consider the execution of the two code fragments below. The first is for a one-delay–slot MIPS, with a `blt` instruction added. If we want to avoid squashing or stalling for a taken branch, the branch target must be at the input to the PC when the branch is in ID, the end of cycle 2 below. That leaves only one cycle to retrieve `r1` and `r2` from the register file and perform a magnitude comparison. The second code fragment is for a hypothetical two-delay-slot MIPS also with the `blt` instruction. Because there are two delay slots the branch target address does not need to be at the input of the PC until the end of the cycle when the branch is in EX, cycle 2 in the example. That gives the implementation two cycles to retrieve the register values and perform the comparison.

Note that in the example below the second delay slot is filled with a useful instruction. That won't happen all the time for real code.

```
# SOLUTION - Execution on a one-delay-slot MIPS that includes blt.
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 addi r8, r8, 1    IF ID EX ME WB
 blt r1, r2  TARG     IF ID EX ME WB
 add r4, r5, r6          IF ID EX ME WB
 lw r10, 4(r12)

TARG:
 xor r7, r8, r9             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18

# SOLUTION - Execution on a hypothetical two-delay-slot MIPS.
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 blt r1, r2  TARG  IF ID EX ME WB
 add r4, r5, r6       IF ID EX ME WB
 addi r8, r8, 1          IF ID EX ME WB
 lw r10, 4(r12)

TARG:
 xor r7, r8, r9             IF ID EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```

(*b*) The code fragment below includes the hypothetical MIPS instruction, `blt`.

```
# Hypothetical MIPS Instruction
blt r1, r2, targ
xor r4, r5, r6
```

☑ Show an equivalent MIPS code fragment without using `blt`.

Solution appears below, in which a `slt` instruction is used to perform the magnitude the was performed by the `blt`.

```
# SOLUTION
slt r3, r1, r2
bne r4, r0, targ
xor f4, r5, 46
```

☑ Show an equivalent SPARC V8 code fragment.

Solution appears below. SPARC branch instructions use the value of the integer condition code register, `icc`, to determine whether to take the branch. The `icc` is written by `cc` versions arithmetic and logical instructions, their mnemonics end with `cc`, a common example is `subcc`.

```
# SOLUTION
subcc r1, r2, g0   # g0 = r1 - r2;  icc bits (Negative,oVerflow,Zero,Carry) also set
blt targ           # Branch if negative.
xor r5, r6, r4
```

Problem 4: [20 pts] For the ISA design tradeoff questions below consider the following data: A typical program executes 10^{10} dynamic instructions, of these 1.38×10^9 are branches (half of which are taken), 9.12×10^7 are indirect jumps (`jr` and `jalr`), 2.22×10^8 are direct jumps (`j` and `jal`).

Consider a debate by those developing the MIPS ISA: don't include format J, which means no `j` and no `jal` instructions.

(*a*) Which MIPS instructions can be used to replace `j` and `jal` in the code below, paying attention to the hints in the target names?

☑ Replacement for `j` and `jal` in code below, paying attention to target names.

The solution appears below. The `j` can be replaced by a `beq` instruction that uses a condition that will always be true. The use of a branch was possible because the target of the `j` was not every far away, and so the displacement would fit within the branch's 16-bit displacement (immediate) field.

A branch is not a suitable replacement for the `jal` for two reasons: the `jal` needs to save a return address and because the target is too far away (the jump uses a 26-bit field to store the target, which could be too distant for the branch's 16-bit field). For that reason a `jalr` instruction is used, along with two instructions to load the target address. The `upper_half` and `lower_half` assembler macros (made up) return the corresponding parts of the target address. MIPS assembler programs can use the synthetic `la` instruction to replace the `lui` and `ori` instructions.

```
# SOLUTION
 beq r0, r0, NOT_TOO_FAR_AWAY
 xor r1, r2, r3

 lui r4, upper_half(A_FAR_AWAY_PROCEDURE)
 ori r4, r4, %lower_half(A_FAR_AWAY_PROCEDURE)
 jalr r4
 xor r1, r2, r3
```

(*b*) Determine how much longer (as a percentage or absolute time) it would take to run the typical program described above without the format J instructions. Assume execution is always at 1 CPI. (A formula is okay.)

☑ Percent increase in execution time without format J:

Assume that 25% of the direct jumps are `j` instructions that can be replaced by a single branch. The remainder of the direct branches must be replaced by a set of three instructions. No changes are made to indirect jumps or branches. The increase in the number of cycles will be $2 \times 0.25 \times 2.22 \times 10^8 = 1.11 \times 10^8$. Expressed as a percentage, that is $100\frac{1.11 \times 10^8}{10^{10}} = 1.11\%$.

(*c*) What parts of out implementation would be unnecessary without format J? Approximately how many bypass paths could you add with the cost saved by not having the format J instructions?

☑ What parts would be eliminated?

☑ How many bypass paths could be added with cost savings?

The connection from the `ii` field, which goes to the `IF`-stage mux.

Problem 5: [15 pts] Answer the following ISA design questions.

(*a*) Suppose that *one of the first* modern computer designers, working in the 1940s, made the following statement: "Let's define an 'Instruction Set Architecture' [the last three words spoken slowly, for emphasis] and then later design some hardware 'im-ple-men-ta-tions' of that architecture." Would that be a good idea? Explain. *Hint: Pay attention to the slanted words.*

☑ ISA before implementation in 1940s, good or bad? Explain.

No, because computer engineers did not have enough experience to complete the final design of an ISA before working on its implementation. Lacking this experience, they might have included features in the ISA which would not be practical to build.

Grading note: Many answers gave the usual benefits for separating ISA design from implementation. Some tried to use features of 1940's technology in their answers, such as costliness. Cost is always an issue. The key difference is in the amount of computer design experience of the first computer engineers.

(*b*) CISC ISAs have variable-length instructions.

☑ Benefit of variable-length instructions for CISC.

Allows for small programs. Allows for instructions with long immediates.

☑ Benefit of fixed-length instructions for RISC.

The logic needed to fetch and decode instructions is simple because the address of the next instruction is the current instruction plus (usually) four. Branch displacements can be in units of instructions giving them a four-times longer reach, per bit.

Problem 6: [15 pts] Answer each question below.

(*a*) In our π program demo we found that turning optimization on reduced execution time by half (yay!) but reduced the instruction count even further, from 325 million to 100 million dynamic instructions. This means that optimization *increased* (made worse) CPI from 2.77 to 4.93. *Note: The data is from an earlier semester, so the numbers won't exactly match.*

☑ Optimization is supposed to eliminate stalls, why did the CPI go up?

The divide instructions by far had the longest latency and were responsible for the most stalls. The optimized code had the same number of divides as the unoptimized code, but many fewer instructions overall, and so CPI went up.

☑ What category of instruction was completely eliminated from the loop body of the π program by optimization.

Loads and stores. They were unnecessary because there were enough registers for the values that were being loaded and stored.

(*b*) SPECcpu currently has two sets of results, *base* and *peak*. Consider a third set, *shrink-wrap*, for people who buy mass-marketed software (say, buying a word processor at an office supply store).

☑ Who should use the peak numbers and who should use the base numbers?

Base: Those who plan to use code in which an ordinary effort at optimization will be made. Peak: Those using code developed with a heroic effort put into optimization.

Grading Note: Many answers implied that it was the buyers of software who were responsible for setting up the compile switches. The answers should have indicated that it was those developing software for such buyers who would have to choose compiler switches.

☑ Explain how the run and reporting rules might be modified for the new shrink-wrap results.

The key difference is that shrink-wrap software cannot be custom compiled for your particular implementation (unless you are a really important person, if so send LSU lots of money and we'll be happy to name something after you). Therefore the rules would have to forbid setting switches for the particular implementation on which the benchmarks will be run. Writing a rule to that effect would be tricky, perhaps: The compile switches shall not identify the system being tested for purposes of optimization and the compiler shall not generate the same code as would be generated when specifically targeting the implementation.

Name Solution

Computer Architecture
EE 4720
Final Examination

8 May 2012, 12:30–14:30 CDT

Problem 1		(15 pts)
Problem 2		(10 pts)
Problem 3		(10 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Problem 6		(30 pts)
Exam Total		(100 pts)

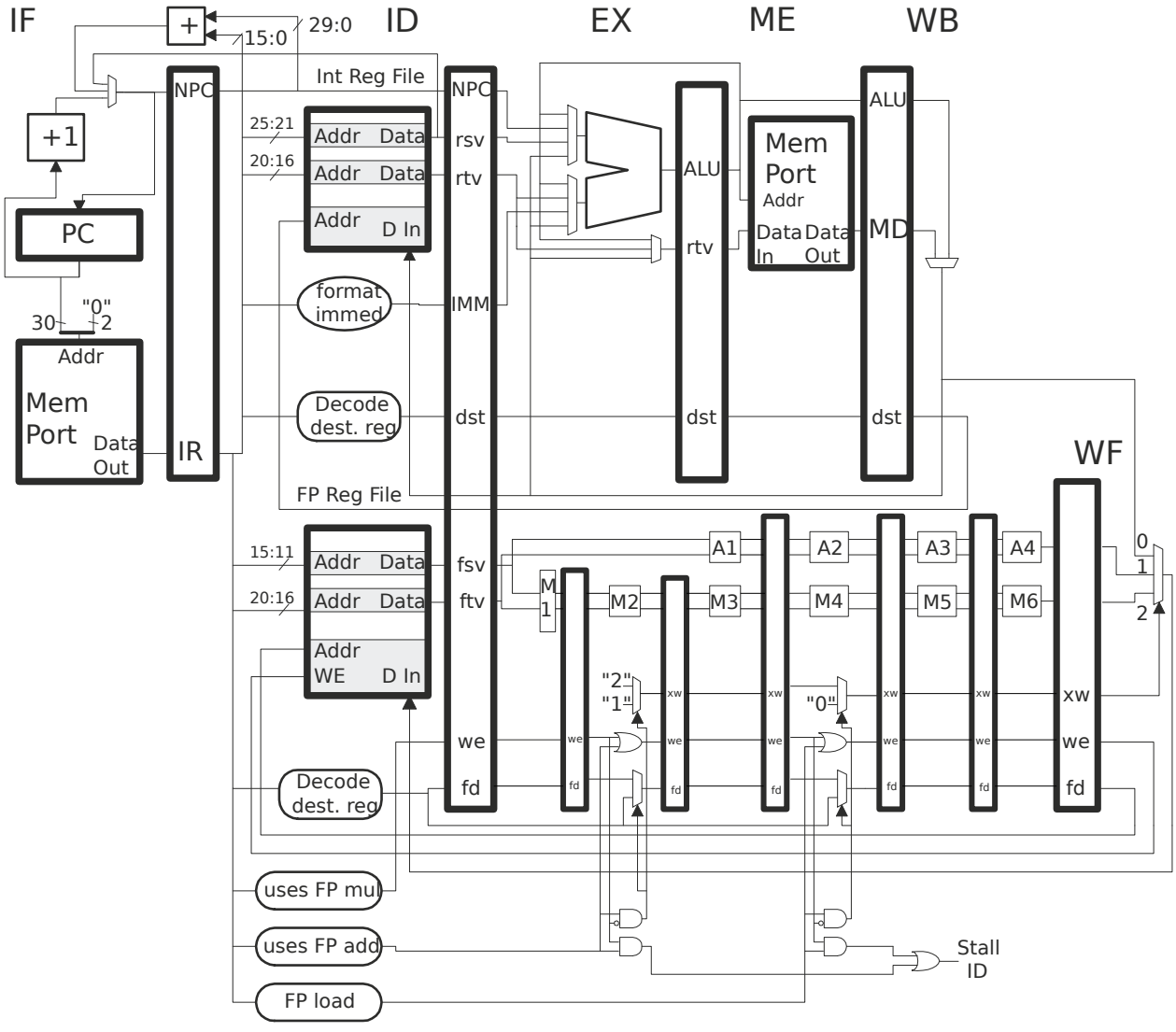
Alias Click Here

Good Luck!

Problem 1: (15 pts) Consider the following method for implementing the MIPS32 integer multiply instruction `mul` (the one that writes ordinary registers, not to be confused with `mult`) on the implementation below. The full set of stages `M1` to `M6` perform floating point multiply, but stages `M2` to `M4` perform integer multiplication. The integer multiply `mul` will read and write registers from the integer register file but will use `M2` to `M4` to perform the multiplication. A sample execution appears below. *Grading Note: In the original exam there was an ID-stage stall in cycle 6, implying that there was no* `WB` *to* `EX` *bypass for the* `mul`*.*

```
# Cycle          0  1  2  3  4  5  6  7  8
add r1, r2, r3   IF ID EX ME WB
mul r4, r1, r5      IF ID M2 M3 M4 WB
xor r9, r10, r11       IF ID -> EX ME WB
sub r6, r4, r7            IF -> ID EX ME WB
```

USE NEXT PAGE FOR SOLUTION

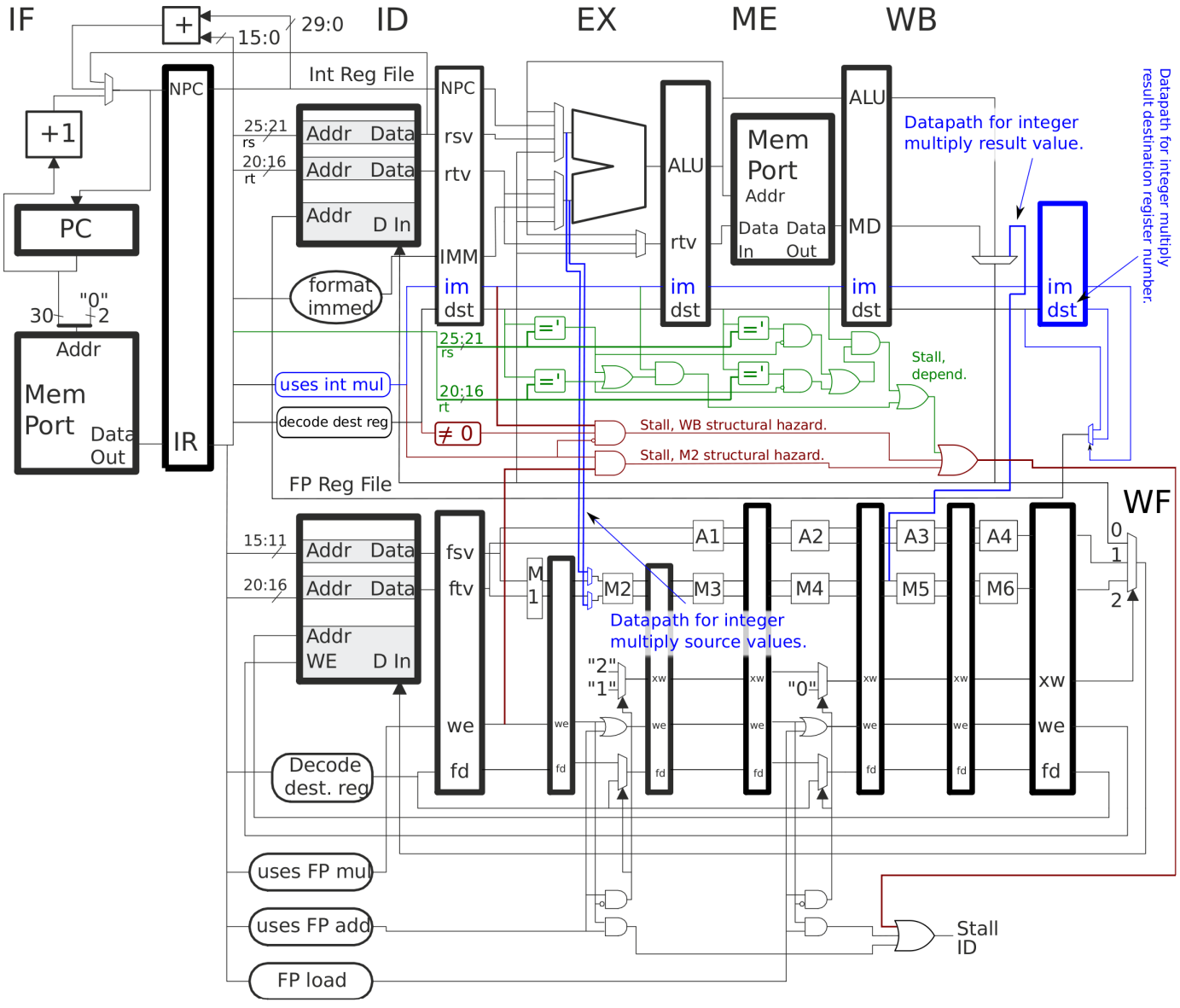


USE NEXT PAGE FOR SOLUTION

CONTINUED ON NEXT PAGE

(*a*) Modify the implementation below so that `mul` uses `M2` to `M4`. Provide any necessary bypass paths so that the code above executes as shown. For this part do not consider control logic. *Hint: Pay attention to source and destination registers.*

☑ Non-control logic modifications for integer `mul`.



Solution appears in blue above. Connections for the integer multiply source values have been added from the ALU inputs to the inputs of `M2`. A connection for the result value has been added from the `M4`/`M5` pipeline latch to the `WB`-stage multiplexor. A new pipeline latch has been added to hold the destination register number (`dst`), creating a sixth stage in the integer pipeline which is only used by the integer multiply instruction. A new mux selects the destination register in the fifth or sixth stage.

Starting the source value connection at the *output* of the ALU multiplexor provides access not just to the `rsv` and `rtv` values, but also to values bypassed from `ME` and `WB`. In this solution source values must pass through two multiplexors before reaching `M2`. A higher-cost, but potentially faster, solution would take the source values from `EX.rsv` and `EX.rtv`, but the `M2` multiplexors would need additional inputs for bypassed values. Either solution would get full credit.

The destination value is taken from the output of the M4/M5 pipeline, where it will be available at the beginning of the cycle, to give sufficient time for the register to write back the value. Credit was deducted for solutions that took the value directly from the output of M4 to the WB multiplexor, because this would likely increase the critical path length and so lower clock frequency.

(*b*) Add control logic related to this implementation of `mul` to detect the structural hazard on M2 and on WB. Also add control logic needed for a data dependence between `mul` and an immediately following instruction. All those signals should connect to the existing *Stall ID* signal and use a new uses int mul logic block.

☑ Control logic for WB structural hazard, ☑ M2 structural hazard, ☑ and the data dependence.

The control logic for the structural hazards appears in maroon and the control logic for the dependence appears in green.

A stall for the WB structural hazard is generated when there is an integer multiply instruction in EX and there is an instruction in ID, except integer multiply, that needs to write back an integer register (in which case the destination register would be nonzero). See the logic generating the signal labeled Stall, WB structural hazard. Such a stall occurs in cycle 2 in Example S1 below.

The stall for the M2 structural hazard is generated when there is an integer multiply in ID and a floating-point multiply in M1, this is detected with the AND gate whose output is labeled Stall, M2 structural hazard. See cycle 2 in Example S2 below.

```
# SOLUTION -- Example S1, for the WB structural hazard.
 # Cycle         0  1  2  3  4  5  6
 mul r1, r2, r3  IF ID M2 M3 M4 WB
 add r4, r5, r6     IF ID -> EX ME WB

# SOLUTION -- Example S2, for the M2 structural hazard.
 # Cycle           0  1  2  3  4  5  6   7  8
 mul.s f4, f5, f6  IF ID M1 M2 M3 M4 M5 M6 WF
 mul r1, r2, r3       IF ID -> M2 M3 M4 WB
```

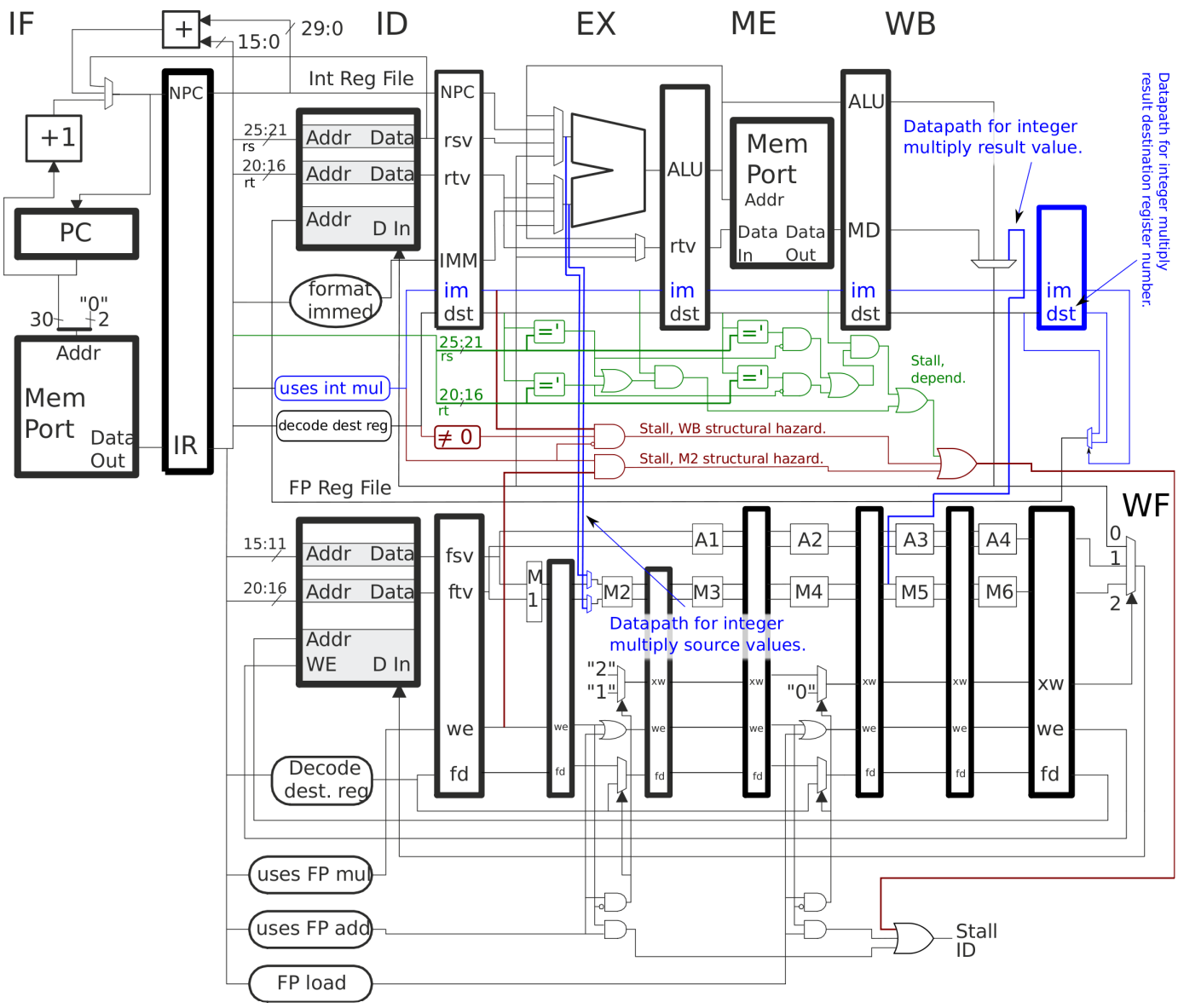
The dependence control logic, shown in green, checks whether the source register numbers of the instruction in ID matches the destination register numbers of the instructions in EX and ME, and the instruction is an integer multiply. The logic ignores the ME-stage destination register if there is already a dependence with the EX-stage destination register (because of the WB structural hazard this particular situation can't happen for the `mul` but might occur for instructions that write back normally).

In example D1 below the `add` stalls two cycles due to a dependence with the integer `mul`. In cycle 2 the `rs` register of the instruction in ID, the `add`, matches the destination of the instruction in EX, the `mul`, and so there is a stall. (For the `mul`, EX and M2 are the same stages.) In cycle 3 the `add` is still in ID but the `mul` has moved to ME (equivalent to M3). The logic again detects the dependence and so there is another stall.

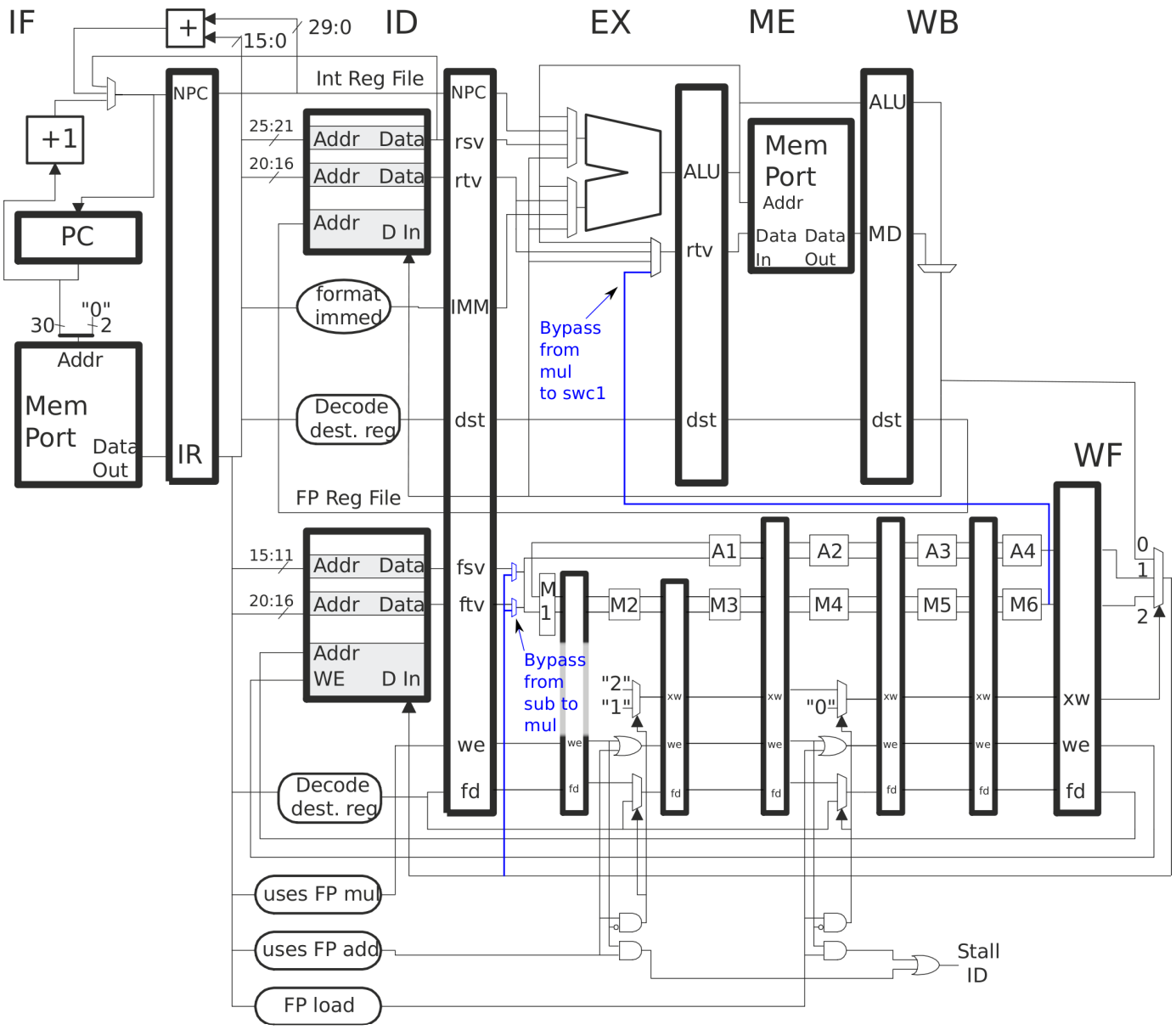
Notice that in cycle 2 there are two reasons for the stall, the dependence discussed immediately above, but also the WB structural hazard. If the WB structural hazard stall always occurred when a `mul` instruction were in EX then there would be no reason to check for a dependence with the EX-stage instruction. But the WB structural hazard stall is not generated if the instruction in ID does not write a register, and for those cases the EX stage must be checked for a dependence. See example D2.

In example D2 the `add` is replaced by a `sw`. The stall in cycle 2 is due only to the dependence through register `r1`, there is no structural hazard. The stall in cycle 3 is also due to the dependence.

```
 # SOLUTION - Example D1, mul dependence with next instruction.
 # Cycle         0  1  2  3  4  5  6  7
 mul r1, r2, r3  IF ID M2 M3 M4 WB
 add r3, r1, r4     IF ID ----> EX ME WB

 # SOLUTION - Example D2, mul dependence with next instruction.
 # Cycle         0  1  2  3  4  5  6  7
 mul r1, r2, r3  IF ID M2 M3 M4 WB
 sw r1, 0(r4)       IF ID ----> EX ME WB
```


IF
ID
EX
ME
WB
WF
+
+1
PC
NPC
IR
Addr
Mem
Port
Data
Out
30
"0"
2
15:0
29:0
Int Reg File
25:21
rs
20:16
rt
Addr Data
Addr Data
Addr D In
format immed
uses int mul
decode dest reg
NPC
rsv
rtv
IMM
im
dst
25:21
rs
20:16
rt
≠ 0
=
ALU
rtv
im
dst
Mem
Port
Addr
Data In
Data Out
ALU
MD
im
dst
Datapath for integer multiply result value.
Datapath for integer multiply result destination register number.
Stall, depend.
Stall, WB structural hazard.
Stall, M2 structural hazard.
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr WE D In
fsv
ftv
we
fd
Decode dest. reg
uses FP mul
uses FP add
FP load
M1
M2
M3
M4
M5
M6
A1
A2
A3
A4
"2"
"1"
"0"
xw
we
fd
Datapath for integer multiply source values.
0
1
2
Stall ID

Problem 2: (10 pts) Show the execution of the instructions below on the illustrated implementation. Add any needed datapath and reasonable bypass paths.



☑ Show execution. ☑ Add datapath and reasonable bypass paths. ☑ Double-check for dependencies.

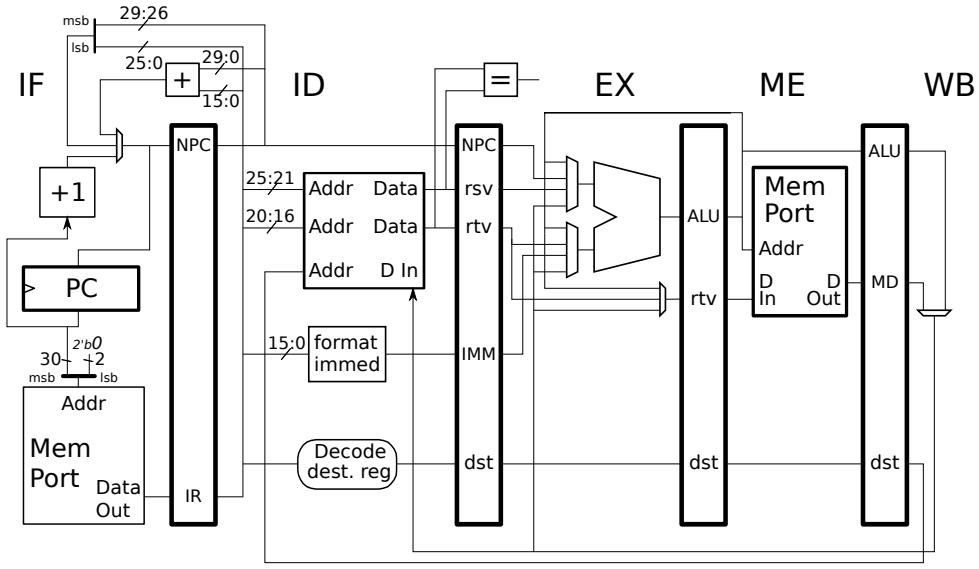
```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 sub.s f8, f10, f12     IF ID A1 A2 A3 A4 WF
 add r1, r2, r3            IF ID EX ME WB
 mul.s f14, f2, f8            IF ID ----> M1 M2 M3 M4 M5 M6 WF
 swc1 f14, 0(r1)                 IF ----> ID -------------> ME WB
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

The added bypass appears above in blue. A bypass path from WF to M1 (and A1 too) has been added for the sub.s to mul.s dependence. For the given code example only the lower multiplexor is needed since the dependence is through the second source operand of mul.s (through the ft field, set to f8 in the example).

For the mul.s to swc1 dependence a bypass path has been added from the end of M6 to EX. A lower-performance option would be to bypass from WF to ME, this is lower performance under the assumption that the memory ports are always on the critical path and so inserting a multiplexor between ME.rtv and the memory port Data In connection would lower the clock frequency.

Grading Note: Many solutions had the add *instruction stall until after the* sub.d *wrote back.*

Problem 3: (10 pts) The code fragments below execute on several different MIPS implementations. In all cases the loop iterates many times. A five-stage scalar system is shown for reference.



(*a*) Show the execution of the code below on our familiar scalar pipeline, above. Show enough iterations to compute the CPI, and compute the CPI.

☑ Execution for enough iterations to determine CPI.

See solution below. The first iteration starts in cycle 1 (by definition, when the first instruction of the loop body is in `IF`), the second iteration starts in cycle 8, and the third in cycle 14. We know we have enough iterations when the state of the pipeline is identical at the start of two consecutive iterations; we can use one of those iterations to compute CPI. The second and third iterations start identically: with `lw` in `IF`, `add` in `ID`, and `bne` in `EX`. So we know that the third iteration (for which only the `lw` is shown) will be identical to the second and so there is no need to show any more.

☑ Find the CPI.

☑ Doublecheck for dependencies.

☑ Note that the first instruction in not part of the loop body.

The second iteration is $14 - 8 = 6$ cycles, and contains 4 instructions, and so the CPI is $6/4 = 1.5$.

```
 # SOLUTION
 lw r2, 0(r10) IF ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)     IF ID -> EX ME WB
 addi r2, r2, 4      IF -> ID EX ME WB
 bne r2, r4 LOOP           IF ID ----> EX ME WB
 add r5, r5, r1               IF ----> ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)                          IF ID EX ME WB
 addi r2, r2, 4                           IF ID EX ME WB
 bne r2, r4 LOOP                             IF ID ----> EX ME WB
 add r5, r5, r1                                 IF ----> ID EX ME WB
LOOP:  #       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)                                           IF ID EX ME WB
```

Problem 3, continued:

(*b*) Show the execution of the code below on a 4-way superscalar statically scheduled system without branch prediction. The superscalar system has five stages, aligned fetch, and can bypass between the same stages as can our scalar system. This means there are no bypass paths to the branch condition. Compute the CPI.

☑ Execution for enough iterations to determine CPI.

Solution appears below. None of the instructions stall in the first iteration, but in the second iteration the branch stalls due to a dependency with `addi` (remember that there are no bypass paths to the branch condition hardware).

Note that because fetch is aligned and because the first instruction of the loop body (`add`) has an aligned address (the address is a multiple of 16 [superscalar width of 4 times instruction size of 4 bytes]), all four instructions of the loop body are fetched at the same time.

☑ Find the CPI.

☑ The code below is *different* than the previous part.

☑ Doublecheck for dependencies.

☑ Note that first instruction not part of loop body.

The CPI is $(9-6)/4 = 0.75$ or $\frac{4}{3}$ IPC, less than half the 4 IPC potential of the hardware.

```
# SOLUTION
 addi r1, r0, 0     IF ID EX ME WB
 # Note: Address of insn below is 0x1000
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1        IF ID EX ME WB
 lw r1, 0(r2)          IF ID EX ME WB
 bne r2, r4 LOOP       IF ID EX ME WB
 addi r2, r2, 4        IF ID EX ME WB
                          IFx
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1           IF ID EX ME WB
 lw r1, 0(r2)             IF ID EX ME WB
 bne r2, r4 LOOP          IF ID -> EX ME WB
 addi r2, r2, 4           IF ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
                             IF ->x
 add r5, r5, r1                 IF ID EX ME WB
 lw r1, 0(r2)                   IF ID EX ME WB
 bne r2, r4 LOOP                IF ID -> EX ME WB
 addi r2, r2, 4                 IF ID -> EX ME WB
LOOP: #             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
                                   IF ->x
 add r5, r5, r1                       IF ID EX ME WB
```

(*c*) Show the execution of the code below on a four-way superscalar system with perfect branch prediction.

☑ Execution for enough iterations to determine CPI.

Because there is a branch predictor the branch does not need to be resolved in **ID**. Instead it will be resolved in **EX** where it can use the ALU to compute the condition, and where it can benefit from the existing ALU bypass paths. Therefore the branch instruction does not stall at all because the value of **r2** that the branch needs is bypassable starting when **addi** is in **ME** (such as in cycle 4 and 6). (Note that the stalls in cycle 4, 6, 8, etc are due to the dependence between the **lw** and the **add**.)

☑ Find the CPI.

☑ The code below is *different* than the first part.

☑ Doublecheck for dependencies.

☑ Note that first instruction not part of loop body.

The CPI is $(5-3)/4 = 0.5$ or 2 IPC, still half the full potential, due to the load-use dependence.

```
 # SOLUTION
 addi r1, r0, 0      IF ID EX ME WB
LOOP: #              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1         IF ID EX ME WB
 lw r1, 0(r2)           IF ID EX ME WB
 bne r2, r4 LOOP        IF ID EX ME WB
 addi r2, r2, 4         IF ID EX ME WB
LOOP: #              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1            IF ID -> EX ME WB
 lw r1, 0(r2)              IF ID -> EX ME WB
 bne r2, r4 LOOP           IF ID -> EX ME WB
 addi r2, r2, 4            IF ID -> EX ME WB
LOOP: #              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1               IF -> ID -> EX ME WB
 lw r1, 0(r2)                 IF -> ID -> EX ME WB
 bne r2, r4 LOOP              IF -> ID -> EX ME WB
 addi r2, r2, 4               IF -> ID -> EX ME WB
LOOP: #              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                  IF -> ID -> EX ME WB
 lw r1, 0(r2)                    IF -> ID -> EX ME WB
 bne r2, r4 LOOP                 IF -> ID -> EX ME WB
 addi r2, r2, 4                  IF -> ID -> EX ME WB
LOOP: #              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
 add r5, r5, r1                     IF -> ID -> EX ME WB
```

Problem 4: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{10}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 12-outcome local history, and one system uses a global predictor with a 12-outcome global history.

```
Insn        Branch
Addr        Outcomes
0x1000: B1  r   r   r   r   r   r   r   r    r   r   r   r   r   r   r   r
0x1010: B2   T   N   N   N   N   T   N   N    T   N   N   N   N   T   N   N
0x1020: B3    R   R   R   R   R   R   R   R    R   R   R   R   R   R   R   R
0x2000: B4     T   T   T   T   T   T   T   T    T   T   T   T   T   T   T   T
```

Branch B1 is random, and can be described by a Bernoulli random variable with $p = .5$ (models a fair coin toss). The outcome of branch B3 is the same as the most recent execution of B1 (perhaps they are testing the same condition). For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal predictor on B2?

In the diagram below the value of the bimodal predictor's 2-bit counter corresponding to B2 is shown before each execution of the branch. Counter values are shown only for one pass through the TNNNNTNN pattern. At the start of the second pass the counter has the same value, 0, as it had at the start of the first and so there is no need to continue (because we know the counter values will repeat). The bimodal predictor will predict not-taken when the counter value is 0 or 1, and so all predictions will be not-taken, resulting in an accuracy of $6/8$.

Solution Tips: Be sure to base your answer on a repeating pattern. Obviously it would be wrong to base the accuracy only on the N outcomes (that would be 100% accuracy), but basing a solution on ten outcomes or twelve outcomes would also be wrong. Another thing to look out for is making sure the pattern repeats. Here we only needed to work out the counter values for one iteration to find a repeating pattern. If we used an initial counter value of 3 then we'd need to work counter values out for two passes through the TNNNNTNN pattern, and we'd need to base the prediction accuracy only on the second pass.

```
SOLUTION: Diagram to work out the values of the bimodal 2-bit counter.
BIMODAL COUNTER: 0   1   0   0   0   0   1   0   0
0x1010: B2         T   N   N   N   N   T   N   N    T   N   N   N   N   T   N   N
PRED RESULT:       x                   x
```

☑ What is the accuracy of the bimodal predictor on B3?

The 2-bit counter used to predict B3 is not affected by B1. Because the probability that B3 is taken is .5 the accuracy is .5.

☑ Considering BHT size, what is the approximate accuracy of the bimodal predictor on B4? ☑ Explain.

Since the predictor has 2^{10} entries it will be indexed by bits $11:2$ of the address of the branch being predicted. Notice that those bits are all zero for both branches B4 (its address is 0x1000) and B1 (its address is 0x2000), and so they will share an entry. Since both branches occur at the same frequency and B4 is always taken, there will be no way for the counter to be decremented. It will change between 3 and 4 and so B4 will be predicted with 100% accuracy. *Grading Note: The problem would have been more interesting if $p < 0.5$ or if* B1 *were executed more frequently than* B4.

☑ What is the accuracy of the local predictor on B2?

The pattern length of B2, 8, easily fits in the local history of 12 and so the accuracy is 100%.

☑ What is the minimum local history size needed to predict B2 with 100% accuracy?
Six outcomes.

Five is insufficient, for example, pattern NNTNN occurs twice, once followed by a N and once by a T.

☑ What is the accuracy of the global predictor B3? ☑ Explain

The 12-bit global history is long enough so that B3 can "see" B1's most recent outcome. The PHT entries will eventually warm up and predict B3 with 100% accuracy.

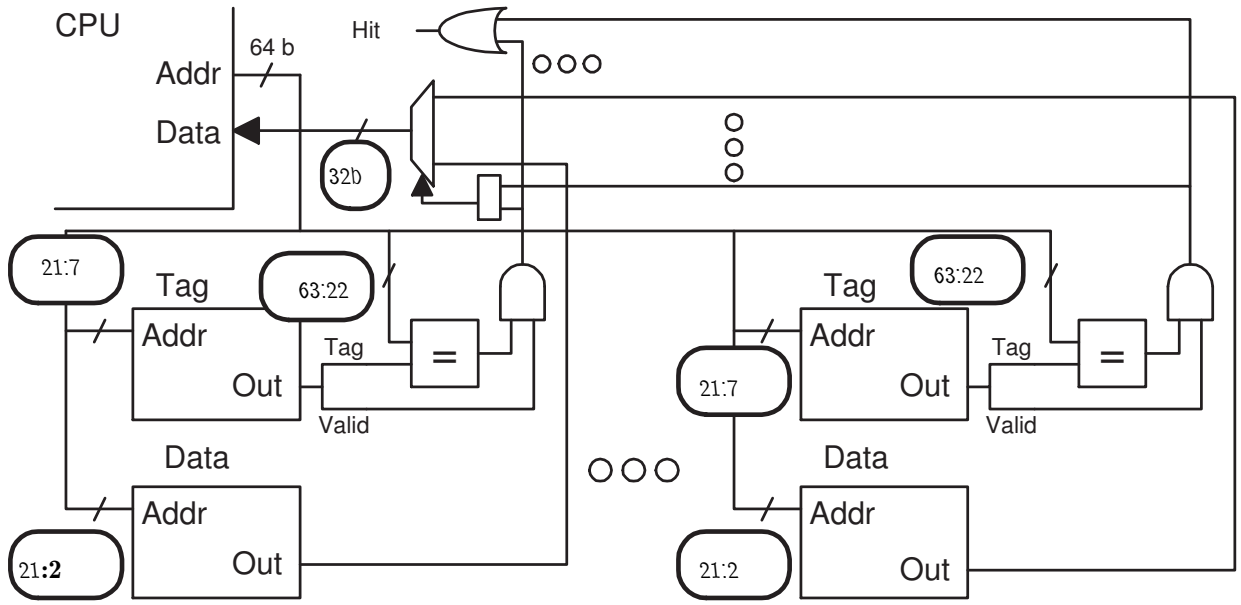
☑ How many PHT entries are used by the global predictor to predict B2?

To answer this question we need a way to represent the GHR contents. Lets use r, R, and T for branches B1, B3, and B4, respectively. For B2 use a 2. A GHR value pattern is then 2RTr2RTr2RTr. Each R can have two values, so for all three there are $2^3 = 8$ possibilities. The first two r's each match one of the R's and so don't contribute additional patterns. The last (rightmost or most recent) r does not correlate with any Rs in the pattern so there are two more possibilities for a total so far of $8 \times 2 = 16$. The three 2's are for branch B2. Those three outcomes can have four possible values, TNN, NNN, NNT, and NTN. (For example, three consecutive outcomes of B2 can never be TTT.) The total number of patterns now is $8 \times 2 \times 4 = 64$. Therefore, 64 entries are used to predict B2.

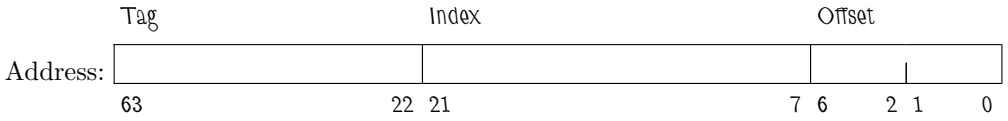
Problem 5: (15 pts) The diagram below is for an eight-way set-associative cache. The size of a tag is 42 bits and the size of a line is $128 = 2^7$ bytes.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



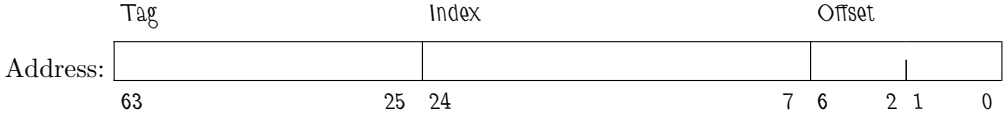
☑ Cache Capacity (Indicate Unit!!):

The cache capacity can be determined from the following pieces of information: the 42-bit tag (given explicitly in the first paragraph), the 8-way associativity (also given explicitly), and the 64-bit address space (from the number of address bits shown in the diagram near the upper-left). From the 42-bit tag and 64-bit address space we know that the low tag bit is $64 - 42 = 22$, and so the capacity of a way is 2^{22} bytes or 4 MiB. Since the cache is 8-way the total capacity is $2^{22} \times 8 = 2^{25}$ bytes or 32 MiB.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{25} bytes, plus $8 \times 2^{22-6}\,(64 - 22 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 5, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 128-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

(*b*) Find the hit ratio executing the code below.

☑ What is the hit ratio running the code below? Show formula and briefly justify.

```
int sum = 0;
half *a = 0x20000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;     // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of $2^7 = 128$ bytes is given, the size of an array element, of type half, is $2 = 2^1$ characters, and so there are $2^7/2 = 2^{7-1} = 2^6 = 64$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^6 elements, and so the next $2^6 - 1$ accesses will be to data on the line. The access at `i=64` will miss and the process will repeat. Therefore the hit ratio is $\frac{63}{64}$.

(*c*) Find the minimum positive value of `OFFSET` needed so that the code below experiences a hit ratio of 0% on accesses to `a`. Explain.

☑ Minimum positive `OFFSET` to achieve 0% hit ratio. ☑ Explanation.

```
int sum = 0;
half *a = 0x20000000; // sizeof(half) == 2
int i;
int ILIMIT = 1 << 11;


int OFFSET =   1 << 24;  // <----------------------------- SOLUTION


for ( i=0; i<ILIMIT; i++ ) sum += a[ i ] + a [ i + OFFSET ];
```

To achieve a zero percent hit ratio each line brought into the cache must be evicted before it is used again. The code above accesses `a` sequentially at two different places: at `i` and at `i+OFFSET`. Sequential accesses of two-byte elements should yield a hit ratio of $\frac{63}{64}$ on this cache, as computed in the previous part. To reduce that to zero one must choose `OFFSET` so that `a[i]` and `a[i+OFFSET]` fall in the same set, meaning that their addresses have the same index and different tags. Since it's a direct-mapped cache there cannot be two different lines with the same index in the cache at the same time. The index bits are at positions `24:7`. Let x be some memory address. Address y will have the same index as x if $y = x + 2^{25}$ (note that adding 1 to the 25th bit position won't affect the index bit positions). For our particular problem we need to choose `OFFSET` such that the difference between `a[i]` and `a[i+OFFSET]` is 2^{25}. Since the size of an element of `a` is two bytes, that means `OFFSET` must be $2^{25-1} = 2^{24}$ =`1<<24`.

Problem 6: (30 pts) Answer each question below.

(*a*) For the SPECcpu benchmark suite results can be reported using two tuning levels, *base* and *peak*. Consider a new level, *no-opt* in which the code is compiled without optimization. How valuable would no-opt tuning scores be to the people that care about SPECcpu scores? How different is no-opt tuning from base tuning?

☑ Value of no-opt tuning level?

No-opt tuning would not be of much value because it reflects the performance of a system running improperly prepared software. In other words, no-opt tuning would be useful to people who care about performance but for some silly reason don't compile with optimization turned on, or they get software from developers that don't turn optimization on.

Grading Notes: Some answered that the performance on unoptimized code might help predict the performance on optimized code, and for that reason no-opt is useful. That's wrong for two reasons. First, there are many reasons why system A can be faster than B on unoptimized code but B can be faster than A with optimization on. Second, peak and base both measure system performance on optimized code, so there is no need to have a third tuning level just to predict the first two.

☑ Difference between no-opt tuning and base tuning?

Programs prepared under the base tuning rules are optimized using normal effort by an experienced programmer. Such a program would at least use a no-brainer flag like `-O3` or `-fast`. That's far from no optimization.

(*b*) The `lw` below will experience a TLB miss exception when it first executes, remember that this exception is considered routine and is not due to any kind of error. The word in memory at location `0x12345678` is `0x222`. Show the execution of this code on our five-stage static pipeline until the `add` instruction reaches writeback, and show the value in register `r1` when the handler starts (see code).

☑ Show execution from `lui` to when `add` reaches `WB`.

☑ `FILL IN` the value that will be in `r1` when the handler starts.

```
# SOLUTION
# Cycles            0  1  2  3  4  5  6  7  8  9    1010 11 12 13 14 15 16
lui r1, 0x1234      IF ID EX ME WB

lw r1, 0x5678(r1)      IF ID EX ME*                      IF ID EX ME WB

add r2, r2, r1            IF ID EX*                         IF ID EX ME WB


HANDLER:
# Cycles            0  1  2  3  4  5  6  7  8  9    1010 11 12 13 14 15 16
# Value of r1 is          0x12340000   <- SOLUTION
sw ...                           IF ID EX ME WB
...
eret                                                IF ID EX ME WB
```

Solution appears above. The value of `r1` reflects correct execution up to but not including `lw`. The handler ends by executing an `eret` (exception return) instruction which jumps to the `lw`, which executes completely this second time around.

(*c*) The instruction below is not a good candidate for a RISC ISA. Explain why in terms of hardware, not just in terms of a rule the instruction would violate. `add (r1), r2, (r3)`

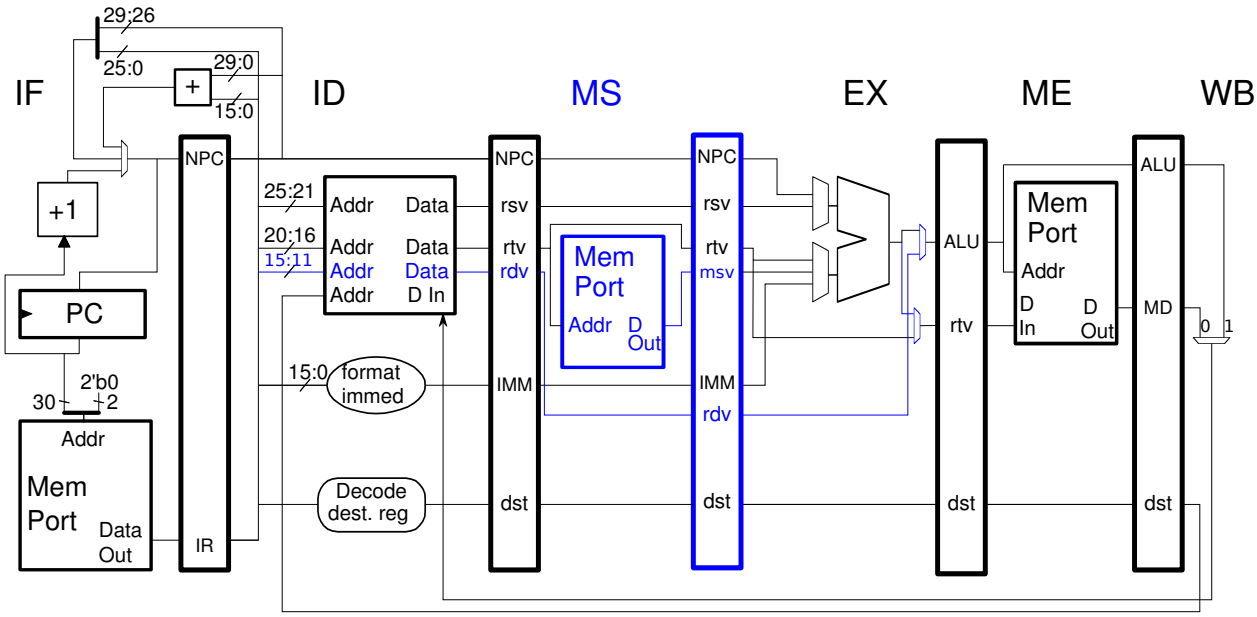
☑ Instruction above unsuitable for RISC because the implementation . . .

. . . would either need two memory ports (memory ports are expensive) or else would require this `add` instruction to use the ME stage twice, which would make the control logic far more complex. The former implementation would be too expensive, the later would go against the goal of simple pipelined implementations.

☑ Provide a quick sketch to illustrate your answer.

See the diagram below in which the added hardware is in blue and in which bypass paths have been omitted for clarity. Assuming that memory addresses for the source and destination operands are just register values (no offsets added), then a sixth memory source stage could be added between ID and EX to fetch a source operand that comes from memory, see stage MS the pipeline diagram below. The existing ME stage can write the result, though additional multiplexors were added so the `rd` register value can be used as a memory address and the ALU output can be used for the write data.

```
# Execution with new memory source stage added.
add (r1), r2, (r3)    IF ID Ms EX ME WB
```



(*d*) With the aid of a diagram, explain why a $5n$-stage pipeline would need fewer bypass paths than a 5-stage, n-way superscalar implementation. (Both implementations are statically scheduled.)

☑ Diagram showing why there are fewer bypass paths in $5n$-stage pipeline than 5-way superscalar.

Both systems would have $2n$ results that could be bypassed (from the original ME and WB stages). However the superscalar would have n instructions in EX to which a result could be bypassed, while the deeply pipelined would have just 1. (There are two ALU inputs for each instruction.)

(*e*) Why is it more important to have a good compiler for a superscalar statically scheduled system than a scalar statically scheduled system?

☑ Compiler more important for superscalar because . . .

. . . dependent instructions must be further apart to avoid a stall and so the compiler must be better at scheduling (finding instructions to put between dependent pairs). In the scalar system there is no need to stall even for adjacent dependent ALU instructions (thanks to the bypass paths). In an n-way superscalar system there must be up to $n-1$ instructions between dependent ALU instructions to avoid a stall, and so the compiler must be good at finding instructions to put between such dependent pairs.

(*f*) Consider the executions of the MIPS code below on a 4-way superscalar dynamically scheduled system of the type discussed in class (method 3). The first execution is correct, the others, though they would run the program correctly, have problems. Describe the problems by completing the statements below.

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q        RR EX WB C
lh r1, 0(r6)      IF ID Q  RR EA ME WB    C
sub r7, r1, r3    IF ID Q           RR EX WB C
```

☑ The one-commit-per-cycle execution below is silly because . . .

. . . it will execute at a maximum rate of one instruction per cycle, and yet it has enough fetch, decode, and presumably execute hardware to sustain 4-instruction-per-cycle execution. The reasonable thing to do is to allow 4 IPC commit.

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q        RR EX WB C
lh r1, 0(r6)      IF ID Q  RR EA ME WB       C
sub r7, r1, r3    IF ID Q           RR EX WB    C
```

☑ The stalls shown below would be necessary on a statically scheduled pipeline because . . .

. . . there is a single pipeline and instructions must remain in program order within the pipeline. If one instruction must wait, so must the unlucky instructions ahead of it.

☑ . . . but should not occur on a dynamically scheduled one because . . .

. . . it is designed to execute instructions out of program order. The `IF/ID/Q` and `RR/EX/WB` are each separate pipelines, and so the `add` instruction waits for the `lw` in buffers, not in pipelines, allowing the `lh` to proceed without delay.

```
lw r1, 0(r2)      IF ID Q  RR EA ME WB C
add r3, r1, r4    IF ID Q  ----> RR EX WB C
lh r1, 0(r6)      IF ID Q  ----> RR EA ME WB C
sub r7, r1, r3    IF ID Q  ----------> RR EX WB  C
```

55 Spring 2011 Solutions

Name Solution

Computer Architecture

EE 4720

Midterm Examination

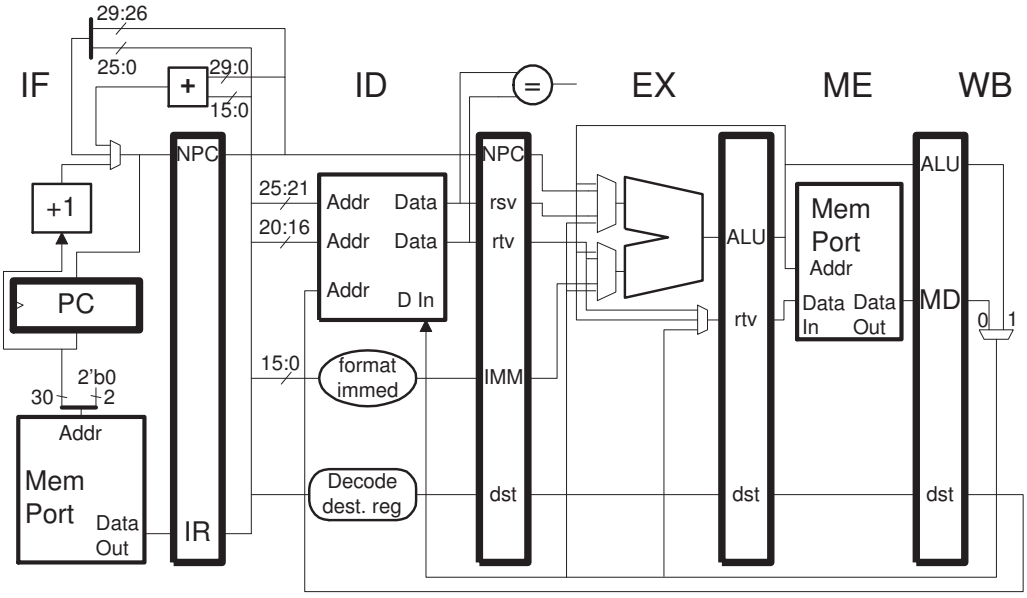
Wednesday, 30 March 2011, 9:40–10:30 CDT

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(10 pts)
Problem 6		(15 pts)
Problem 7		(15 pts)
Exam Total		(100 pts)

Alias The end of history?

Good Luck!

Problem 1: [15 pts] The MIPS code below executes on the illustrated hopefully familiar implementation.



n

```
      # SOLUTION
LOOP: # Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         IF ID EX ME WB                          First Iteration
addi r4, r4, 4          IF ID EX ME WB
andi r3, r1, 0xf0f0        IF ID EX ME WB
bne r3, r0, LOOP              IF ID ----> EX ME WB
lw r2, 4(r2)                     IF ----> ID EX ME WB
#       Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         Second Iteration     IF ID -> EX ME WB
addi r4, r4, 4                               IF -> ID EX ME WB
andi r3, r1, 0xf0f0                                IF ID EX ME WB
bne r3, r0, LOOP                                      IF ID ----> EX ME WB
lw r2, 4(r2)                                             IF ----> ID EX ME WB
#       Cycle        0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
lh r1, 0(r2)         Third Iteration                               IF ID -> EX
```

(*a*) Show a pipeline execution diagram for the code below running on the illustrated MIPS implementation for enough iterations to determine CPI.

☑ Show a pipeline execution diagram.

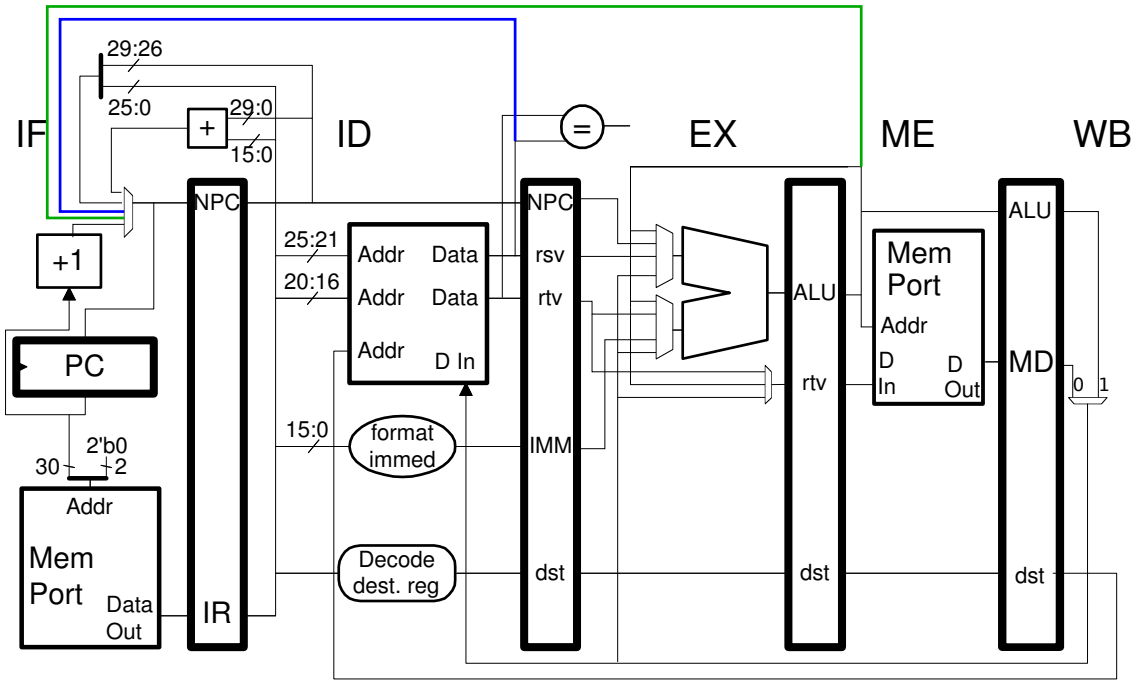
Diagram appears above.

(*b*) Determine the CPI.

☑ Find the CPI.

The state of the second and third iterations are identical when they start, in cycles 7 and 15, respectively. (At both cycles 7 and 15 there is an `lh` in `IF`, `lw` in `ID`, and a `bne` in `EX`, the other stages are empty. Therefore the third iteration will be just like the second, and so on. The time for the second iteration is $15 - 7 = 8$ cycles (using the `IF` of the first instruction as a reference point), and so the CPI is $\frac{15-7}{5} = 1.6$ CPI.

Problem 2: [20 pts] In the implementation below the datapath for the `jalr` and `jr` instructions is not shown.



(*a*) Add datapath (but not control logic) needed for the `jr` instruction to the diagram above.

☑ Add datapath (but not control logic) for `jr`.

Change appear in blue. All that's needed is a path from the register file output for `rsv` to the IF-stage mux. Note that the register values are available early enough to avoid the risk of lengthening the critical path.

(*b*) Show the execution of the code below based upon the answer above. Show execution starting from the first instruction (as usual) and continue until `ori` executes or eight instructions have executed whichever is longer. (In the correct answer `ori` is the eighth instruction to execute.) Note that the `jalr` will jump to `ROUTINE` the first time it executes.

☑ Show execution consistent with previous answer.

Solution appears below. Notice that the `jalr` must stall two cycles since it can only get the source register from the register file.

```
# SOLUTION
# Cycles                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 addi r1, r1, 16        IF ID EX ME WB
 jalr r31, r1              IF ID ----> EX ME WB
 addi r4, r31, -8             IF ----> ID EX ME WB
 ori r6, r6, 0xff

# Cycles                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
ROUTINE:
 addi r4, r4, 4                         IF ID EX ME WB
 jr r4                                     IF ID ----> EX ME WB
 lw r31, 0(r7)                                IF ----> ID EX ME WB
```

```
# Cycles                0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 # NOTE: These are the instructions that follow the jalr.
 addi r4, r31, -8                                      IF ID -> EX ME WB
 ori r6, r6, 0xff                                         IF -> ID EX ME WB
```

(*c*) If your answer to the previous part was correct the code above would suffer stall(s). Add bypasses to avoid as many of the stalls as possible without significant critical path impact.

☑ Bypass paths to eliminate stalls.

The solution has to balance "as many as possible" with "significant impact." The value at the output of the ALU would not be available until close to the end of the clock cycle, so connecting it to the IF-stage mux would likely increase the critical path by a small amount. Instead, an ME-stage bypass is used. There will still be one stall cycle but the clock frequency will be untouched. That stall cycle is only suffered when a `jr` or `jalr` instruction immediately follows the instruction writing the target register, and that won't happen very often.

The changes appear in green on the diagram.

Problem 3: [15 pts] In the code fragment below the `lw` raises an exception due to a TLB miss. Remember that a TLB miss is something that happens all the time to almost any load.

Solution omitted.

```
 # Cycle          0   1   2   3   4   5   6   7
 lw r1, 0(r2)     IF
 add r3, r5, r6
 xor r4, r3, r9
...
HANDLER:
 sw r10, ..
```

(*a*) Based on the exception hardware presented in class, at which cycle will the first instruction of the handler be in `IF`?

☑ Handler in `IF` in cycle:

For the next parts of this problem don't consider the exception hardware presented in class, instead the `lw` will trigger a deferred exception. That is, the `lw` will raise a TLB miss exception but `add` will successfully complete `WB` before the handler is called. Register `r1` will have some garbage value but the `add` will execute correctly.

(*b*) This is **not** a good example for why precise exceptions are needed for loads. Why not?

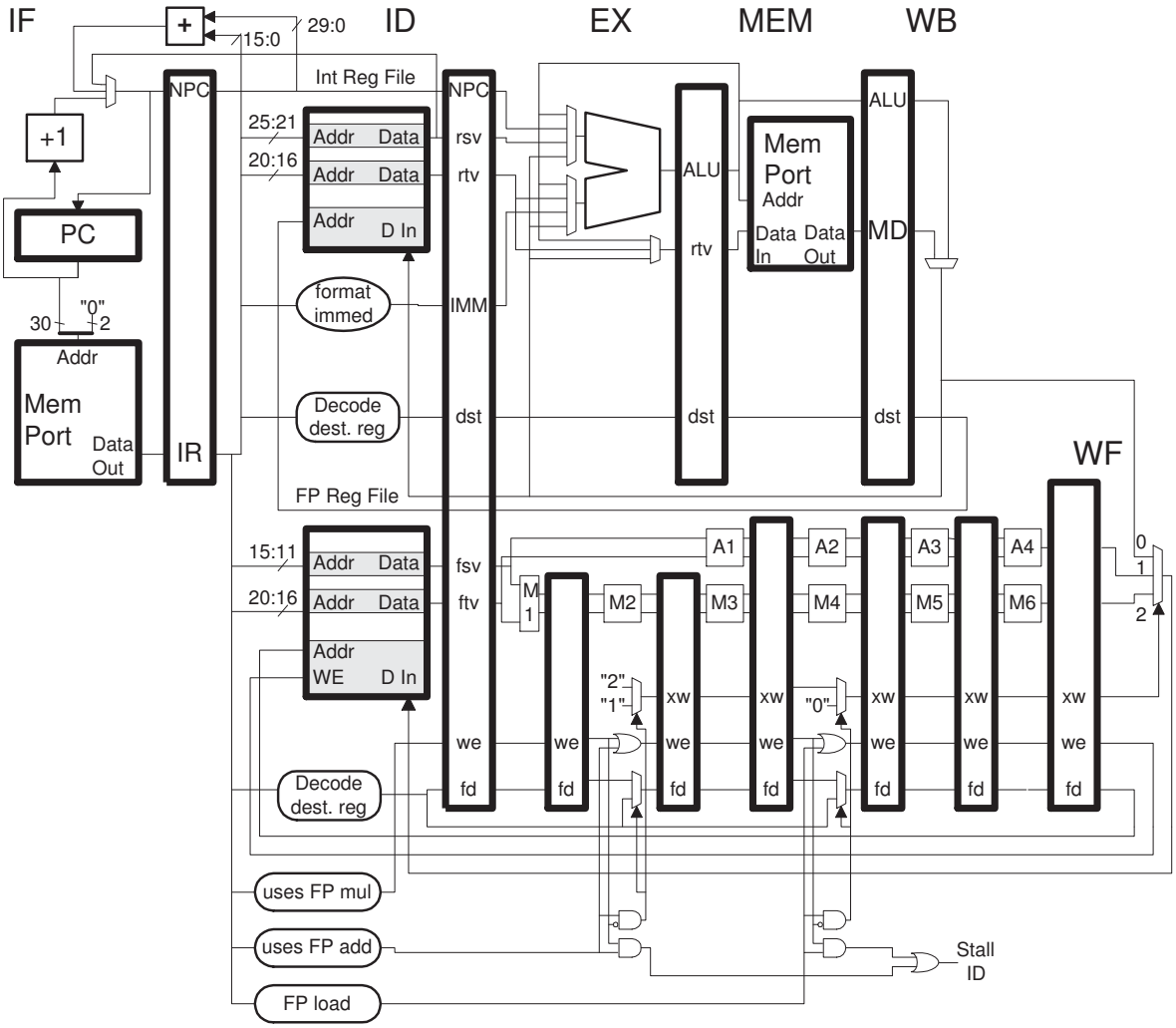
☑ Not a good example because:

(*c*) Modify the code after the `lw` so that this *is* a good example of why precise exceptions are needed. Briefly explain why.

☑ Modify code.

☑ Explain why it is a better example.

Problem 4: [10 pts] Show the execution of the MIPS code below on the illustrated implementation. Assume that all needed bypass paths are available.



☑ Execution of code.

Solution omitted.

```
add.d f2, f4, f6

sub.d f8, f4, f10

mul.d f12, f2, f14

addi r1, r1, 8
```

Problem 5: [10 pts] Answer each question about the SPECcpu suite.

(*a*) With SPECcpu the tester is responsible for providing compilation tools and choosing optimization switches. Which feature or goal of the SPEC benchmarks does that support?

☑ Compilation tools and optimization choices supports:

Test the full potential of the newest implementations and ISAs.

(*b*) Part of the SPEC rules requires that compilation tools be actively marketed as products, not just available to someone who knows exactly how to ask for them. What sort of abuse does that prevent?

☑ Actively marketed compilation tools prevents:

It prevents compiling the suite using a compiler with optimizations that are really fast but which in many cases will generate incorrect code. Specific compiler bugs encountered when building the SPECcpu code might be fixed, but other compiler bugs might remain. A company would not want to hurt its reputation by marketing a buggy compiler, and so the rule helps avoid this abuse.

Problem 6: [15 pts] Answer each question below.

(*a*) ISAs are frequently updated, examples mentioned in class include MIPS I, MIPS II, SPARC v8, SPARC v9, and the multimedia extensions such as MMX, SSE, SSE 2, etc. for IA-32. With all these updates it sounds like an ISA is not really frozen for All Time. Does that mean the main advantage in separating ISA design from implementation design has been lost?

☑ Has main advantage of ISA/implementation separation been lost? Explain.

No. The main advantage is that code written for an ISA will run correctly on any of the ISA's implementations. ISA updates and extensions are done in such a way that code written for an earlier version will run on an implementation of a later version. For example, code written for the SPARC v8 ISA will run correctly on an implementation of the SPARC v9 ISA.

(*b*) Why is the typical RISC program larger than an equivalent CISC program?

☑ RISC larger than CISC because:

Because RISC instruction sizes are all the same, meaning that some instructions take up more space than the equivalent CISC instruction.

(*c*) Explain how the approaches to encoding immediate arithmetic instructions differ in the SPARC and MIPS ISAs.

☑ How approaches differ.

In MIPS integer instructions that require an immediate, such as `addi`, use a different opcode and encoding than their two-source-register counterparts, such as `add`. In contrast, many instructions that use an immediate, such as `add`, use the same opcode and opcode extension, `op3`, for their immediate and two-source-register forms; one bit in the encoding, `i`, distinguishes the two.

Problem 7: [15 pts] Answer each question below.

(*a*) Provide the following optimization examples:

☑ An optimization that can be performed well without knowing the particular implementation being targeted.

Dead code elimination.

☑ An optimization that can be performed well only if the compiler knows the particular implementation being targeted.

Scheduling to avoid stalls.

(*b*) In an alternate universe, when designing the ALT-MIPS ISA, computer engineers insisted, perhaps for cost reasons, that the data memory port and ALU had to be in the same stage (that is, the `EX` and `ME` stages would be combined), resulting in a four-stage pipeline. How would the matching ALT-MIPS be different from our MIPS ISA? Note that elegance and performance are important for both ALT-MIPS and MIPS. Provide two code samples, one for which the four-stage ALT-MIPS is better and one for which the five-stage MIPS is better.

☑ ALT-MIPS differences with MIPS.

Because there is no time to add an immediate to a base register, load and store instructions in ALT-MIPS would not have an offset.

☑ Code sample better for ALT-MIPS.

☑ Explain.

Because the memory port and ALU are in the same stage it is possible to bypass a value from a load instruction to an arithmetic instruction. The example below has such a pair of instructions. Note that the load instruction does not have an offset, making it appropriate for ALT-MIPS. The combined `EX` and `ME` stages are called `EM`. In cycle 3 the value loaded by the `lw` is being bypassed to the `add`.

```
# SOLUTION          0  1  2  3  4  5
lw r1, (r2)         IF ID EM WB
add r3, r1, r3         IF ID EM WB
```

☑ Code sample better for MIPS.

☑ Explain.

In the first code fragment below, for MIPS, the `lh` has an offset, and so it could not execute on ALT-MIPS. The ALT-MIPS version follows, an `addi` had to be inserted to compute the load address.

```
# SOLUTION  MIPS Code (could not run on ALT-MIPS)
lw r1, 0(r2)
lh r3, 4(r2)

# The corresponding ALT-MIPS code.
lw r1, (r2)
addi r2, r2, 4
lh r3, (r2)
```

Name Solution

Computer Architecture
EE 4720
Final Examination

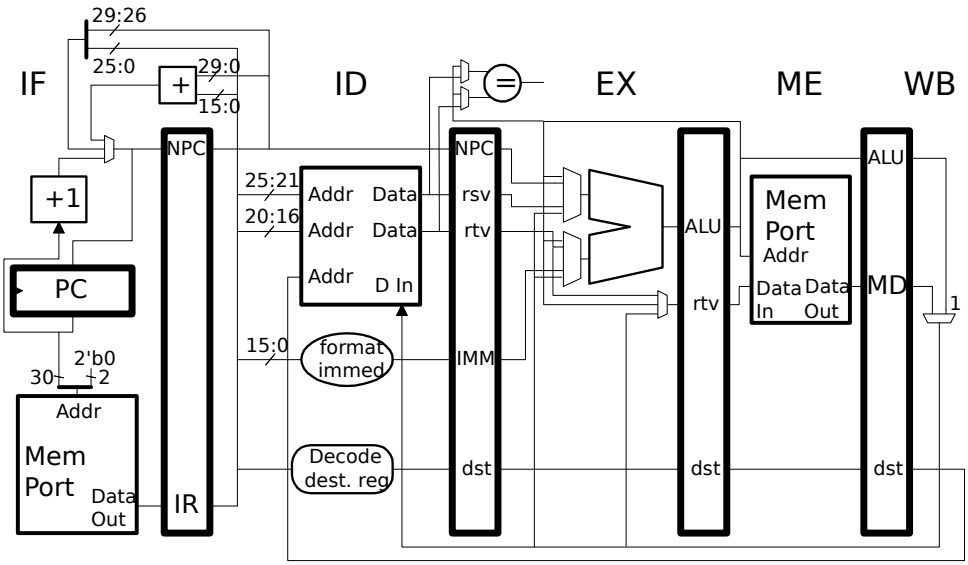
9 May 2011, 15:00–17:00 CDT

Problem 1		(20 pts)
Problem 2		(10 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

Alias Mississippi Rising

Good Luck!

Problem 1: (20 pts) The MIPS implementation below is similar to the one frequently used in class, except that it has bypass paths to the branch condition comparison unit.



(*a*) Show the execution of the code fragments below on the illustrated implementation up until the fetch of the first instruction of the second iteration. Be sure to account for the dependence on the branch condition.

☑ Pipeline execution diagram of code below.

```
#          SOLUTION
LOOP1:   # Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)        IF ID EX ME WB
 addi r2, r2, 4         IF ID EX ME WB
 bne  r2, r3 LOOP1         IF ID -> EX ME WB
 add r4, r4, r1               IF -> ID EX ME WB
#          Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)                        IF ID EX ME WB
```

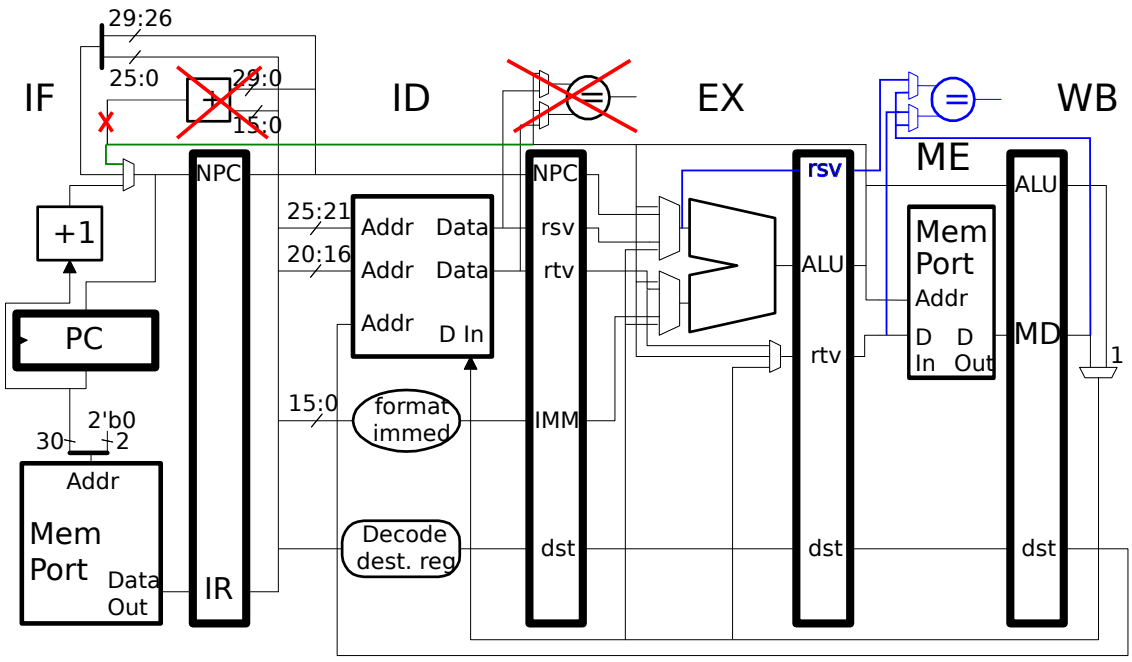
The solution appears above. Note that because of the dependence with the `addi` instruction the branch must stall one cycle, until cycle 4, at which time the branch can use the bypass from ME to ID.

☑ Pipeline execution diagram of code below.

```
#       SOLUTION
LOOP2: # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)       IF ID EX ME WB
 lw r2, 4(r2)          IF ID EX ME WB
 bne  r2, r0 LOOP2        IF ID ----> EX ME WB
 add r4, r4, r1               IF ----> ID EX ME WB
#       # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)                           IF ID EX ME WB
```

Solution appears above. Because the branch is dependent on a load, the `lw r2`, the bypass doesn't help and so the branch must stall until `lw r2` reaches WB, in cycle 5. At that time the load value will be passed through the register file to the branch condition comparison unit (the circled equal sign).

(*b*) The implementation below has hardware in `IF` (not shown) to predict branches. Since it has prediction hardware the branch resolution hardware (the hardware that computes the branch condition and branch target) could be moved to the `ME` stage. Move the branch resolution hardware. Note that the resolution hardware will be used only if the branch is mispredicted.



☑ Move resolution hardware to the `ME` stage.

☑ Add any bypasses to resolution hardware needed by code samples in this problem.

☑ Cross out unused hardware in `ID`.

☑ Avoid adding unnecessary hardware.

Solution appears above.

The comparison unit has been moved to the `ME` stage, where it gets inputs from an added `ME.rsv` latch and the existing `ME.rtv` latch. In the figure the old comparison unit is crossed out and the new (or moved) comparison unit and new connections for it appear in blue. Those connections provide bypasses of values from the two preceding instructions, but won't work if the immediately preceding instruction is a load. For that a bypass has been added from the `WB.MD` latch. (The output of the `WB`-stage mux could also be used, but any value in `WB.ALU` could also have been bypassed when the branch was in `EX`.

The `ID`-stage adder for computing the branch target has been removed. Instead, the `ALU` will compute the branch target using the existing `EX.NPC` and `EX.IMM` inputs. A new connection from `ME` to the `IF`-stage mux has been added so the branch target can reach the `PC`, that appears in green.

Problem 1, continued: Consider the implementation from the previous part.

(*c*) Resolving the branch in ME with prediction can hurt performance with the code below compared to resolving in ID with prediction when the branch is hard to predict. By how much will it hurt performance? *Note: An exact number was not required on the original exam.* Explain, use a diagram if necessary. *Hint: Resolve is where recovery starts for a misprediction. Another hint: pay attention to dependencies on the branch condition.*

```
#         SOLUTION
LOOP1:  # Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)       IF ID EX ME WB
 addi r2, r2, 4        IF ID EX ME WB
 bne  r2, r3 LOOP1        IF ID EX ME WB
 add r4, r4, r1               IF ID EX ME WB
 xor                             IF IDx
 or                                 IFx
#         Cycle     0  1  2  3  4  5  6  7  8  9
 lw r1, 0(r2)                          IF ID EX ME WB
```

☑ Resolving in ME hurts performance here by __1__ cycles.

☑ Explanation.

Because the branch resolution hardware is in ME, when there is a misprediction the correct path instruction will only reach IF when the branch is in WB (the cycle after ME). For the code fragment above this occurs in cycle 6. Compare this to the execution of the code in part (a) of this problem, in which the branch target is fetched in cycle 5. So performance is hurt by one cycle.

Note that though on a misprediction execution takes one cycle longer, when there is not a misprediction execution takes one cycle less since the dependence stall is avoided. See the next part for an example of that.

(*d*) Resolving the branch in ME with prediction should help performance with the code below compared to resolving in ID with prediction. Explain why, preferably with an execution diagram. *Hint: Same hints as previous part.*

```
#      SOLUTION
LOOP2: # Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)       IF ID EX ME WB
 lw r2, 4(r2)          IF ID EX ME WB
 bne  r2, r0 LOOP2        IF ID EX ME WB
 add r4, r4, r1               IF ID EX ME WB
#       Cycle      0  1  2  3  4  5  6  7  8  9  10
 lw r1, 0(r2)                     IF ID EX ME WB
```

☑ Resolving in ME helps performance because...

. . . because we can bypass the load value, in cycle 5 for the example, and so avoid stalls. Contrast this with the execution of the fragment in part (a) in which there is a two cycle stall every iteration. With prediction there is never a dependence stall for the branch, and two-instruction squashes only occur when the branch is mispredicted.

Problem 2: (10 pts) The diagrams below show the execution of code on two-way superscalar dynamically scheduled systems of the type described in class. Each diagram **has mistakes**. Identify and fix them. Also explain why code should not execute as illustrated. The answer should specify what would go wrong, or why the system would be particularly inefficient, as appropriate.

(*a*) Find and fix the two problems with commit in the execution below, and explain why execution would be incorrect or inefficient.

```
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
sub r4, r5, r6   IF ID Q  RR EX WB C
or  r7, r4, r8      IF ID Q  RR EX WB    C
# Cycle          0  1  2  3  4  5  6  7  8

# SOLUTION BELOW, UNFIXED PROBLEM ABOVE.
# Cycle          0  1  2  3  4  5  6  7  8
lw r1, 0(r2)     IF ID Q  RR EA ME WB C
sub r4, r5, r6   IF ID Q  RR EX WB    C
or  r7, r4, r8      IF ID Q  RR EX WB    C
# Cycle          0  1  2  3  4  5  6  7  8
```

☑ Fix the two problems above.

☑ Describe one error or inefficiency with execution.

Commit is supposed occur in program order, but `sub` commits before the `lw`. That's fixed in the solution. If commit did occur out of order it might not be possible to recover the correct register values (in particular the register mappings) should an instruction raise an exception. For example, if some instruction raised an exception in cycle 6 it would be too late to stop the `sub`.

☑ Describe another error or inefficiency with execution.

The code is committing at a rate of one per cycle, but since it's two-way superscalar it should be able to commit at a rate of two per cycle. (If it could only commit at one per cycle then there would be little point in having hardware to fetch, decode, and execute two per cycle.) That is also fixed in the solution.

(*b*) Find and fix the problem with the code below. Note that the processor has two FP adders, named A and B. *Hint: Check dependencies.*

```
add.d f0, f2, f4      IF ID Q  RR A1 A2 A3 A4 WB C
sub.d f6, f0, f8      IF ID Q              RR A1 A2 A3 A4 WB C
add.d f10, f11, f12      IF ID Q           RR B1 B2 B3 B4 WB C
addi r1, r1, 4           IF ID Q              RR EX WB         C

# SOLUTION BELOW, UNFIXED PROBLEM ABOVE.
add.d f0, f2, f4      IF ID Q  RR A1 A2 A3 A4 WB C
sub.d f6, f0, f8      IF ID Q              RR A1 A2 A3 A4 WB C
add.d f10, f11, f12      IF ID Q  RR B1 B2 B3 B4 WB          C
addi r1, r1, 4           IF ID Q  RR EX WB                     C
```

☑ Fix the problem above.

☑ Describe one error or inefficiency with execution.

The `addi` and the second `add.d` start execution later than they need to. Because neither of these instructions has dependencies with the others they can start execution without delay. That is shown in the solution above. Note that in this example there is no impact on execution time but if the second `add.d` were instead a divide instruction then the delayed start would slow down execution.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history. The outcomes of branch B1 form a repeating pattern, the outcome of B2 can be modeled by a Bernoulli random variable with $p = .7$ (the branch is taken with probability .7), and B3 is always taken.

```
    0  0   1   2   3   3   3   2   1     0   1   2   3   3   3   2   1 <- Bimo Ctr
B1:  N   T   T   T   T   T   N   N     N   T   T   T   T   T   N   N
         x   x               x   x         x   x               x   x  <- Bimo pred outc
B2:    R   R   R   R   R   R   R   R     R   R   R   R   R   R   R   R
B3:     T   T   T   T   T   T   T   T     T   T   T   T   T   T   T   T
```

For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal predictor on B1?

The counter used to predict B1, and the outcome of those predictions is shown in the diagram above. (The counter is labeled Bimo Ctr.) Based on these outcomes the accuracy is $\boxed{\frac{4}{8} = .5 = 50\%}$.

☑ What is the approximate accuracy of the bimodal predictor on B2? Explain. *Hint: A correct answer would be slightly more or less (pick one) than.... For the exact answer one would need the state probability formula for a Markov chain.*

If the counter for B2 were always 2 or 3, the accuracy would be .7. However two consecutive not-taken outcomes are far from impossible, and that would switch the prediction to not taken. Therefore the accuracy is less than .7. The exact probability that the counter value is i is $p_i = \frac{\omega-1}{\omega^4-1}\omega^i$, where $\omega = \frac{.7}{.3}$. The probability of a correct prediction is $p_2 + p_3 = .844828$. The probability of a correct prediction is then $0.7(p_2 + p_3) + 0.3(p_0 + p_1) = 0.5914 + 0.0466 = 0.638$.

☑ What is the accuracy of the local predictor on branch B1?

The branch B1 pattern has a length of 8, which easily fits in the 10-outcome local history, and so $\boxed{\text{the accuracy is 100\%}}$.

☑ What is the warmup time of the global predictor on branch B1? Be sure to consider the effect of B2.

First, find the possible GHR values used to predict B1. The GHR patterns are: tTrtTrtTrt, tTrtTrtNrt, tTrtNrtNrt, tNrtNrtNrt, tNrtNrtTrt, tNrtTrtTrt. The lower-case ts are an outcome of B3, the rs are the outcomes of B2, and the upper-case letters are outcomes of B1; the rightmost outcome is the most recent. The reason these are referred to as patterns and not actual GHR values is because the outcome of branch B2 is shown as an r, which can be either a t or n. Therefore for a pattern such as tTrtTrtNrt there are $2^3 = 8$ actual GHR values. Branch B1 needs to be executed twice to warm up the GHR, and each subsequent GHR value needs to be seen three times to warm up the PHT entries. So the total warmup time is $3 + 6 \times 2^3 \times 2 = 99$ outcomes.

☑ What is the minimum local history size needed for a local predictor to predict B1 with 100% accuracy?
$\boxed{\text{Five outcomes.}}$

If it were four outcomes the local history TTTT would occur before both a taken and not-taken outcome.

☑ What is the accuracy of a global predictor with a three-outcome global history on branch B1?

Branch B1 can only be predicted using its own prior outcomes. With a global history length of 3 all B1 can see is one of its prior outcome. If that prior outcome is N (making pattern Nrt) then the outcome will be N 2 out of 3 times (over a repetition of the

length-8 pattern). If that prior outcome were T the outcome would be T 4 out of 5 times. The PHT entries would predict the more common outcomes, so the accuracy would be $\frac{2+4}{3+5} = 0.75$.

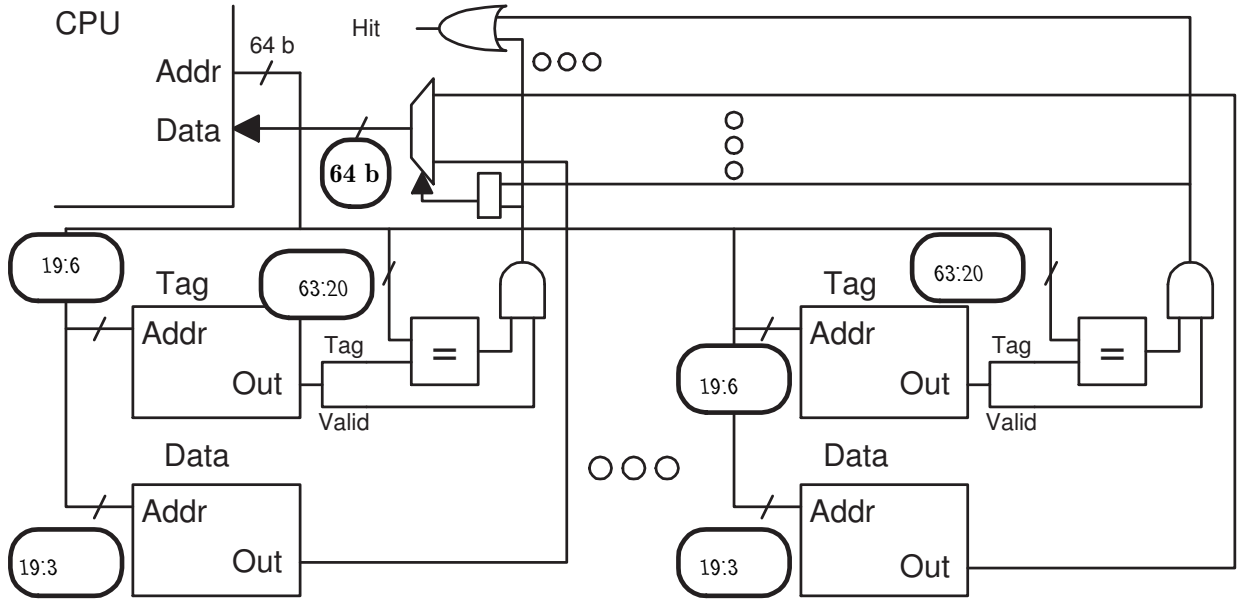
☑ What is the minimum global history size needed for a global predictor to predict B1 with 100% accuracy?

Branch B1 must see 5 prior outcomes to be predicted perfectly. That would require a global history length of $5 \times 3 = 15$.

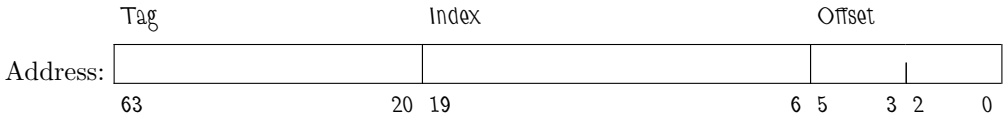
Problem 4: (20 pts) The diagram below is for a set-associative cache with a capacity of 16 MiB (2^{24} bytes), and a line size of 64 bytes. The system has the usual 8-bit characters. Each data store below has a capacity of 2^{20} bytes.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



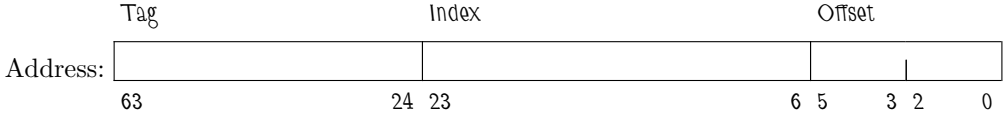
☑ Associativity:

The associativity is 16. The associativity is determined based on the given cache capacity, 2^{24} bytes, and the given capacity of an individual data store, 2^{20} bytes. There must be 16 data stores for the given cache capacity.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{24} characters, plus $16 \times 2^{20-6}\,(64 - 20 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 4, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 256-byte line size. Each code fragment starts with the cache empty; consider only accesses to the arrays, `a`, `ax`, and `ay`.

(*b*) Find the hit ratio executing the code below.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
double *a = 0x20000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 2^8 characters is given, the size of an array element is $8 = 2^3$ characters, and so there are $2^{8-3} = 2^5$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^5 elements, and so the next $2^5 - 1$ accesses will be to data on the line. The access at `i=32` will miss and the process will repeat. Therefore the hit ratio is $\frac{31}{32}$.

(*c*) The two code fragments below, labeled Method 1 and Method 2, perform the same computation but organize the data differently.

```
// Method 1
 struct Our_Struct { int x;  int y; };
 Our_Struct a[SIZE];

 for (int i=0; i<SIZE; i++) sum += a[ i ].x;
 for (int i=0; i<SIZE; i++) myfunc(sum,a[ i ].y);

// Method 2
 int ax[SIZE];
 int ay[SIZE];

 for (int i=0; i<SIZE; i++) sum += ax[ i ];
 for (int i=0; i<SIZE; i++) myfunc(sum,ay[ i ]);
```

☑ Which method has a higher hit ratio if the cache is small (or `SIZE` is large)? Explain

On each miss a line's worth of data is brought into the cache. With Method 1, because the data is organized into structures, a miss will bring in data which is not soon needed (the `y` values for the first loop and the `x` values for the second loop). When the cache is small those not-soon-needed values will be evicted from the cache before they are needed. Suppose the line size is 256 bytes and the integer sizes are 4 bytes each. A line with Method 1 holds $256/8 = 32$ elements, but a line using Method 2 holds $256/4 = 64$ elements. Therefore the hit ratio will be higher on the code above with Method 2 when the cache is small. If the cache were larger Method 1 would have about the same performance as Method 2 because the cache would be large enough to hold the not-soon-needed values long enough for the second loop.

Problem 5: (30 pts) Answer each question below.

(*a*) Eliminating bypass paths from an n-way superscalar statically scheduled processor would have a much larger effect than eliminating them from a scalar statically scheduled processor (like our 5-stage MIPS implementation).

☑ Describe a positive benefit of eliminating the bypass paths that's much larger for the n-way superscalar.

Full-Credit Answer: The cost of bypass paths in a superscalar processor is $\propto n^2$, much larger than their cost in a scalar processor, and much larger than an acceptable n-times higher cost.

Long Explanation: In an n-way, 5-stage superscalar processor derived from our familiar 5-stage MIPS implementation there would be bypasses from each of n ALU results in the ME stage and n writeback values in the WB stage, to both inputs to each of n ALU's in the EX stage, for a total of $4n^2$ multiplexer inputs. The cost of these bypass paths is much larger in the superscalar processor, even after taking into account the expected n times higher cost.

☑ Describe a disadvantage of eliminating the bypass paths that's a much bigger disadvantage for the n-way superscalar.

No bypass is needed in the scalar processor if dependent instructions are separated by two instructions (see diagram below), but in an n-way superscalar the separation must be at least $2n$ instructions. Therefore there would be many more added stalls in the superscalar processor.

```
add r1, r2, r3  IF ID EX ME WB
or  r6, r7, r8     IF ID EX ME WB
and r9, r7, r8        IF ID EX ME WB
sub r4, r1, r5           IF ID EX ME WB
```

(*b*) Consider a deeply pipelined system with $5n$ stages without bypass paths and that runs programs in which dependent instructions are far apart. Name two factors that will limit clock frequency for larger values of n. *Note: bypass paths were not mentioned in the original exam.*

☑ One frequency limiter.

The clock period includes time for the logic within a stage and time for the latch itself. Ideally, the time for the logic is reduced by n, but the time for the latch remains constant. So the clock period can be no lower than the latch time.

☑ Another frequency limiter.

In the $5n$-stage system logic within each stage in the original 5-stage system must be split into n pieces. If that is done ideally the time for that logic would be t_l/n, where t_l is the logic time in a stage in the 5-stage system. But because logic cannot be perfectly divided some stages might take longer.

(*c*) Consider the BHT of a bimodal branch predictor. An entry would contain a CTI type, a two-bit counter, a branch target, and perhaps a tag. The tag would be used like a cache tag, but does not need to be as large. For this problem only consider branches.

☑ How large would the target need to be to predict MIPS branches? *Hint: The answer is not 32 bits.*

Since the address of the branch is known, the target field need only hold the displacement. The predictor hardware can then compute `PC + 4 + 4 * DISP`. So the target field size would be 16 bits.

☑ Explain what the predictor can do with the tag to improve prediction accuracy.

If the tag doesn't match then the target will very likely be wrong, in fact, the instruction being fetched might not be a branch (or other CTI) at all. So when the tag doesn't match predict not taken, which is correct for non-CTIs and about half the branches.

(*d*) A four-way statically scheduled processor is much less expensive than a four-way dynamically scheduled processor. Why would the dynamically scheduled processor run some code faster than the statically processor even when the code was compiled with a good compiler that targeted the specific four-way implementation?

☑ Specific advantage of dynamically scheduled processor for some code.

The compiler can schedule for fixed (always the same) latencies, such as those for floating point instructions, but the latency of load instructions depends upon where the data is found (*e.g.*, L1 cache, L2 cache, memory). In a region of code containing several loads, a compiler might be able to schedule one of those loads assuming an L2 cache hit (an L1 miss), but it would be difficult to find enough non-dependent instructions to schedule assuming all loads missed the cache. In contrast the dynamically scheduled processor could execute non-dependent instructions following any load that missed. Note: An L2 hit has on the order of a 15 cycle latency.

Another advantage is for branches. The compiler has most freedom to schedule within basic blocks, but in many types of programs they are small, about 10 instructions. Though a compiler can draw instructions from before and after a basic block this can slow code down because some of those *boosted* instructions will not need to be executed for some branch outcomes. In contrast the dynamically scheduled processor works with the actual path taken (actual branch outcomes) and so is not affected by branches.

☑ Describe properties of code that would run at the same rate on the two processors.

Code that has large basic blocks, loads that always hit the L1 cache, and that has large distances between dependent instructions. A compiler can do a good job scheduling this kind of code, and so the dynamically scheduled processor's advantages are won't apply.

(*e*) In many VLIW ISAs the bundle size is 3 slots. Describe a disadvantage of making a VLIW ISA with a larger bundle size.

☑ Disadvantage of larger bundle size.

Branch targets in VLIW ISA's must be at the start of a bundle. So if the bundle size were larger the compiler might need to insert more `nop` instructions at the end of a bundle to push a branch target to the beginning of a bundle.

(*f*) A tester measuring the performance of a system using SPECcpu has the source code to SPECcpu programs. Why is that important to the goals of the suite?

☑ Source code important to SPECcpu goals?

A goal of SPECcpu is to measure the performance potential of new implementations and new ISA's. For the performance potential of a new implementation of an existing ISA to be reached the code has to be compiled for that implementation. This can be done by the tester using the source code. (With the alternative of having SPECcpu distribute binaries [compiled code] the code might be compiled for some older implementation.) The source code is completely necessary new ISA's (say, if the tester had the very first system with this new ISA), since there would be no way for the binary to be packaged with SPECcpu.

56 Fall 2010 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

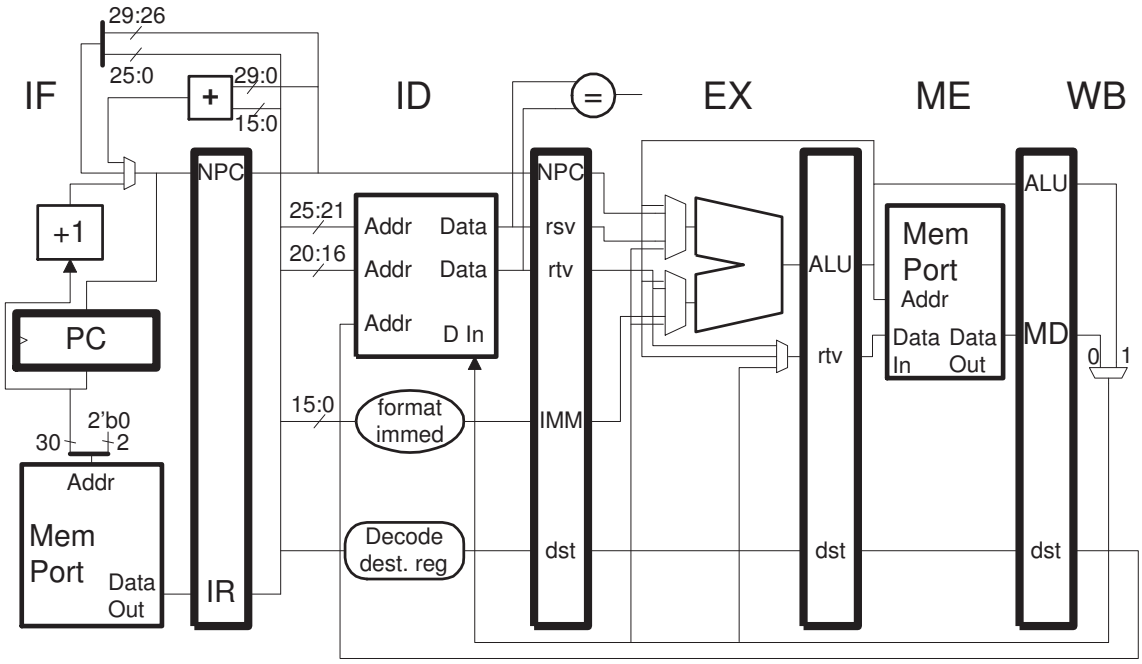
Wednesday, 3 November 2010, 10:40–11:30 CDT

Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(23 pts)
Problem 5		(10 pts)
Problem 6		(7 pts)
Exam Total		(100 pts)

Alias ¡Mi! ¡Mi! ¡Mi! ¡Ips! ¡Ips! ¡Ips! ¡Cinco etapes hacen MIPS!

Good Luck!

Problem 1: Show the execution of the following code fragments on the illustrated MIPS implementations.



(*a*) [20 pts] The code below executes for many iterations. Show a pipeline execution diagram for the execution of the code on the implementation above for enough iterations to determine the CPI, and determine the CPI.

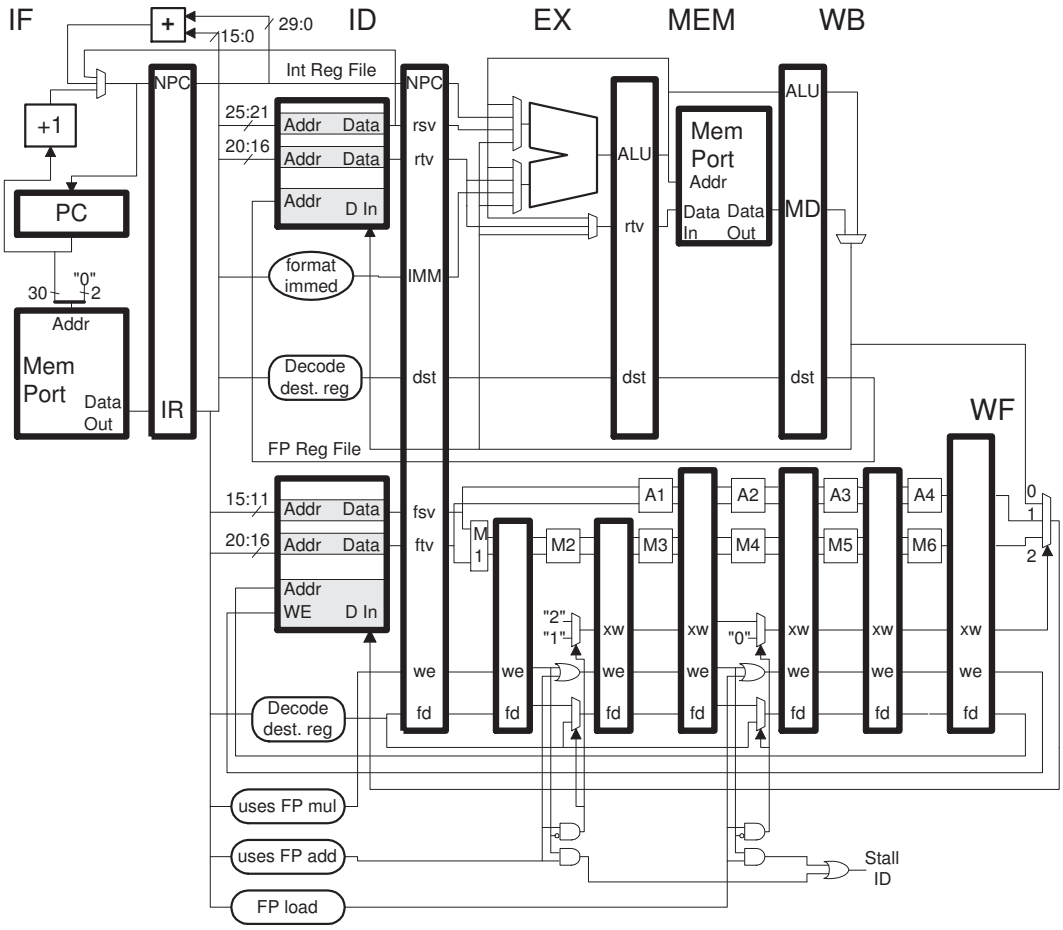
☑ Pipeline diagram of execution. Don't forget to check for dependencies!

```
# SOLUTION
LOOP: # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)  IF ID EX ME WB                                 First Iteration
 lw r3, 0(r1)     IF ID -> EX ME WB
 bne r3, r4          IF -> ID ----> EX ME WB
 lw r2, 8(r1)              IF ----> ID EX ME WB
LOOP: # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)                      IF ID -> EX ME WB        Second Iteration
 lw r3, 0(r1)                         IF -> ID -> EX ME WB
 bne r3, r4                                IF -> ID ----> EX ME WB
 lw r2, 8(r1)                                    IF ----> ID EX ME WB
LOOP: # Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
 lw r1, 0(r2)        Third Iteration                          IF ID -> EX ME WB
```

☑ CPI for a large number of iterations.

The first iteration starts in cycle 0, the second in cycle 7, the third in cycle 15. The state of the pipeline at the beginning of the second and third iterations is identical: `lw r1` in `IF`, `lw r2` in `ID`, and `bne` in `EX`. Therefore the third iteration will execute identically to the second and the time for the second iteration, $15 - 7 = 8$ cycles, will be the same as the third, etc. The CPI is $\frac{8}{4} = 2$.

Problem 1, continued:



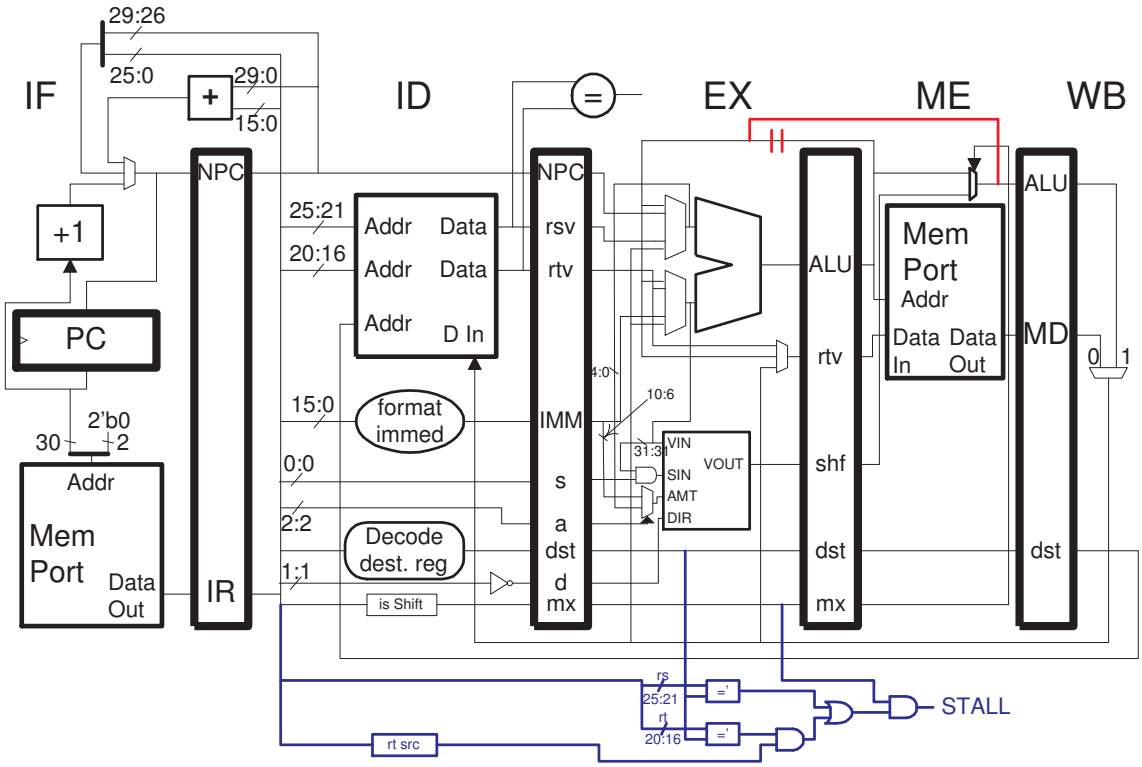
(*b*) [10 pts] Show the execution of the code below on the implementation above.

☑ Pipeline diagram of execution.

```
# SOLUTION
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.s f1, f2, f3  IF ID M1 M2 M3 M4 M5 M6 WF
add.s f4, f5, f6     IF ID A1 A2 A3 A4 WF
add.s f7, f8, f9        IF ID -> A1 A2 A3 A4 WF
lwc1 f10, 0(r1)            IF -> ID ----> EX ME WF
```

Problem 2: The MIPS implementation below includes the shift unit (taken from the Homework 3 solution). Notice that it is not possible to bypass a shift result to the EX stage. For that reason the `add` in the code below stalls:

```
# Cycle          0  1  2  3  4  5  6
sll r1, r2, r3   IF ID EX ME WB
add r4, r1, r5      IF ID -> EX ME WB
```



(*a*) [10 pts] Design control logic to generate a signal named `STALL` which will be logic `1` when a stall is needed due to a shift result that cannot be bypassed, as in the example above. (Generate the signal, but don't do anything with it.)

☑ Logic to generate stall signal.

Solution appears above in blue, on the bottom of the diagram. (Ignore for now the changes appearing in red at the top of the diagram.) Consider the state of the pipeline in cycle 2 in the example above, this is the cycle that the pipeline stalls. The two comparison units, =', compare the destination of the instruction in `EX`, `r1` in the example, with the sources of the instruction in `ID`, `r1` and `r5` in the example. The `rs` register is always compared (which isn't 100% right), but the `rt` register comparison is set to false if the instruction in `ID` does not use the `rt` register as a source. The stall signal should only be generated if the instruction in `EX` is a shift, that is the purpose of the last AND gate.

(*b*) [5 pts] The stall above can be avoided by disconnecting the `ME`-to-`EX` bypass from the `EX/ME.ALU` latch and instead connecting it to the output of the `ME`-stage mux. What was the reason for **not** doing something like that in Homework 3.

☑ Problem with `ME`-to-`EX` bypass for shift results.

The connection appears on the top of the diagram, in red. Note that the parallel red lines indicate that the wire is broken. In Homework 3 the shifter was to be added to two different systems, one in which the ALU was not on the critical path, and one in which

it was. If the ALU is on the critical path then adding logic between the pipeline latch outputs and the input to the **EX/ME.ALU** latch will, by definition of a critical path, slow things down. The change shown in red adds the ME-stage mux to the ALU critical path, which was the reason for not doing it.

Problem 3: Answer the following questions about interrupts.

(*a*) [7 pts] An important part of an ISA's interrupt mechanism is a separate privileged mode (also called system or supervisor mode) and user mode.

☑ Why is it necessary to have these two modes?

Acceptable answer on test: So that an OS (which runs in privileged mode) can control access to resources such as disk I/O that are requested by user code (in user mode).

Longer description: These modes enable one to set up one group of software that manages certain resources and another group of software that can only get access to the resources by requesting them from the first group. A common example of the first group is the operating system kernel and the second group might be a computer science homework assignment, or your word processing program, etc.

Examples of resources at a higher level are disk (or filesystem) access, memory, and CPU time.

☑ Explain the difference between privileged and user mode in how the CPU operates.

Under privileged mode any valid memory address can be accessed. Under user mode only certain memory address can be accessed, an attempt to access other addresses would cause an exception. Under privileged any instruction can be executed, such as instructions to manage the memory system. Under user mode such instructions cannot be executed.

(*b*) [8 pts] In class our 5-stage implementations resolved exceptions only in `ME`, and by doing so all integer-pipeline exceptions were precise. *Note: In the original exam the phrase "and by doing so..." was not present.* Suppose instead we resolved exceptions in `WB`. Show a code fragment in which an exception could not be precise on such a system.

☑ Simple code fragment.

```
# SOLUTION
# Cycle       0   1   2   3   4   5
lw r3, 0(r4)  IF  ID  EX  ME  WB
sw r1, 0(r2)      IF  ID  EX  MEx   Too late to stop store.
```

In the code fragment above, the `lw` raises an exception in `ME` but because it is not resolved until `WB`, the store initiated by the next instruction cannot be stopped.

☑ What can't the handler do, and why can't it do it?

The handler cannot see the state of the program as it would be before the `lw` because the `sw` has written memory.

Problem 4: Answer each question below.

(*a*) [8 pts] Consider the distinction between an ISA and its implementation.

☑ What was to be achieved by separating computer design into separate ISA design and implementation design?

Software compatibility between the first implementation and implementations done years later. This avoids the effort needed to port code to a newer design, and the risk of having customers switch vendors.

When designing the ISA separately features are chosen not just based on what can be done with the current implementation, but what would be easy or hard on future implementations.

Grading Note: Surprisingly few answers explicitly mentioned software compatibility.

☑ Provide a reason not to have separate ISA and implementation design, consider the point of view of a conservative computer engineer from the 1960s.

When designing an ISA for longevity one might make decisions that hobble the first implementation. An engineer accustomed to designing implementations for a particular purpose (perhaps considering a few benchmark programs) would consider the need to provide for a large address space or have "unnecessary" instructions wasteful of time and vacuum tubes (or those newfangled transistors).

(*b*) [8 pts] One reason to not compile a program with optimization turned on is because you plan to debug the program. Explain why stepping through a program in a debugger can be confusing when the code has been optimized.

☑ Debugging optimized code is confusing because . . .

. . . when single-stepping through a program execution will not be in the same order as the high-level code specifies. In fact, certain lines of code might not be executed at all and the value of certain variables cannot be printed.

☑ Name a particular type of optimization and explain how it can cause confusing results when single stepping through a program.

Instruction Scheduling: Can cause high-level statements to be executed out of order.

Dead-Code Elimination: A line of dead code will not executed, and a variable that the line may write cannot be printed.

The following answers are wrong in the sense that they would not make debugging difficult. Points were not deducted because they are legitimate optimizations.

Profiling / Code Layout: First profile to determine more likely branch directions, then layout code so that the more likely direction is not taken. This won't by itself make debugging more difficult because it does not change the order in which high level statements are executed. (Location in memory is not the same as execution order.)

Operation Substitution: For example, replacing a multiply by a small constant by a sequence of shifts and adds. Another example is replacing multiplication, division, and modulus operations with power-of-two operands by shifts or ANDs, and then reporting the user to the Bad Programmer Registry. This is wrong because it does not change execution order nor does it affect values of variables. The person using the debugger can't tell whether it was a real multiply (or divide or modulus) or something else. (Unless the person prints the assembly code or is using really slow CPU.)

(*c*) [7 pts] Do you agree or disagree with the statement below regarding the rules for building the SPECcpu benchmarks? Answer with respect to the goals for the SPECcpu benchmarks.

"Testers should not be allowed to use their own compilers because the SPECcpu benchmarks are supposed to test CPUs, not compilers. All testers of a particular ISA should use the same compiler." *Grading note: the phrase "of a particular ISA" was not in the original exam.*

☑ Agree or disagree? Explain.

Note: the key phrase is "are supposed to test CPUs, not compilers."

Disagree. When used properly the compiler will optimize code for a particular implementation, for example, scheduling code to avoid stalls. Certain implementation features may have been designed hand-in-hand with the compiler optimizations needed to fully exploit them. In a sense, the compiler back end is part of the implementation. So to not test the compiler is to not fully test the implementation.

Problem 5: [10 pts] CISC programs generally are smaller than RISC programs.

(*a*) Show how the instruction below is encoded in MIPS and in VAX. For MIPS the name and bit position of each field should be known, and the value of all but one field should be known. For VAX one should know the fields and their sizes, but not every field value needs to be known. *Hint: VAX has 16 general-purpose registers. The size of the two instructions should be the same.*

```
add r1, r2, r3
```

☑ Encoding in MIPS.

```
SOLUTION:

-------------------------------------------------------------------------
! opcode    ! rs      ! rt      ! rd      ! sa      ! function  !
! 0         ! 2       ! 3       ! 1       ! 0       ! 0x20      !
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

☑ Encoding in VAX.

```
SOLUTION:
Note: Encoding is for VAX instruction addl3 r1, r2, r3

  !    Opcode     !  !   Source 1    !   !   Source 2    !   ! Destination   !
  !               !  ! Mode  ! Rn    !   ! Mode  ! Rn    !   ! Mode  ! Rn    !
  !    0xc1       !  !   5   ! 2     !   !   5   ! 3     !   !   5   ! 1     !
   7 6 5 4 3 2 1 0    7 6 5 4 3 2 1 0     7 6 5 4 3 2 1 0     7 6 5 4 3 2 1 0
```

(*b*) Identify unused field(s) in the MIPS instruction.

☑ Unused MIPS field.

The `sa` (shift amount) field.

(*c*) Since VAX instruction sizes can vary they should be able to use space more efficiently. Yet even though the MIPS instruction has unused field(s) the two instructions are the same size. Explain why the VAX instruction is larger than one might expect and explain what advantage that provides.

☑ Why is VAX larger than one might expect?

Because **each** operand has an 4-bit field specifying the addressing mode, whereas in MIPS the addressing mode is: part of the opcode, the same for all three operands, and for integer add instructions there are only two choices (the second source is either a register or an immediate).

In more detail: MIPS Opcode: $6 + 6 = 12$ bits. MIPS register numbers $3 \times 5 = 15$ bits. Unused 5 bits (5 more than VAX). VAX Opcode, including operand specifiers: $8 + 3 \times 4 = 20$ bits (8 more than MIPS). Register numbers: $3 \times 4 = 12$ (3 less than MIPS).

☑ What advantage does this larger size provide?

A large variety of addressing modes. That can reduce the total instruction count (compared to MIPS) by eliminating loads, stores, initializing large constants, etc.

Problem 6: Answer each question below.

(*a*) [7 pts] Unlike MIPS, SPARC integer branches use condition codes.

☑ Why might this allow SPARC branch targets to be further away than MIPS branch targets?

Because in MIPS one needs two register fields to indicate the registers to compare, that takes ten bits. In SPARC V8 integer instructions there is just one set of condition codes and so zero bits are needed for them (and in V9 there are just two, taking 1 bit). The space saved for the register numbers in SPARC can be used for a larger displacement.

☑ Why might this enable certain SPARC implementations to have a higher clock frequency than comparable MIPS implementations?

Because in MIPS I one needs to compare up to two 32-bit registers. For resolve-in-ID implementations (such as the one used in class) the register values are not available until partway through the cycle, likely forcing the branch direction logic to be on the critical path. In SPARC one just needs to look at a single 4-bit condition code register, and so the logic would be faster.

Name Solution

Computer Architecture

EE 4720

Final Examination

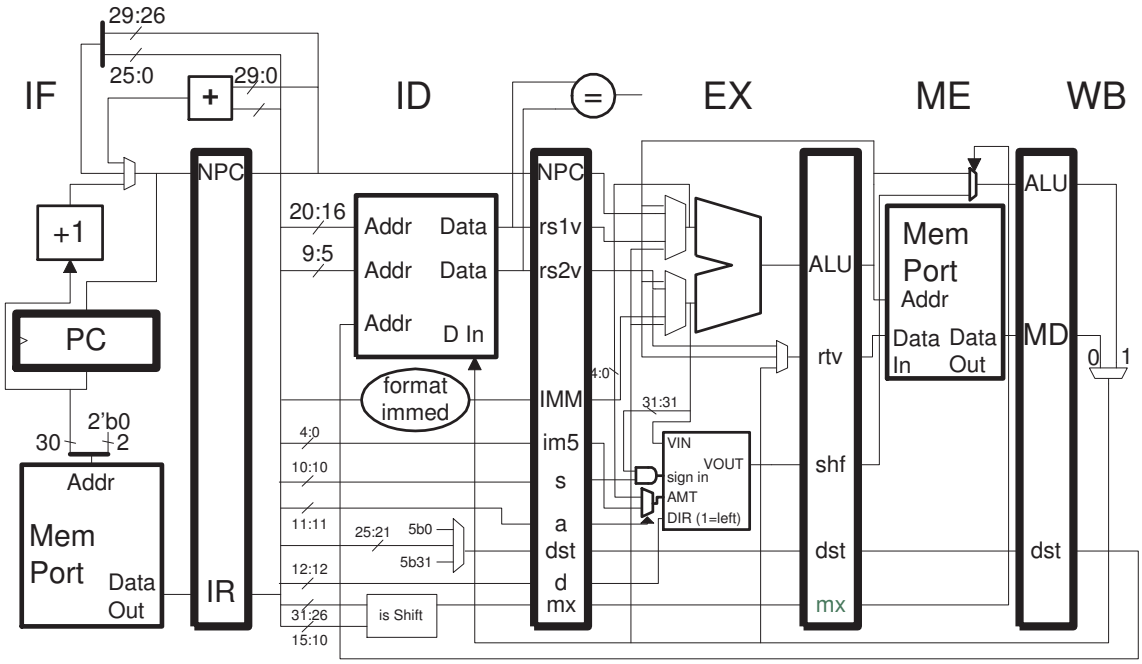
6 December 2010, 7:30–9:30 CST

Problem 1		(10 pts)
Problem 2		(14 pts)
Problem 3		(14 pts)
Problem 4		(14 pts)
Problem 5		(15 pts)
Problem 6		(15 pts)
Problem 7		(18 pts)
Exam Total		(100 pts)

Alias On File

Good Luck!

Problem 1: (10 pts) The diagram below shows a 5-stage pipeline that looks alot like our familiar MIPS implementation but it's actually an implementation of ISA X. (The diagram is based on the solution to Homework 3, in which a shift unit was added to MIPS.)



(a) ISA X instruction format T encodes the shift instructions and others, it is the equivalent of format R in MIPS. Based on the diagram above show the encoding for ISA X format T.

☑ Format T encoding, including bit positions and field names.

Solution appears below. The source registers are named `rs1` and `rs2`, those names are taken from the ID/EX pipeline latches, the bit positions at the input to the register file provide their place in the instruction encoding. The input to the is Shift unit provide the location of the opcode fields. Through its connection to the shifter one finds that bits `4:0` are the equivalent of the MIPS `sa` field. The mux at the input to the `dst` pipeline latch provides the destination register field bits, `25:21`.

(b) Consider the shift instructions `sll`, `sllv`, `srl`, `srlv`, `sra`, and `srav`. Suppose that the encoding of one of these instructions is zero (meaning that every field value is zero). Show the opcode field value(s) for each of these instructions based on the diagram above. *Hint: The control signal for each top mux input is 0, etc.*

☑ Opcode field value(s) for: `sll`, `sllv`, `srl`, `srlv`, `sra`, and `srav`.

From inspection of the diagram we see that bit 10 determines whether the shift is arithmetic (signed) (bit 10 is `1`) or logical (unsigned) (bit 10 is `0`). From inspection of the diagram we see that bit 11 determines whether the shift amount is obtained from the `rs1` register (possibly bypassed) or if bit 11 is `1` whether the shift amount is obtained from the `im5` field. From inspection of the diagram we see that bit 12 indicates the direction, with `1` for a left shift. Since one of the shift instructions has a zero opcode, the `opc` field must be zeros and bits `15:13` of `opT` must be zero. Putting these bits together we get the `opT` values shown below:

```
Solution:
   sll    sllv    srl    srlv    sra   srav
   110    100     010    000     011    001   <- Bits 12, 11, and 10
000110 000100  000010 000000  000011 000001   <- Full opT value.
```

(*c*) Explain why the implementation of instructions such as `sw r1,2(r3)` and `beq r1, r2 TARG` would be less elegant for ISA X than for MIPS. *Hint: It has something to do with registers.*

☑ `sw` and `beq` less elegant because...

Consider instructions `lw r1,2(r3)` and `sw r1,2(r3)`. In MIPS these can use the same format because the `rt` field can be either a destination (as its used for `lw`) or as a source (as its used for `sw`). But in ISA X, based upon the diagram above, there is only one destination field. So the `lw` encoding might be in a format where the `rs2` field is omitted (and so bits `9:5` can be part of the immediate) while the `sw` instruction would be in a format where `rd` was omitted. As a result the `format immed` unit needs to pick different sets of bits based on format.

Problem 2: (14 pts) Answer the questions below.

(*a*) Complete the execution diagram for the MIPS code below on a two-way superscalar statically scheduled implementation of the type described in class.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8
add r1, r2, r3   IF ID EX ME WB
add r4, r5, r6   IF ID EX ME WB
add r7, r4, r8      IF ID EX ME WB
lw  r9, 0(r7)       IF ID -> EX ME WB
addi r1, r9, 3         IF -> ID -> EX ME WB
xor r10, r11, r12      IF -> ID -> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8
```

☑ Complete the diagram above.

The solution appears above. The `lw` stalls due to a dependence on the preceding `add`. Notice that the stall affects all following instructions. That is, though in cycle 3 the `lw` is stalled in $\texttt{ID}^1$ the $\texttt{ID}^0$ stage is empty. It would be possible to design the pipeline so that the `addi` could move into $\texttt{ID}^0$ in cycle 3, however that would greatly complicate control logic since instructions would be out of order. (The instruction in $\texttt{ID}^0$ would come after the one in $\texttt{ID}^1$ in program order, whereas usually the instruction in $\texttt{ID}^0$ is before the one in $\texttt{ID}^1$.) So to keep instructions in order the `addi` must stall.

(*b*) Show the execution of the code below on an 8-way superscalar statically scheduled processor of the type described in class. Branches are not predicted. Find the CPI for a large number of iterations.

```
# SOLUTION
LOOP:  # Cyc     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
and r1, r1, r5   IF ID EX ME WB
add r3, r3, r1   IF ID -> EX ME WB
lw r1, 0(r2)     IF ID -> EX ME WB
bne r2, r4 LOOP  IF ID -> EX ME WB
addi r2, r2, 4   IF ID -> EX ME WB
or               IF ID -> x
or               IF ID -> x
or               IF ID -> x
xor                 IF -> x
xor                 IF -> x # Six more instructions squashed.
LOOP:  # Cyc     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
and r1, r1, r5            IF ID EX ME WB
add r3, r3, r1            IF ID -> EX ME WB
lw r1, 0(r2)              IF ID -> EX ME WB
bne r2, r4 LOOP           IF ID -> EX ME WB
addi r2, r2, 4            IF ID -> EX ME WB
or                        IF ID -> x
or                        IF ID -> x
or                        IF ID -> x
xor                          IF -> x
xor                          IF -> x # Six more instructions squashed.
LOOP:  # Cyc     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
and r1, r1, r5                    IF ID EX ME WB
```

☑ Complete diagram above for enough iterations to determine CPI.

The solution appears above.

☑ Find the CPI for a large number of iterations.

The third iteration starts, in cycle 6, with the pipeline in the same state as the start of the second iteration, in cycle 3. (The pipeline state is **and** in **ME**, **add**, **lw**, **bne**, and **addi** are all in **EX**, etc.) Therefore the time for the second iteration, $6 - 3 = 3$ cycles will be the same as the time of the third. The CPI is then $\boxed{\frac{6-3}{5}\,\text{CPI}}$.

(*c*) Complete the execution diagram for the MIPS code below on a two-way superscalar dynamically scheduled implementation of the type described in class. The execution of the first instruction is shown. The `lw` instruction uses stages `EA` and `ME` in place of `EX`.

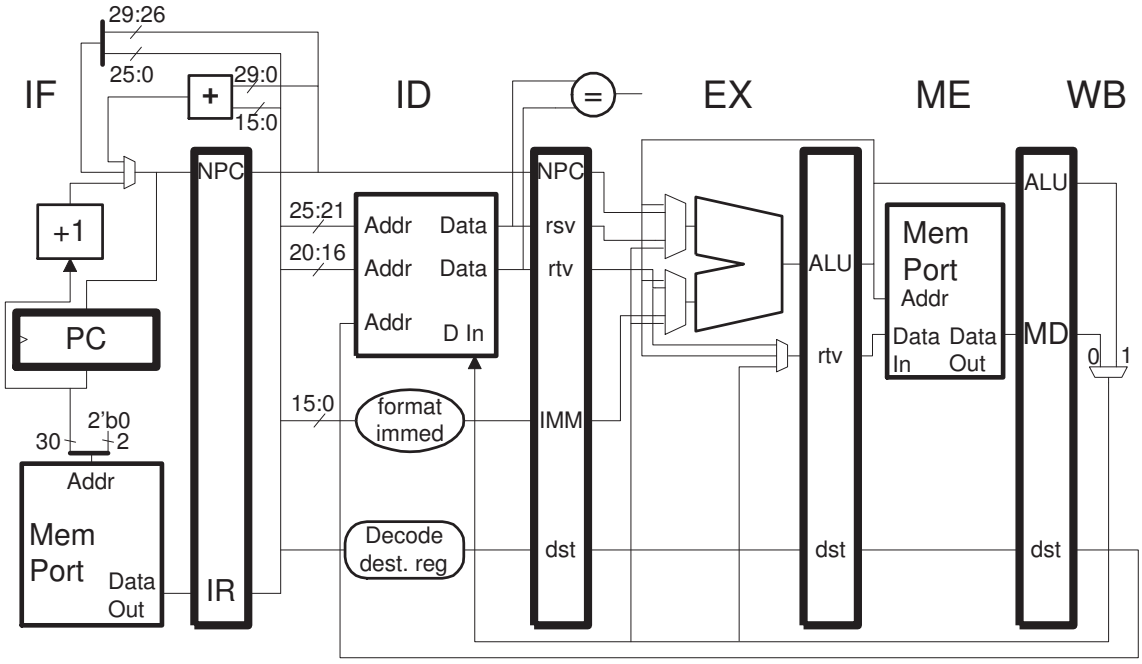
```
# Solution
LOOP:  # Cyc     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
add r1, r2, r3   IF ID Q  RR EX WB C
add r4, r5, r6   IF ID Q  RR EX WB C
add r7, r4, r8      IF ID Q  RR EX WB C
lw  r9, 0(r7)       IF ID Q     RR EA ME WB C
addi r1, r9, 3         IF ID Q        RR EX WB C
xor r10, r11, r12      IF ID Q  RR EX WB       C
LOOP:  # Cyc     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

☑ Complete diagram above.

Solution appears above. Notice that the dependence chain from the `add r4` through the `lw` and `addi` does not affect when the `xor` executes, though it does delay its commit.

Some important things to remember: Commit must occur in program order. Since the processor is 2-way superscalar at most two instructions per cycle can commit (by default for classroom examples).

Problem 3: (14 pts) A deeply pipelined MIPS implementation can be constructed by dividing some stages of our familiar 5-stage statically scheduled scalar implementation (shown below) into two or more parts. In this problem the technique is applied to construct several 8-stage implementations. All have just one `ID` and `WB` stage, and in all implementations **it takes** 4.4 ns **for an instruction to pass through all 8 stages**, from the beginning of `IF1` to end of `WB1`. The stages are divided without changing what they do. For example, if an "original" MIPS stage, say `EX`, is divided into multiple stages, say `EX1 EX2 ... EXn`, then all values from `ID` and bypass paths are needed when `EX1` starts, and values reach `ME` in the cycle after `EXn`. Our familiar 5-stage implementation is shown below for reference:



(*a*) Consider the 8-stage *baseline* implementation below (indicated by the stage labels). What is the execution rate of a friendly program (one written to maximize performance) on the implementation, in units of instructions per second? The answer can be given as a formula of constants (as opposed to putting down just x as an answer).

`Baseline Implementation: IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1`

☑ Execution rate in instructions per **second**.

If it takes 4.4 ns to pass through 8 stages then the clock frequency must be $\frac{8}{4.4\,\text{ns}} = 1.82\,\text{GHz}$. So the execution rate is 1.82 billion insn per second.

(*b*) Call a program *favorable* for an implementation if it runs faster on the implementation than on the baseline (repeated below). If it runs slower, call it *unfavorable*. For each implementation below write a two- or three-instruction favorable program and a two- or three-instruction unfavorable program. Also provide a concise but clear explanation of what it is about the program and implementation that makes it favorable or unfavorable. If a program is favorable on one implementation and unfavorable on another write it once, but provide an explanation for each. *Hint: The three implementations differ in how they are affected by certain dependencies.*

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 1:  IF1 IF2 IF3 ID1 EX1 ME1 ME2 WB1
```

☑ Favorable program. ☑ Explanation.

☑ Unfavorable program. ☑ Explanation.

In Implementation 1 there is one more IF stage than the baseline but one less EX stage. A favorable program would have a close dependence from ALU output to ALU input. An unfavorable program would have a dependence from anywhere to the IF input, which would be a control transfer, including a branch. From the pipeline diagram below for the unfavorable program one can see that because of the extra IF stage the target is fetched one cycle later and so one more instruction is squashed than would be in the baseline. Without a branch predictor these squashes would occur for every taken branch, with a branch predictor they would only happen when the branch is mispredicted, but that's still worse than the Baseline.

```
# SOLUTION
# Favorable for 1:  Second instruction has true dep with first, both are ALU.
 add r1, r2, r3
 sub r4, r1, r5

# Unfavorable for 1: Taken branch
 beq r1, r2 TARG   IF1 IF2 IF3 ID1 EX1 ME1 ME2 WB1
 and                   IF1 IF2 IF3 ID1 EX1 ME1 ME2 WB1
 or                        IF1 IF2x
 sub                           IF1x

TARG:
 xor                                 IF1 IF2 IF3 ID1 EX1 ME1 ME2 WB1
```

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 2:  IF1 ID1 EX1 EX2 EX3 ME1 ME2 WB1
```

☑ Favorable program. ☑ Explanation.

☑ Unfavorable program. ☑ Explanation.

Implementation 2 has one fewer IF stage, but one more EX stage. So the favorable program from the last part is unfavorable here, and the unfavorable program from the last part is favorable here.

```
Baseline Impl.:    IF1 IF2 ID1 EX1 EX2 ME1 ME2 WB1
Implementation 3:  IF1 ID1 EX1 EX2 ME1 ME2 ME3 WB1
```

☑ Favorable program. ☑ Explanation.

☑ Unfavorable program. ☑ Explanation.

Implementation 3 also has one fewer IF than the baseline, but it has one more ME than the baseline. So the favorable program is the same as the favorable program from the last part. The unfavorable program has a dependence through memory.

```
# SOLUTION - Unfavorable Program
lw r1, 0(r2)       IF1 ID1 EX1 EX2 ME1 ME2 ME3 WB1
add r3, r1, r4         IF1 ID1 --------------> EX1 EX2 ME1 ME2 ME3 WB1
```

Problem 4: (14 pts) Answer the following branch predictor questions.

(*a*) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 16-outcome local history, and one system uses a global predictor with a 16-outcome global history.

```
0x1000: B1:  T N  T T N  T T T N   T N  T T N  T T T N     ...
0x1010: B2:     T      T        N     T      T        N    ...
0x1020: B3:                       T                    T   ...
```

For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal predictor on `B1`?

The accuracy is $\frac{6}{9}$.

☑ What is the accuracy of the local predictor on branch `B1`?

The branch `B1` pattern has a length of 9, which easily fits in the 16-outcome local history, and so the accuracy is 100%.

☑ What is the warmup time of the local predictor on branch `B1`?

The branch must first be seen 16 times to warm up the local history, then it must be seen 2×9 times to warm up each of the 9 entries that it uses. The warmup time is $16 + 2 \times 9$ executions.

☑ What is the minimum local history size needed for a local predictor to predict `B1` with 100% accuracy?
Five outcomes.

If it were four outcomes the local history `TTNT` would occur before both a taken and not-taken outcome.

☑ What is the accuracy of a global predictor with a three-outcome global history on branch `B2` (not `B1`)?

Two global histories are possible: `TTN` and `NTN`. the `NTN` global history is always followed by a taken outcome. The `TTN` global history can be followed by both a taken and a not taken outcome. The patterns occur in the following repeating sequence `TTN.N`, `TTN.T`, `NTN.T`, `TTN.N`, `TTN.T`, `NTN.T`,.... Assuming one of the two `TTN` patterns mispredicts the accuracy would be $\frac{2}{3}$. It is also possible that both `TTN` occurrences mispredicts, in that case the accuracy would be $\frac{1}{3}$.

☑ What is the minimum global history size needed for a global predictor to predict `B2` (not `B1`) with 100% accuracy?

Five outcomes.

Problem 4, continued:

(*b*) Consider code producing the patterns below running on two systems, one using a global predictor and the other using a gshare predictor. Both systems use a 2^{16}-entry BHT and a 12-outcome global history. The hexadecimal numbers indicate the address of the branch instruction. For example, `B5: 0x1100` indicates that the instruction we call `B5` is at address `0x1100`.

```
Pattern L:  T T T ... N    (A hundred iteration loop.)

B5: 0x1100:  L       L       L       L
B6: 0x1110:    N       N       N       N
B7: 0x1200:      L       L       L       L
B8: 0x1210:        T       T       T       T
```

☑ Why is the accuracy of the gshare predictor so much better than the global predictor on this example?

The difference is on branches B6 and B8. Branch B6 is highly biased not taken and B8 is highly biased taken, a bimodal predictor could easily predict each of these branches accurately since it uses the branch address to find a two-bit counter. In contrast the global predictor uses the global history for an address in the PHT to retrieve a two-bit counter. The global history for both branches will be TTTTTTTTTTTN, because branches B5 and B7 have the same long pattern. Therefore branches B6 and B8 will share an entry and so be predicted inaccurately. The gshare predictor uses the global history exclusive-ored with the branch address as an address in the PHT. Because the PC is used branches B6 and B8 will use different entries and so be predicted accurately.

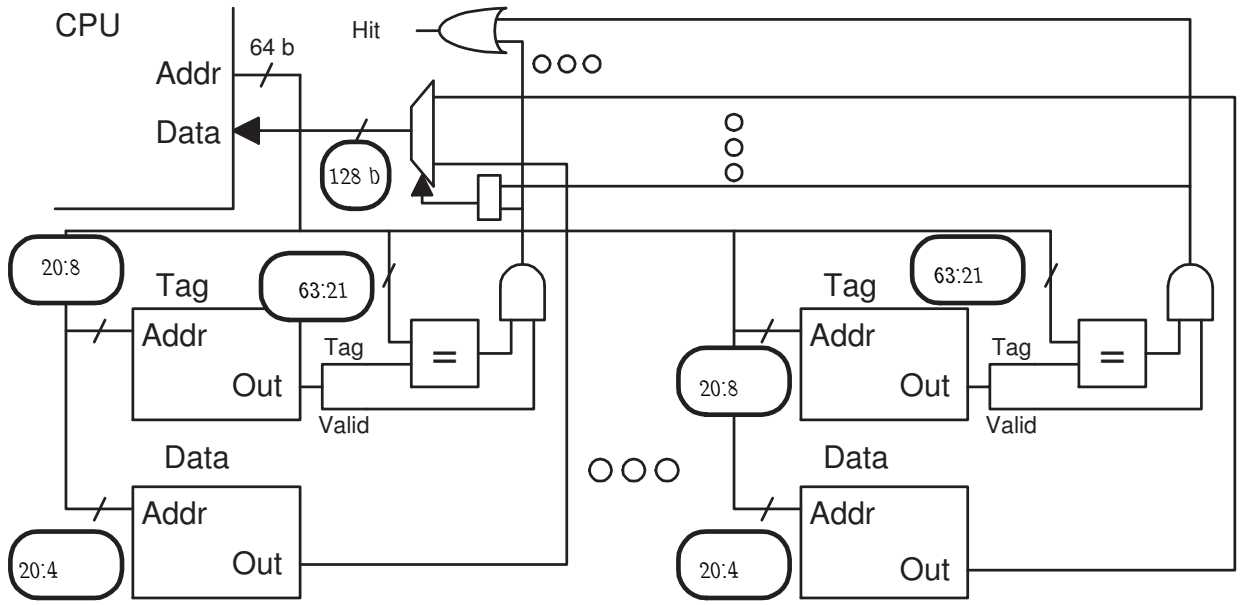
☑ What is the minimum number of BHT entries for which the gshare predictor will outperform global on this code? *Hint: Look at the branch addresses.*

The minimum number is 2^7 entries. If there are any fewer than address B7 and B6 will map to the same entry because the lower $6 + 2 = 8$ bits of their addresses are identical. (The $+2$ is needed because the two least significant bits, which are always zero, are not used in computing the PHT address.)

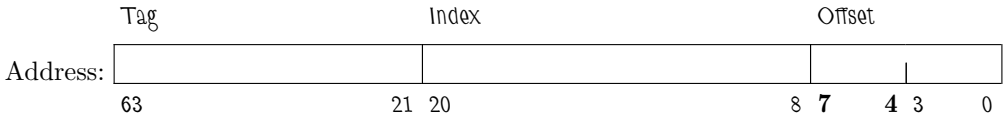
Problem 5: (15 pts) The diagram below is for a **4-way** set-associative cache with a capacity of 8 MiB (2^{23} bytes). The system has the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



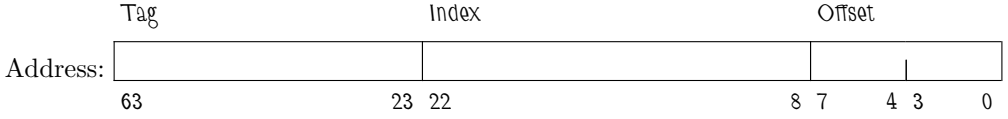
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{23} characters, plus $4 \times 2^{21-8}\,(64 - 21 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^8 = 256$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 5, continued: The problems on this page are **not** based on the cache from the previous page. The code fragments below run on a 32 MiB (2^{25} byte) direct-mapped cache with a 64-byte line size. Each code fragment starts with the cache empty; consider only accesses to the array, `a`.

(*b*) Find the hit ratio executing the code below.

☑ What is the hit ratio running the code below? Explain

```
float sum = 0.0;
float *a = 0x2000000; // sizeof(float) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 2^6 characters is given, the size of an array element is $4 = 2^2$ characters, and so there are $2^{6-2} = 2^4$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^4 elements, the next $2^4 - 1$ accesses will be to data on the line, but sixteenth access after the miss will miss again. Therefore the hit ratio is $\frac{15}{16}$.

(*c*) Find the smallest value of `STRIDE` for which the cache hit ratio in the program below will be zero.

☑ Fill in smallest value for `STRIDE` for which hit ratio 0.

☑ Briefly explain.

The value of `STRIDE` should be chosen so that the index of `a[i]` and `a[i+STRIDE]` are the same (but of course their tags will be different). We want the difference in addresses to be 2^{25}. Since an array element is 4 bytes that means that `STRIDE` must be $\frac{2^{25}}{4} = 2^{23}$.

```
float sum = 0.0;
float *a = 0x2000000; // sizeof(float) == 4
int i;
int ILIMIT = 1 << 11;    // = 2^11


int STRIDE =      1 << 23;   // <-- FILL IN.   SOLUTION FILLED IN.


for (i=0; i<ILIMIT; i++) sum += a[ i ] + a[ i + STRIDE ];
```

Problem 6: (15 pts) Answer each question below.

(*a*) What is the difference between instruction-level parallelism (ILP) and explicit parallelism?

☑ ILP and explicit parallelism difference.

Short answer, sufficient to get full credit: ILP is a property of machine language code that is written for a serial system, it is a measure of how many instructions can execute at the same time. Explicit parallelism is something specified by a programmer or compiler.

More detailed explanation: Explicit parallelism is something specified by the programmer, perhaps using a special language or more often an API such as pthreads, OpenMP, or MPI to specify how parts of the program should execute in parallel. Explicit parallelism might also be specified by a parallelizing compiler. In contrast, instruction level parallelism is something that is present in serial (non-parallel) programs at the machine-language level. It is a measure of how many instructions can execute at the same time. Programs with lots of ILP execute well on superscalar processors.

(*b*) A company has a large customer base for its ISA Y products, the most advanced of which is a 4-way superscalar dynamically scheduled implementation. The company is considering three possible next-generation systems: Develop an 8-way superscalar dynamically implementation of ISA Y, develop a chip with 16 scalar implementations of ISA Y, or develop a VLIW ISA and implementation. For each strategy indicate how much effort it will take for customers to use the new system.

☑ Customer effort to use 8-way superscalar dynamically scheduled system. Explain.

The company's current product is a 4-way superscalar chip. Code should run on the 8-way chip without any modification and so the customer effort (in preparing the code) is zero.

☑ Customer effort to use chip with 16 scalar implementations. Explain.

Presumably the company's customers have not parallelized their code. Since the chip has sixteen separate processors customers will have to parallelize their code, otherwise the code will just run on one core and so run slowly (since its scalar). Parallelizing code is hard, and so customer effort is high.

☑ Customer effort to use VLIW implementation.

The term VLIW refers to a style of ISA. To use the new chip customers will have to re-compile their code. That will require moderate to low effort.

(*c*) Consider the three systems from the previous part. For each system indicate an advantage over the others. The advantage should specify an assumed workload, an answer might start "The advantage, those customers that run programs that are _____, is ....

☑ Advantage of 8-way superscalar dynamically scheduled system.

It can run existing code without modification. Dynamic scheduling is well suited to typical "integer" workload, with its many medium difficulty branches and irregular data access patterns.

☑ Advantage of 16 scalar system chip.

Because the processors are simple, many can fit on a chip. Workload that can be efficiently parallelized will run fastest on this system.

☑ Advantage of VLIW implementation.

The intended advantage of VLIW is simpler implementations compared to superscalar techniques. So far this has only achieved success in DSP (digital signal processing) chips. So lets assume the workload consists of DSP programs.

Problem 7: (18 pts) Answer each question below.

(*a*) Why might a 1% change in branch prediction accuracy have a larger impact on the performance of a 4-way superscalar processor than on a 2-way superscalar system?

☑ Larger impact on 4-way over 2-way because...

Because the 4-way system will fetch twice as many instructions per cycle. Since the time to resolve the branch will likely be the same on the two systems, the 4-way system will waste more of its execution potential. For example, suppose that the 4-way system takes 10 seconds to execute a program and the 2-way system takes 15 seconds. Suppose that the time to resolve mispredicted branches on both systems (in total) is 3 seconds. That 3 seconds is a bigger percentage of 10 than 15, so the 1% improvement will have a bigger impact on the 4-way system.

(*b*) Dynamically scheduled systems use a technique called register renaming in which an instruction's architected registers are renamed into physical ones. Provide a brief example illustrating why register renaming is necessary.

☑ Example illustrating need for register renaming and explanation.

```
# Example for solution.
mul.s f0, f1, f2   IF ID Q  RR M1 M2 M3 M4 M5 M6 WB C
add.s f3, f0, f5      IF ID Q                 RR A1 A2 ..
lwc1  f0, 0(r1)          IF ID Q  EA ME WB
sub.s f6, f0, f7            IF ID Q  RR A1 A2
```

In the example above the lw writes back before the mul. Without register renaming the value in the `f0` register would be wrong after the `mul.s` writes back. Any later instruction would get the `mul.s` result, not the `lwc1` result which is correct.

(*c*) The SPECcpu rules require that the compilers used to prepare a run of the suite be real products that the company (or some other company) makes an honest effort to sell (or give away). Explain how the SPECcpu scores might be inflated if the compilers could be products in a technical sense, but not something the company is trying to put in customer hands.

☑ Scores inflated by unmarketed compilers because...

The compilers might include buggy optimizations which work for the SPECcpu benchmarks, because those specific problems have been fixed, but would still be buggy for other code. Actual users would be reluctant to use a buggy compiler, even if avoiding the bugs would yield a performance benefit.

☑ If the compilers produce faster programs, why not promote them?

Based on the answer above, because that would alienate customers.

(*d*) What is the difference between a trap and an exception?

☑ Difference between trap and exception.

A trap is an instruction which is intended to start a handler. It's usually used to implement the interface to an operating system, among other purposes. The programmer might insert a trap instruction to request that the operating system, say, open a file. An exception is the response of the system to something going wrong with an instruction, for example, a bad opcode or memory address. As with a trap the handler is called, but unlike a trap an exception can happen to most any instruction.

57 Spring 2010 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 26 March 2010, 10:40–11:30 CDT

Problem 1		(40 pts)
Problem 2		(12 pts)
Problem 3		(14 pts)
Problem 4		(10 pts)
Problem 5		(24 pts)
Exam Total		(100 pts)

Alias Buffer Size + 1

Good Luck!

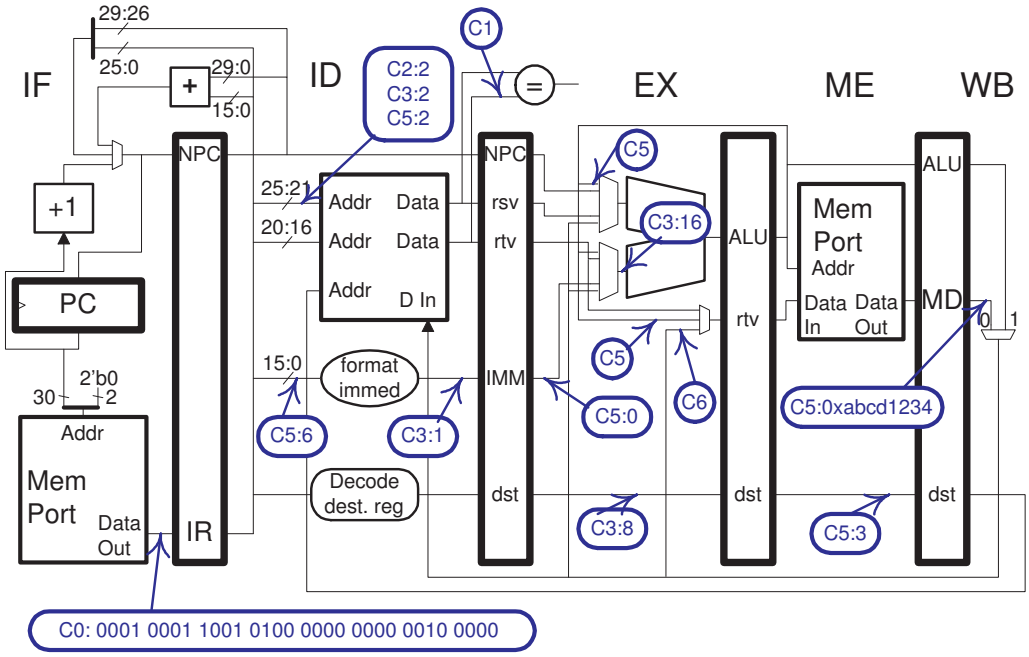
Problem 1: [40 pts] In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c5:6 indicates that at cycle 5 the wire will hold a 6. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Write a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ The first instruction address has been provided, **show the addresses of the remaining four instructions**.

☑ The third instruction is an `addi`, don't forget to show its registers and immediates.

☑ If an instruction is a load or store, show all possible size and sign possibilities. For example, (lw,lh)




```
# SOLUTION
#
# Cycle                 0  1  2  3  4  5  6  7  8
0x1000 beq r12, r20 TA  IF ID EX ME WB
0x1004 lw r8, 16(r2)       IF ID EX ME WB
TA:0x1084 addi r3, r2, 1      IF ID EX ME WB
0x1088 sb r3, 0(r3)              IF ID EX ME WB
0x108c sh (or sb) r3, 6(r2)         IF ID EX ME WB
# Cycle                 0  1  2  3  4  5  6  7  8
```

Easy Stuff: The destination for second instruction, `r8`, and the third instruction, `r3`, can be read off the `ID/EX.dst` and `EX/ME.dst` pipeline latches. Similarly, the first source register of several instructions can be read directly at the input to the register file in `ID`. Three immediate values are directly provided on both sides of the `ID/EX.IMM` pipeline latch. If this doesn't seem obvious within five minutes, please please please see the Statically Scheduled System Study Guide for tips and other sample problems, also consider asking for help. Don't bother reading the next paragraphs until everything above is easy.

Dependencies: The use of bypass paths shown by the two C5 bubbles and C6 bubble in **EX** reveals three dependencies. For example, the upper C5 means that the **rs** source of the fourth instruction is the same as the destination of the third (otherwise the bypass path would not be used). In this particular problem the result of the third instruction is bypassed to three places, two in the fourth and one in the fifth instruction. The destination register of the third instruction, **r3**, was directly determined; with the dependencies we just found we see that the three source registers must all be **r3**.

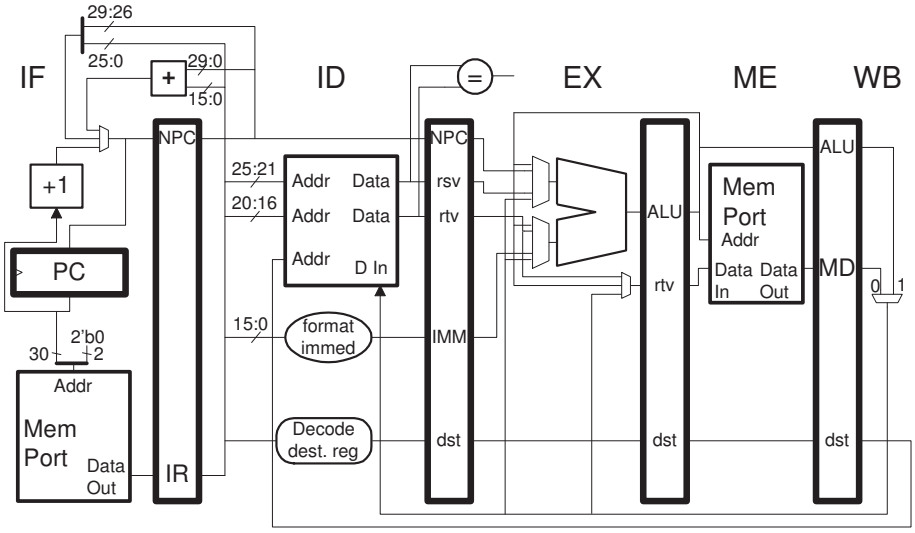
Rough Instruction Identities: Only store instructions can use the **EX/ME.rtv** latch, so we know that the fourth and fifth instructions must be stores. Only load instructions use the **ME/WB.MD** latch, and so the second instruction must be a load. Only branches use the = in the **ID** stage, and so the first instruction must be a branch.

The lower ALU input: At this point we know that the instruction in **EX** in cycle 3 is some kind of load, and so the C3:16 must be its immediate value.

Load and Store Data Sizes: Since the Mem Port sign-extends or pads the loaded value, the C5:0xabcd1234 could only be from a **lw**. (If it were a **lh**, which loads 16 bits, the upper 16 bits, instead of being **0xabcd**, would have to be either **0** or **0xffff**.) MIPS' load and store addresses must be aligned, meaning, they must be a multiple of the data size (in characters [popularly called bytes]). One does not need to be an expert on number theory to figure out that in that case the contents of register **r2** must be a multiple of 4 (given that **r2** + 16 is a multiple of 4). We should also be able to figure out that **r3**, after the **addi**, cannot be a multiple of 4 or of a 2. Therefore the fourth instruction can only a **sb**. The fifth instruction can be a **sb** or a **sh**.

That long thing in **IF***:* The bubble in **IF** provides the entire first instruction. Test takers were not expected to memorize opcodes, but were expected to know the instruction formats. Since we already know it's a branch we should know it uses a type I format. Parsing that we find that **rs** is 12, **rt** is 20, and the immediate is **0x20**. The immediate is used to find the branch target: $1000_{16} + 4 + 4 \times 20_{16} = 1084_{16}$. The solution is written as though the branch was taken, but there is not enough information to tell whether it was taken or not and so a solution with the third instruction at the fall-through address, **0x1008**, would also be correct.

Problem 2: [12 pts] Consider the following cost-reducing design options for the MIPS implementation shown below. Performance is still important, and a compiler will be able to optimize for this lower-cost implementation, but even with skillful optimization there may be some performance loss.



(*a*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all lower-ALU-input bypass paths. Which would you choose? *Note: The following sentence did not appear on the original exam.* (Only bypass paths are eliminated, other mux inputs remain.)

☑ Eliminate: upper or lower. (Circle one.)

☑ Briefly explain why.

Eliminate bypasses in lower mux. If upper mux bypasses were eliminated then we could not bypass any type I instructions. With the lower-mux inputs eliminated the compiler could accommodate at least some type R instructions that would have needed the lower mux by switching the `rs` and `rt` operands. (That would only work for commutative operations.)

(*b*) Suppose one had to choose between eliminating all upper-ALU-input bypass paths or all rtv (not to the ALU) bypass paths. Which would you choose (Note: Only bypass paths are eliminated, other mux inputs remain.)

☑ Eliminate: upper or rtv. (Circle one.)

☑ Briefly explain why.

This is an easy choice: Eliminate rtv bypasses. Stores are less frequent than type R instructions. *Grading Note: The opposite argument would get partial credit.*

(*c*) Suppose that the mux in the WB stage was moved to the ME stage.

☑ How would that reduce cost?

There would be one fewer ME/WB pipeline latch.

☑ How might that affect performance?

It would likely hurt performance because the memory port is most likely on the critical path. The clock frequency would have to be slowed to give time for the signal to propagate through the mux.

Problem 3: [14 pts] The branch delay slot ISA feature eliminates the need for the stall or squash following a branch that occurs in implementations such as our 5-stage pipeline.

(*a*) The two MIPS code fragments below do not exactly make a strong case for delay slots. In both cases there is no squash or stall, which sounds good. For each code fragment indicate whether performance is slower, equal to, or faster than corresponding non-delay-slot ISA code on a corresponding implementation (one that will suffer a squash after every branch). If faster indicate if the improvement is full (based on the eliminated squash) or partial. *Hint: it won't be full.*

```
# -------------------- Code Fragment 1 --------------------
beq r1, r2, TARG
nop
lw r3, 0(r1)
```

☑ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☑ Explain.

```
# SOLUTION
# Execution with branch not taken.
# Delay Slot ISA  0  1  2  3  4  5  6      # Non Delay Slot ISA  0  1  2  3  4  5
beq r1, r2, TARG  IF ID EX ME WB           beq r1, r2, TARG      IF ID EX ME WB
nop                  IF ID EX ME WB        lw r3, 0(r1)             IF ID EX ME WB
lw r3, 0(r1)            IF ID EX ME WB

# Execution with branch taken.
# Delay Slot ISA  0  1  2  3  4  5  6      # Non Delay Slot ISA  0  1  2  3  4  5
beq r1, r2, TARG  IF ID EX ME WB           beq r1, r2, TARG      IF ID EX ME WB
nop                  IF ID EX ME WB        lw r3, 0(r1)             IFx
lw r3, 0(r1)
...
TARG: xor r4, r3, r5    IF ID EX ME WB     TARG: xor r4, r3, r5        IF ID EX ME WB
```

Slower performance.

Brief explanation that would get full credit: In the delay-slot ISA code the **nop** guarantees that nothing is useful is done after the branch, but in the non-delay-slot code the **lw** would do something useful if the branch were not taken.

Longer explanation: Two executions of the given code fragment are shown above on the left-hand side, in the first the branch is not taken, in the second it is taken (with an added target instruction). On the right-hand side a version of the code is shown for an ISA without delay slots; it too is shown executing both with the branch not taken, and taken.

In the not-taken case the **lw** is the first instruction after the branch doing something useful. That is executed one cycle earlier in the non-delay-slot code. In the taken case the **xor** instruction (added for this explanation) does useful work. That is executed at the same time (cycle 4) for both ISAs.

Therefore the delay-slot code is slower (with the reasonable assumption that the branch is sometimes not taken).

```
# -------------------- Code Fragment 2 --------------------
beq r1, r2 SKIP
add r9, r4, r5
or r6, r9, r8
SKIP:
sub r3, r6, r5
lw r9, 0(r3)
```

☑ Delay slot in code above results in: slower, equal, partial improvement, full improvement. (CIRCLE ONE).

☑ Explain.

```
# SOLUTION
# Execution with branch not taken.
#                 Delay Slot ISA            Non Delay Slot ISA
# Cycle           0  1  2  3  4  5  6       0  1  2  3  4  5  6
beq r1, r2 SKIP   IF ID EX ME WB            IF ID EX ME WB
add r9, r4, r5       IF ID EX ME WB            IF ID EX ME WB
or r6, r9, r8           IF ID EX ME WB            IF ID EX ME WB
SKIP:
sub r3, r6, r5
lw r9, 0(r3)

# Execution with branch taken.
#                 Delay Slot ISA            Non Delay Slot ISA
# Cycle           0  1  2  3  4  5  6       0  1  2  3  4
beq r1, r2 SKIP   IF ID EX ME WB            IF ID EX ME WB
add r9, r4, r5       IF ID EX ME WB            IFx
or r6, r9, r8
SKIP:
sub r3, r6, r5          IF ID EX ME WB            IF ID EX ME WB
lw r9, 0(r3)               IF ID EX ME WB            IF ID EX ME WB
```

Equal performance.

The `add` instruction in the delay slot is only useful if the branch is not taken. So when the branch is taken the delay-slot version of the code executes the `add` instruction but ignores the result (`r9` is overwritten by the `lw` before being read), while the non-delay-slot version fetches but then squashes the `add` instruction. Either way the first useful instruction after the branch is executed at the same time in both ISAs (cycle 3 if the branch is not taken, and cycle 4 if the branch is taken).

(*b*) How can Code Fragment 2 (above) be improved by profiling? Show the optimized fragment and how profiling led to the optimized fragment.

☑ Optimized version of Code Fragment 2

```
# SOLUTION
# Code identical for both ISAs.
# Execution shown for delay-slot ISA.
# Cycle            0  1  2  3  4  5  6  7  8  9
bne r1, r2  LINEX  IF ID EX ME WB
SKIP:
sub r3, r6, r5        IF ID EX ME WB
                             IF ID EX ME WB    (Second execution of sub).
lw r9, 0(r3)                    IF ID EX ME WB
...

LINEX:
J SKIP:                  IF ID EX ME WB
or r6, r9, r8               IF ID EX ME WB
```

☑ Specifically, how did profiling help?

Brief answer that would get full credit: Profiling told us which branch outcome is more likely, and so we could re-arrange the code so that the more-likely outcome is not taken, resulting in fewer wasted or squashed instructions.

Longer answer: The code above works best on both systems if the branch is not taken. Assume that with profiling we find that the branch is mostly taken. In the code above the **beq** was changed to a **bne** and the following code was re-arranged, so now the branch is mostly not taken. In the original code the **add** would be executed unnecessarily (or squashed) when the branch is taken. In the profile-optimized code above the **sub** is executed one extra time if the branch is taken.

Problem 4: [10 pts] The SPECcpu2006 integer suite has 12 benchmarks. Suppose instead it included only four benchmarks, but those benchmarks were chosen fairly, given that only four could be chosen.

(*a*) Considering just the base score (or ignoring the base/peak distinction altogether), what is the disadvantage of having just four benchmarks?

☑ Disadvantage of having four benchmarks.

Four is not enough to represent the broad range of characteristics found CPU and memory bound applications.

(*b*) Suppose that with this smaller set of benchmarks the difference between the base and peak scores was smaller than if 12 benchmarks were used. Assume that in both cases the testers were skilled and followed the rules. Also assume that the base score values were about the same with 4 or 12 benchmarks.

☑ Give a reason for the smaller difference that has to do with the different rules for base and peak tests.

In base tuning the same optimization switches must be used to prepare all benchmarks sharing a language. In peak, each benchmark can have its own set of optimization switches. Base rules are designed so that testers do not perform time-consuming per-benchmark tuning.

There are some optimizations that are good for a few benchmarks but bad for some others. Suppose such a switch helps 1% of the benchmarks and hurts 20%, and has no effect on the remaining 79%. With twelve benchmarks testers are more likely to leave out such switches because they might help just one benchmark but hurt several others. But with only four benchmarks there is a lower chance that a switch that helps one benchmark would hurt the other three. Therefore testers would spend lots of time tuning benchmarks under base tuning, which is not what the base tuning is supposed to measure.

Grading Note: Nobody got this right, nor did anyone mention the common-switch rule for base that was the key to answering this question.

☑ Does the smaller difference indicate that base and peak are not measuring what they should?

Yes, they are not measuring what they should because base is supposed to indicate normal compiling effort, but with so few benchmarks testers could still get away with per-benchmark tuning (see previous answer).

Problem 5: Answer the questions below.

(*a*) [14 pts] Write the following MIPS code fragments with the minimum number of instructions. If you don't know the exact mnemonic, such as `mtc1` or `cvt.w.f`, make up something that sounds right.

☑ Put constant `0x12345678` into register `r1`.

```
# Solution
lui r1, 0x1234
ori r1, r1, 0x5678
```

☑ Jump to address `0x72345678` from address `0x1000`.

```
# Solution
lui r1, 0x7234
ori r1, r1, 0x5678
jr r1
```

☑ Jump to address `0x72345678` from address `0x1000` using a SPARC-like `jmpl` (but in MIPS, like Homework 2).

```
# Solution
lui r1, 0x7234
jmpli r0, r1, 0x5678
```

☑ Registers `r1` and `r2` contain integers. Write register `f4` with the sum as a single-precision floating point number.

```
# Solution
add r3, r1, r2
mtc1 f2, r3
cvt.w.s f4, r2
```

(*b*) [10 pts] Typical CISC ISAs have a range of immediate sizes for use in ALU instructions, up to the data size limit. (Say, 8, 16, 24, and 32 bits). MIPS, and other RISC ISAs, have just one immediate size for ALU instructions.

☑ Why can't MIPS have a 32-bit immediate?

Because the entire instruction is 32 bits, there would be no place to put it.

☑ Why can a CISC ISA have a 32-bit immediate?

CISC ISAs have variable-size instructions, and so the instruction size can be adjusted to accommodate whatever immediate size is needed.

☑ Why would there be no benefit for MIPS to have both 8- and 16-bit immediate sizes for arithmetic instructions?

☑ What is the benefit for CISC ISAs in having 8-, 16-bit, (and more) immediate sizes for arithmetic instructions?

The instructions would be smaller, and then so would the programs. On modern general purpose systems the benefit would be a smaller instruction cache (not yet covered).

Name Solution

Computer Architecture

EE 4720

Final Examination

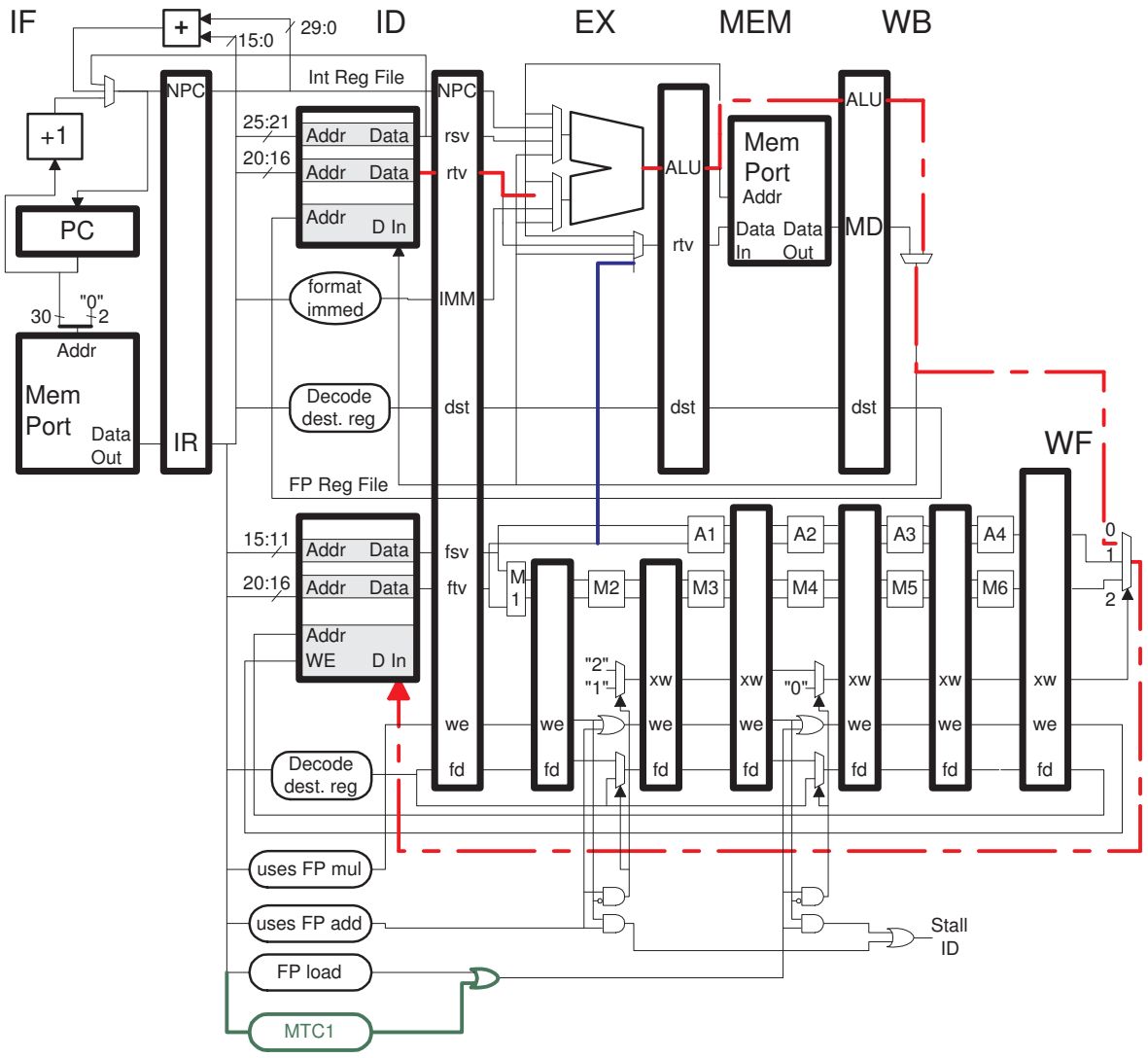
11 May 2010, 12:30–14:30 CDT

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(10 pts)
Problem 4		(15 pts)
Problem 5		(15 pts)
Problem 6		(10 pts)
Problem 7		(15 pts)
Exam Total		(100 pts)

Alias A(H1N1) · · · chooo!

Good Luck!

Problem 1: (15 pts) The statically scheduled MIPS implementation including the floating-point pipeline is illustrated below.



(*a*) Consider the instruction `mtc1 f2, r4`. On the diagram above show the path taken by the data on its trip from `r4` to `f2`.

☑ Show path taken by value using a squiggly line on the diagram above.

The path is highlighted in red with dashes instead of squiggles. Note that `mtc1` uses the `rt` field for the integer register. The ALU will set its output to the value at its lower input.

(*b*) The control logic for the FP pipeline needs only a small change to handle `mtc1`. Make that change above. (This has nothing to do with the bypass problem below.)

☑ Control logic for `mtc1` in diagram above. (Ignore bypasses.)

Change appears in green. Both the `lwc1` and `mtc1` instructions take a value from the integer pipeline and write it to the FP register file in the integer WB stage. So for detecting `WF` structural hazards the same logic can be used with the addition of a box for detecting the `mtc1` opcode.

(*c*) Add the hardware needed to implement `swc1`. Add only datapath, not control logic.

☑ Datapath for `swc1`.

The added hardware will provide a way for a value read from the FP register file to hop over to the "Data In" connection on the ME-stage memory port. The EX stage is the critical-path friendly place to do that, the change appears in blue.

Problem 1, continued:

```
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13
lui r1, 0x4593      IF ID EX ME WB
ori r1, r1, 0x819c     IF ID EX ME WB
mtc1 f1, r1               IF ID EX ME WF
add.s f2, f2, f1             IF ID A1 A2 A3 A4 WF
mtc1 f4, r4                     IF ID EX ME WF
sub.s f6, f4, f1                   IF ID A1 A2 A3 A4 WF
swc1 f6, 0(r5)                        IF ID -------> EX ME WF
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

(*d*) The code fragment execution (pipeline diagram) above could not occur on the pipeline above because certain bypass paths are needed.

☑ Add those bypass paths for the code above.

The bypass paths appear in blue in the diagram below.

☑ Show the cycle in which each added bypass path is used by the code above.

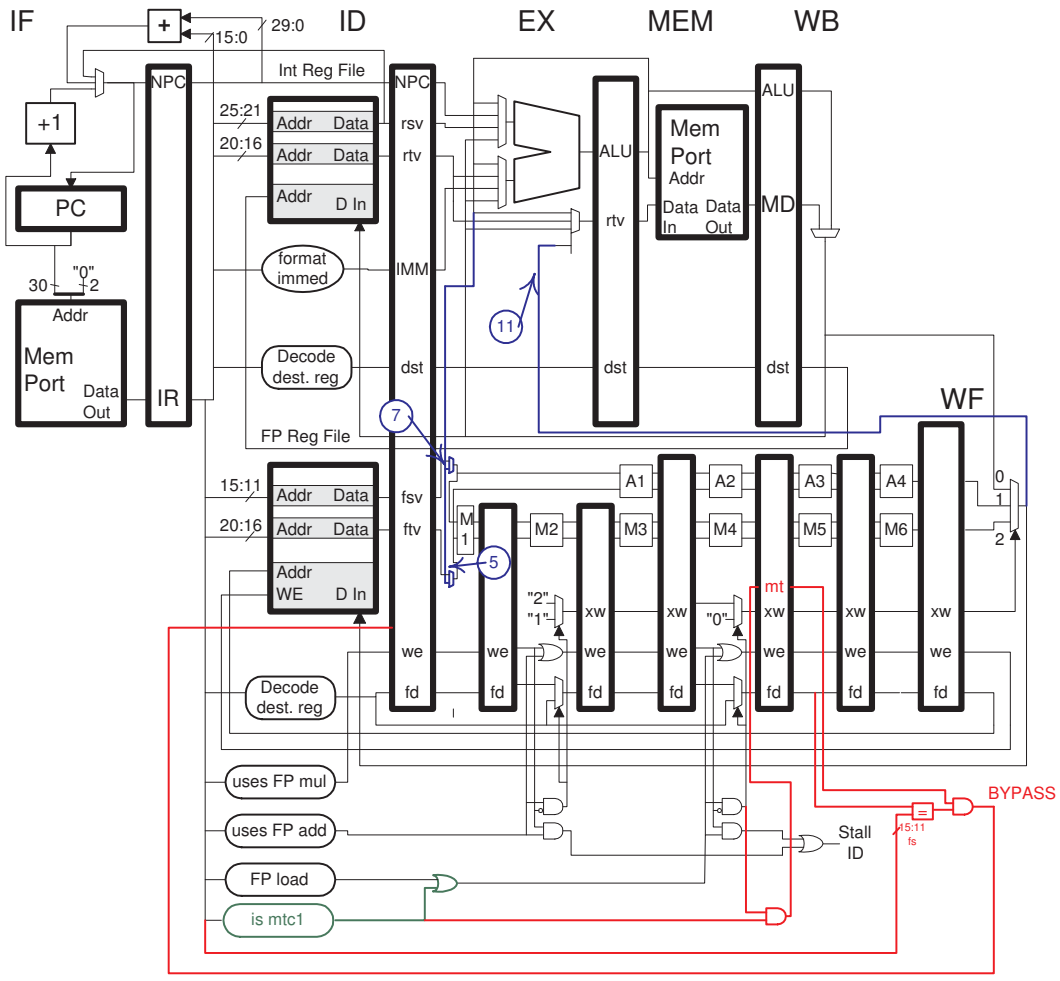
See the blue circles in the diagram below.

(*e*) Add control logic needed to detect the bypass used from `mtc1` to `sub.s`. The logic should deliver a signal, `BYPASS`, to the stage containing the bypass multiplexors. The `BYPASS` signal should be true if the bypass is needed.

☑ Logic generating `mtc1` to `sub.s` bypass signal.

Solution appears in red below; the green changes are from part (b). The dependence is detected by the [=] box, comparing the `fs` source of the instruction in `ID` (register `f4` of the `sub.s`, compared in cycle `6`) to the destination of whatever instruction is two stages from `WF`, that would be the `mtc1` in the example (also during cycle `6`). The comparison unit output is connected to an AND gate which makes sure that there is an `mtc1` two stages from `WF`. (Just checking `we` would tell us that something is two stages from `WF`, but for our purposes we need to know whether it's an `mtc1` because that is the only instruction that can use the bypass path.) The problem just asked for a signal named BYPASS, not that it be connected to anything, but the solution goes a bit further showing that BYPASS is connected to the `ID/EX` latch. Connecting BYPASS directly to the added `fsv` multiplexor **would be wrong**.

Diagram on next page.


IF
ID
EX
MEM
WB
WF
+
+1
PC
NPC
IR
Mem Port
Addr
Data Out
30
"0"
2
29:0
15:0
Int Reg File
25:21
20:16
Addr Data
Addr D In
rsv
rtv
format immed
IMM
Decode dest. reg
dst
FP Reg File
15:11
Addr WE D In
fsv
ftv
we
fd
ALU
Mem Port
Addr
Data In
Data Out
MD
11
7
5
M1
M2
M3
M4
M5
M6
A1
A2
A3
A4
xw
"2"
"1"
"0"
mt
0
1
2
uses FP mul
uses FP add
FP load
is mtc1
Stall ID
BYPASS
15:11
fs

Problem 2: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system uses a bimodal predictor, one system uses a local predictor with a 16-outcome local history, and one system uses a global predictor with a 16-outcome global history.

Branch B1 is random, and can be described by a Bernoulli random variable with $p = .5$. The outcome of branch B3 will always be the same as the most recent outcome of branch B1. (That is, if an execution of B1 is taken, the next execution of B3 will be taken.) Branch B2 has a repeating pattern, it repeats twice below.

```
B1: r   r       r           r               r   r       r           r
B2:  T   N   T   N   N   T   N   N   N   T    T   N   T   N   N   T   N   N   N   T
B3:   R       R           R               R    R       R           R               R
```

For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal predictor on B2?

The 2-bit counter values are shown below. Note that the same counter value is present at the beginning and end, so the first occurrence of the pattern can be used to compute the prediction ratio. That would be an unimpressive $\boxed{\frac{4}{10} = .4}$ (40%).

```
# SOLUTION Work
    1  2   1   2   1   0   1   0   0   0  1    <-- Counter values
B2:  T   N   T   N   N   T   N   N   N   T    T   N   T   N   N   T   N   N   N   T
     x   x   x   x       x               x     <-- Prediction outcome.
```

☑ What is the accuracy of the bimodal predictor on B3?

The branch is random and uncorrelated with itself so there is no way prediction accuracy can be anything other than $\boxed{.5}$.

☑ What is the accuracy of the local predictor on B2?

The pattern length is ten outcomes, which can easily fit within the 16-outcome local history, and so the accuracy will be $\boxed{1}$.

☑ How small can the local history size be made without affecting the accuracy of branch B2? Explain.

The local history can be as small as $\boxed{\text{5 outcomes}}$. A local history size of four would be too small, in that case local history pattern NTNN could be followed by both a T or N outcome, precluding 100% accuracy. The table below shows all of the five-outcome patterns that are possible (sorted and original order), followed by the outcome. We know that five are sufficient because each distinct history is followed by only one possible outcome.

```
---Sorted-----        -- Original Order--
Hist   Outcome        Hist   Outcome
NNNTT  N              TNTNN  T
NNTNN  N              NTNNT  N
NNTTN  T              TNNTN  N
NTNNN  T              NNTNN  N
NTNNT  N              NTNNN  T
NTTNT  N              TNNNT  T
TNNNT  T              NNNTT  N
TNNTN  N              NNTTN  T
TNTNN  T              NTTNT  N
TNTNN  T              TTNTN  N
TTNTN  N              TNTNN  T
```

☑ What is the accuracy of the local predictor on B3?

The local predictor can only "see" B3 and so is just as helpless as the bimodal predictor, the accuracy will be .5.

☑ What is the accuracy of the global predictor on B3?

Branch B3 is perfectly correlated with B1, and the global predictor can "see" B1, thanks to the generous 16-outcome global history register. So after warmup accuracy will be 100%.

☑ How small can the global history size be made without affecting the accuracy of branch B3? Explain.
Five outcomes.

Five is the size needed to hold the `r N N N T` pattern, which has the longest separation between B1 and B2.

☑ How many PHT entries are used by B2 in the system using the local predictor?

Not counting warmup, ten entries. During warmup up to 15 additional patterns would be used for the first 15 times branch B2 was encountered. What those 15 patterns are is determined by the contents of the local history field in the BHT entry for B2 before B2 was first executed. That local history field might contain the outcomes for some aliased branch, or else whatever values were set when the power was turned on (random, or perhaps intentionally initialized to 0).

☑ What is the warmup time of the global predictor on branch B3?

There are four patterns of GHR value that will be present when predicting B3, for example, `TRrNTRrNNTRrNNNT`. The `r` and `R` in this pattern can be `T` or `N`. If B1 and B3 were not correlated then the six arrs could have $2^6 = 64$ possible values. But they are and so the big R will always have the same outcome of the little r to its left. There are two such pairs in the pattern, and two unpaired ars to the total number of instances of this pattern is $2^4 = 16$. Now, assuming that there are 16 instances of each of the remaining three patterns there would be a total of 64 possible GHR values and so 64 PHT entries would be used, each of which requires two updates to warm up in the worst case. The warmup time would then be at least 128 executions of B3.

Problem 3: (10 pts) Suppose that in a MIPS-I system using a bimodal predictor there were BHT collisions on 5% of the predictions. (A BHT collision occurs when two branches use the same BHT entry.) A BHT entry stores both a 2-bit counter and the branch's 16-bit displacement. *Grading Note: The contents of a BHT entry was not in the original exam.*

Consider a design alternative in which a tag were used to detect BHT collisions, in the same way a cache uses a tag to detect hits (or misses).

(*a*) How large would the tag have to be to perfectly detect misses on a MIPS-I system using a 2^{14}-entry BHT?

☑ Tag size needed, reason:

The BHT is indexed using the low bits of the branch PC, omitting alignment bits of course. That would be bits 15:2 in this case. The higher bits, 31:16 would form the tag, and so that tag size would be 16 bits.

(*b*) Suppose that the storage budget for the BHT was fixed at the number of bits in a 2^{14}-entry BHT without tags.

☑ About how many entries would there be in a BHT with tags?

Two solutions appear below. One is for a BHT entry that only stores a 2-bit counter. That was what the original question implied. The second is for a BHT that stores the branch displacement.

Solution Ignoring Displacement: The existing BHT uses $2^{14} \times 2 = 2^{15}$ bits. If the BHT held the tag, not just the two bit counter, its capacity would be $2^x \times 18 \approx= 2^{x+4}$ bits, where 2^x is the number of BHT entries. Solving $2^{x+4} = 2^{15}$ for x yields $x = 11$, so the tagged BHT would have 2048 entries, one eighth its original size. The net result would be an increase in collisions, but now at least we can detect those collisions. *Note: That last sentence needs to be read with an air of foolish confidence.*

Solution Using Displacement: An existing BHT entry requires 18 bits, 16 for the displacement and 2 for the counter. Also including the tag would nearly double the size. To keep the costs equal the number of entries in the tagged BHT would have to be halved. Therefore, The tagged BHT would have $2^{13} = 8192$ entries.

(*c*) Based on the answer to the previous part, could we use a tagged BHT if the benefit of detecting collisions were small, moderate, or large?

☑ Benefit needs to be small, medium, or large. Explain.

Solution Ignoring Displacement: With a factor of eight reduction in BHT size the benefit of detecting a collision needs to be large. Extra large, maybe.

Solution Using Displacement: Since the BHT is half the size we would expect more collisions, exactly how much more is hard to say. Therefore there would have to be at least a moderate benefit in doing so.

(*d*) What is the benefit of detecting BHT collisions on a four-way superscalar statically scheduled MIPS implementation?

☑ Benefit for 4-way MIPS:

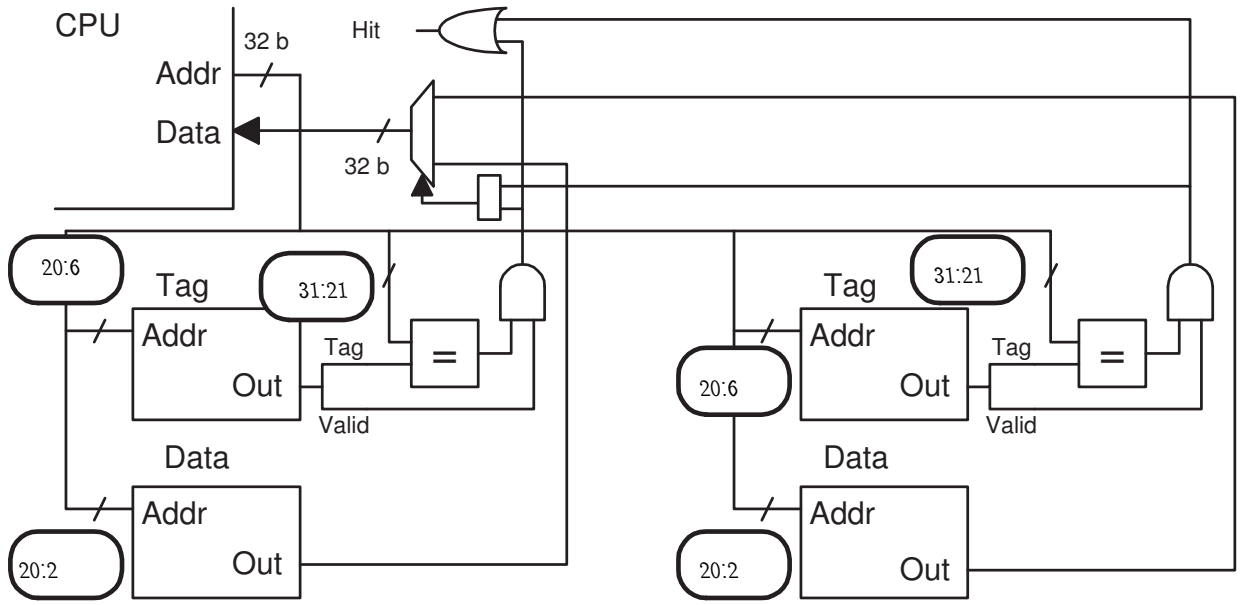
Solution Ignoring Displacement: Here we are assuming that a correct branch target is available despite the BHT collision. There is no benefit at all because knowing that there is a collision does not help in making a prediction. It's better to use an unreliable prediction that might be right than to stall the pipeline until the branch resolves (thereby guaranteeing performance loss).

Solution Using Displacement: If there is a BHT collision then the branch displacement is likely wrong. Therefore, predicting the branch taken is likely going to result in a target misprediction. Therefore, on a collision we should predict the branch not taken, which might have just a .5 chance of being correct, but that's better than predicting taken and having a nearly certain chance of getting the target wrong.

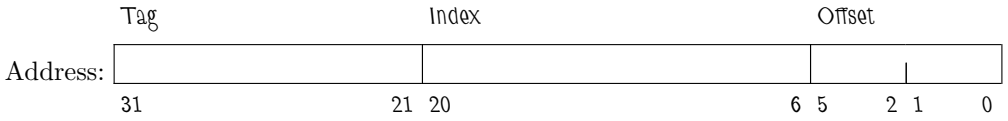
Problem 4: (15 pts) The diagram below is for a set-associative cache with a line size of 64 bytes and a tag size of 11 bits. The system has the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Cache Capacity (Indicate Unit!!):

The cache capacity can be determined from the following pieces of information: the 11-bit tag (given explicitly in the first paragraph), the 2-way associativity (by noting the diagram has 2 "ways"), and the 32-bit address space (from the number of address bits shown in the diagram near the upper-left). From the 11-bit tag and 32-bit address space we know that the low tag bit is $32-11=21$, and so the capacity of a way is 2^{21} bytes or 2 MiB. Since the cache is 2-way the **total capacity is $2^{21} \times 2 = 2^{22}$ characters or 4 MiB**.

☑ Associativity:

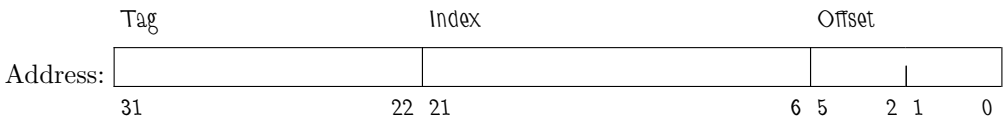
Associativity is 2

by inspection of the diagram.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{22} characters, plus $2 \times 2^{21-6}\,(32-21+1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.


Tag
Index
Offset
Address:
31
22
21
6
5
2
1
0

Problem 4, continued:

(*b*) The code below runs on a 32 MiB direct-mapped cache with a 256-byte line size. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
double *a = 0x2000000; // sizeof(double) == 8
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ 4 * i ];
```

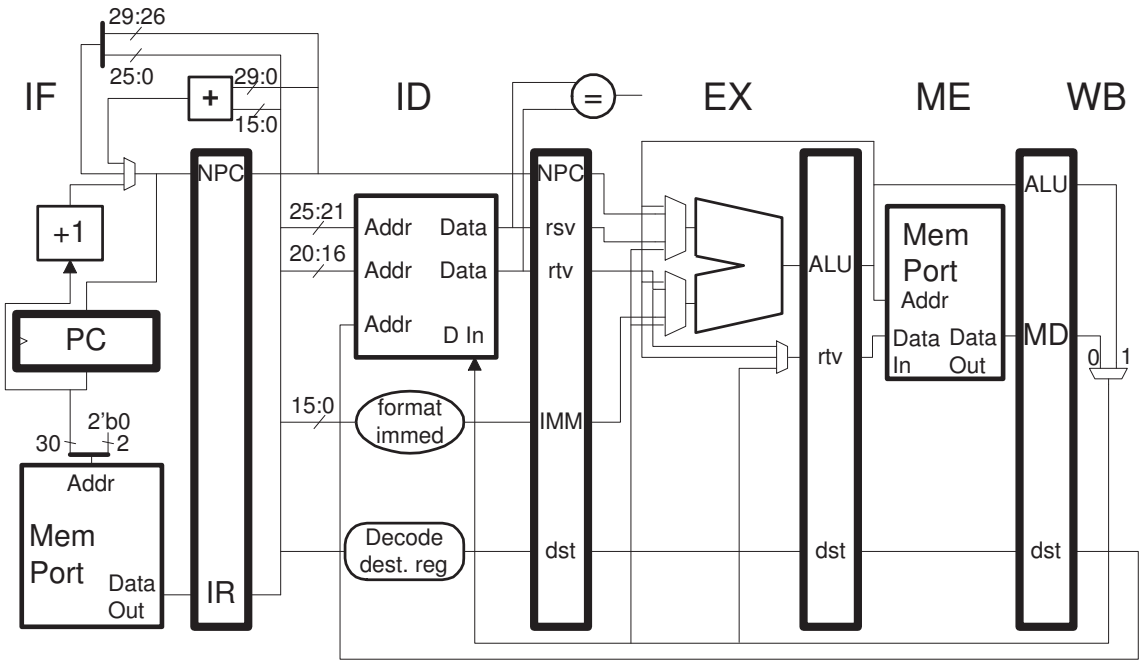
The line size of 2^8 characters is given, the size of an array element is $8 = 2^3$ characters, and so there are $2^{8-3} = 2^5$ elements per line. However, because the array index is multiplied by four only every fourth element is read, or $2^5/4 = 2^3$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^5 elements, the next $2^3 - 1$ accesses will be to data on the line, but eighth access after the miss will miss again. Therefore the hit ratio is $\frac{7}{8}$.

Problem 5: (15 pts) Several possible new MIPS instructions appear below. Show how each instruction can be encoded and show datapath changes needed to implement the instruction.

- The changes cannot break existing instructions.
- The changes can not have a large impact on clock frequency.
- A shift unit is present, but not shown.

Note: The original exam only asked for a summary of datapath changes, and did not mention the shift unit or clock frequency.

Also indicate the relative difficulty of implementing the instruction. If an instruction is deemed moderate or difficult indicate the most important reasons why.



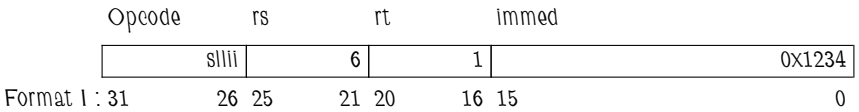
Continued on next page.

(*a*) The `sllii` instruction is like an `lui` except it can shift the immediate by any amount. For example, `sllii r1, 0x1234,16` would be equivalent to an `lui r1, 0x1234`.

```
sllii r1, 0x1234, 6   ! r1 = 0x1234 << 6
```

☑ Show possible encoding (instruction format (R, I, etc) and field (rs, rt, etc) usage):

Because it has an immediate and register it would have to be format I. The `rs` field will be used to hold the shift amount, leaving the role of the `rt` field unchanged.

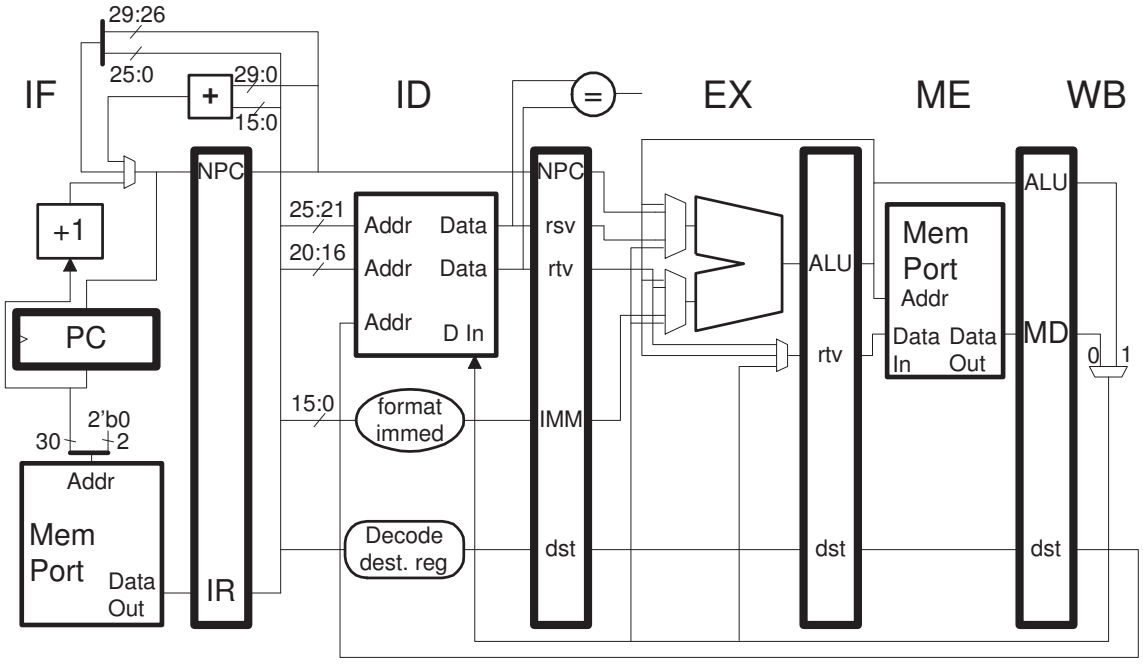


☑ Easy, moderate, or difficult to implement:

☑ Show datapath changes.

This should be easy because there already is a shift unit (for the `sll` and friends). For the new instruction the "shiftee" input to the shift unit will also have to connect to the `ID/EX.IMM` latch, and the shift amount input would have to connect to a new `ID/EX.rs` latch (not to be confused with `ID/EX.rsv`).

Problem 5, continued:

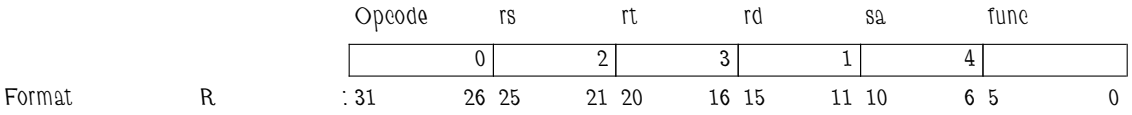


(*b*) The `adds` (add scaled) can shift the second operand left by any amount.

```
adds r1, r2, r3, 4   ! r1 = r2 + ( r3 << 4 )
```

☑ Show possible encoding:

The encoding is easy, just put the shift amount in the `sa` field. In fact, there would be no reason to choose a new `func` field value because `add` is ordinarily defined with an `sa` field value of 0.



☑ Easy, moderate, or difficult to implement:

☑ Show datapath changes.

This would be moderately difficult to implement because the shift needs to be performed before the ALU operation. Since arbitrary shifts cannot be done quickly, a stage would have to be added between `ID` and `EX`.

(*c*) The `addsid` instruction produces a sum like `add` but one operand is obtained from memory.

```
addsid r1, r2, (r3)   ! r1 = r2 + Mem[r3]
```

☑ Show possible encoding:

The encoding here is also easy. Make the memory address register `rt`.

☑ Easy, moderate, or difficult to implement:

☑ Show datapath changes.

This is difficult because the memory port is in the stage after the ALU. There are no reasonable ways to get the value read from memory back to the ALU. Here are some unreasonable ways:

Add a new stage between ID and EX, and put a memory port there. This isn't a good idea because memory ports are expensive.

Add a new stage, call it MS (memory source). Unlike the solution above, the MS stage will share a memory port with ME. This will require a multiplexor on the memory port address input, as well as control logic to detect the structural hazard. Further, it will likely strain critical path since it's the memory port that is hardest to squeeze into one cycle.

Add a new stage, EX2, after ME, the new stage will have an ALU to do any operations that could not be done in EX. Besides the cost, there will be bypass issues with this solution.

Problem 6: (10 pts) Consider a single four-way superscalar implementation and a chip with four scalar implementations (like our five-stage MIPS). Both are statically scheduled and have similar features.

☑ Why might the clock frequency of the superscalar system be lower than the scalar systems?

Bypass paths are the main culprit in several ways. First, physical distances are larger and so there will be higher propagation delays. Bypass multiplexors will also have more inputs, $2n$ inputs each for an n-way superscalar. If control logic is placed in ID then that should not have an impact on clock frequency for something as small as a 4-way system.

☑ Why might the chip area (or cost) of the four scalar systems be less than the superscalar system?

An n-way, 5-stage superscalar might have about $2n^2$ bypass paths, whereas n scalar systems would just have $2n$ paths. For that reason the scalar systems would cost less. There might be some savings with the superscalar systems, such as fewer memory ports or floating-point units.

☑ How does the average program run less efficiently on the superscalar system than on one of the scalar systems? (Less efficiently means more stalls and squashes.)

There are only a few instruction pairs that can cause the scalar system to stall, such as a load followed by an instruction that uses the loaded value. There are no stalls for consecutive dependent arithmetic instructions, and no stalls or squashes for taken branches (assuming the needed bypasses are present). In contrast, dependent instructions in the same group (fetched together) on the superscalar system will cause a stall. For that reason, execution will be less efficient.

☑ Given the answers above, why does a typical chip have two four-way superscalar implementations rather than eight scalar implementations? (Please do not confuse *eight scalar implementations* with *one eight-way superscalar system.*)

Lets consider the simpler tradeoff of one 4-way superscalar vs. four scalar processors. If a program could be perfectly parallelized (achieve linear speedup) then it would execute faster with four scalar processors because their clock frequency would be higher and because there would be fewer stalls. However, writing parallel code is often difficult, for some classes of programs very very difficult. The 4-way superscalar is faster for the programs you already have. (Beyond 4-way efficiency drops sharply, which is why chips have several 4-way cores.)

Problem 7: (15 pts) Answer each question below.

(*a*) One use for exceptions is to implement rare or specialized instructions in software (called *emulation*). One example is the SPARC quad-precision arithmetic instructions, such as faddq f4, f8, f12 (floating-point add quad). No existing SPARC implementation can execute these instructions in hardware.

☑ Why would `faddq` have to raise a precise exception in order to be emulated?

The emulation code will reproduce what the `faddq` was supposed to do, meaning it will read the source registers, compute a 128-bit floating-point sum, and write that sum to the destination registers. It will then resume execution at the instruction following `faddq`. To do that the exception handler (which calls the emulation code) must be reached as though execution jumped to the handler just before the `faddq` (the precise exception definition). If the exception were not precise then execution might have reached several instructions after `faddq`, perhaps overwriting one of its source registers, making it impossible to emulate, and anyway making it impossible to return to the instruction after `faddq` (which would then execute a second time).

(*b*) Consider an implementation similar to our pipelined MIPS in which a floating-point overflow on a `fmuld` did not raise a precise exception (because the hardware to do so would not be worth the trouble).

☑ Does that mean it would not be possible for the `faddq` to raise the precise exception needed for emulation? Explain.

No, because the `faddq` would raise an illegal instruction exception, which is detected early enough to be precise.

(*c*) Answer the following questions about the SPECcpu benchmarks.

☑ Why are branches in the FP suite easier to predict than branches in the integer suite?

Because many programs in the FP suite perform scientific calculations, which typically operate on large sets of uniform data. Loops have many iterations, making them easier to predict.

☑ Why is it important that the source code is available for the SPECcpu benchmarks?

SPECcpu is intended to measure the potential of new ISAs and new implementations. The tester compiles the source code for the particular implementation being tested, presumably in such a way that brings out the tested system's full potential. If source code were not included the suite would have to provide pre-built executables. These could not be used to test new ISAs since SPEC wouldn't have the tools to build code for them (because they are new). New implementation might benefit from new compilers, which SPEC could also not be expected to have.

(*d*) Answer each ISA question below:

☑ Describe a feature of VLIW ISAs that distinguishes them from RISC and CISC ISAs.

Instructions are handled in groups called bundles. A bundle includes several (usually 3) instructions plus a field (sometimes called a template) providing dependency and other information.

☑ Describe a feature of CISC that distinguishes it from RISC.

Instructions can vary in length. Memory access is not restricted to load and store instructions. For example, an arithmetic instruction can load operands from memory and write a result to memory.

58 Spring 2009 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

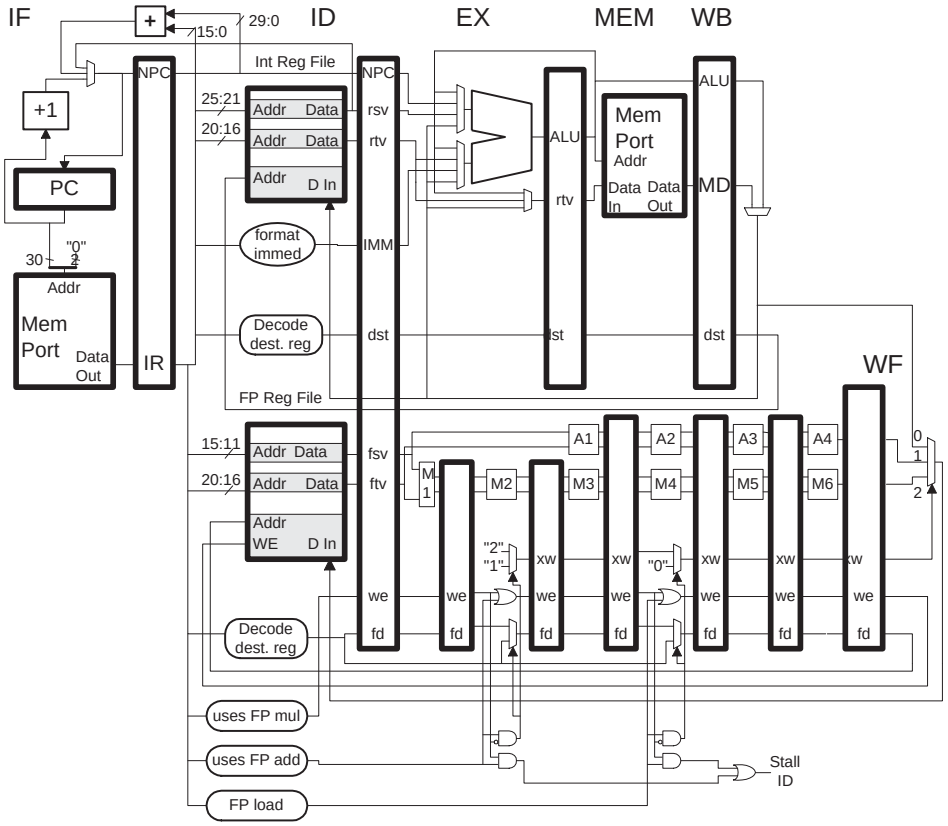
Friday, 27 March 2009, 11:40–12:30 CDT

Problem 1		(30 pts)
Problem 2		(30 pts)
Problem 3		(10 pts)
Problem 4		(20 pts)
Problem 5		(10 pts)
Exam Total		(100 pts)

Alias Blue sunset?

Good Luck!

Problem 1: [30 pts] The code below executes on the illustrated FP pipeline.




```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

(*a*) Add reasonable bypass paths needed by the code above, for both the integer and FP pipelines.

☑ Add reasonable bypass paths. Don't add unneeded bypass paths.

See discussion in the next part.

(*b*) Analyze the performance of the code using your bypasses:

☑ Show a PED for the code above using your bypasses. (Use this page or next page.)

The PED appears below. Three additional bypasses are needed for the code below. In cycle 4 `mul.d` uses two bypasses, from `WF` to `M1`, for the value loaded by `ldc1`. Also in cycle 4, `bneq` uses a bypass from `ME` to `ID`. In cycle 10 the `add.d` uses a bypass from `WF` to `A1` (this would be one of the bypasses to `mul.d`).

```
# SOLUTION
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)     IF ID EX ME WF
addi r1, r1, 8        IF ID EX ME WB
mul.d f2, f0, f0         IF ID M1 M2 M3 M4 M5 M6 WF
bneq r1, r2, LOOP           IF ID EX ME WB
add.d f6, f6, f2               IF ID ----------> A1 A2 A3 A4 WF
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)                    IF ----------> ID EX ME WF
addi r1, r1, 8                                   IF ID EX ME WB
mul.d f2, f0, f0                                    IF ID M1 M2 M3 M4 M5 M6 ..
bneq r1, r2, LOOP                                      IF ID EX ME WB
add.d f6, f6, f2                                          IF ID ----------> ..
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
ldc1 f0, 0(r1)                                               IF ----------> ..
```

☑ Compute the CPI of the code for a large number of iterations.

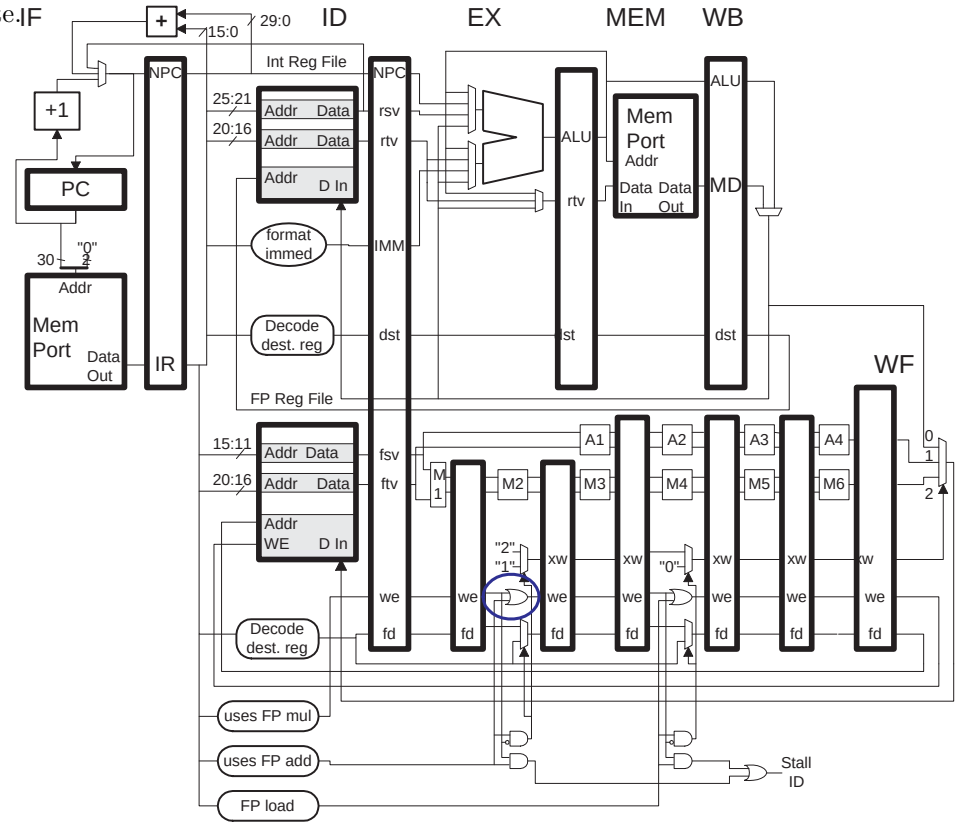
The iteration starting at cycle 5 and 14 start with the pipeline in the same state (`ldc1` in IF, `add.d` in ID, `bneq` in EX, etc.) and so the iteration starting at cycle 5 will match future ones. The iteration starting at cycle 5 takes $14 - 5 = 9$ cycles and uses five instructions, so the average instruction execution time is $\frac{14-5}{5=1.8\,\mathrm{CPI}}$.

Use this page for PED, if needed.

```
LOOP:
ldc1 f0, 0(r1)

addi r1, r1, 8

mul.d f2, f0, f0

bneq r1, r2, LOOP

add.d f6, f6, f2
```

Problem 1, continued:

(*c*) A component failure in the MIPS implementation below has changed the circled OR gate into an AND gate.



☑ Is it still possible to perform a FP multiply? If yes, show how with PED, if no explain.

```
# Original Code. (For answer show modified code with PED.)
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.d f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WF
nop                  IF ID EX ME WB
nop                     IF ID EX ME WB
```

Yes. First consider the original code with two `nop`s added, above. In cycle 3 the `mul.d` is in M2, the circled gate will have a `1` input from the `M1/M2.we` latch and a `0` input from the uses FP add logic, and so its output will be zero, snuffing out the `mul.d` instruction.

Now consider the code execution below in which the second `nop` is changed to an `add.d`. In cycle 3 the `mul.d` in M2 will be "helped" by the `add.d` in ID and so the output of the circled gate will be `1`; with this help the `mul.d` will complete normally.

The `add.d` stalls in ID in cycle 3 (as it should) but in cycle 4 it suffers the same problem as the `mul.d` in the original code: the `we` signal is `0` when it should be a `1`. As a result the `add.d` will not finish completely. All other logic is working correctly so there is no way to "fix" the `add.d`'s we signal in the later stages and so there is no way the `add.d` can finish.

```
# SOLUTION Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
mul.d f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WF
nop                  IF ID EX ME WB                <- Other insns would work.
add.d f8, f2, f4        IF ID -> A1 A2 A3 A4 WF    <- Let mul go through.
```

☑ Is it still possible to perform a FP add? If yes, show how with PED, if no explain.

```
# Original Code. (For answer show modified code with PED.)
add.d f0, f2, f4
```

No. See the solution to the previous part.

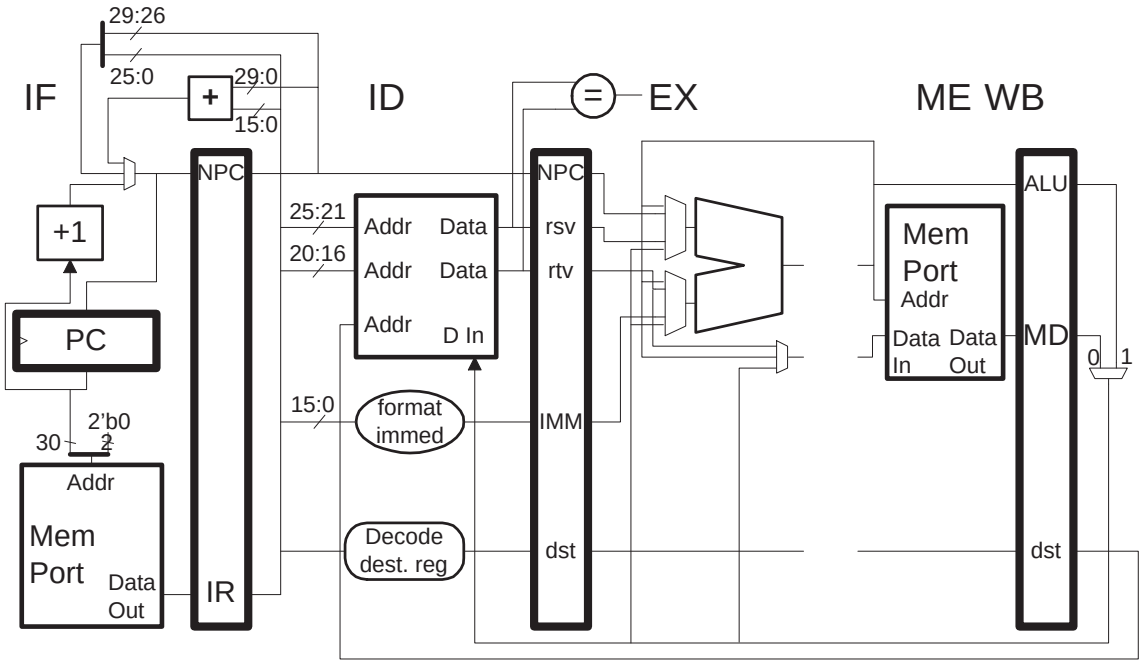
Problem 2: [30 pts] The MIPS-A ISA is like MIPS-I except that load and store instructions use only the `rs` register value for an address, they don't add an offset (or anything else). As a result MIPS-A can be implemented with a four-stage pipeline.

(*a*) Our familiar 5-stage MIPS-I implementation appears below with one of the pipeline latches missing. Make additional changes so that this is a reasonable four-stage implementation of MIPS-A.

☑ Show all connections to the memory port.

☑ Cross out wires and other items that are not needed, add what is needed.

The memory port address input should be connected to the output of the upper ALU mux. The bypass connections that used to be from ME should be removed. Ask for a diagram if needed!



(*b*) Describe two ways in which this MIPS-A implementation costs less than the five-stage MIPS-I.

☑ Two reasons for lower cost.

Reason 1: No **ME** to **EX** bypass connections (including the wires themselves, multiplexor inputs and control logic). Reason 2: No **EX/ME** pipeline latches.

Problem 2, continued:

(*c*) The four-stage MIPS-A implementation may be faster or slower than the five-stage MIPS-I.

☑ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is faster. *Hint: The MIPS-I code will have a familiar stall.*

Because the memory port and ALU are in the same stage the MIPS-A implementation can bypass a load value to the next instruction, avoiding a stall that occurs in MIPS-I. See the two code fragments. Note that the fact that WB occurs in four rather than five cycles of IF is not in itself a performance advantage, the advantage is in avoiding the stall.

```
# SOLUTION
# MIPS-I
lw r1, 0(r2)    IF ID EX ME WB
add r3, r1, r4     IF ID -> EX ME WB

# MIPS-A
lw r1, (r2)     IF ID EM WB
add r3, r1, r4     IF ID EM WB
```

☑ Provide a pair of equivalent code fragments, one for MIPS-I that runs on the five-stage implementation and one for MIPS-A that runs on the four-stage, in which the MIPS-A version is slower. *Hint: Think about the difference in the ISAs.*

MIPS-A will have to perform additions to retrieve data at an offset (small distance) from some base address, something that MIPS-I can avoid. See the examples below.

```
# SOLUTION
# MIPS-I
lw   r3, 0(r2)
lw   r4, 4(r2)

# MIPS-A
lw   r3, (r2)
addi r2, r2, 4
lw   r4, (r2)
```

(*d*) A company needs to decide whether to develop MIPS-I or MIPS-A. How does it decide? Assume that at this point software compatibility is not an issue.

☑ How should company decide which to develop?

Benchmarks based on customers' workloads should be analyzed to determine how often loads ands stores use non-zero offsets and how often a load/use stall cannot be avoided. If there are more unavoidable load/use stalls than non-zero offsets then MIPS-A might be better, otherwise MIPS-I might be better.

☑ Will having skilled compiler writers tilt the decision towards MIPS-A or MIPS-I? Explain.

MIPS-I, because the compiler can schedule to avoid load/use stalls, but there is little the compiler can do to avoid using offsets in most cases.

Problem 3: [10 pts] Answer the questions below.

(*a*) Why does MIPS have a `beq` but does not have a `blt` (branch less than), even though a `blt` instruction would be frequently used?

☑ Why `beq` but no `blt`?

Because the magnitude comparison needed for `blt` would take more time than the equality comparison needed for `beq`, long enough to lengthen the critical path.

(*b*) Explain why a branch delay slot can be though of as a short-sited feature in an ISA.

☑ Delay slots are good when...

The implementation has five stages and is scalar. Execution proceeds without a stall or squash whether or not a branch is taken (assuming no difficult dependencies).

☑ ...but delay slots are bad when...

The implementation is superscalar or deeper than five stages (roughly). In that case more than one instruction will be fetched by the time a branch is resolved and so instructions will have to be squashed. The benefit is smaller, but all of the complexity is still there.

Problem 4: [20 pts] The doing-it-the-hard-way MIPS code below loads the constant $\frac{1}{3}$ in IEEE 754 single-precision format, `0x3eaaaaab`, into register `f2`.

```
addi r10, r0, 1
addi r20, r0, 3
mtc1 f12, r10
mtc1 f22, r20
cvt.s.w f14, f12
cvt.s.w f24, f22
div.s f2, f14, f24
```

(*a*) Describe what the `mtc1` and `cvt.s.w` instructions do.

☑ Explain `mtc1`

Move to co-processor 1. This moves the value in `r10` to `f12`. It only moves it, it does not do format conversion.

☑ Explain `cvt.s.w`

Convert word (32-bit integer) to single (IEEE 754 single-precision floating point).

(*b*) The code above uses more instructions than are necessary and also uses a wastefully time-consuming instruction.

☑ What is the wastefully time-consuming instruction?

The divide, `div.s`. It's time-consuming because it's a divide, it's wasteful because it's computing something at run time that could have been computed at compile (or assembler-writing) time. See the next part.

☑ Re-write the code so it uses fewer instructions without using load instructions. *Hint: A correct answer uses three instructions and a piece of information slipped into the first sentence of the problem.*

Rather than computing it, just load the pre-computed FP value of $\frac{1}{3}$ into a register:

```
# Solution
lui r10, 0x3eaa
ori r10, r10, 0xaaab
mtc1 f2, r10
```

Problem 5: [10 pts] Answer the following SPECcpu questions.

(*a*) In the SPECcpu2000 suite profiling was allowed for both base and peak results but in SPECcpu2006 profiling is allowed for peak results but not for base results.

☑ Why isn't profiling allowed for base results in SPECcpu2006? Your answer should say something about the difference between base and peak.

The base results should reflect the performance attainable with normal effort. Profiling requires selecting a training set, compiling the program with instrumentation on, running the code on the training set, then re-compiling. Most programmers don't go to so much trouble.

(*b*) Company A has a reputation for reliable compilers, company B has a reputation for buggy compilers. Both companies and their customers are okay with this. (Think Italian sports cars.)

Optimization X results in a substantial improvement in SPECcpu base scores. Company B has it in their shipping compilers (those sold to customers) but company A only has optimization X in their experimental compilers (not available to customers), but they are working hard on it. The optimization achieves the same results for company A and B. Company B's compiler will not run on company A's system (as though they'd want to!). *Grading Note: The last line was not in the original exam, but a student did see the possibility of using B's compiler for A's spec run.*

☑ Can Company A use optimization X to prepare SPECcpu2006? Explain.

No, because they do not sell the compiler as a product.

☑ Can Company B use optimization X to prepare SPECcpu2006? Explain.

Yes, because the compiler is a product.

☑ Your answers above should reflect the letter of SPEC's rules. Do you think this achieves SPECcpu's goals or what you would like to see in benchmark results? Explain.

Yes, because SPECcpu does not claim to measure factors other than performance, such as cost, power consumption, or reliability. A person deciding between Company A and Company B's systems would balance the performance as measured by the SPECcpu numbers against the companies' reputations for quality. If A and B used the same compiler there would be no way to make this judgment.

Name Solution

Computer Architecture
EE 4720
Final Examination

7 May 2009, 17:30–19:30 CDT

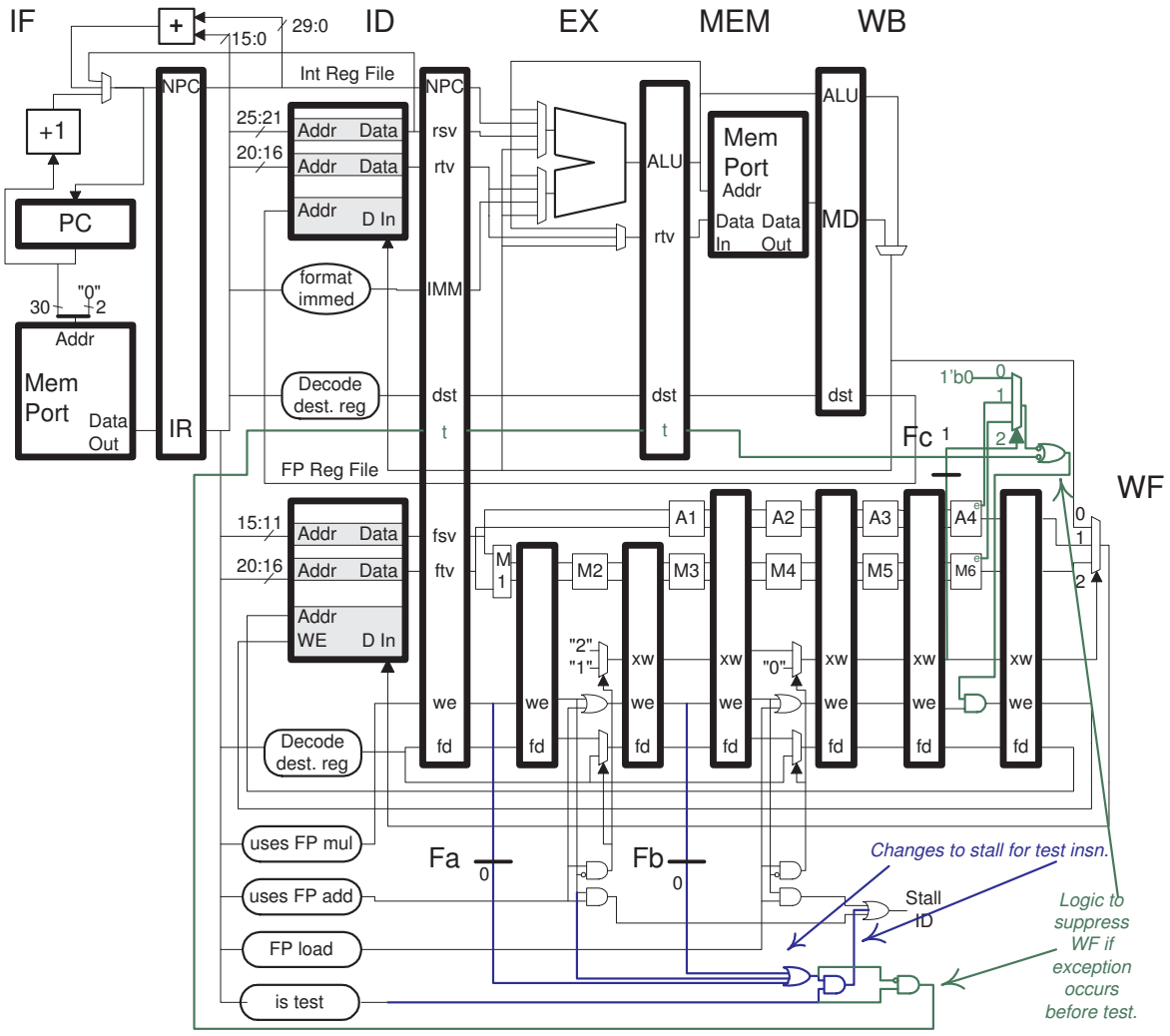
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias A(H1N1) · · · chooo!

Good Luck!

Problem 1: (20 pts) The MIPS implementation below, taken from the solution to last semester's final, includes hardware to implement an exception test instruction.

Several wires on the diagram are broken with heavy solid lines and marked with `Fx` and a value (mostly `0`). These indicate *potential* fault locations. If there is no fault the wire acts normally, if there is a fault the wire is broken at the heavy line and the free half takes on the indicated value. For example, if fault `Fa` is present the bottom input to the OR gate is always zero however the `ID/EX.we` signal on the other side is unaffected.



The problem here is to detect which fault, if any, is present by running test programs. One test program, and a pipeline diagram, appears below. A handler has been set up that will set a `goodException` variable to 1 if register and memory values are as expected, otherwise it is set to -1. The `goodException` variable is initialized to 0 before each test.

Problem 1, continued:

```
# Test 1: The mul should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 * f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
mul.s f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6x
test                  IF ID -------> EX MEx
sw r1,0(r2)              IF -------> ID EXx
```

(*a*) When a test is run the exception handler is called (because the tests intentionally raise an exception). A handler is shown below, written in C, but the handler does not set `goodException` correctly (not even close). Modify the code so that `goodException` is set correctly based on the information provided by the Test 1 code and comments. That is, `goodException` is set to -1 if the exception could not be precise.

```
int handler_fp(Regs *regs){
 // Code below wrong, but shows how to read registers and memory.

 // SOLUTION BELOW (Including commented-out line.)
 // if ( regs->f10 == regs->f12 && is_nan(f14) && MemW(regs->r31) == 0x1234 )
 if ( regs->f0 == 1 && MemW(regs->r2) == 50 )
  goodException = 1; else goodException = -1;

}
```

The exception raised by an instruction is precise if it appears that execution cleanly reached the instruction and then jumped to the handler (without executing the instruction). The solution, above, checks for two problems that might occur when the exception originates in Test 1 (or other routines specially written for this handler): the faulting instruction should not write a register (`f0` in this case) and no following instruction should change a register or memory. The `regs->f0 == 1` condition makes sure the multiply did not write back (it would have written back a non-numeric value) and the `MewW(regs->rs) == 50` condition makes sure that the store did not execute (it would have changed the memory location to 20).

(*b*) Suppose Test 1 is run and `goodException` is set to -1 (it would be 1 in the no-fault case). Which of the faults (Fa, Fb, or Fc) could have been responsible for the `goodException` value?

☑ Faults that are definitely present. Explain.

None. Any of the faults could be present. If `Fa` or `Fb` were present then the multiply would stall one less cycle than necessary allowing the `sw` to write memory. If `Fc` were present then the `mul` would write memory (assuming that the adder exception output was 0).

☑ Faults that could be present. Explain.

See solution above.

Problem 1, continued:

(*c*) Develop tests to determine for certain whether each of the faults is present. Test 1 and each of your tests will be run, and based on the `goodException` values from each test one can say for certain which faults are present.

Assume that at most one of the faults is present. Your tests should look similar to Test 1.

☑ Tests (Code like Test 1)

```
# SOLUTION
#
# Test 2: Will only fail if Fc is present.
#         The mul should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 * f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
mul.s f0, f2, f4   IF ID M1 M2 M3 M4 M5 M6x
test                  IF ID -------> EX MEx
nop                      IF -------> ID EXx

# Test 3: Will only fail if Fb is present.
#         The add should raise a precise exception.
#         Initial: f0 = 1;  r1 = 20;  Mem[r2] = 50;   f2 + f4 = NaN
# Cycle            0  1  2  3  4  5  6  7  8  9
Many nops.
add.s f0, f2, f4   IF ID A1 A2 A3 A4x
test                  IF ID -> EX MEx
sw r1,0(r2)              IF -> ID EXx
```

Test 2 above is like Test 1 except there is no `sw`, so fault Fa and Fb won't cause incorrect operation, but Fc still will. Note that Test 2 would not be necessary if the handler could set a variable indicating the reason for failure (either the `sw` writing memory or the faulting instruction writing `f0`).

☑ Which combination of `goodException` values conclude `Fa` for certain.

`Fa` is present if Test 1 fails (`goodException`=-1) and Tests 2 and 3 pass (`goodException`=1). See the discussion about the Test programs.

☑ Which combination of `goodException` values conclude `Fb` for certain.

Test 3, `goodException`=-1. Test 3 can't be affected by `Fa` because the add instruction never passes through the `M1` stage and it can't be affected by `Fc` because the add uses input 1 to the multiplexor anyway.

☑ Which combination of `goodException` values conclude `Fc` for certain.

Test 2, `goodException`=-1. Since there is no `sw` after the `test` instruction it can't fail because of there were not enough stalls, the only possibility is `Fc`.

Problem 2: (20 pts) Answer each question below. **Be sure to check each code fragment carefully for dependencies.**

(*a*) The loop below runs on a statically scheduled 4-way superscalar MIPS implementation.

☑ Show a pipeline execution diagram.

```
#                 SOLUTION
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
addi r2, r2, 8    IF ID EX ME WB
lw r1, 0(r2)      IF ID -> EX ME WB
add r3, r1, r4    IF ID -------> EX ME WB
bneq r2, r5 LOOP  IF ID -------> EX ME WB
sw r3, 4(r2)         IF -------> ID EX ME WB
(XXX)
addi r2, r2, 8                   IF ID EX ME WB
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

☑ Determine the IPC for a large number of iterations and assuming no cache misses.

Iterations start at cycle 0 and 5. There are no reasons why the second iteration would be any different than the first so the execution rate is $\frac{5}{5}$ IPC.

☑ Comment on the difference between the IPC and the potential IPC of the processor.

The potential is 4 but we only get 1. That's very inefficient for code without cache misses.

☑ Schedule the code to improve execution time.

```
# First iteration + 0x8
addi r2, r2, 8
lw r1, 0(r2)
add r3, r1, r4

# Second iteration + 0x10
addi r2, r2, 8
lw r1, 0(r2)

LOOP: # Cycles    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw r3, -4(r2)     IF ID EX ME WB
add r3, r1, r4    IF ID EX ME WB
lw r1, 8(r2)      IF ID EX ME WB
bneq r2, r5 LOOP  IF ID EX ME WB
addi r2, r2, 8       IF ID EX ME WB
(xxx) # Cycles    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
sw r3, -4(r2)           IF ID EX ME WB
add r3, r1, r4          IF ID EX ME WB
lw r1, 8(r2)            IF ID EX ME WB
bneq r2, r5 LOOP        IF ID EX ME WB
addi r2, r2, 8             IF ID EX ME WB
```

The scheduled code appears above. Execution is shown on an implementation that can bypass a value from **EX** to the branch condition comparison unit in **ID**. Work that was done in a single iteration in the original code is spread out over three iterations in the scheduled code above. For example, the value loaded by the **lw** in iteration 1 is used by the **add** in iteration 2 and the sum produced by the **add** in iteration 2 is used by the **sw** in iteration 3. This technique is called software pipelining.

Problem 2, continued:

(*b*) The code below (same as the previous problem) executes on a 4-way superscalar dynamically scheduled machine. Assume that branch prediction on this machine is perfect. Load instructions use the `EA` and `ME` stages, branch instructions use the `B` stage, store instruction only use `EA` (they write memory when they commit).

Though the machine is 4-way, assume an unlimited number of `WB`, `EX`, and `RR` stages.

☑ Show a pipeline execution diagram for two iterations.

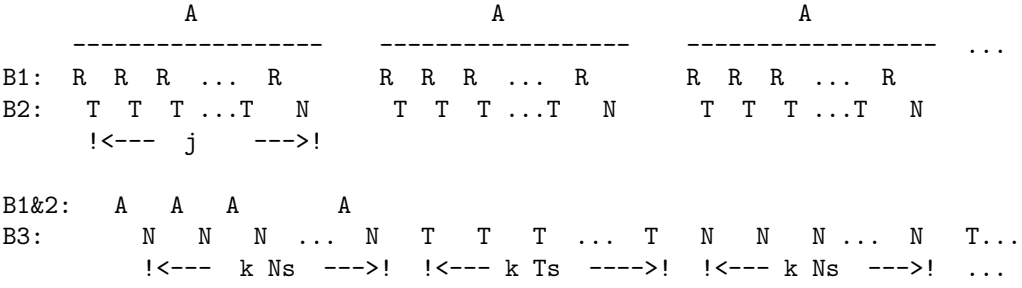
```
# SOLUTION
LOOP: # Cycles    0  1  2  3  4  5  6  7  8  9  10 11
addi r2, r2, 8    IF ID Q  RR EX WB C
lw r1, 0(r2)      IF ID Q     RR EA ME WB C
add r3, r1, r4    IF ID Q           RR EX WB C
bneq r2, r5 LOOP  IF ID Q     RR B  WB       C
sw r3, 4(r2)         IF ID Q  RR EA WB       C
(XXX)
addi r2, r2, 8          IF ID Q  RR EX WB    C
lw r1, 0(r2)            IF ID Q     RR EA ME WB C
add r3, r1, r4          IF ID Q           RR EX WB C
bneq r2, r5 LOOP        IF ID Q     RR B  WB       C
sw r3, 4(r2)               IF ID Q  RR EA WB       C
LOOP: # Cycles    0  1  2  3  4  5  6  7  8  9  10 11
```

☑ Determine the IPC for a large number of iterations.

The execution rate is $\frac{5}{2} = 2.5$ IPC.

Problem 3: (20 pts) Code producing the branch patterns shown below is to run on three systems, each with a different branch predictor. All systems use a 2^{14}-entry BHT. One system has a bimodal predictor, one system uses a local history predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history.

The code has three branches, B1, B2, and B3. The outcome of B1 is random, described by a Bernoulli random variable with $p = .5$ and is independent of everything. Branch B2 has a simple j-iteration loop pattern ($j-1$ T's and an N) and branch B3 is a repeating pattern of k N's followed by k T's (see diagram below). Also from the diagram notice that B1 occurs just before B2 but that a set of j B1 and B2s occur between each B3, (similar to a branch in the homework and last semester's final).



For the questions below accuracy is after warmup.

☑ What is the accuracy of the bimodal on B1?

The branch is random and uncorrelated so there is no way prediction accuracy can be anything other than $\boxed{0.5}$.

☑ What is the accuracy of the bimodal on B2 in terms of j?

The Ts will all be correctly predicted the Ns will all be mispredicted, so the $\boxed{\text{accuracy is } \frac{j-1}{j}}$, for $j = 5$ that would be $\frac{4}{5} = 0.8$.

☑ What is the accuracy of the bimodal on B3 in terms of k?

When $k > 2$ there will be two mispredicts after each change from N to T or T to N. Therefore $\boxed{\text{the accuracy is } \frac{k-2}{k}}$. For $k = 4$ the accuracy is 0.5.

☑ What is the accuracy of the local predictor on B3 when $k \gg 10$ and $j < 5$ in terms of j and k?

When $j < 5$ the patterns generated by B2 cannot match the patterns generated by B3. The only PHT entries that will result in a misprediction of B3 are the all Ns, and all Ts, and they will only be wrong just at the N to T or T to N transition. Therefore $\boxed{\text{the accuracy is } \frac{k-1}{k}}$.

☑ What is the accuracy of the local predictor on B3 when $k \gg 10$ and $j > 10$ in terms of j and k?

With $j > 10$ both B2 and B3 will have local history TTTTTTTTTN. For B2 the next outcome is T, but for B3 the next outcome is N. B2 will generate the local history far more frequently and so it will keep the respective PHT entry at 3. So now branch B3 will mispredict at the transition (where its local history will be all Ns or all Ts) and at pattern TTTTTTTTTN. It's overall $\boxed{\text{accuracy will be } \frac{2k-3}{2k}.}$

☑ What is the accuracy of the global predictor on B3 when $j = 10$ and k is very, very large, in terms of j and k?

Because the local history size is 10 and $j = 10$ the GHR (global history register) contents seen when predicting B3 does not contain any past outcomes of B3, it only contains B1 and B2 outcomes, neither of which are helpful. Therefore at each transition there will be mispredicts until the PHT entries warmup, after which accuracy will be 1.0 until the next transition. If k is very very large we can say that the $\boxed{\text{accuracy will be very very close to 1.0}}$. Not satisfied? See the next problem.

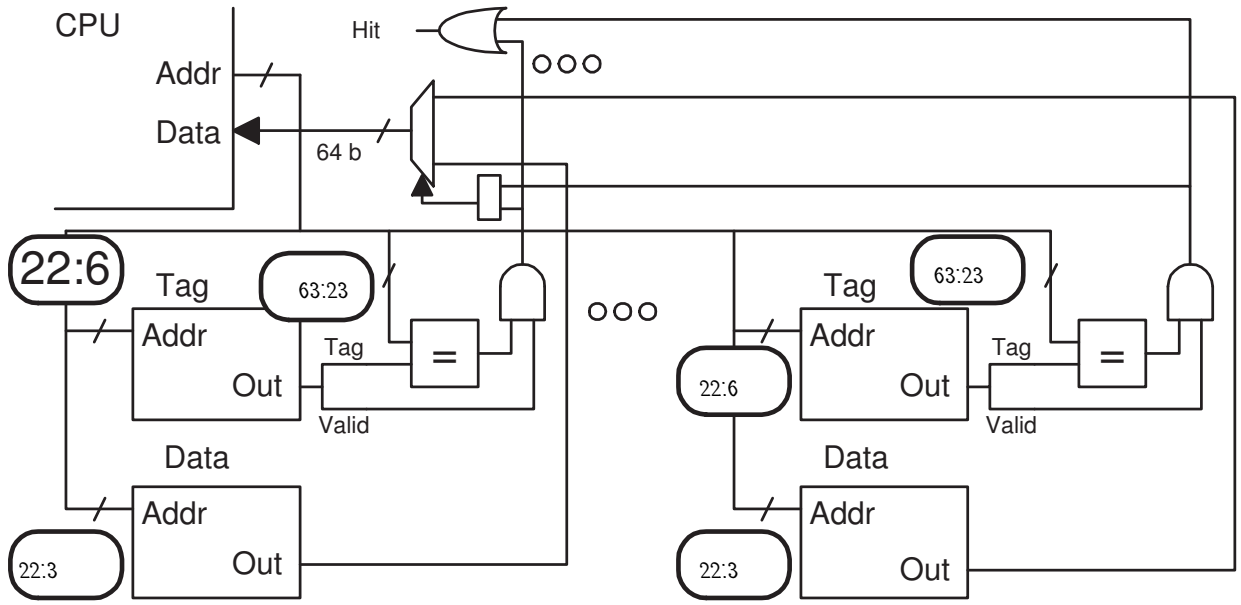
☑ What is the warmup time for B3 on the global predictor. (This warmup occurs whenever B3 switches from Ns to Ts or Ts to Ns.) in terms of j and k?

The global history encountered when predicting B3 contains 5 B1 outcomes and 5 B2 outcomes. The GHR contents will be rTrTrTrTrN, where r are the B1 outcomes and the other elements are B2 outcomes. Since B1 is random the r can be either N or T. There are five of them and so there are $2^5 = 32$ patterns of this form. Each corresponding PHT entry needs to warm up on a B3 transition, which requires two occurrences. The $\boxed{\text{warmup time is then } 2 \times 2^5}$ and the accuracy of the global predictor is $\frac{k-2\times 2^5}{k}$.

Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25}-character) set-associative cache with the usual 8-bit characters and a 64-bit address space.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

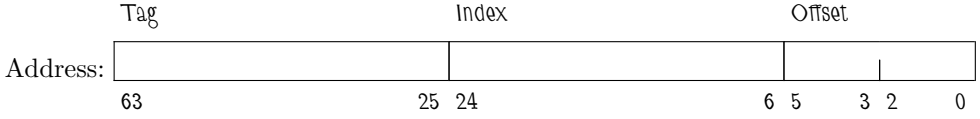
☑ Associativity:

From the given tag store address bits, 22:6, we know the data store has a capacity of 2^{23} characters so for a cache capacity of 2^{25} characters the associativity must be $\frac{2^{25}}{2^{23}} = 4$.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{25} characters, plus $4 \times 2^{23-6}\,(64 - 23 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 4, continued:

(*b*) The code below runs on the cache from the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;
int i;
int ILIMIT = 1 << 11;    // = 2^11

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 2^6 characters is given, the size of an array element is 1 character, and so there are 2^6 elements per line. The first access, at `i=0`, will miss but bring in a line with 2^6 elements, the next $2^6 - 1$ accesses are to subsequent elements on the line and so will hit, so the hit ratio to the first line is $\frac{2^6-1}{2^6}$.

(*c*) The code below runs on a direct-mapped cache unrelated to the one above. When the code starts running the cache is cold, for the solution only count accesses to `array`.

```
struct My_Struct {
  double val;
  double something;
  double relative;
  double more_data[29];
};  // Total size: 32 * sizeof(double) = 256 bytes

const int SIZE = 1 << 12;
My_Struct array[SIZE];  // &array[0] = 0x1000000

void tri()
{
  double sum = 0;
  for ( int i=0; i<SIZE; i++ ) sum += array[i].val;
  const double avg = sum / SIZE;
  for ( int i=0; i<SIZE; i++ ) array[i].relative += array[i].val - avg;
}
```

☑ Determine the minimum cache and line size needed so that there are no misses in the second `for` loop.

The minimum line size is 32 characters, the minimum cache size 128 kiB.

☑ Explain your answer.

Both `for` loops iterate over the entire array, the first `for` loop only accesses the `val` member but the second `for` loop accesses both the `val` and `relative` member. The line size will be chosen so that on a miss in the first `for` loop both members will be on a line. If line size were not restricted to a power of two then a line size of 24 characters would be sufficient, but they are so the minimum line size is 32 characters. Such a line can hold both `val` and `relative`, but whether it will depends on the alignment of the structure. From the given address of `array[0]` (which is the address of `array[0].val`) the offset bits are obviously zero (meaning it is at the beginning of a line). Since the struct size is a multiple of 32 `array[i].val` will also be on the beginning of a line.

The number of lines needed to store the array is 2^{12} and so the total cache size needed is $32 \times 2^{12} = 2^{17}$ or 128 kiB.

Problem 5: (20 pts) Answer each question below.

(*a*) In a dynamically scheduled machine one would like to be able to have a large number of instructions in flight to, say, find something to do while waiting for data from memory. Which part of the dynamically scheduled machine is most difficult to scale up to support a large number of in-flight instructions?

☑ Part that's difficult to scale, and reason why.

The scheduler. It needs to monitor the status of the source registers of every instruction in the instruction queue so that it can find instructions that are ready for execution. It does this each cycle based on the destination registers that will be written by the earlier rounds of instructions. It is difficult to do this quickly for a large number of instructions.

(*b*) How might profiling improve the performance of the following C code:

```
if ( a > b ) { x = d / e; } else { y = q / f; }
```

☑ Explain how profiling is used.

First the code is compiled with a special profiling switch. It is then run on what is hoped to be typical input data, while running it will record branch direction counts and other information to a profile file. The code will be compiled a second time using another profiling switch. The compiler will read the profile file and use that information to make optimization decisions. See the next answer.

☑ Explain why this profiled code might be faster than code compiled without profile feedback.

A taken branch or jump slows execution more than a not-taken branch, even with correct prediction. Based on a profile run the compiler might learn that in the `if` statement above the `else` part is rarely executed. It will organize the code so that two taken branches are needed to execute the `else` part but zero branches (or jumps) are needed for the `if` part.

(*c*) In the first implementation of a company's ISA the integer multiply instruction was slow. An Engineer working on the second implementation is deciding whether to speed up the multiply instruction. To decide he plans to analyze some benchmark runs, but he can't decide whether the code should be optimized. The compiler was designed for the first implementation.

The engineer is trying to determine if the multiply instruction is used often enough to justify the effort to improve it. To do that he runs benchmark programs using some kind of simulator that can tell him how many times a multiply was used while running the code. If, say, the multiply accounts for less than 0.00001% of executed instructions then it's not worth the effort.

☑ What is the disadvantage of counting multiply usage in code compiled with optimization?

Since the multiply was slow in the first implementation the compiler might have used it less often, perhaps substituting adds and shifts. However, the compiler would only look for alternatives to a multiply if optimization were turned on. With optimization off the compiler generates straightforward code without tricks.

Since the compiler is avoiding use of the multiply instruction the number of multiplies would be undercounted. That would be bad if the compiler's alternative were slower than the speeded-up multiply instruction in the second implementation.

☑ What is the disadvantage of counting multiply usage in code compiled without optimization?

Without optimization there might be too many multiplies. Techniques like dead-code elimination and constant folding would eliminate multiplies no matter how fast the instruction is.

(*d*) A memory system that can fetch 2^w chars of data is less expensive and faster if the data address is a multiple of 2^w. (That's one reason for the alignment restriction.)

☑ How do VLIW ISAs take advantage of that?

Instruction bundle sizes are set to a power of two and control-transfer instructions can only jump to the beginning of a bundle. Therefore the memory port used for instruction fetch can enjoy the cost and performance benefits of alignment.

(*e*) Compared to a RISC implementation, say the 5-stage MIPS, what additional logic does a CISC implementation require between the IF-stage memory port output and decode?

☑ Additional logic for CISC.

Since instruction length varies it will need logic to shift the instruction into position and logic to combine the bits fetched in one cycle with a piece of the instruction fetched in an earlier cycle.

59 Fall 2008 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 31 October 2008, 10:40–11:30 CDT

Problem 1		(50 pts)
Problem 2		(10 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Well ARMed

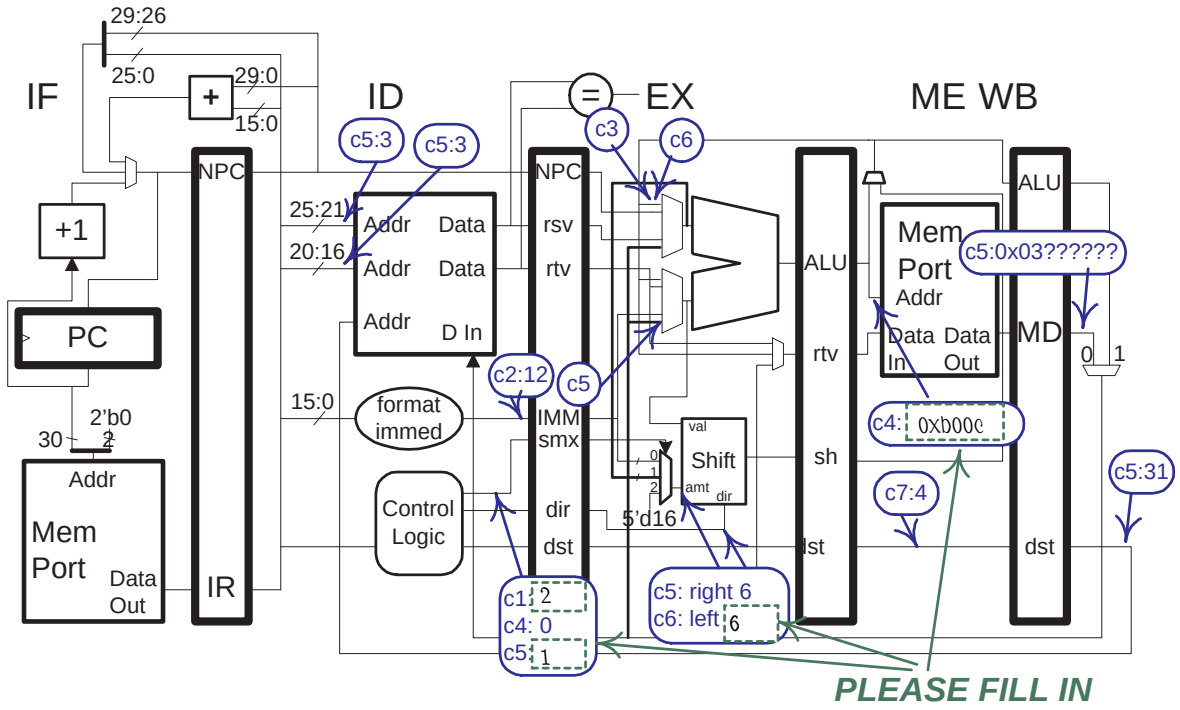
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c5:3 indicates that at cycle 5 the wire will hold a 3. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Instruction addresses and the first instruction have been provided. [50 pts]

(*a*) Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Be sure to fill the **three** blocks marked *PLEASE FILL IN*.




```
# SOLUTION
#  Cycle                    0  1  2  3  4  5  6  7  8
0xb0000 lui r2, 0xb         IF ID EX ME WB
0xb0004 lw r31, 12(r2)         IF ID EX ME WB
0xb0008 j TARG                    IF ID EX ME WB
0xb000c srl r3, r31, 6               IF ID EX ME WB
0xe0000 sllv r4, r3, r3                 IF ID EX ME WB
#  Cycle                    0  1  2  3  4  5  6  7  8
```

Solution discussion on next page.

0xb0004: The c4 item in ME indicates that this is a load or store, the c5:0x03?????? in WB reveals that it must be a load and because of the size of the value, it must be a load word. The destination register is directly given (the c5:31 in WB), the bypass with `lui` reveals the `rs` register. The address can be calculated using the known immediate values (12 from `lw` itself and `0xb` from `lui`). Note that the load address matches the address of the fourth instruction.

0xb0008: See the next part for a discussion of this instruction.

0xb000c: The identity of the instruction is revealed by the shifter control signals, the dependence with the fifth instruction reveals the destination register.

0xedd900: The control signals reveal the type of instruction. The interesting part is determining the shift amount for the C6:left...: The `lw` instruction loads the fourth instruction into `r31`. The fourth instruction shifts the value (itself) right by six bits, putting the `sa` field into the least significant bits of `r3`. We know from c5:right 6 that the `sa` field of the fourth instruction is 6, so the least significant bits of `r3` must be a 6, and that is what is used for the shift amount in this last instruction: c6:left 6.

Problem 1, continued: Continue referring to the implementation on the previous page.

(*b*) Explain whether each instruction below could be the instruction at address `0xb0008` (on the previous page). If it could be the instruction write "Possible" otherwise write "Impossible because ..." (The grade will be based on the reason.)

☑ `add`

Could not be. Since the address of the fifth (and last) instruction, `0xedd900`, is not the address of the fourth instruction, `0x0b000c`, plus 4 the third instruction must be some kind of a control transfer. Therefore it can't be an `add`.

☑ `j`

It could be. As explained above, the third instruction has to be some kind of control transfer instruction, and there's nothing to rule out a `j`.

☑ `jal`

Could not be. The `jal` instruction always writes the return address to `r31`. A bypass path is used to send the value of `r31` written by the second instruction to the fourth instruction. If the third instruction also wrote `r31` that bypass path would not be used, and so the third instruction can't be a `jal`.

☑ `beq`

Could not be. Yes, a `beq` is a control transfer, but its range is limited to $\pm 2^{15}$ instructions, so it could not reach address `0xedd900` from `0x0b0008`.

Problem 2: [10 pts] Based on the experience of preparing SPECcpu benchmark runs manufacturers A and B each release a new compiler that improves their respective SPECcpu scores.

(a) In the course of preparing a SPECcpu benchmark run manufacturer A discovers a new optimization technique that improves the performance of most of the SPECcpu programs, and other programs. This optimization technique is added to the compiler for use at the `-O1` and higher optimization levels. The manufacturer sells the compiler, proudly boasting about the performance benefits.

☑ Is it in the spirit of the SPECcpu rules to use this new optimization technique for the base results? Explain.

Yes. The only difference between it and other optimizations is that it was invented in the process of running the SPECcpu benchmarks.

(b) To prepare a SPECcpu benchmark run manufacturer B has its best programmers prepare hand-written assembly code for the most time consuming portion of each benchmark. The compiler will recognize each benchmark based on the source code and substitute the hand-written routines where needed; the hand-written code will only work for these benchmarks.

These optimizations are included in manufacturer B's compiler and used at `-O1` and higher levels. Manufacturer B sells this new compiler.

☑ Is it in the spirit of the SPECcpu rules to use this new optimization technique for the peak results? Explain.

It is not in the spirit of peak rules because the score does not reflect the performance that users can obtain on similar programs. Re-writing benchmark code is against the SPECcpu rules and that is what this testing procedure effectively does. It is likely that this technique might violate the letter of the rules too.

☑ Explain why it is not in the spirit of SPECcpu rules to use such optimizations for base results.

Since it's not in the spirit of the peak rules it can't be in the spirit of the stricter base rules either. One reason that applies to base but not peak: The amount of effort needed to produce the hand-optimized code is well beyond base intent, and so it's not in the spirit of base rules.

☑ Assuming that the compiler can not be reverse engineered (there is no way to inspect the compiler itself) and assuming that manufacturer B can keep secrets, how might this cheating be discovered?

The performance on similar programs would be disappointing in comparison to the SPECcpu benchmarks under peak rules. In fact, one might change a few lines of the SPECcpu benchmarks and see a large performance drop.

Problem 3: Answer the following ISA questions.

(*a*) [10 pts] A RISC advocate claims that by having fixed-length instructions and alignment restrictions a branch can reach twice as far for a given displacement field size than would be possible in CISC ISAs.

☑ Explain why.

RISC instruction addresses (in most RISC ISAs) are multiples of 4 (since the size is four bytes and the alignment feature imposes the multiple of 4 restriction). Therefore the displacement can be the number of instructions to skip. In variable-instruction-length ISAs the displacement would have to be the number of bytes to skip and because most instructions are more than one byte a given displacement size can not go as far.

A CISC advocate responds that branches in CISC programs would take less space anyway.

A primary feature of CISC ISAs is variable instruction size. Among other things this allows instructions' immediate size to match their needs. So, for a small displacement control-transfer there might be a branch instruction with a one-byte immediate field and for a large displacement control transfer there might be a branch instruction with a four-byte immediate field.

☑ Provide a reason for small-displacement branches.

A small displacement branch instruction could be encoded in two bytes, half the size of any MIPS branch.

☑ Provide a reason for large-displacement branches.

Consider MIPS. A control transfer to a target more than 2^{15} instructions away would have to be realized using three instructions: two to put the target address in a register, and a `jr` to perform the jump. A fourth instruction would be needed for a conditional control transfer. A CISC ISA might just use a five-byte instruction.

(*b*) [10 pts] Indicate whether each item below is usually an ISA feature or an implementation feature.

Grading Note: Easy. Maybe too easy.

☑ Number of bits in immediate.

ISA.

☑ Clock frequency.

Implementation.

☑ Number of branch delay slots (if any).

ISA, though chosen with certain implementations in mind.

☑ Floating-point format.

ISA.

☑ Minimum distance between load instruction and a dependent instruction to avoid a stall.

Implementation.

Problem 4: Equivalent MIPS and SPARC code fragments appear below.

(*a*) [10 pts] Taking advantage of SPARC's condition code features, modify the SPARC code to use one fewer instruction (without changing what it does).

```
# MIPS Branch Example
 addi $t1, $t1, -1
 bne  $t1, 0  LOOP
 add  $t2, $t2, $t3
...

! Equivalent SPARC Branch Example
 add l1, -1, l1      ! l1 = l1 - 1
 subcc l1, 0, g0     ! g0 = l1 - 0
 bne LOOP
 add l2, l3, l2
...
```

There is no need to use `subcc` to set the condition code bits since the `add` writes register `l1` and it could also set the condition code bits. Do that using an `addcc` instead of an `add`:

```
! SOLUTION: Modified SPARC code.
 addcc l1, -1, l1      ! l1 = l1 - 1
 bne LOOP
 add l2, l3, l2
...
```

(*b*) [10 pts] For branch-in-ID implementations, why might it be possible to attain higher clock frequencies for condition-code ISAs, like SPARC, than for register-test branche ISAs, like MIPS.

☑ Condition code performance advantage.

The branch logic in a SPARC implementation only needs to examine four bits (the condition code), in MIPS-32 implementations two 32-bit values need to be compared.

Name Solution

Computer Architecture
EE 4720
Final Examination

9 December 2008, 17:30–19:30 CST

Problem 1		(10 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(15 pts)
Problem 5		(20 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias ISA were, was I?

Good Luck!

Problem 1: (10 pts) Consider a new floating point instruction `nin.d` which uses a 5-stage computation unit, N1 - N5 (in contrast to FP add's A1-A4). The format and register usage for `nin.d` is the same as the other FP arithmetic instructions. Modify the implementation below so that it can execute `nin.d`. *Hint: This is an easy question,* `nin.d` *is just like the FP add and multiply, except it's 5 stages.*

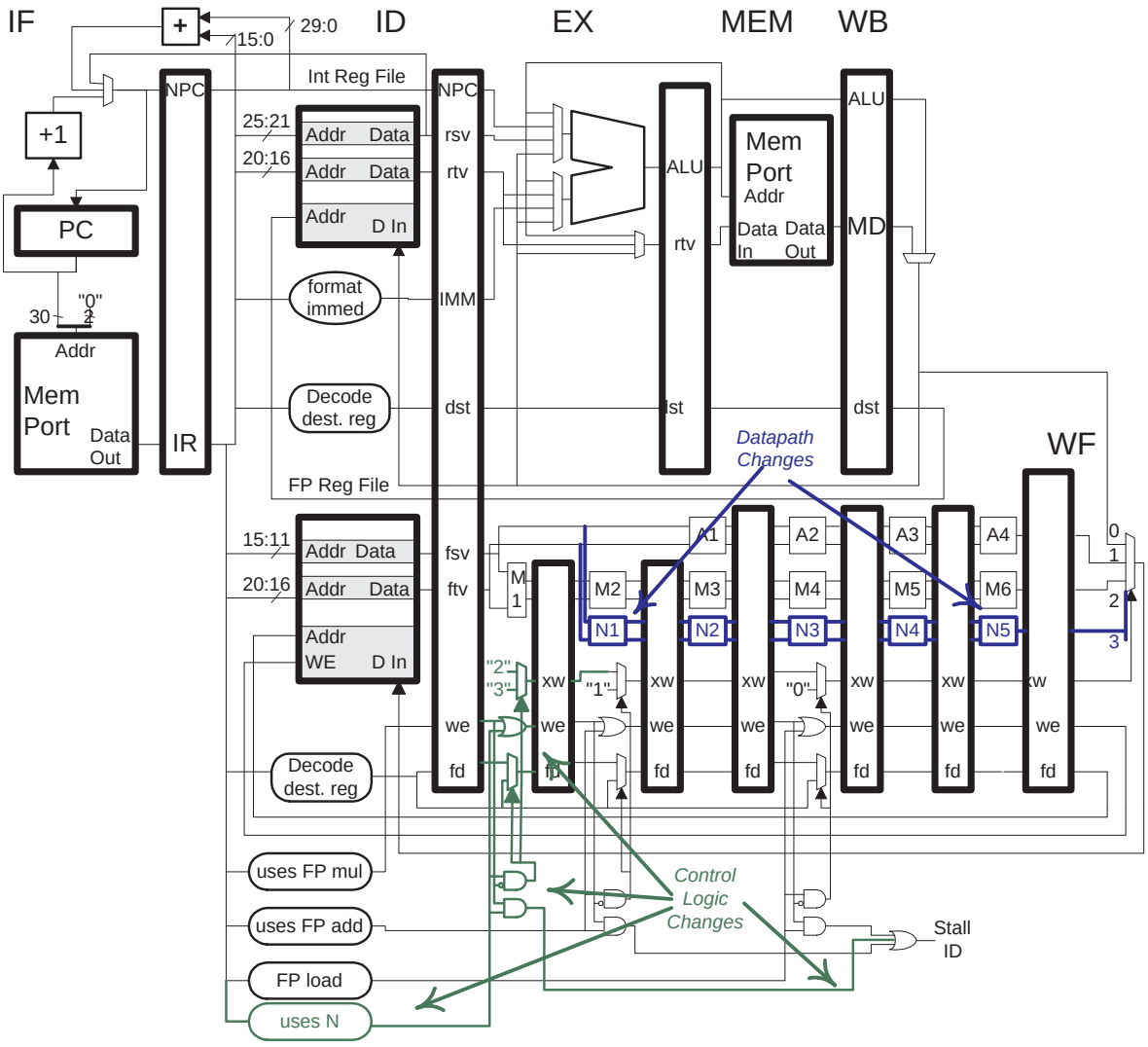
☑ Add the datapath components, including the functional unit stages (N1 to N5).

Solution appears below in blue.

☑ Add control logic for proper write-float and structural hazard detection (as is already present for add and multiply). Be sure not to break existing instructions.

Solution appears below in green.

☑ The changes must fit in efficiently with what is already present.



Problem 2: (15 pts) With the MIPS implementation illustrated below floating-point add and multiply instructions do not raise precise exceptions. If a programmer needs a precise exception for a particular FP instruction he or she can follow it with a `test` (test for exception) instruction. In the code below `test` is used to provide a precise exception for `mul.d`.

```
mul.d  f2, f2, f6
test
sub.d  f16, f14, f20
```

The `test` instruction will stall in ID for **the minimum number of cycles necessary** to ensure a precise exception and will suppress a WF if necessary.

(*a*) Add hardware to implement the `test` instruction. The output of is test is `1` if a `test` instruction is in `ID`. Assume there are A4 and M6 outputs that indicate whether an exception was raised. These can be checked in the middle of the cycle.

☑ Add control logic to stall the pipeline using a new input to the *Stall ID* or gate. *Hint: Very little logic is needed.*

Changes shown in blue on the next page. When a `test` instruction is in ID it will stall the pipeline until there is no FP instruction more than three cycles from `WF`. That is achieved by simply ORing the `we` bits from stages that are more than three cycles from `WF`. If an instruction three or less cycles from WF raises an exception there will be enough time to squash any instruction after the test. (See part b.)

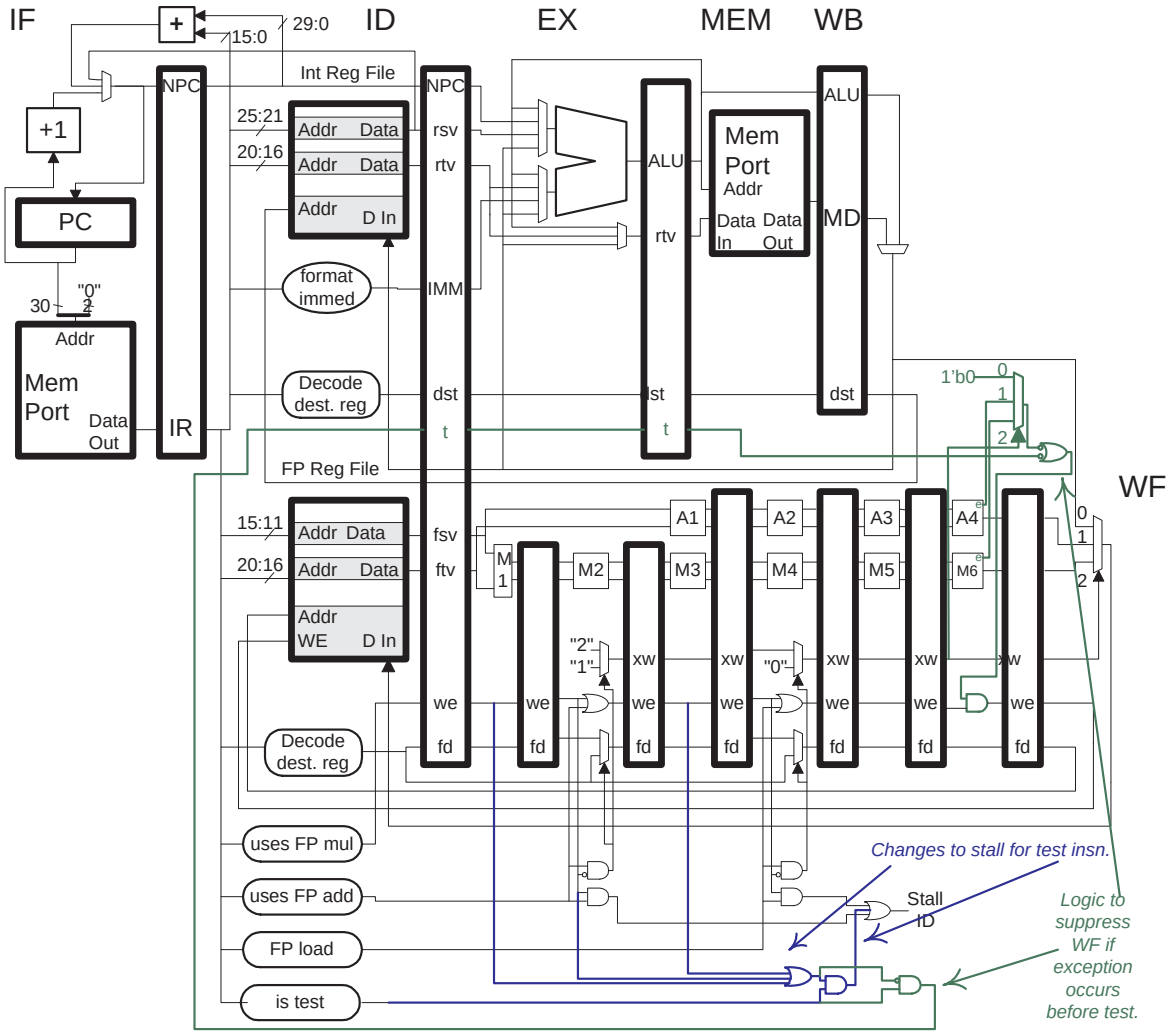
☑ Add logic to suppress WF when necessary.

Changes shown in green on the next page. The WF needs to be suppressed for a FP instruction that raises an exception and is followed by a `test` instruction.

New "e" outputs were added to A4 and M6, these are logic 1 if there is an exception. The added multiplexer will select the e value for the unit in use (or a 0). The instruction in A4/M6 will not be allowed to write back (its `we` bit will not be be propagated) if it raises an exception and it is right before a test instruction. Note that an excepting instruction that doesn't precede a test instruction is allowed to write back, it would write back a NaN (not a number) or some other special value.

A much simpler solution would just AND the we with the negated e signal. That would suppress writeback for all instructions that raised an exception, which is not the desired behavior.

☑ The hardware should work correctly for floating-point multiplies and adds.


IF
ID
EX
MEM
WB
WF
+
+1
PC
NPC
IR
Addr
Mem
Port
Data
Out
30
"0"
2
15:0
29:0
Int Reg File
25:21
20:16
Addr Data
Addr Data
Addr D In
format
immed
IMM
Decode
dest. reg
dst
t
rsv
rtv
ALU
Mem
Port
Addr
Data In
Data Out
MD
1'b0
FP Reg File
15:11
fsv
ftv
Addr WE D In
M1
M2
M3
M4
M5
M6
A1
A2
A3
A4
"2"
"1"
"0"
xw
we
fd
uses FP mul
uses FP add
FP load
is test
Changes to stall for test insn.
Stall
ID
Logic to
suppress
WF if
exception
occurs
before test.

Problem 2, continued:

(*b*) Show an example in which, in a faulty implementation, the test instruction stalls one less than the minimum number of cycles and as a result an exception is not precise.

☑ Example code and pipeline execution diagram, including fetch of first handler instruction.

```
# SOLUTION
#
# Cycle           0  1  2  3  4  5  6  7  8  9
mul.d  f0,f2,f4   IF ID M1 M2 M3 M4 M5*M6*x
test                 IF ID ----> EX ME WBx
sw r6, 0(r7)            IF ----> ID EX MEx
add                              IF ID EXx
or                                  IF IDx
..
HANDLER:
xor                                    IF ID ..
```

☑ Explain why the exception is not precise.

See the pipeline diagram above. The `mul.d` raises an exception in cycle 7, that's too late to stop the `sw` from writing memory. (The memory write started in the beginning of the cycle but the exception was detected later, in the middle of the cycle.)

As a result the handler, which starts fetch in cycle 7, will "see" the change the `sw` made to memory and that violates a requirement for a precise exception.

If the `test` instruction had stalled one more cycle the `sw` could be squashed in time and the exception would have been precise.

Problem 3: (20 pts) The code below has two branches, branch `B1` implements a 3-iteration loop with outcome pattern `T T N T T N`$\cdots$. The outcome pattern for branch `B2` is shown below. **Note that `B1` executes three times for each execution of `B2`.**

The code runs on three systems, the branch predictor on all systems uses a 2^{14}-entry BHT. One system has a bimodal predictor, one system uses a local history predictor with a 10-outcome local history, and one system uses a global predictor with a 10-outcome global history.

```
BIGLOOP:    Note: B1 and B2 are the only branches in this code.
T1:
B1:  bne r3, r4 T1   .. 3 iteration loop ..

# .. no CTIs ..
B2:  beq r1, r2 T2   N N N T N N T N N N T N N T N N N T N N T N N N T N N T ...
T2:

j BIGLOOP
```

☑ What is the accuracy of the bimodal predictor on branch `B2`?

The prediction accuracy is $\frac{5}{7}$.

☑ What is the accuracy of the local predictor on branch `B2`?

The period-seven pattern can easily be captured in the local predictor's ten-outcome history, and so the prediction accuracy is 100%.

☑ What is the size of the BHT and PHT, in bits, needed to implement the local predictor? Only take into account the storage need for the branch predictor, omit things like CTI type.

The BHT size is $2^{14} \times 10$ bits (each BHT entry holds a local history, which is 10 bits). The PHT size is $2^{10} \times 2$ bits (there is one entry for each possible outcome pattern, each entry is two bits).

☑ What is the minimum local history size needed to achieve 100% accuracy on branch `B2` using the local predictor? Explain.

Six outcomes.

A five-outcome pattern is too short, for example, `NNTNN` can be followed by an `N` or a `T`, but a six-outcome pattern is unambiguous. See the table below in which all seven shifts of the pattern `NNNTNNT` are shown. None of the patterns match in the first six bits. (That's a sufficient, but not a necessary condition for 100% accuracy.)

```
123456 7
NNNTNN TN
NNTNNT NN
NTNNTN NN
TNNTNN NN
NNTNNN NT
NTNNNN TN
TNNNNT NN
NNNNTN NT
```

☑ What is the minimum global history size needed to achieve 100% accuracy on branch `B2` using the global predictor? Explain.

Based on the answer above, the global predictor needs to "see" the last six outcomes of branch B2. However, after each outcome of B2 is a set of 3 outcomes of B1, and so the global history size must be at least $\boxed{4 \times 6 = 24 \text{ outcomes}}$ long to hold six outcomes of B2.

Problem 4: (15 pts) Consider two alternatives for improving the performance of our familiar scalar 5-stage implementation: an n-way superscalar implementation or a $5n$-stage superpipelined implementation.

(a) Both systems can *potentially* reduce execution time by a factor of n. Explain how this is achieved for each system in terms of CPI (cycles per instructions), clock frequency, and other relevant factors. The answers might read, "Because of ... the CPI is x^2 times larger which causes ... but ... **and so overall execution is n times faster**."

☑ Explain how the superscalar system is potentially n times faster.

Because each stage has n copied of most hardware, the superscalar system will execute up to n instructions for each instruction executed by the scalar system. The duplicated hardware might have only a small impact on clock frequency (if n is small), so IPC numbers reflect performance.

☑ Explain how the superpipelined system is potentially n times faster.

Because each stage is carefully divided into n parts, the superpipelined system's clock frequency can be n times faster (a bit lower due to overheads). Though the execution rate expressed in IPC won't go up, there are now n times more cycles per unit time.

(b) Bypass paths add substantially to the cost of sufficiently large superscalar systems. Provide expressions in terms of n for the cost of bypass paths in both systems. Briefly justify your answers, using a diagram if necessary.

☑ Expression for the cost of bypass paths in superscalar systems, with quick diagram and brief description.

The cost of bypass paths will be measured in multiplexor inputs needed. The EX stage has n ALUs, each with 2 inputs, for a total of $2n$ bypass path destinations. The ME and WB stages together hold $2n$ destination values, so the number of bypass multiplexor inputs needed is $4n^2$. One component of $\boxed{\text{cost is thus } 4n^2}$.

☑ Expression for the cost of bypass paths in superpipelined systems, with quick diagram and brief description.

The ALU gets its input values in the EX0 stage (the first part of the EX stage which was divided into n pieces), so the number of bypass path destinations is 2. Each of the stages from ME0 to WBn-1 can hold one bypassable value, for a total of $2n$ values. The total number of multiplexor inputs is thus $\boxed{4n}$.

Note that the superpipelined system has fewer mux inputs because it is doing with time what the superscalar system does with space.

(c) Ideally the $5n$-stage superpipelined system would have a speedup of n (the scalar system would take n times longer than the superpipelined system to run a program). Consider a program that has no nearby dependencies so that the superpipelined system does not have to stall.

☑ Explain why the speedup of the superpipelined system might be $\frac{t_0+t_1}{t_0+\frac{t_1}{n}}$, where the clock frequency of our familiar scalar system is $\frac{1}{t_0+t_1}$.

The clock period is shown as a sum of two components, t_0 and t_1. This sum appears in the numerator of the speedup expression, the denominator has the clock period of the superpipelined system. If stages were divide without overhead the superpipelined clock period would be $\frac{1}{n}(t_0+t_1)$ but instead it is $t_0+\frac{t_1}{n}$, indicating that one component of the critical path is not being divided. A likely candidate for that would be the setup time needed for the latches, something which is not split.

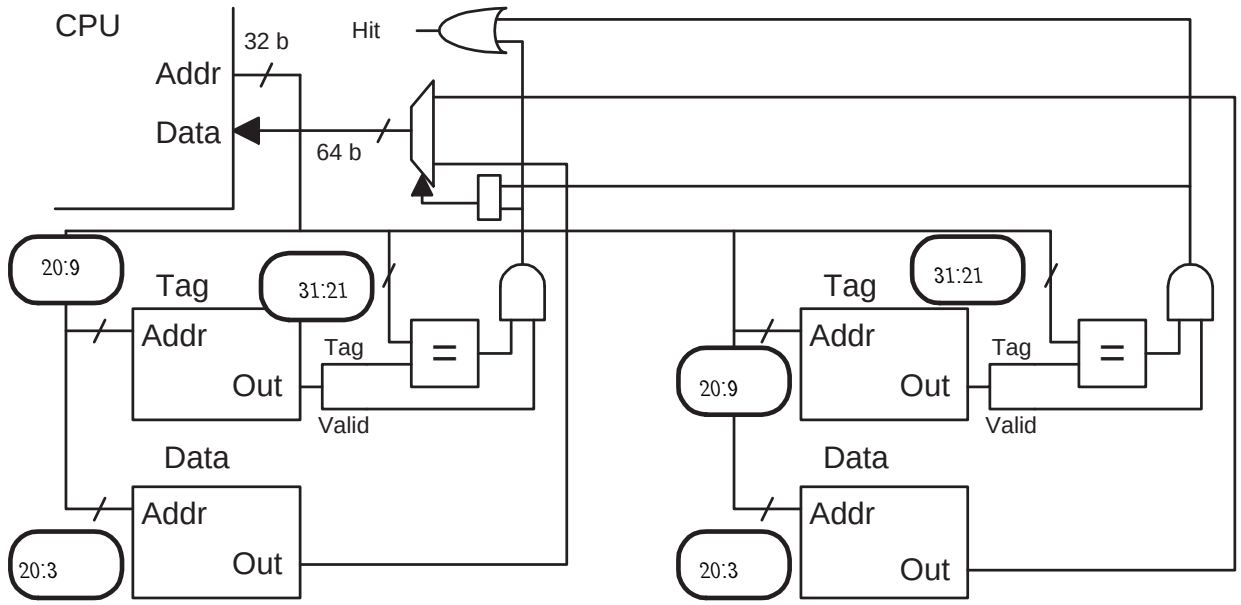
☑ Explain what t_0 is likely to be.

See solution above.

Problem 5: (20 pts) The diagram below is for a 4-MiB (2^{22}-character) set-associative cache with a line size of 512 (2^9) characters, with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

☑ Associativity:

Based on the number of ways in the diagram (there is no ellipses), the associativity is 2.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{22} characters, plus $2 \times 2^{21-9}\,(32 - 21 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Problem 5, continued:

(*b*) The code below runs on the cache from the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 2^9 characters is given, the size of an array element is $2^1 = 2$ characters, and so there are 2^8 elements per line. The first access, at `i=0`, will miss but bring in a line with 2^8 elements, the next $2^8 - 1$ accesses are to subsequent elements on the line and so will hit, so the hit ratio to the first line is $\frac{2^8-1}{2^8}$.

(*c*) The code below runs on a direct mapped (not set-associative) cache with a line size of 512 characters and a capacity of 4 MiB. *Grading note: The cache size was omitted from the original problem.*

☑ Determine an address for `b` that will result in a 0% hit ratio when running the code below.

☑ Briefly explain.

If the index bits of `a[i]` and `b[i]` are identical then `b[i]` will evict the line holding `a[i+1]` (except when `i` is 255, 511, etc.). The index bits are positions 22 to 9, so choose a `b` address in which those bits match `a`, the smallest possibility is `0x400000`.

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.

short *b =  0x400000;                              // <--- SOLUTION

int i;
int ILIMIT = 1 << 10;    // = 2^10

for (i=0; i<ILIMIT; i++) sum += a[ i ] + b[ i ];
```

Problem 6: (20 pts) Answer each question below.

(*a*) Describe restrictions (or lack of) on instruction addresses and the placement of instructions that distinguish RISC, CISC, and VLIW ISAs.

☑ RISC address and placement restrictions.

☑ Reason for each restriction (if any).

Instruction addresses must be a multiple of four, beyond that there are no placement restrictions. The address restriction simplifies instruction fetch since less shifting (or none at all) is needed to correctly position instructions. Also, the alignment restriction quadruples the reach of CTIs that hold less than a full address.

☑ CISC address and placement restrictions.

☑ Reason for each restriction (if any).

There are no restrictions.

☑ VLIW address and placement restrictions.

☑ Reason for each restriction (if any).

Instruction bundle addresses must be aligned. There might be placement restrictions within a bundle, for example, one might not be allowed to put a load instruction in the first slot.

The reason for the alignment restriction for VLIW is the same as RISC. Restrictions within a bundle can reduce the number of paths to functional units. For example, if the first slot cannot hold a load there is no need to have a connection from the slot-0 position in the memory stage to the memory port.

(*b*) A feature of the SPECcpu benchmarks is that they come with **source code** and it is the tester's responsibility to **compile** (and run) them. Note: "are these" in the questions below refers to source code and compiling.

☑ Are these necessary, important, or irrelevant for testing an implementation of a new ISA? (This is not a yes-or-no question.) Explain.

If source code is not provided then instead executable code must be provided. If the ISA is new then there is not likely to be an executable compiled for it (especially if the new ISA is still being developed!). So, source code is necessary.

☑ Are these necessary, important, or irrelevant for testing a new implementation of an existing ISA? Explain.

Source code is important, but not 100% necessary. Most new implementations will substantially improve the performance of code compiled for an older implementation, which is a good thing for those upgrading their computers. So lack of source code is not necessary. Source code will help show the full potential, and so it is important.

(*c*) A corrupt member of the committee choosing benchmarks for the next SPECcpu suite has successfully bribed the other committee members so that the benchmarks that were selected favor the corrupt member's company.

☑ How might this be discovered? Indicate who would discover the problem and what public information draw their suspicion?

Call the company providing the corrupt member company C, and let V be some other participating company that manufactures computers. Experts from V are aware of characteristics of different benchmarks, especially those which their machines handle well or poorly. They will become suspicious when none of the benchmarks that their machines run well are chosen, benchmarks that they knew were nominated for the suite. Perhaps their suspicions will be raised when their SPEC representative arrives at work up in a new luxury car.

☑ Can we expect those involved to be sufficiently motivated to correct the situation? Explain.

Buying decisions are influenced by benchmarks, so those from company V have a strong financial incentive to take action against their own bribed employee and see to it that the benchmarks are re-chosen.

60 Spring 2008 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 10 March 2008, 12:40–13:30 CDT

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(10 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Well ARMed

Good Luck!

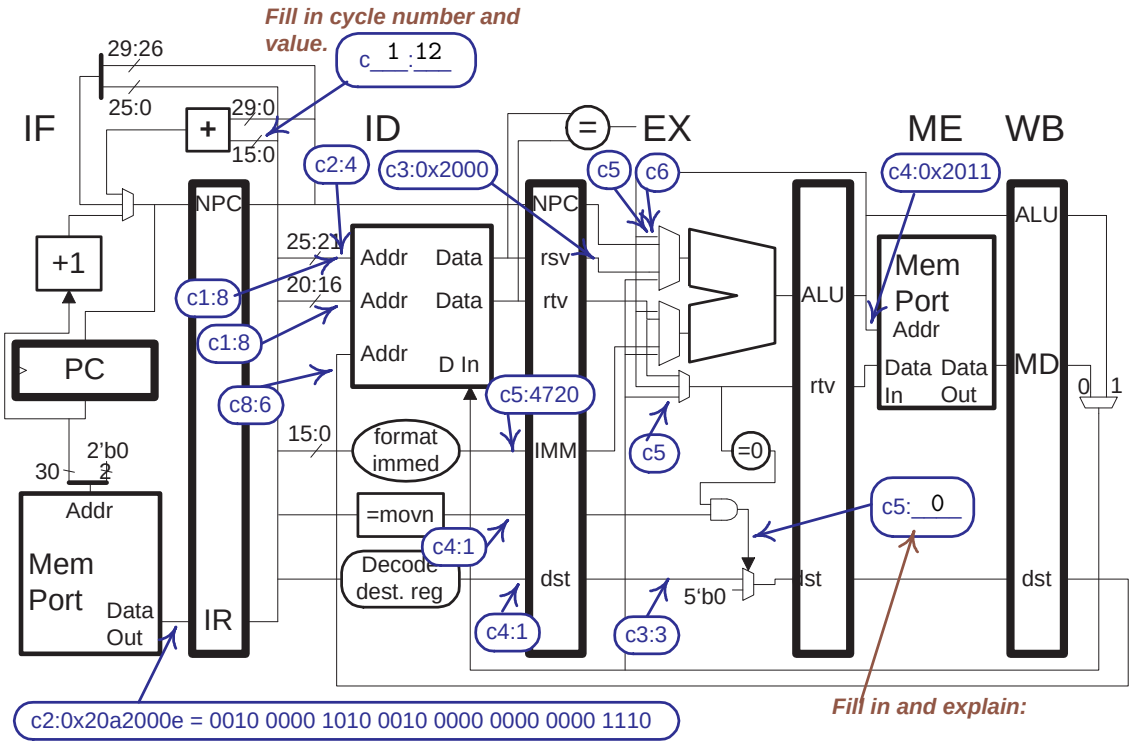
Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c2:4 indicates that at cycle 2 the wire will hold a 4. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that instruction addresses have been provided. [50 pts]

☑ Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Be sure to fill the two blocks marked *Fill In*.

☑ Provide an explanation for the EX-stage fill-in block.




```
# SOLUTION

#  Cycle                    0  1  2  3  4  5  6  7  8
0x1000 beq r8, r8, 0x1034   IF ID EX ME WB
0x1004 lb r3, 0x11(r4)         IF ID EX ME WB
0x1034 ADDi r2, r5, 14            IF ID EX ME WB
0x1038 movn r1, r2, r3               IF ID EX ME WB
0x103c ORi  r6, r1, 4720                IF ID EX ME WB
#  Cycle                    0  1  2  3  4  5  6  7  8
```

Solution discussion on next page.

Solution Discussion

0x1000: This has to be some kind of control transfer because of the break in consecutive addresses after the next instruction (the address of the third instruction is 0x1034 instead of 0x1008). The ID-stage fill-in points at the dedicated adder, which is used only for branches (not jumps or traps). Therefore this instruction must be a branch, the cycle number on the fill in must be 1 (when this instruction is in ID), and the value must be the displacement which is $\frac{\texttt{0x1034}-\texttt{0x1004}}{4} = 12$. (The displacement is the number of instructions to skip starting at the delay-slot instruction which in this case is at address 0x1004.) The c1:8 bubbles tell us the registers to use for comparison and also that it must be a **beq**. It can't be a **bne** because the branch is taken and it can't be a branch such as **bgtz** because those branches only read one register value.

0x1004: The c4:0x2011 tells us this is a memory operation that operates on byte-sized data (since the address is not a multiple of 4 or 2). The c3:3 tells us that it writes a destination register so it must be a load. The c2:4 reveals the source register and c3:3 in EX provides the destination. The displacement, 0x11, is determined using c3:0x2000 in EX and c4:0x2011 in ME.

0x1034: The bubble in IF provides the encoded form of this instruction. It can't be a type-J instruction because the last instruction follows the penultimate one, and so it is either type-R or type-I. Either way, the rs register is **r5** and the **rt** register is **r2**. If it were a type-R the rd register would be **r0** but since, as the c5 at the upper ALU mux indicates, the result is bypassed to another instruction and so it can't be type-R because **r0** would not be bypassed. Therefore it's a type-I. Unless one memorized MIPS opcodes there is no way to determine exactly which type I instruction it is so any arithmetic instruction would be considered correct. The opcode is for an **addi**.

0x1038: The lower c4:1 in ID provides the destination register, **r1**, and the one above that shows that this is a **movn** instruction. The ALU c5 provides the rs register number, **r2** from 0x1034, the rtv mux c5 provides the rt register number, **r3** from the 0x1004 **lb**.

If the c5: EX-stage fill-in is 0 then the **movn** instruction completes the move, otherwise the destination register remains unchanged. The instruction at 0x103c bypasses the **movn** destination, **r1**. This would only be correct if the **movn** was to write **r1** and therefore the fill-in value must be 0. (If the value were 1 the **movn** would not change **r1** but the instruction at 0x103c would read the value of **r2** rather than **r1**.)

0x103c: The c5:4720 in ID provides the immediate, indicating that this must be a type-I instruction. The c6 in EX gives the rs register, **r1** from the **movn**. The c8:6 gives the destination register, **r6**. This can be any arithmetic type I instruction, **ori** is assumed.

Problem 2: Answer the following questions about, or inspired by, ARM.

(*a*) [10 pts] MIPS lacks a counterpart to the ARM instruction shown below. *Hint: This has nothing to do with MIPS'* `movn`.

```
mov r1, #5   // Move the constant 5 to register r1.
```

☑ Explain how `r0` makes such a MIPS instruction unnecessary.

You can use an `addi` instruction with `r0` as the first source register.

☑ Show how to perform the same operation using MIPS instruction(s).

```
addi r1, r0, 5
```

(*b*) [5 pts] The ARM ISA states that the result of executing an instruction like `str r15, [r1]` is that either `PC+8` or `PC+12` is stored in memory, depending on the implementation. (Remember that ARM `r15` is an alias for `PC`.)

☑ What is the benefit of making the result of the store implementation dependent?

Some implementations would be lower cost because they wouldn't need to send a "correct" value of `PC` through the pipeline, instead they would use the `PC` value of whatever instruction was being fetched when the store reached `ME`.

(*c*) [5 pts] Consider an ISA which stated that the number of branch delay slots could be either zero or one, depending on the implementation.

☑ As an ISA feature, how does this delay-slot implementation dependence compare in practicality to ARM's store `PC` implementation dependent behavior?

The delay slot implementation dependence is a really bad idea because portable code (code that runs on any implementation of the ISA) could not put anything other than a NOP in delay slots. Since branches are frequent the impact on performance would be large (unless the code were compiled for a specific implementation). In contrast, the stored `PC` could always be added to a constant (-8 or -12) to obtain the correct `PC` value. That would take a little extra time but storing the `PC` in memory using a store instruction is not something programs need to do very often.

Problem 3: In RISC ISAs instructions are of fixed size, that is, all instructions are the same size. For example, in MIPS, all instructions are 32 bits. The character size in MIPS (and all ISAs mentioned in this test) is 8 bits.

(*a*) [5 pts] Describe a benefit of having fixed-size instructions. (This answer can be mentioned in the next answer's explanation.)

☑ Fixed-size instruction benefit for RISC.

Benefits: Instruction fetch is simple because the PC is incremented by a constant amount and instructions are fetched already properly aligned (because instruction addresses are required to be a multiple of four).

(*b*) [5 pts] In the Itanium VLIW ISA all instructions are 41 bits. Why would 41-bit instructions be difficult or wasteful to implement in a RISC ISA, such as MIPS, but cause no difficulties in VLIW ISA implementations, including Itanium implementations.

☑ Forty-one bit RISC instructions difficult or wasteful because:

An instruction would use six bytes of memory, that would either waste 7 bits or else require having the end of one instruction and the beginning of the next instruction together in one memory location. Branch targets would be harder to compute since one would have to multiply by 6 (and that's the space-wasting alternative). A multiply by six is just an add of two shifted values, but that's still one more trouble than just shifting a displacement.

☑ Forty-one bit VLIW instructions not wasteful and make sense because:

Fetch operates on 3-instruction bundles, which are 128 bits (16 bytes), and there is no way to jump to a middle of a bundle. There is no wasted space because $128 - 3 \times 41 = 5$ bits are used to store dependency information. It is easy to compute displacement, just shift left by 4 bits.

Problem 4: Answer the following compiler questions.

(*a*) [10 pts] A company is considering removing a bypass connection in a design of an implementation. Analysis on good test programs shows that performance drops by 5% with the bypass removed (and no other changes).

☑ What can compiler writers do about the performance drop?

They can re-do code scheduling so that the bypass is rarely used. For example, suppose the bypass is used when an `sll` uses the result of a preceding `lw`. The compiler might place two (or whatever) instructions between the `lw` and `sll`.

(*b*) [10 pts] Dead-code elimination is a commonly used compiler optimization, while profiling is used less often because it is a multi-step process.

☑ Using an example briefly describe dead-code elimination.

```
# Solution:
 x = a + b;   // The compiler would produce no code for this line ...
 x = c + d;   // ... because this line overwrites the first x before it's used.
```

☑ Briefly describe the steps used in profiling.

First, compile the program using a profile switch. Second, run the program on what is expected to be typical input data. The program will write profile information. Third, compile again, this time telling the compiler to read the profile information.

☑ Which is more likely to result in disappointing results that surprise the programmer? Explain.

Profiling is more likely to disappoint because the "typical input data" chosen for the profiling run may not match what most end-users use closely enough to get good profiling results. This might happen because there are many different inputs in common use and the compiler would optimize differently for different inputs so no profile run would result in good performance on all, or even most, inputs.

Name Solution

Computer Architecture
EE 4720
Final Examination

7 May 2008, 10:00–12:00 CDT

Problem 1		(10 pts)
Problem 2		(30 pts)
Problem 3		(20 pts)
Problem 4		(15 pts)
Problem 5		(15 pts)
Problem 6		(10 pts)
Exam Total		(100 pts)

Alias ISA were, was I?

Good Luck!

Problem 1: (10 pts) The tables below show the state of a dynamically scheduled system using method 3 during the execution of a code fragment.

- The code is preceded by many nop instructions.
- Instructions use either a 4-stage FP multiply unit (M1-M4) or a 2-stage FP add unit (A1, A2).
- Assume that there are unlimited RR, WF and execute (A,M) units.

(*a*) Show a program and pipeline execution diagram consistent with these tables.

☑ Show program and all stages used.

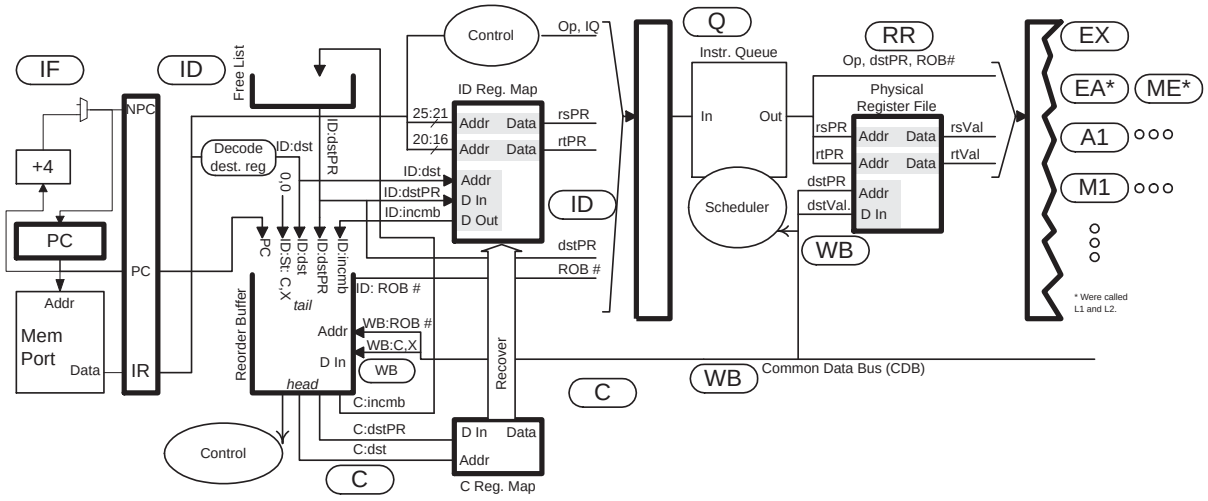
☑ On the physical register file show where physical registers are removed from the free list, using a `[`, and where they are put back in the free list, using a `]`.

☑ All destination registers can be determined exactly.

☑ The time for all stages can be determined exactly.

```
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
 add.s f20,f0,f1  IF   ID   Q    RR   A1   A2   WF   C
 mul.s f15, f20, f2    IF   ID   Q         RR   M1   M2   M3   M4   WF   C
 add.s f20, f3, f4          IF   ID   Q    RR   A1   A2   WF                  C
 add.s f20, f20,f5               IF   ID   Q         RR   A1   A2   WF             C
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14

ID Register Map
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
f15  9                    39
f20  16              43        82   93

Commit Register Map
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
f15  9                                                                  39
f20  16                                             43                       82   93

Physical Register File
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
9      11
16     22
39                                                                 33
43                                        44
82                                                  55
93                                                                 66
# Cycle           0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

Problem 1, continued: The implementation diagram is provided below for references. Don't forget to answer the question below.



(*b*) In the code fragment on the previous page only two true dependencies are possible.

☑ Show them (by connecting the destination and source with an arrow).

The only true dependencies are through `f20`. That is the second instruction depends on the first, and the fourth instruction depends on the third. One can conclude that the second depends on the first because it waits until the first instruction's result is bypassable, the same for the fourth and third.

☑ Briefly explain why other dependencies are not possible.

The third instruction cannot depend on the second because it starts before the result from the second is available.

Problem 2: (30 pts) Answer the following branch predictor questions.

(*a*) The code below runs on two branch predictors, a bimodal predictor with a 2^{10} entry BHT and a local predictor with a 2^{10} entry BHT and a 9-outcome history.

The outcomes of branch B1 form a repeating pattern as shown below.

The outcome pattern for branch B2 can be constructed by tossing a fair coin, adding a `N N N` for heads or a `T T T` for tails, then repeating. More precisely, outcome $3i$ is a Bernoulli random variable with $p = .5$ and outcomes $3i + 1$ and $3i + 2$ match outcome $3i$, for $i = 0, 1, 2, \ldots$. For example, the following is a possible pattern of outcomes for B2: `N N N T T T T T T`. The following is **not possible** for B2: `N N N T T N T T T`, it's not possible because the number of consecutive N's or T's must be a multiple of 3.

```
LOOP:

B1:      N  N  N  N  T  T  T  N  N  N  N  T  T  T  N  N  N  N  T  T  T...

B2:       N  N  N  T  T  T  T  T  T  N  N  N  T  T  T  N  N  N  (See text.)

j LOOP
nop
```

For the questions below assume all tables are initially zero. All accuracies are after warmup.

☑ What is the accuracy of the bimodal predictor on branch B1?

Accuracy is $\frac{3}{7}$.

☑ What is the accuracy of the local predictor on B1?

The accuracy is close to 100%.

The pattern of B1 outcomes repeats with a period of 7, which is shorter than the 9-element local history. Thee is possible interference with B2, however in part because B2 is random those interfering patterns will rarely occur frequently enough to affect B1's accuracy.

☑ What is the minimum local history size needed to predict B1 with 100% accuracy ignoring branch B2. Briefly explain.

Four outcomes.

At three outcomes pattern `NNN` can appear before an `N` or `T`.

☑ What is the approximate accuracy of the bimodal predictor on branch B2? Explain.

Accuracy is $\approx \frac{4}{6}$.

B2's outcomes come in groups of 3 and after each group the counter will be saturated at either 0 (for `N`) or 3 (for `T`). If the next group of outcomes is the same there will be 3 correct predictions, if the next group differs there will be 1 correct prediction. Since each case is equally likely the accuracy is $\frac{3+1}{6}$.

☑ What is the approximate accuracy of the local predictor on branch B2 ignoring B1? Explain.

Accuracy is $\approx \frac{5}{6}$.

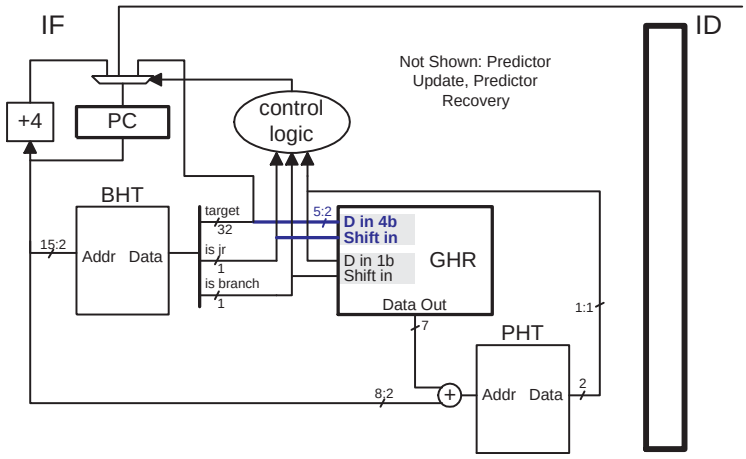
Consider the local history `NNNNNNNNT`, in which the last outcome is taken. Since the next outcome will always be taken the PHT entry, after warmup, will always provide the correct prediction. Lets call this a position 1 local history since the most recent branch is the first in a sequence of 3. All predictions made with position 1 local histories are predicted correctly after warmup, the same for position 2 local histories. For a position 3 local history, say, `NNNNNNTTT`, the accuracy will be 50% because the next outcome is random. Considering position 1, 2, and 3 histories the accuracy is $\frac{1+1+\frac{1}{2}}{3} = \frac{5}{6}$. Finally, consider local histories `NNNNNNNNN` and `TTTTTTTTT`. For these B2 can be at any position, if at 1 or 2 the outcome will be `N` for the first and `T` for the second, if at 3

the outcome is random. Therefore the PHT entries will saturate to 0 or 1, respectively and offer correct predictions $\frac{2}{3}$ of the time. These patterns occur 2^{-3} to 2^{-4} of the time, and so their impact will be small.

☑ What is the approximate warmup time for the local predictor on B2? *Note: This problem was alot more difficult to get perfectly correct than originally intended.*

A local history can span 4 groups of 3 outcomes (two complete, the oldest and the newest incomplete) or 3 groups of 3 outcomes. For the first case there are 2^4 possible outcome sets, each in two different positions, say TTTTTNNNT and TTTTNNNTT. Thus there are 2^5 local histories for this case. For the 3 groups of 3 case there are 2^3 possible local histories, for a total of $2^5 + 2^3$ local histories. Each must be seen twice to warm up. The tricky part is determining how long this will take.

(*b*) The predictor to the right is from the solution to last semester's final exam (and this semester's Homework 4). It is similar to gshare except in how the GHR is updated. Like global and gshare, when a branch is predicted the predicted outcome is shifted into the GHR, but when a `jr` is predicted four bits of the target are shifted into the GHR.



The code fragment below is the one used to justify the predictor, except that now the code is in a four-iteration loop. Important assembler code is shown in the comments.

```
for ( iter=0; iter<4; iter++ ) {
  int c = getchar(); // Unpredictable
  switch (c) {                    // jr $t0  # Jump to correct case.
  case 'a': x = 3; j++; break;    // addi $t1, $0, 3;  j ENDSWITCH;  addi $t2, $t2, 1
  case 'b': x = 7; break;
    ...
  case 'z': x = 1; i++; break;
  }                               // ENDSWITCH:
  if ( x < 5 ) foo(); else bar();
}
```

☑ Why might the prediction accuracy of the `for` loop branch be worse with the predictor above than an ordinary gshare?

The `for` loop branch is the one after the test `iter<4` (the branch itself is not shown). The `for` loop iterates four times, and so can be predicted perfectly with just 3 outcomes of the `for` loop branch in the GHRusing a global or gshare predictor. Even with the `if` branch present the GHR should still be large enough (though barely). However, if four bits of the `jr` target are shifted into the GHR there will not be enough room for three outcomes of the `for` branch and so accuracy will suffer.

☑ Considering the assembler code above, why might it make more sense to shift in the PC of the jump (`j`) instructions rather than the target of the `jr`? *Hint: This would only be useful when the case statements had branches.*

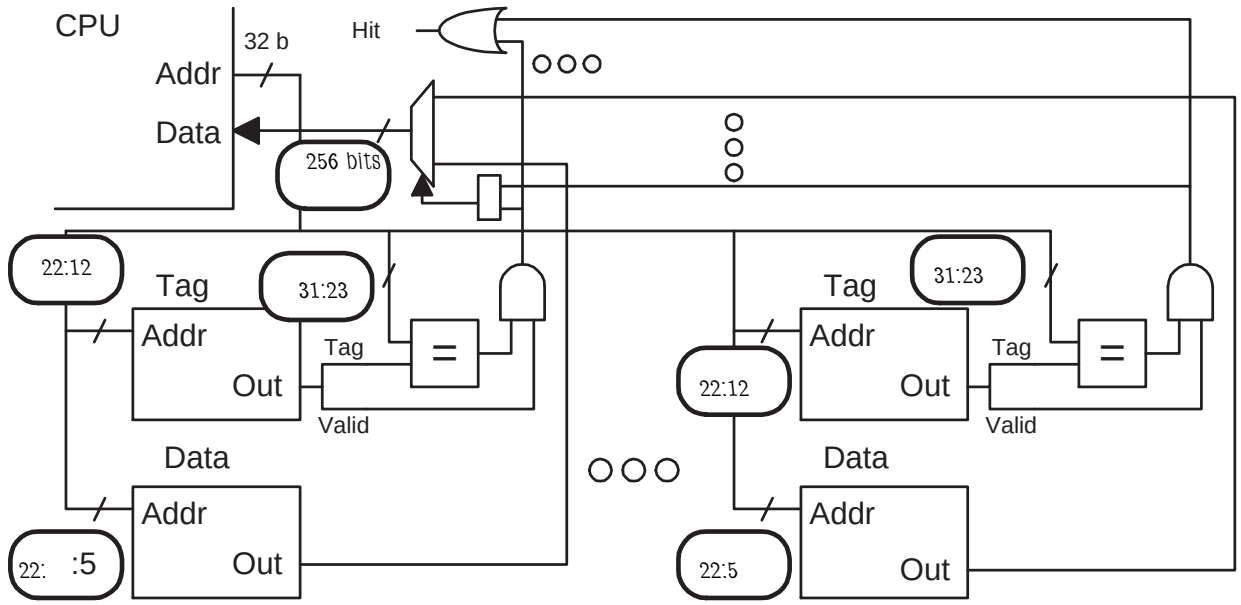
Shifting `jr` bits into the GHR would not help branches in the case statements, and might hurt them. It would not help them because the same bits would be shifted in every time (the address of the start of the case). It would hurt if the `jr` bits shifted in to the GHR pushed out branch outcomes that were needed. For example, suppose a branch in the case statement has the same outcome as the fourth previous branch (encountered before the switch was entered). Shifting in the `jr` bits would push out the outcome of that branch and so accuracy would suffer.

Shifting the PC of the jump instructions would help the prediction of the `x < 5` branch. Note that if the branch is reached via case `a` or `z` it is always not taken (assuming `foo` is on the not-taken path) but if the branch is reached via case `b` it is always taken. Suppose the PC of jump instructions (not the jump target) was shifted into the GHR, then when the `x < 5` branch is reached the value of the GHR would depend on the case statement and so for each case statement a different PHT entry would probably be used. The PHT entries for case `a` and `z` would saturate at zero and the entry for `b` would saturate at 3, resulting in perfect predictions.

Problem 3: (20 pts) The diagram below is for a 32-MiB (2^{25}-character) 4-way set-associative cache with a line size of 4096 (2^{12}) characters, with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.

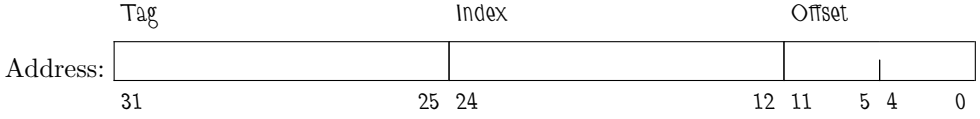


☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity, 2^{25} characters, plus $4 \times 2^{23-12}\,(32 - 23 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 3, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
long *a = 0x20000000;     // sizeof(long) = 8 characters.
int i,j;
int ILIMIT = 1 << 10;     // = 2^10

for (j=0; j<2; j++)
  for (i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 2^{12} characters is given, the size of an array element is $2^3 = 8$ characters, and so there are 2^9 elements per line. The first access, at `i=0`, will miss but bring in a line with 2^9 elements, the next $2^9 - 1$ accesses are to subsequent elements on the line and so will hit, so the hit ratio to the first line is $\frac{2^9-1}{2^9}$, and the same hit ratio will be achieved for all lines accessed with `j=0`. Since `ILIMIT * 8` is much smaller than the cache capacity every access on the second `j` iteration will hit. Therefore the overall hit ratio will be $\frac{1}{2}\frac{2^9-1}{2^9} + \frac{1}{2} = \frac{2^{10}-1}{2^{10}}$.

(*c*) The code below runs on a fully associative cache with 2^7-byte lines, **not the same as the previous cache**. Let h denote the hit ratio of the code below for a cache size of 8 MiB (2^{23} bytes). As always, assume the cache is empty before the code starts.

```
void p4(double *a, double *b) {
  // sizeof double = 8 characters.
  double sum = 0;
  int size = 1 << 8;

  for ( int d=0; d<size; d++ )
    for ( int col=0; col<size; col++ )
      sum +=  b[ col * size + d ]  *  a[ d * size + col ];
}
```

☑ What is the minimum cache size for which the hit ratio is h? Explain.

The `a` array is accessed sequentially and each element is accessed once and since this is a fully-associative cache it only need have two lines to keep `a`'s data safe from eviction. The `b` array is accessed at a stride of `size * 8` characters. Call the line brought in on the first miss to `b`, at `d=0` and `col=0`, `b[0]`. Because `size * sizeof(double)` is larger than the line size the next access to `b`, when `col=1`, will be on a different line. The `b[0]` line will not be accessed again until much later, when `d=1` and `col=0`.

With a large cache the `b[0]` line will remain during its long wait, but with a smaller cache it might be evicted. For the cache to be large enough it must be able to hold the `b[0]` line, plus all lines accessed until the `b[0]` is accessed a second time.

Recall that the line size is 2^7 bytes and `size` is 2^8. Since the access to `a` is sequential the number of lines due to `a` is $\frac{\texttt{size}\times\texttt{sizeof(double)}}{2^7} = \frac{2^8 2^3}{2^7} = 2^4$.

The number of lines due to `b` (including `b[0]`) is `size`, so the total number of lines needed is at least $\texttt{size} + 2^4 = 2^8 + 2^4$ which corresponds to a cache size of $2^7(2^8 + 2^4)$ characters.

Grading Note: No one came close to answering this.

Problem 4: (15 pts) Answer each question below.

(*a*) Describe what's wrong with each execution below.

☑ What's wrong with this execution on our usual bypassed 5-stage MIPS implementation.

```
# Cycle        0  1  2  3  4  5
lw r1, 0(r2)   IF ID EX ME WB
add r3, r1, r4    IF ID EX ME WB
```

Because the value for `r1` created by `lw` is not available until the end of the ME stage, in cycle 3, there is no way to bypass its value when needed, by the EX stage for the `add`, at the beginning of cycle 3.

☑ What are the two problems below with this execution on our usual scalar statically scheduled MIPS implementation?

```
add.d f0, f1, f2  IF ID A1 A2 A3 A4 ME WF
```

The register numbers for an `add.d` instruction must be even (it uses pairs of registers for 64-bit operands). Also, the FP pipeline does not have an ME stage.

☑ What's wrong with this execution on a 2-way superscalar dynamically scheduled machine?

```
# Cycle       0  1  2  3  4  5  6  7  8  9
add r1,r2,r3  IF ID Q  RR EX WB C
add r4,r1,r6  IF ID Q     RR EX WB C
add r7,r8,r9     IF ID Q  RR EX WB    C
add r8,r2,r1     IF ID Q     RR EX WB    C
# Cycle       0  1  2  3  4  5  6  7  8  9
```

Because the processor is 2-way superscalar, the commit in cycle 8 should occur in cycle 7, otherwise execution would be limited to 1 instruction per cycle. Note that the last `add` instruction waits one cycle because there are only two execute pipelines (meaning two RR, two EX, etc.)

(*b*) Alas, it is unlikely that there will soon be 16-way, statically scheduled, superscalar implementations that anyone would want to buy. Three reasons are started below, complete them.

☑ It would be too expensive because ...

The cost of the bypass paths would be much too high, proportional to 16^2. The output of each of the 16 ALUs would have to bypass to each of the 16×2 ALU inputs.

Note: The answer "because 16 copies of most parts are needed" is **wrong** because one should expect to pay 16 times as much as long as there is the potential to get 16 times the performance.

☑ The clock frequency would be too low because ...

To find dependencies, control logic would have to compare an instruction's source registers to up to 15 other instructions in ID, plus 16 instructions per downstream stage (two in a 5-stage design). That would put a strain on critical path.

Having 16 copies of everything plus the necessary bypass to avoid stalls would require substantially more area than, say, a 4-way processor. The increased physical distance would increase the propagation time of signals and so require a lower clock frequency.

☑ A few programs might realize the full potential of the processor, but most won't because ...

True dependencies within 16 instructions of each other could cause a stall, it would be hard for a compiler to avoid those with scheduling.

Problem 5: (15 pts) Answer each question below.

(*a*) In ARM the PC is a general purpose register, `r15`. It could have been defined in ways consistent with ISA design principles, but it wasn't.

☑ Describe how the use of `r15` appears contrary to the usual goals of an ISA.

The value written when storing `r15` or using it in a calculation depends upon the implementation. The point of separating an ISA from an implementation is to avoid making the program dependent upon details of an implementation (thus loosing portability).

(*b*) The two code fragments below are supposed to do the same thing, the first is MIPS the second is SPARC. In both a branch is taken if a less-than comparison is true. The code would work correctly if the called subroutine returned immediately.

☑ Explain why the SPARC code may not execute as intended.

The SPARC branch tests a condition code. It's possible that an instruction in `some_subroutine` has overwritten the condition codes to something else.

☑ Suggest a fix.

Move the `subcc` after the call (or re-execute it).

```
# MIPS Code
slt $s1, $s2, $s3       # Registers %s0-%s7 preserved.
jal some_subroutine
nop
bneq $s1, $0 SKIP1
...

! SPARC code
subcc %l2, %l3, %l1     ! Registers %l0-%l7 preserved.
call some_subroutine
nop
blt SKIP1
...
```

Hint: The following two questions are really asking about exceptions.

(*c*) A compiler writer is wondering whether dead-code elimination should remove the first assignment to x in the code fragment below (which sure looks dead, right?).

```
x = a / b;
x = a + c;
```

The guy down the hall wrote a program that included these two lines and that program would run differently if dead-code elimination removed the first assignment (the one with the division). The difference has nothing to do with timing or the size of the program.

☑ Why might the program have run differently?

The program might have used a signal handler to be run on a division by zero exception.

☑ The compiler writer wants to DTRT (do the right thing), how does he or she find out what the right thing is?

It's possible to argue that a signal handler can be useful. A stronger argument is that dead-code elimination is even more useful and that most similar needs for a signal handler could be met by tricking the compiler into believing the code isn't dead. (Say, by using it in an if statement.)

However, the compiler must generate code that runs correctly *as defined by the language specification*. The writer then must look at the language specification to see what the correct behavior should be. If the specification says nothing then see what most other compilers do.

Answers which did not mention some kind of a language specification or common behavior were considered wrong.

(*d*) The code fragment below may have come from the code in the previous problem.

```
div.d f0, f2, f4
add.d f0, f2, f6
```

Designers of an implementation are considering whether to avoid WAW hazards like the one above by converting the `div.d` to a nop when the `add.d` reaches `ID`. This will work correctly under ordinary circumstances.

☑ Show a pipeline execution diagram illustrating the WAW hazard.

```
# SOLUTION
# Cycle           0  1  2  3  4  5  6  7  8  9
div.d f0, f2, f4  IF ID DI DI DI DI DI DI DI WF
add.d f0, f2, f6     IF ID A1 A2 A3 A4 WF
```

☑ Under what circumstances will this produce the wrong answer?

If the `add.d` raises an exception. In that case the signal handler won't see the value written by the `div.d` because it was converted to a `nop`.

Problem 6: (10 pts) SPECcpu provides a separate set of training inputs to be used for feedback-directed optimization (FDO) techniques, such as profiling.

☑ Why is a separate training input set needed?

Using the same input set to train and run will produce the best possible results, but it's not something that can be used in real life because input data changes with each run and there is usually no way to determine it in advance.

Profiling using the training set provides a better indication of the benefits of profiling than the inflated numbers obtained by using the same data to profile and run.

☑ Why might an honest and good tester want to substitute a different training set?

There may be newer techniques for choosing good training sets, techniques that anyone can use. The SPEC training sets however used older less-effective training set selection techniques. The tester would like the results to reflect the benefits of his new and perfectly legitimate training selection methods.

The run and reporting rules don't allow such a substitution.

☑ Why do you think the run and reporting rules don't allow a training set substitution when they allow so much flexibility on compilers, optimizations, and libraries?

The rule would have to say that the training set can't be chosen based on an exact knowledge of the reference (data-collection run) inputs. The problem is determining whether a tester's training data violates this rule.

61 Fall 2007 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 8 November 2007, 10:40–11:30 CST

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Exam Total		(100 pts)

Alias 630 bugs to EE3

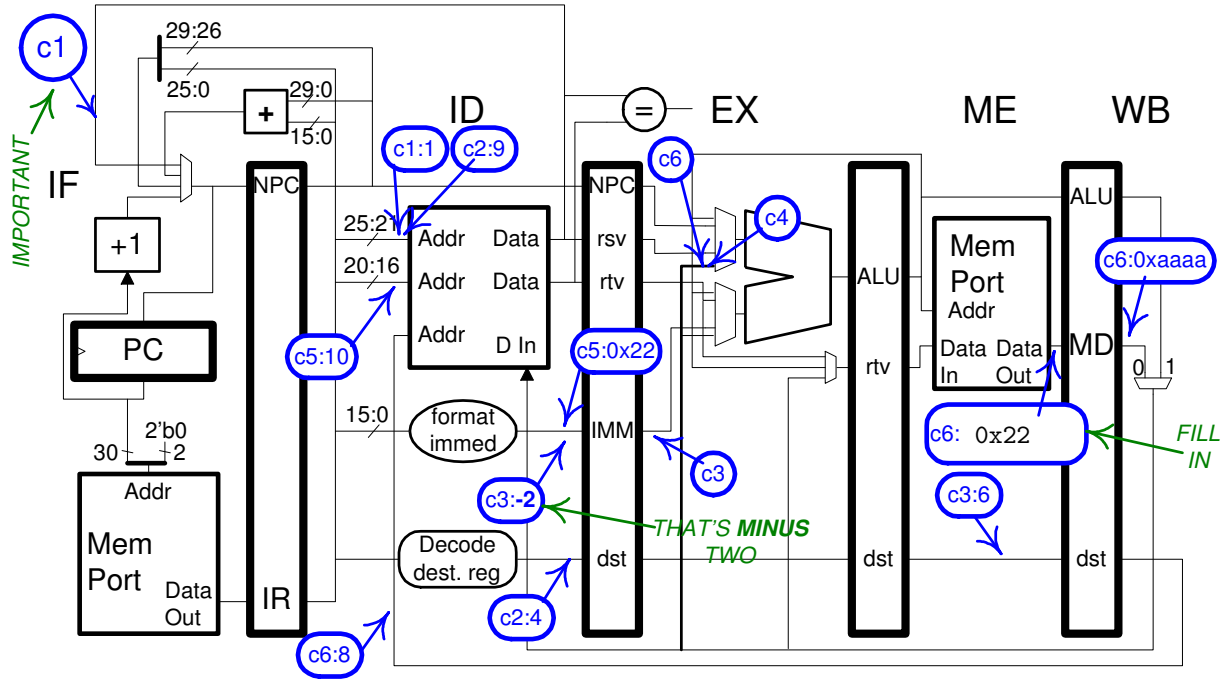
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that the fourth instruction has been provided. [50 pts]

☑ Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Be sure to fill the block marked *Fill In*.




```
# SOLUTION

#  Cycle                        0  1  2  3  4  5  6  7  8
0x1000 jalr r6, r1              IF ID EX ME WB
0x1004 ADDi r4, r9, 0xaaaa         IF ID EX ME WB
0x2000 lhu r8,-2(r6)                  IF ID EX ME WB
0x2004 lb r3, 0xB(r1)                    IF ID EX ME WB
0x2008 sW r10, 0x22(r8)                     IF ID EX ME WB
#  Cycle                        0  1  2  3  4  5  6  7  8
```

Uppercase is used to indicate parts of an instruction that could vary. For example, the `ADDi` could have been a `SUBi` but it could not have been an `ADD`.

To determine the value for the fill-in block, `0x22`, one has to know that `r1` holds the target address of the `jalr`. To determine the immediate for the second instruction, `ADDi`, one has to know that `jalr` will write `r6` with the return address.

The last instruction has to be a store because no other instruction (extensively covered in class) uses an immediate, uses an rt register value, and uses the EX stage.

Problem 2: The VAX `locc` instruction (from Homework 3) appears below, along with its encoding (taken from the Homework 3 solution).[20 pts]

```
 locc #65, r2, (r3)

Instruction Encoding:

 -opcode-    -- 1st operand ----     -- 2nd op -    -- 3rd op -
  locc        imm   PC*    immed      reg   r2       reg-d r3
              mode         value      mode           mode
  0x3a        0x8   0xf    0x41       0x5   0x2      0x6   0x3   <- Encoded value.
 7     0     7   4 3   0   7   0     7   4 3   0    7   4 3   0  <- Bit position.
```

(*a*) A MIPS implementation can easily retrieve its two source register values in one clock cycle.

☑ What about the VAX instruction formats makes one-cycle source register value retrieval more difficult in a VAX implementation (without lowering clock frequency)? Consider instructions with at most two source register operands.

In MIPS formats the source register fields are always in the same place so the corresponding instruction register bits can be hard wired to the register file read ports. In VAX the position varies. In the example above the second source register field is in the fourth byte, but if the first operand used register addressing the second source register field would be in the third byte. Therefore one has to partially decode the instruction before one can determine where the register bits are, and so both the decode and the path through the multiplexer add to the cycle time.

(*b*) The constant 65 (`0x41`) is encoded in immediate mode, taking a total of two bytes, rather than literal mode, which would take only one byte if 65 were small enough for literal mode.

☑ Given the way VAX encodes operands one might expect the maximum literal size to be only four bits. Why?

Because the mode field is four bits, so there is only four bits remaining. (The four remaining bits are normally used for the register number.)

☑ In fact, the maximum literal size is six bits. How is that accomplished? (If you don't know or remember the details then make up something reasonable.)

Literal mode is specified by setting the first two mode bits to zero. This consumes four modes but provides for a six-bit literal.

Problem 3: Under SPECcpu2006 base rules at most four compiler optimization switches can be used per language. [15 pts]

☑ What is the rationale for that restriction?

The base scores are supposed to reflect normal effort. It's too vague to stipulate something like *valid compiler optimization switches shall be those that can be determined using normal effort*. With such a rule there would be endless arguments over whether the effort to determine a particular set of switches was normal or not. The four-switch rule, though arbitrary, is specific.

Suppose a tester would like to use five options:

```
cc -O4 --optimization-a --optimization-b --optimization-c --optimization-d
```

The tester modifies the compiler by combining options `a` and `b`, the modified compiler is made available as a product. Now compilation can be done consistent with the rules:

```
cc -O4 --optimization-ab --optimization-c --optimization-d
```

☑ Should that be considered cheating? Explain why or why not.

It should be considered cheating because it renders the four-switch rule irrelevant. The impetus for combining the switches was to get performance on SPECcpu benchmarks, so it's probably not the kind of switch combination that would make sense for other uses otherwise it would already be present.

Problem 4: Answer each question below.[15 pts]

(*a*) In MIPS the branch instruction uses a 16-bit immediate to specify a displacement target, and the `jal` instruction uses a 26-bit immediate. Why does it make sense to choose a coding for `jal` that has a larger immediate?

Branches are typically used for loops and if/else constructs. In both cases the target will be within a procedure. The `jal` instruction is usually used for a procedure call in which case the target will be in another procedure. Therefore `jal` jumps much further and so a coding with a larger immediate makes sense.

Grading Note: The question originally asked *Why does it make sense for* `jal` *to have a larger immediate?* The answer above would be considered correct for that question. The following answer is correct for the original question but not the one appearing further above: Branch instructions can specify as many as two registers for a condition, so there is less space for an immediate.

(*b*) An ISA should be designed to support decades of implementations but invariably ISA designers are biased towards a first implementation at the expense of later ones. A branch delay slot (as present in MIPS and SPARC, for example) is a good example of such a phenomenon.

☑ Explain why the branch delay slot is a good example of a shortsighted ISA feature.

In the classic five-stage MIPS implementation a branch target and direction can be determined (resolved) in the cycle before they are needed, this perfect timing is due to the delay slot. In superscalar implementations and those that are more deeply pipelined the branch would be resolved after the target was needed and so the delay slot provides less of a benefit, while complicating the design. That is, with a classic five-stage implementation there is no branch penalty because of a delay slot. In a superscalar or more deeply pipelined implementation there is a branch penalty (the squashed wrong-path instructions) despite the delay slot, and because of the delay slot control logic is more complex.

Name Solution

Computer Architecture
EE 4720
Final Examination

11 December 2007, 15:00–17:00 CST

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias ISA were, was I?

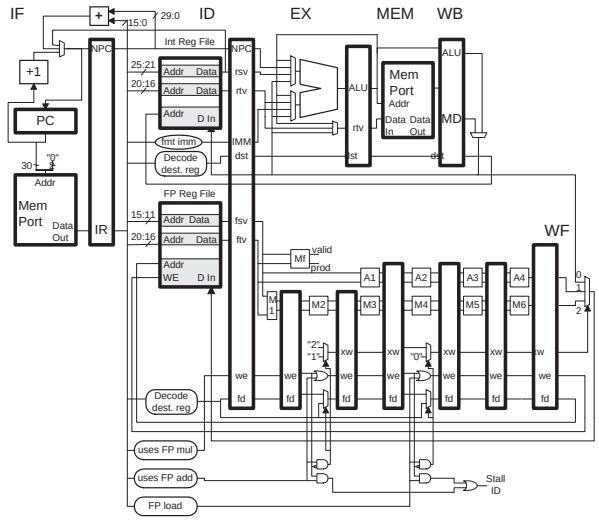
Good Luck!

Problem 1: (25 pts) Suppose it turns out that many `mul.d` instructions are executed with a 0 or 1 for one of the operands, in that case the product could be computed in less than the six stages used in the class implementation. The MIPS implementation below includes a new *multiply fast unit*, `Mf`, for such situations.

`Mf` has two inputs (unlabeled) and two outputs, `valid` and `prod` (they are not yet connected to anything). As with `A1` and `M1`, IEEE 754 doubles are expected at the inputs of `Mf`. If one of the inputs is 0 or 1 then the `valid` output will be `1` (otherwise it is `0`). If `valid` is `1` the `prod` output is the product of the two inputs; both outputs are available by the end of the cycle.

Call a multiply *fast* if one of its operands is 0 or 1.

USE NEXT PAGE FOR SOLUTION!



USE NEXT PAGE FOR SOLUTION!

(*a*) Add datapath and control logic for `Mf` meeting the requirements below:

☑ Add datapath so the `Mf` output will be written to the FP register file at the right time.

Datapath shown in blue on diagram on next page. The solution has the fast-multiply product pass through an extra stage so that `WF` is fifth stage, to avoid the special case of a fast multiply being the only four-stage instruction (making it hard to squash in certain circumstances).

☑ In order of decreasing priority: the added datapath must be correct, must not increase critical paths, and should use as little new hardware as possible.

The datapath does not add multiplexors at the inputs or outputs of any functional unit (such as `A4`), so it does not strain critical path. If hardware savings were more important than critical path the fast multiply product might have been muxed in after `A3` or `A4`.

☑ The control logic must detect and handle the new `WF` structural hazard with preceding instructions that's possible when writing a fast product.

The control logic appears in green on the next page. The logic labelled Control logic to insert fast multiply in the diagram detects and handles the new structural hazard. If one exists the fast multiply path is not taken. (The alternative would be to stall.)

☑ If `Mf` is used then the `mul.d`'s usual `WF` slot should be available for other instructions. For example, suppose the second instruction after `mul.d` is an `add.d`. If the multiply is normal the `add.d` would stall, if the multiply is fast the `add.d` shouldn't stall (at least for the `WF` conflict with `mul.d`).

The logic labelled Control to stop mult at M2 makes the multiply's `WF` slot available to other instructions by setting `we` to `0` if the fast multiply will use `WF`.

(*b*) Add bypass hardware and control logic so that the code below executes without a stall if the `mul.d` turns out to be fast. For this part assume that multiplexors at FP unit inputs won't add to critical path.

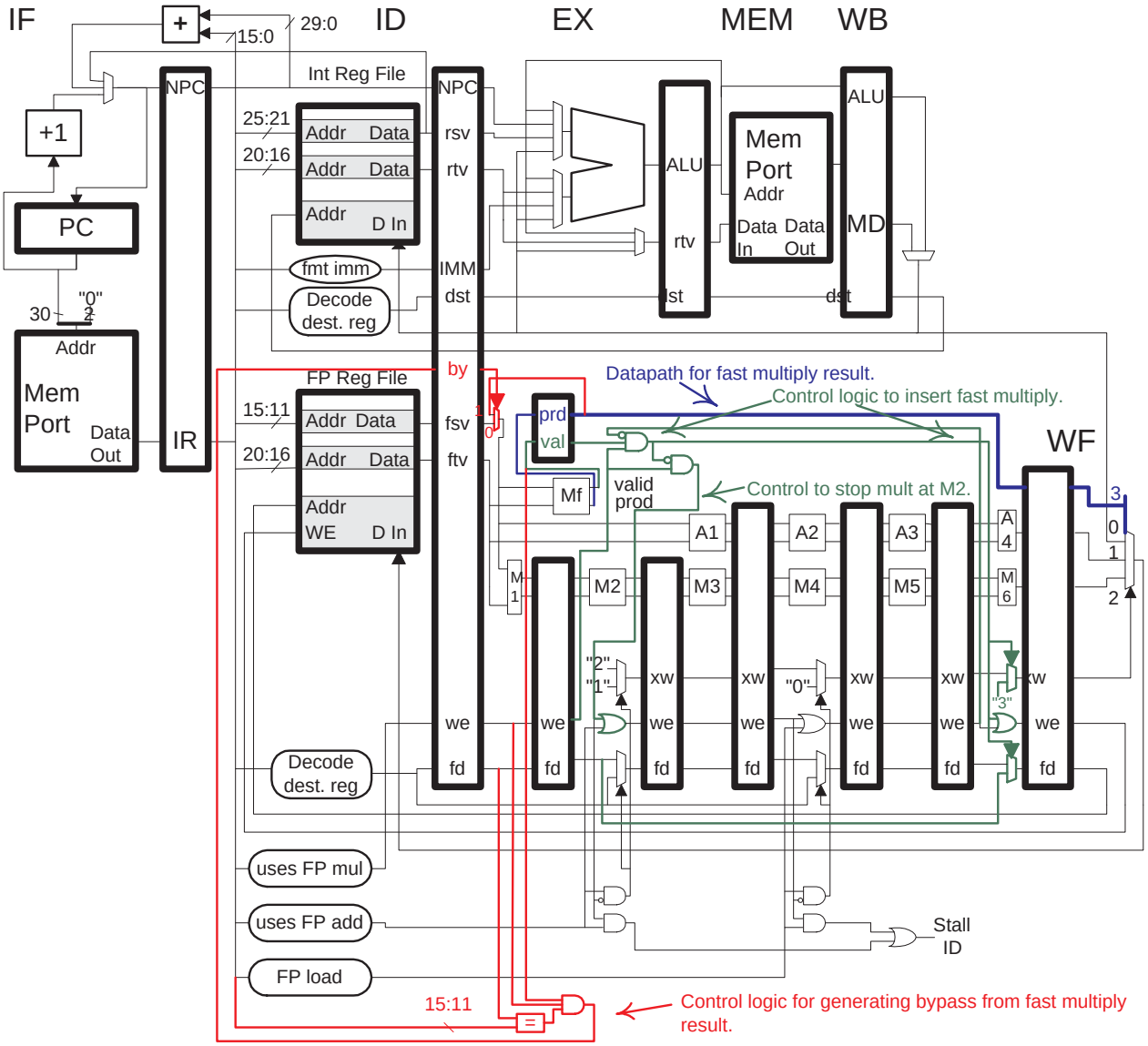
```
# Cycle              0  1  2  3  4  5  6  7
mul.d f2, f4, f6     IF ID MF Mx WF
add.d f8, f2, f10       IF ID A1 A2 A3 A4 WF
```

☑ Bypass hardware for case above.

The bypass logic appears in red. See the pipeline diagram above for stage abbreviations. Note that the bypass logic is in `Mf` because the `add.d` must make a stall decision when it is in `ID` and so it can't be done any later.

The logic for selecting the `WF` stage is computed when the `mul.d` is in `Mx` because that logic is more elaborate and would add to the critical path if it were in `Mf` (there is less time pressure in `Mx`).

Note that the `add.d` can bypass a fast multiply result even if the fast multiply won't write back.



Problem 2: (25 pts) Answer the following predictor questions.

(*a*) The code below executes on three systems, one uses a bimodal predictor with a 2^{14}-entry BHT, one uses a local predictor with a 2^{14}-entry BHT and a 7-outcome local history, and one uses a global predictor with a 7-outcome global history. Consider the execution of the code below, all branches are shown.

```
BIGLOOP:
...
B1: beq r1, r2, SKIP1   T T T T N T T T T N T T T T N T T T T N ...
...
SKIP1:
...
B2: bne r3,r4 SKIP2      T T T N N N T T T N N N T T T N N N T T T N N N ...
...
SKIP2:
...
j BIGLOOP
nop
```

For partial credit show work, not just answer.

☑ Accuracy of bimodal predictor on B1 after warmup:

Accuracy is $\frac{4}{5} = 80\%$.

☑ Accuracy of bimodal predictor on B2 after warmup:

Accuracy is $\frac{2}{6} \approx 33.3\%$.

☑ Accuracy of local predictor on B2 after warmup:

Accuracy is 100%

☑ Warmup time of local predictor on B2:

Warmup time is $6 + 6 \times 2 = 19$ executions of B2.

☑ Minimum local history size for 100% accuracy of local predictor **on B2** (without ignoring B1) (show work):

Minimum size is 5 outcomes. Four outcomes are not enough because pattern `TTTN` would occur in both B1 and B2 and the next outcomes would disagree: `T` for B1 and `N` for B2.

☑ Accuracy of global predictor on B2 after warmup:

Accuracy is $\frac{30-2}{30} \approx 93.3\%$. There are $5 \times 6 = 30$ possible GHR values (patterns). Twenty four of these will only occur when predicting B1 or predicting B2 (but not both). For example, pattern `tTnTnNn` can only occur when predicting B1 (because B1 never has 3 (or 2) consecutive `n` outcomes). (The upper-case letters are the outcomes of the branch being predicted, the lower-case letters are outcomes of the other branch.) Those 24 will be followed by correct predictions. There are two shared patterns in which the outcomes agree by happy coincidence: `tNtTtTn` and `nNtTtTt`. Pattern `tTtTtTn` is used twice by B1 with differing outcomes so the impact on the PHT entry cancels out, it is also used once by B2 and so the entry reaches a value suitable for B2, 0; the same with pattern `nTtTtTt`. For all patterns discussed so far correct predictions are made for B2. There are two in which B2 is incorrectly predicted: `tTtTtNn` and `nTtTtTt`. It is used once per unit by both B1 and B2, since B1 is more frequent the PHT entry reflects B1's next outcome: `T`. Summing it up, correct predictions will be made for B2 on 28 out of 30 patterns.

☑ Warmup time of global predictor on B2 (explain):

Warmup time is $6 + 5 \times 6 \times 2 = 66$ executions of B2. Based on number of distinct GHR patterns.

Problem 2, continued: Consider the execution of the code fragment below on a system using branch prediction. The value of `c` used in the switch statement is random, uniformly distributed over `a` to `z`, and outcomes are independent (like a 26-sided die). The branch implementing the `if ( x < 5 )` statement, which will be called the *if* branch, is taken 50% of the time.

```
int c = getchar(); // Unpredictable
switch (c) {
 case 'a': x = 3; j++; break;
 case 'b': x = 7; break;
 ...
 case 'z': x = 1; i++; break;
}
if ( x < 5 ) foo(); else bar();
```

As shown below, the `switch` construct is implemented using a dispatch table and a `jr`, and other jumps. The `switch` construct itself and the case blocks use no branches.

```
 # Part of code implementing switch construct.
 # Value in register $t1 has been computed using variable c.
 lw $t0, 0($t1)   # Load the address of the case statement corresponding to c.
 jr $t0           # Jump to case statement.
 nop
# Case 'a'
 addi $t5, $0, 3  # x = 3;
 j endswitch
 addi $t7, $t7, 1 # j++
...
```

(*b*) The predictors covered in class would all achieve just a 50% prediction accuracy on the *if* branch. Explain why they might do better if the case statements contained branches, but the values assigned to `x` are the same as the example above and the if branch is the same. *Note: The original exam did not have this part.*

☑ Why predictors might do better on *if* branch when cases have branches.

They might be able to tell which case statement was executed by the pattern of recent branch outcomes.

(*c*) Modify one of the predictors used in class so that it does better than 50% on the if branch. The predictor should also work well on similar switch statements and on predict well on code not having switch statements. Don't design a predictor for exactly the code above. For example, the predictor shouldn't somehow find or guess the value of `x`. *Hint: Think about the answer to the previous part. A correct solution requires just a small modification of one of the predictors shown in class.*

☑ Briefly explain the idea behind your predictor.

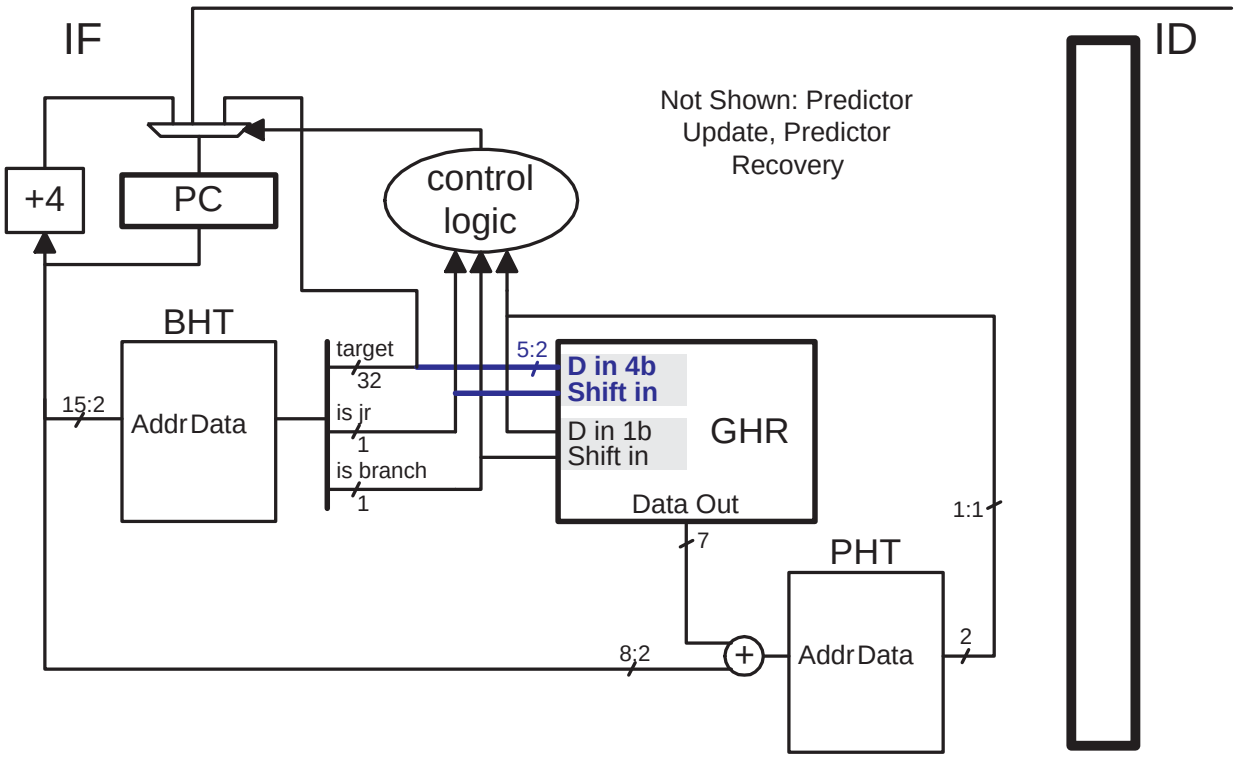
If the predictor could determine which case statement was executed then it could better, or maybe perfectly, predict the branch. A global or gshare predictor "knows" the outcome of prior branches (they are in the GHR) and uses that to make a prediction. Here the predictor needs to know which case statement was executed, or equivalently what the target of the `jr` instruction was. So a predictor for the code above would shift `jr` target addresses in the GHR, just as branch outcomes are shifted into the GHR. The entire target address would be too large, but the lower order bits, say 4 of them, might do.

☑ Draw a diagram of the predictor.

☑ Show tables such as BHT and PHT.

The diagram below is a gshare predictor modified so that it would accurately predict the branch. The material in blue shows changes specific to this problem. The size of the BHT and the size of the local history match the first part of this problem.

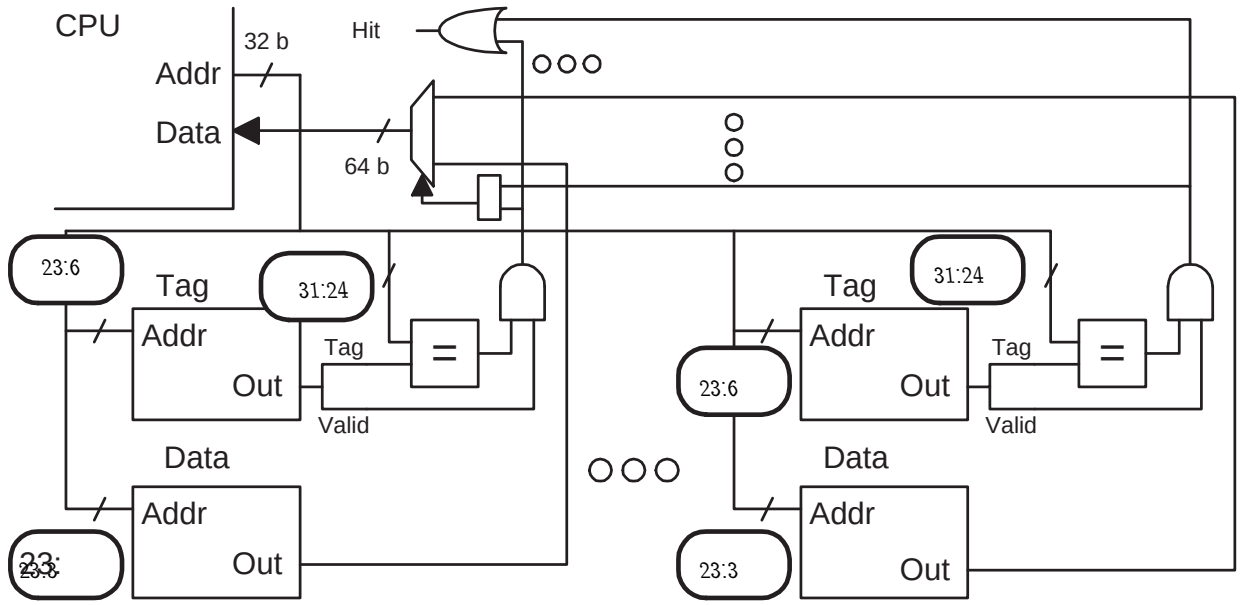
In a more realistic predictor the BHT would just store a branch displacement (16 bits) and a separate jump target buffer (JTB) would hold 30-bit addresses (or maybe just the other 14 bits) for jumps.



Problem 3: (20 pts) The diagram below is for a 256-MiB (2^{28}-character) set-associative cache with a line size of 64 characters on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

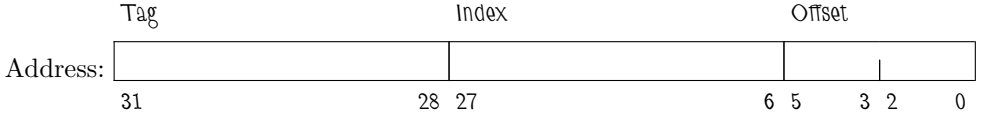
☑ Associativity:

Based on the high bit position used for the data store, 23, the data store size is 2^{24} characters. Since the total cache capacity is 2^{28} characters the cache must be $\boxed{\frac{2^{28}}{2^{24}} = 16\text{-way set associative}}$.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $16 \times 2^{24-6}\,(32 - 24 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 3, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
long *a = 0x2000000;    // sizeof(long) = 8 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 64 characters is given, the size of an array element is 8 characters. The miss on the first iteration will bring in a line which is 64 characters or 8 **long**s, the first **long** will be accessed. The second iteration will access the second **long** on this line. The line will be "used up" at $i = \frac{64}{8} = 8$, and so for each miss there are 7 hits. The hit ratio is $\frac{7}{8}$.

(*c*) Consider a 1 MiB (2^{20} byte) *direct mapped cache* with a line size of 256 characters (not the same as the one from parts a and b) for a system with a 32-bit address space. Suppose this cache has the following defect: a particular bit position in the tag comparison unit will match even if the tags differ. That is, if the bad bit position were 2 then tags `0x5` and `0x1` would match. Other cache hardware functions correctly. The cache is write through and write around.

☑ Complete the program below so that it finds the bad bit position in a small amount of time and assigns it to `badbit`.

☑ For maximum partial credit briefly describe your strategy.

- The cache is empty when the program starts.
- Assume that any address can be read or written.

```
char *a = 0x1000;
int bad_bit = -1;  // At end should be set to position of bad tag bit.


// SOLUTION
unsigned int i;

// Initialize elements to zero.
// Since cache is write-around these initialized elements won't be cached.
// Tag values are: 1, 2, 4, ..., 4096
//
for ( i=20; i<32; i++ ) a[ 1 << i ] = 0;

// Initialize this element to 1.
// Tag value is 0.
a[0] = 4720;
int dummy = a[0]; // Make sure a[0] is in cache.

// Each element fetch should miss the cache and find a zero. If
// the tag comparison is wrong then it will hit and find the 4720.
for ( i=20; i<32; i++ )
 if ( a[ 1 << i ] == 4720 ) break;

if ( i != 32 ) bad_bit = i - 20;
```

Problem 4: Answer each question below.

(*a*) (6 pts) A trap instruction is sort of like a procedure call (*e.g.*, `jal`) to the operating system.

☑ Describe a difference between a trap and `jal` in how the target address is specified.

A `jal` specifies the low bits of the target address. A trap does not specify any address at all, the target address is a particular entry in the trap table, and the trap table address is defined by the MIPS ISA and built into the hardware.

☑ Describe another important difference between a trap and a `jal`.

A trap switches the processor to privileged mode.

(*b*) (6 pts) A `log` (logarithm) instruction is to be added to an ISA. Group E wants to define the `log` instruction as producing the IEEE 754 double representation that is closest to the exact result. Group A would define `log` as producing any result within a certain number of bits of the exact result. Group A argues that the precision of an exact result is not needed and their approximate result is sufficient. *Group E agrees with this,* they want an exact result for other reasons. *Hint: Think about the reasons for separating ISA and implementation.*

☑ Why might group A want an approximate result?

It can be computed faster than an exact result.

☑ Why might group E want an exact result?

Code would run exactly the same way on different implementations. (Note that the A-group definition of `log` would allow to implementations to produce different results, so long as they were close enough to an exact result.)

(*c*) (6 pts) Consider two scalar MIPS implementations, implementation A is similar to the one covered in class with the familiar stages `IF ID EX ME WB` while implementation B has stages `IF I1 I2 I3 I4 EX ME WB`. The two implementations run at the same clock frequency and are similar in other ways.

☑ Explain why implementation A does not need a branch predictor, or at best would only gain a small amount of performance.

Because the branch target and direction are available in time for `IF`, so long as there is no unbypassable dependence. Consider the code execution below which is on a processor with an aggressive EX-to-ID bypass for the branch condition. The branch has the direction and target ready in cycle 2, which is in time for the fetch of the target which starts in cycle 3. There is no need for a branch prediction. If the branch condition were from a load instruction (or if the aggressive bypass were not possible) the branch would have to stall and so branch prediction would help, but only a little.

```
 # SOLUTION example
 # Cycle            0  1  2  3  4  5  6  7
 sub r1, r2, r3     IF ID EX ME WB
 bneq r1,r5  TARG      IF ID EX ME WB
 xor r9, r10, r11         IF ID EX ME WB
TARG:
 add r12,r13,r14             IF ID EX ME WB
```

☑ Explain why a branch predictor would help implementation B much more than implementation A.

Because the branch would not be resolved until after several instructions past the delay slot instruction were fetched, these instructions would have to be squashed if the prediction were wrong. In the example below three instructions, `or` and two others not shown, are squashed because the branch is mispredicted not-taken.

```
 # SOLUTION example
 # Cycle            0  1  2  3  4  5  6  7
 sub r1, r2, r3     IF I1 I2 I3 I4 EX ME WB
 bneq r4,r5  TARG      IF I1 I2 I3 I4 EX ME WB
 xor r9, r10, r11         IF I1 I2 I3 I4 EX ME WB
 or                          IF I1 I2x
TARG:
 add r12,r13,r14             IF I1 I2 I3 I4 EX ME WB
```

☑ Consider the performance of implementation A and implementation B when branch prediction is perfect for both. Which (if any) is faster, and by how much? Explain, state any assumptions made.

They would be the same performance because the only other events that could slow execution, dependence stalls, would be just as frequent and just as long.

(*d*) (6 pts) The code below executes on a two-way superscalar dynamically scheduled machine similar to the one presented in class. The `sub` instruction reads the value of `r1` from the register file in cycle 10, `xori` writes a value for `r1` in cycle 6 and `lw` writes a value for `r1` in cycle 9.

```
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12
lw r1, 0(r2)     IF ID Q  RR EA          ME WB C
add r3, r9, r1   IF ID Q                 RR EX WB C
or  r6, r3, r8      IF ID Q                  RR EX WB C
xori r1, r4, 5      IF ID Q  RR EX WB                C
sub  r5, r1, r3        IF ID Q                  RR EX WB C
```

☑ Briefly explain why the code runs correctly despite the fact that `lw` writes *after* `xori`.

Because registers are renamed, meaning `lw` writes a different physical register than `xori` and so there is no confusion.

(*e*) (6 pts) Modern VLIW ISAs are designed with modern implementations in mind, unlike decades old RISC ISAs.

☑ Show the contents of a typical VLIW bundle.

A bundle typically contains three RISC style instructions plus some dependency information.

☑ Provide an example of a VLIW feature that's designed to make implementation easier. Explain how it does so.

The dependency information eases design of the control logic, perhaps shortening critical paths.

62 Spring 2007 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 28 March 2007, 11:40–12:30 CDT

Problem 1		(40 pts)
Problem 2		(30 pts)
Problem 3		(30 pts)
Exam Total		(100 pts)

Alias 0x501f

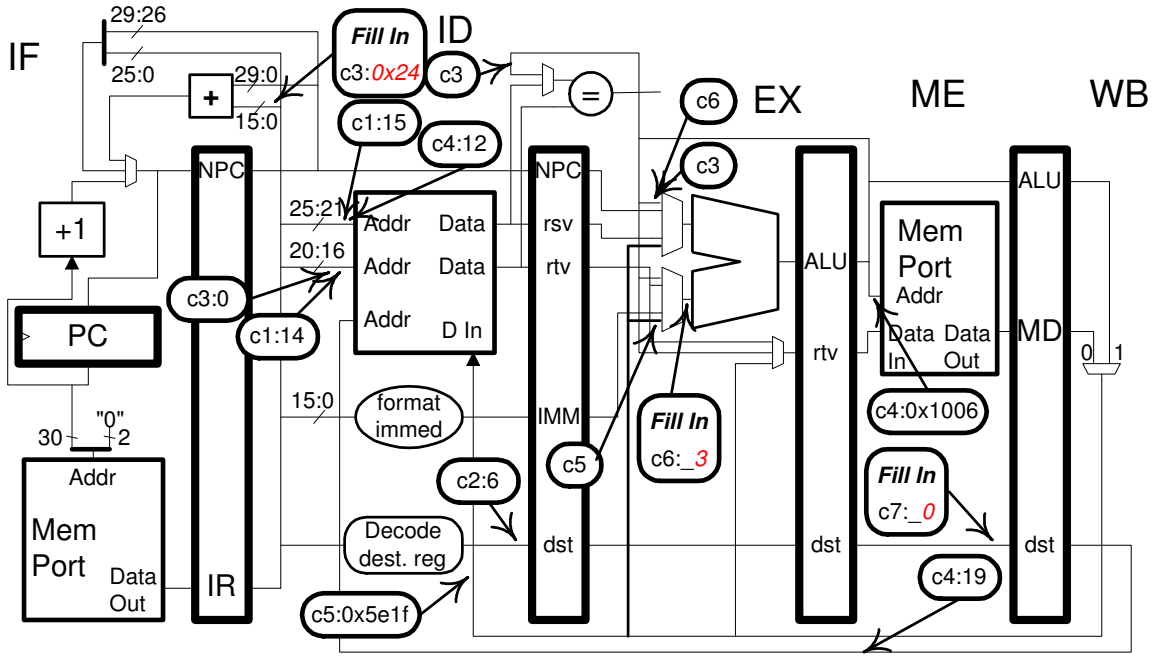
Good Luck!

Problem 1: In the MIPS implementation below some wires are labeled with cycle numbers and values that will then be present. For example, c1:15 indicates that at cycle 1 the wire will hold a 15. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. Note that the last instruction and the address of two instructions have been provided. [40 pts]

☑ Finish a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Be sure to fill the three blocks marked *Fill In*.




```
# Solution Below
#
     #  Cycle               0  1  2  3  4  5  6  7  8
0x1000  xor r19, r15, r14   IF ID EX ME WB
0x1004  lhu r6, 0x5e1f(r19)    IF ID EX ME WB
0x1008  bne r19, r0, TARG         IF ID EX ME WB
0x100c  or r4, r12, r6               IF ID EX ME WB
0x1030  sb  r17, 3(r4)                  IF ID EX ME WB
     #  Cycle               0  1  2  3  4  5  6  7  8
```

Instruction at 0x1000: The c1:15 and c1:14 directly provide the two source registers and c4:19 directly provides the destination register. Because it uses an rt register and writes back it must be a format R instruction, though it could be almost any arithmetic or logical format R instruction (such as `add`, `sub`, `slt`, `or`). Of these `xor` was chosen arbitrarily.

Instruction at 0x1004: The destination register, **r6**, is determined directly by the c2:6. The c3 in EX indicates that this instruction bypasses its rs register value from the previous instruction and so the rs register of this instruction must be the same as the destination of the previous instructions, **r19**. The c4:0x1006 indicates that this is a memory instruction of size either byte or half, and since it writes back it must be some kind of a load. The size of loaded value, c5:0x5elf, rules out a **lb** or **lbh**, so the instruction must be a **lh** or a **lhu**. Peculiarly, the address being loaded from is actually the two least significant bytes of the instruction itself and so the loaded value provides the instruction's immediate field value: **0x5elf**. (Determining this instruction's immediate value was probably the subtlest part of this problem.)

Instruction at 0x1008: The ID-stage c3 indicates that this is a branch, and that the rs register is **r19**. The c3:0 indicates an rt register of 0. The last instruction has address **0x1030** and so the branch must have been taken. Because the previous instruction loaded from address **0x1006** the value in register **r19** could not have been zero, and so the branch must be **bne**. The value for the Fill In: c3: is the number of instructions to skip, the target address minus the delay slot address: **0x1030 - 0x100c = 0x24** or for those proud of their ten fingers, **36**.

Instruction at 0x100c: The c4:12 directly indicates the rs register. The bypass indicated by the EX-stage c6 indicates that this instruction's destination register value is bypassed to the following one and so this instruction must write register **r4**. The c5 in the EX stage indicates that this instruction reads an rt register, since it also writes a register it must be format R. As with the first instruction there is no way to tell which one it is, **or** is an arbitrary choice.

Instruction at 0x1030: This instruction itself is given, but there are the fill-in blocks in EX and ME. The c6: fill-in block points to the lower input of the ALU when this instruction is in EX. For stores (and also loads), the lower input is the immediate value, which in this case is 3. The c7: fill-in block points to the destination register number. This instruction is a store, since stores don't write registers the destination register number is 0. (Register **r17** specifies the value to write to memory.)

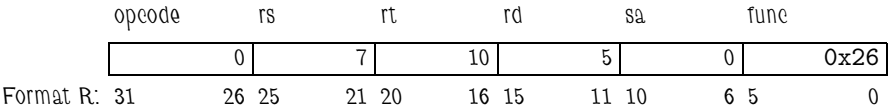
Problem 2: Answer each question below.

(*a*) Show the encoding of the MIPS `xor` instruction below. In particular, show the name of each field in the encoded instruction, the fields' bit positions, and the fields' values. If you don't know the value of a field (there's one or two you're not expected to know) make up a value and label it "made up." *Hint: It is not cheating to look at the diagram for Problem 1.*

```
xor r5, r7, r10
```

☑ [10 pts] Encoding showing field names, bit positions, and values.

Fields shown below. Only the `func` field can have a value marked "made up." One was expected to know that for common format R instructions the `opcode` field value would be zero.



(*b*) Unlike some other benchmark suites, in SPECcpu the tester compiles the benchmarks (rather than having SPEC provide the benchmarks already compiled). [10 pts]

☑ How does this difference make SPECcpu valuable to those in the area of computer architecture?

Those in the area of computer architecture are interested in the performance potential of new designs. Because they are usually testing products they would like to sell, it is in the testers' interests to get results reflecting the full potential of their systems so they will select compilers and other build tools appropriate for the new design and make the best use of these tools.

If SPEC were to provide compiled (built) benchmarks then they might be compiled for some average system, and so would not make full use of a new designs' special features. If a new design implemented a new ISA then SPEC binaries would not run at all, which of course would disappoint the computer architecture research community.

☑ Why might this difference make SPECcpu less valuable to those looking for the fastest computer to run their favorite game?

SPEC shows the performance of a program compiled specially for a particular machine. Games are compiled for some typical machine. If you have a system with some uncommon special feature, the SPECcpu benchmarks might show performance when that feature is used, but games, intended for a broad audience, might not be compiled for that uncommon feature. So one might get lower performance on a game than would be led to expect by looking at similar SPECcpu benchmark numbers.

(*c*) A company is considering adding a BCD data type to their new ISA. An analysis of a suite of Cobol programs, widely used by their customers, shows that the ISA's BCD data type would be extensively used. [10 pts]

☑ What more does the company need to know to make the decision?

The company needs to know how much faster a system would be with the new data type.

If the hardware didn't have the data type then BCD arithmetic would have to be done in software. That would be slower, but would only have a significant impact if a large amount of BCD arithmetic were performed.

Note that issues such as precision or the ability to represent decimal fractions in binary are irrelevant because BCD would be used whether or not the data type were present. If not present, numbers would still be represented in BCD but rather than using, say, a BCD add instruction they'd use a whole bunch of shift, and, and add instructions.

Problem 3: Answer each question below.

(*a*) The MIPS code below loads a character from memory and places it in bit positions 15:8 of register `r3`. [10 pts]

```
lbu r1, 0(r2)   # Note: r2 can be any address.
sll r1, r1, 8
and r3, r3, r4  # r4 = 0xffff00ff
or  r3, r3, r1
```

☑ Explain how the operation performed by this code is similar to, and differs from, the operation performed by `lwl` and `lwr` (from Homework 1). *Note: the phrase "the operation performed by this code" was not present on the original exam.*

Similar: loaded value shifted and combined with existing contents. Difference: Can place any byte in a specific position.

☑ Suppose the operation performed by the code above is something that important benchmarks do often. **Given that MIPS already has `lwl` and `lwr`** is it worthwhile adding an instruction that performs the same operation as the code above? Explain, stating any necessary assumptions or made-up data.

The problem indicates that the operation performed by the code is important and done often. It should also be clear that a machine instruction could perform the operation faster since three of the four instructions it would replace just drop a value in a particular bit range.

Whether it is worthwhile to add this instruction then depends upon how much it would cost. The problem clearly states that `lwl` and `lwr` are already present. Since the hardware to implement these instructions can load something from memory and place it in parts of the destination register without changing the register's other contents, the additional hardware needed for the new instruction would be minimal. And for that reason it would be worthwhile.

Grading Note: Very few saw the point of this question.

(*b*) The first code fragment below, standard MIPS, uses a `slt` to perform a comparison for a branch. The second uses a proposed MIPS branch instruction that does the comparison itself and as a result the target is fetched one cycle sooner. The first execution is on *sImp*, an implementation of standard MIPS, the second execution is on *bImp*, an implementation of the proposed MIPS; bImp includes `blt` but is otherwise identical to sImp. [10 pts]

```
 # Standard MIPS  Cyc: 0  1  2  3  4  5  6  7
 slt r1, r2, r3        IF ID EX ME WB
 bne r1, r0,  TARG        IF ID EX ME WB
 nop                         IF ID EX ME WB
TARG: xor r4, r5, r6            IF ID EX ME WB

 # Proposed MIPS  Cyc: 0  1  2  3  4  5  6  7
 blt r2, r3, TARG      IF ID EX ME WB
 nop                      IF ID EX ME WB
TARG: xor r4, r5, r6         IF ID EX ME WB
```

☑ Why might the clock frequency of bImp be lower than sImp?

The hardware needs to know whether the branch is taken by the end of its ID stage. In sImp the comparison used for the branch starts at the beginning of the branch's ID stage (when `slt` is in EX). In bImp the comparison is also done when the branch is in ID, but it starts only after the registers are retrieved from the register file so there is less time to do so. If comparison takes long enough then bImp's clock frequency would be forced to be lower.

☑ How does knowing the percentage of branches in a program help determine if bImp can run the program faster than sImp (when compiled for the respective implementation)?

With bImp there would be fewer instructions, and if the clock frequency were identical it would run faster. But if bImp had a lower clock frequency then the only way for it to be faster than sImp would be bImp programs to be small enough to overcome the clock frequency disadvantage. The reduction in the number of instructions is based on the number of `slt` / branch pairs, so knowing that can help one estimate performance.

(*c*) All a compiler needs to know about the target (the implementation being compiled for) is its ISA. However, if the compiler also knows the implementation it can produce faster code. [10 pts]

☑ What's wrong with the following statement: *If the compiler also knows the target implementation it can make better register assignment decisions since it knows the exact number of registers available.*

The exact number of registers is something specified in the ISA, not the implementation.

☑ How can the compiler produce better code knowing operation latencies, such as the time needed for a `mul` instruction?

With a knowledge of operation latencies the compiler could choose the best code for an operation that can be performed several ways. For example, to multiply by a small constant, say 6, one might use an integer multiply or one could use a series of shifts and adds. Without knowing operation latencies one can't tell which method is faster.

Scheduling (rearranging instructions) is another thing the compiler can do better knowing operation latencies. (Scheduling is not something covered before the exam and so that was not expected as an answer.)

Name Solution

Computer Architecture
EE 4720
Final Examination

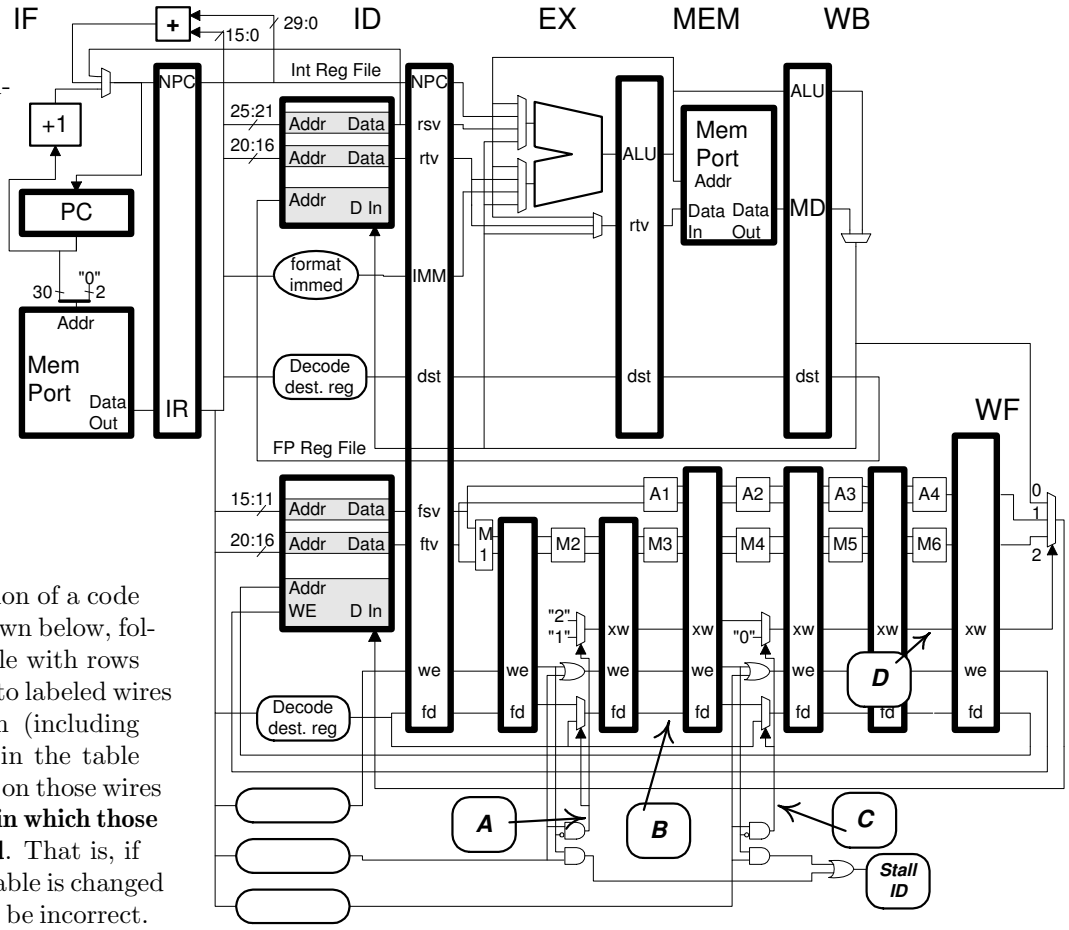
10 May 2007, 7:30–9:30 CDT

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias

Good Luck!

Problem 1: (20 pts) The statically scheduled MIPS implementation illustrated to the right is taken from the class notes. To avoid making things too easy some descriptions were removed from logic blocks in the lower-left corner.



(*a*) The execution of a code fragment is shown below, followed by a table with rows corresponding to labeled wires in the diagram (including Stall ID). Fill in the table showing values on those wires **only for cycles in which those values are used**. That is, if a value in the table is changed execution must be incorrect.

☑ Complete the table, omitting unused values.

```
# Cycle             0  1  2  3  4  5  6  7  8  9  10
 mul.d f2, f12, f18 IF ID M1 M2 M3 M4 M5 M6 WF
 add.d f8, f10, f16    IF ID A1 A2 A3 A4 WF
 sub.d f6, f20, f14       IF ID -> A1 A2 A3 A4 WF
 lwc1 f4, 0(r1)              IF -> ID ----> EX ME WF
# Cycle             0  1  2  3  4  5  6  7  8  9  10
 A                        1  0  1                    <- Solution
 B                           8  2  6                 <- Solution
 C                              0  0  0  1           <- Solution
 D                                       1  2  1  0  <- Solution
 Stall ID:         0  0  0  1  0  1  1  0  0  0  0 <- Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10
```

Solution Discussion

A: This control signal is used to insert `add` and other instructions that use the floating-point adder. When the signal is `0` `fd` passes through unchanged and `xw` is set to take the multiplier output. An instruction is inserted when the control signal is `1`: the `fd` value from `ID` stage will enter and the `xw` control signal will be set to take the adder output.

In the code above the `add.d` and `sub.d` need the FP adder. The 1 in cycle 2 is generated for the `add.d`, it arrives at `A1` in cycle 3. Also in cycle 3 a 0 is generated for the `sub.d` instruction, it is refused entry because it would have arrived at `WF` in the same cycle as `mul.d`. In cycle 4 the signal is 1 and so the `sub.d` enters `A1` in cycle 5. If there is no instruction being inserted and there is no multiply passing through it does not matter what the value is and so those entries are blank (cycles 0, 1, and $5 \cdots 10$).

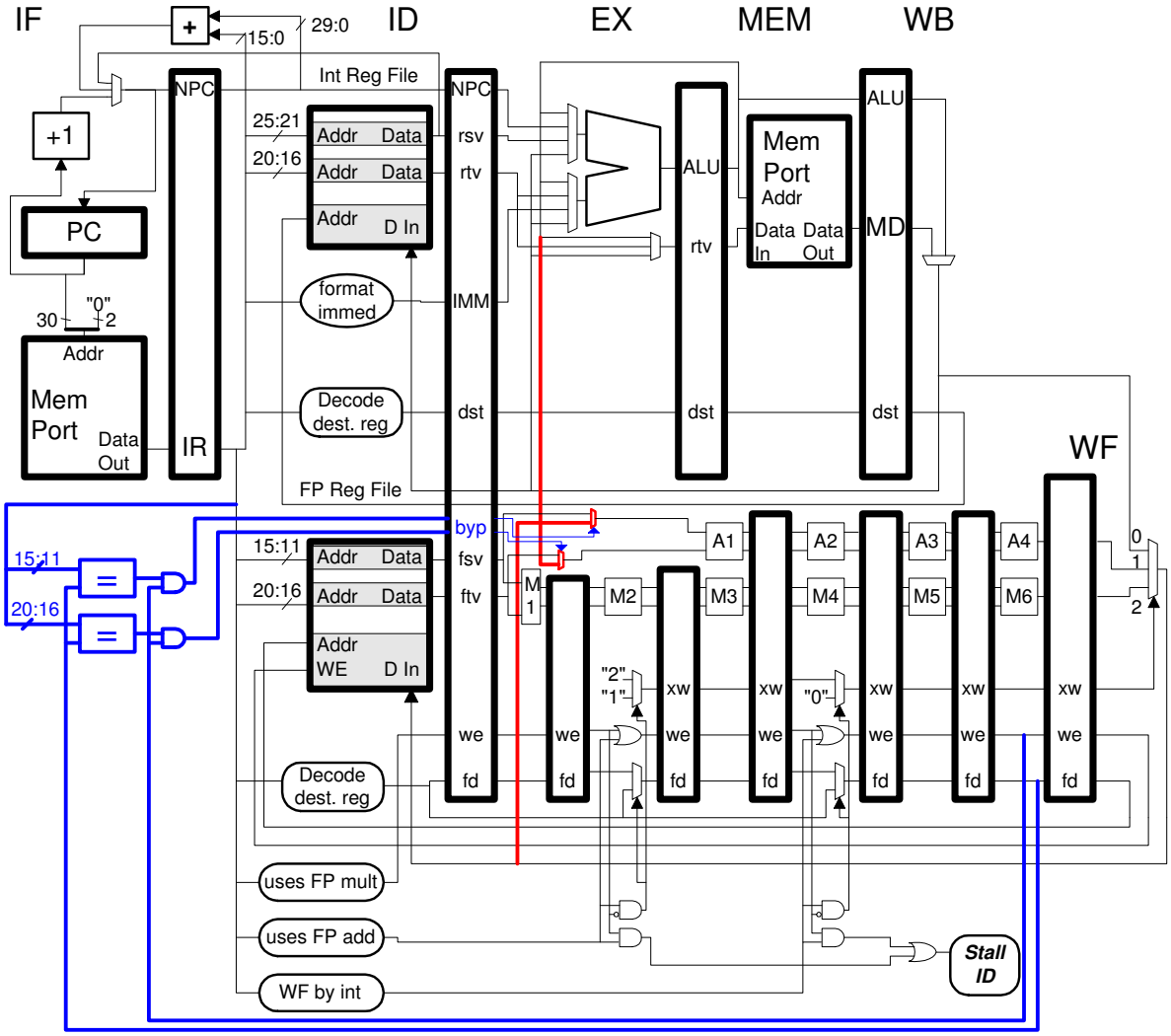
B: This signal provides the floating-point register to write for the instruction in stage `A1`/`M3`. Note that add-unit and multiply-unit instructions pass through this stage but instructions that use the integer pipeline to get a floating-point register value do not pass through this stage. That's why `f4` is not present here (it is inserted in the next stage).

C: This control signal is used to insert instructions that get their value from the integer pipeline. In the example, that's `lwc1`, another such instruction is `mtc1`, seen in part b. The `lwc1` is inserted in cycle 7, which is its last cycle in ID. The signal must be 0 when other instructions pass through, that happens in cycles 4, 5, and 6. At other cycles it doesn't matter what the value is.

D: This is the control signal for the `WF`-stage multiplexor. A value of 0 selects the value from the integer pipeline, 1 selects the floating-point add unit, and a 2 selects the multiply unit. The value has no effect if there isn't an instruction in WF in the next cycle, for that reason values are blank in cycles 0 to 5 and cycle 10.

Stall ID: This value is 1 if there is a stall. Note that a stall starts in the cycle before the beginning of an arrow (cycles 3 and 6) and the stall ends in the cycle before the arrowhead. Unlike the other signals this one must be shown every cycle to achieve the given execution.

Grading Note: For Stall ID very few showed 0's for every cycle without a stall.



(*b*) Show the execution of the code below on the implementation assuming that all needed bypass are present. Don't forget to check for dependencies. (Instruction `mtc1` moves a value from an integer register to a floating-point register.)

☑ Pipeline diagram.

```
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11
 mtc1 f2, r7         IF ID EX ME WF                      <- Solution
 add.s f3, f4, f2       IF ID A1 A2 A3 A4 WF             <- Solution
 add.s f6, f3, f8          IF ID -------> A1 A2 A3 A4 WF <- Solution
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11
```

Diagram shown above. Note that it's possible to bypass `f2` from `mtc1` to the `add.s` without a stall.

Grading Note: Most did not realize a `mtc1`-to-`add.s` bypass was possible and so showed a stall in cycle 3.

(*c*) Add the bypass paths needed by the code above and show the control logic for the added paths. **Do not add unneeded bypass paths.** *Hint: Control logic should consist of two one-bit signals.*

☑ Bypass paths for code above.

Bypass paths shown in red. Note that the bypass from `mtc1` to `add.s` is taken from the ME stage by extending the ME-to-EX bypass connection.

Also, it would probably make more sense to connect the outputs of added bypass multiplexors to both the add and multiply units. The reason for not doing it in the diagram was just space. Those solving the problem might have avoided doing so since unneeded bypass paths were not to be added. Points would not have been deducted for that since those ME-to-M1 paths are free.

☑ Control logic for added bypass paths.

Control logic shown in blue. The control signals are computed in ID, pass through the ID/A1 latch and are used in A1. Computing the control signals in A1 is not a good idea because extra pipeline latch space would be needed for `fs` and `ft` (ten bits versus two) and worse the floating point add would have to wait for the comparison units.

Problem 2: (20 pts) Illustrated is the execution of some code on our dynamically scheduled MIPS implementation along with the contents of the ID register map, the commit register map, and the physical register file. The implementation itself is shown on the next page.

```
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 mul.d f2, f4, f6  IF  ID  Q   RR  M1  M2  M3  M4  WF  C
 add.d f4, f2, f10     IF  ID  Q               RR  A1  A2  A3  WF  C
 ldc1 f2,0(r1)             IF  ID  Q   EA  ME  WF                      C
 sub.d f2, f2, f8              IF  ID  Q   RR  A1  A2  A3  WF              C
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID Register Map
 F2:      12           9       71  99
 F4:      18               51
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
Commit Register Map
 F2:      12                                                  9               71  99
 F4:      18                                                                  51
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
Physical Register File
  9                   [                           2.1                     ]
 12          2.0                                              ]
 18          4.0                                                          ]
 51                      [                                       4.1
 71                          [               2.2                                 ]
 99                              [                       2.3
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

(*a*) Answer the following

☑ Which physical register was allocated for `f4` in `add.d`?

Physical register 51.

☑ If one used the ID map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

Value of 2.3 (from physical register 99).

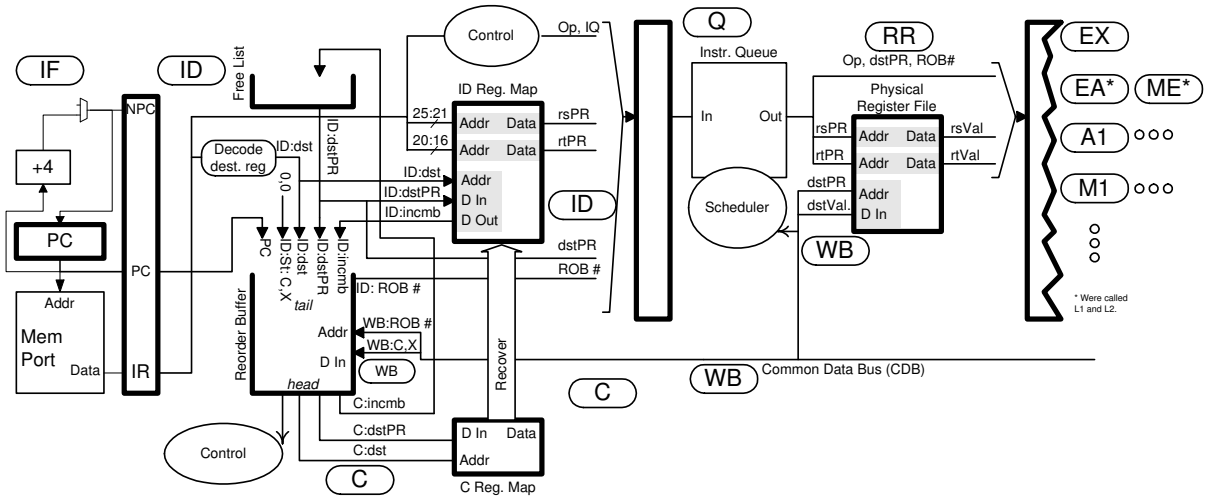
☑ If one used the commit map to determine the value of `f2` in cycle 11, what value would one obtain? *Hint: It's a two-step process.*

Value of 2.1 (from physical register 9).

☑ In cycle 11 where is the value of `f2` from `ldc1` located?

In the physical register file, in physical register 71.

Problem 2, continued: Dynamically scheduled processor shown for reference.



Problem 2, continued:

```
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 mul.d f2, f4, f6  IF  ID  Q   RR  M1  M2  M3  M4  WF  C
 add.d f4, f2, f10     IF  ID  Q                   RR  A1  A2  A3  WF  C
 ldc1 f2,0(r1)             IF  ID  Q   EA  ME  WF                          C
 sub.d f2, f2, f8              IF  ID  Q   RR  A1  A2  A3  WF                  C
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
ID Register Map
 F2:      12           9       71  99
 F4:      18               51
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
Commit Register Map
 F2:      12                                                  9               71  99
 F4:      18                                                                  51
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
Physical Register File
  9                   [                               2.1                 ]
 12          2.0                                                  ]
 18          4.0                                                          ]
 51                      [                                           4.1
 71                          [                   2.2                         ]
 99                              [                           2.3
 # Cycle           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

(*b*) Suppose that in cycle 11 the contents of physical register number 71 was somehow changed, perhaps due to a one-time problem. If the code executes as shown then the program runs correctly. But if a hardware interrupt happens (is taken) at the wrong time execution would be incorrect because of this change. Explain why, illustrate timing details on the diagram.

☑ Reason for incorrect execution.

Physical register 71 holds the value of `f2` used as a source by `sub.d`. If the value were changed in cycle 11 that would not affect `sub.d` since it already read (actually bypassed) the value in cycle 7 (if it got it from the register file it would be in cycle 6). If by chance the handler for the hardware interrupt starts right after `ldc1` then execution would have to resume at `sub.d` (and since it's a hardware interrupt there's no reason to think any of the instructions above caused the problem). When the handler returns `sub.d` will again execute (it didn't commit the first time) but this time it will read the changed 71 from the physical register file.

The key feature is the handler starting between `ldc1` and `sub.d`.

Grading Notes: Many solutions described the handler as starting, say, in cycle 12. Instead, the solution should have indicated the last instruction to commit before the handler starts.

☑ Timing details on diagram.

See the answer above.

Problem 3: (20 pts) The MIPS code below runs on a system using a bimodal branch predictor of the indicated sizes. Branch outcomes are shown for each branch, the outcome patterns will continue to repeat.

```
BIGLOOP:
B1: 0x1000 beq r1, r2    T  T  T  T  T  N  T  N  N  T  T  T  T  T  T  N  T  N  N  T ...
... nonbranch insn.
...
B2: 0x1100 bne r3, r4     N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  N  ...
nop
j BIGLOOP
nop
```

(*a*) What is the accuracy after warmup of a bimodal branch predictor with a 2^{14}-entry BHT on branch B1?

☑ 2^{14}-entry BHT bimodal accuracy on B1.

Accuracy is 60%.

(*b*) What is the accuracy after warmup of a bimodal branch predictor with a 2^4-entry BHT on branch B1?

☑ 2^4-entry BHT bimodal accuracy on B1.

With a 2^4-entry BHT branches B1 and B2 will share the same entry. Since B2 always executes after an execution of B1 it will be as though the counter were unchanged when B1 is taken and decremented by 2 when B1 is not taken. The counter will soon reach zero and only the not-taken outcomes will be correctly predicted and so the accuracy is 30%.

(*c*) What is the smallest BHT size for which one can obtain the same accuracy on branch B1 as a 2^{14} entry table? Explain.

☑ Smallest size for 2^{14} entry accuracy.

Smallest size: 2^7 entries.

☑ Reason.

That is the smallest size at which B1 and B2 use separate entries. It's the sharing of entries that results in the lower accuracy in part b.

(*d*) Normally the BHT in a MIPS implementation is indexed starting at bit position 2 (omitting the 2 least-significant bits) of the branch PC (address). For the following questions think about answers to the preceding parts but answer the question for ordinary programs, not the code sample above.

☑ Why might starting at position 3 or 4 be better?

Better means reducing the number of collisions (sharing of entries, as in part b). Taking, say, the 14 PC bits from 16:3 instead of 15:2 would separate branches that differed at bit 16 but whose lower bits are identical. That's good. That might be offset by collisions from branches which differ only in bit position 2; those branches would be adjacent and since that's unlikely there would probably be a benefit by starting at 3.

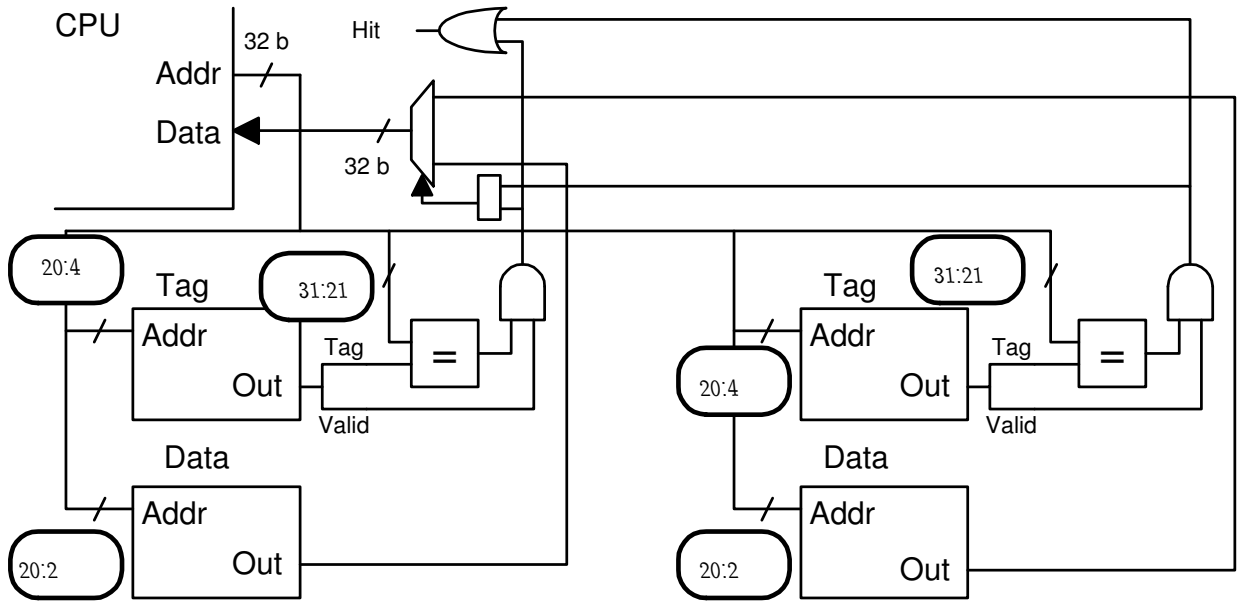
☑ Why might starting at position 10 or 11 be worse?

About one out of six instructions is a branch, so starting at 10 would certainly result in collisions by nearby branches.

Problem 4: (20 pts) The diagram below is for a 4-MiB (2^{22}-character) set-associative cache with a line size of 16 characters on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

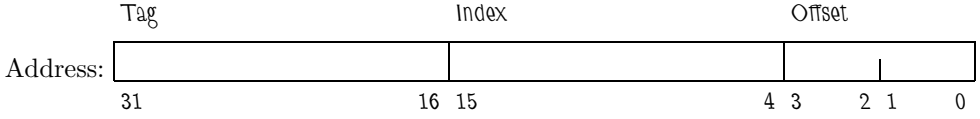
☑ Associativity:

The cache is 2-way set associative.

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $2 \times 2^{21-4}\,(32 - 21 + 1)$ bits.

☑ Show the bit categorization for a 64-way set-associative cache with the same capacity and line size.



Problem 4, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
short *a = 0x2000000;    // sizeof(short) = 2 characters.
int i;
int ILIMIT = 1 << 10;    // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size of 16 characters is given, the size of an array element is two characters. The miss on the first iteration will bring in 16 characters, a line. The second iteration will access data on this line. The line will be "used up" at $i = \frac{16}{2} = 8$, and so for each miss there are 7 hits. The hit ratio is $\frac{7}{8}$.

(*c*) The code below runs on a **direct mapped cache** with the same line size and capacity as the cache from the first part. Initially the cache is empty; consider only accesses to the arrays. Choose `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled in the minimum number of iterations (minimum `ILIMIT`). (Every access should be a miss.)

☑ `b`, `ILIMIT`, and `ISTRIDE`

☑ Briefly explain each choice.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.

char *b = 0x3000010;                                    // SOLUTION

int i;
int ILIMIT = 1 << 17;                                   // SOLUTION

int ISTRIDE = 1 << 5;                                   // SOLUTION


for(i=0; i<ILIMIT; i++)
  sum += a[ i * ISTRIDE ]  +  b[ i * ISTRIDE ];
```

Since the cache is direct mapped generate each index exactly once. Accesses to `a` generate even indices and `b` odd indices.

Problem 5: Answer each question below.

(*a*) (5 pts) Consider trap instructions and instructions that raise exceptions.

☑ What are trap instructions typically used for?

Trap instructions allow code running in user mode to call operating system routines. Those routines can do things that user code is not allowed to do, such as accessing hardware. A trap instruction might be used for system calls, for example, to open or read from a file.

☑ Sometimes when an instruction in a program raises an exception the program ultimately is allowed to continue. Give an example of such an exception, and what the handler might do.

Load and store instructions frequently encounter routine problems that the operating system can fix. This might include moving a page of memory from the disk into memory. After the handler fixes such a problem it will restart the program at the load or store that raised the exception, with the expectation that the load or store would now complete normally.

(*b*) (5 pts) When a MIPS instruction raises an exception the type of exception is written to the cause register. SPARC V8 lacks an equivalent of a cause register, so what does it use as a substitute? Explain.

☑ SPARCs alternative to MIPS' cause register.

SPARC calls different exception handlers for each type of exception so there is no need to check a cause register. For example, suppose a FP instruction raises an exception due to an arithmetic error. On SPARC a handler just for that exception would be called (located at entry 8 in the trap table), in MIPS a generic handler would be called which would have to examine the cause register to discover, for example, that a FP exception was raised. It would then jump to the appropriate handler.

(*c*) (5 pts) An early critic might have said that the improvements realized by dynamically scheduled systems could be achieved on much less expensive statically scheduled systems by using better compilers. The particular compiler improvements would help statically scheduled systems but have no impact on dynamically scheduled ones. Consider two-way superscalar statically and dynamically scheduled systems for the examples needed below.

☑ Explain what the compilers would have to do and why.

A two-way statically scheduled superscalar system would likely suffer from stalls due to true dependencies, see the first code fragment below. A compiler could avoid those stalls by scheduling (re-arranging) instructions, see the second fragment below.

☑ Provide an example, showing code before and after optimization.

```
# Solution

# Before optimization, execution on statically scheduled 2-way superscalar.
#    Execution is same as a less-expensive scalar system.

 add r1, r2, r3   IF ID EX ME WB
 sub r4, r1, r5   IF ID -> EX ME WB
 or  r6, r7, r8      IF -> ID EX ME WB
 and r9, r6, r10     IF -> ID -> EX ME WB


# Before optimization, execution on dynamically scheduled 2-way superscalar.
#    Though SUB and AND instructions wait other instructions aren't delayed.

 add r1, r2, r3   IF ID Q  RR EX WB C
 sub r4, r1, r5   IF ID Q     RR EX WB C
 or  r6, r7, r8      IF ID Q  RR EX WB C
 and r9, r6, r10     IF ID Q     RR EX WB C


# After optimization: Execution is ideal, same as a more-expensive DS system.

 add r1, r2, r3   IF ID EX ME WB
 or  r6, r7, r8   IF ID EX ME WB
 sub r4, r1, r5      IF ID EX ME WB
 and r9, r6, r10     IF ID EX ME WB
```

(*d*) (5 pts) What is it about loads that allow dynamically scheduled systems to outperform statically scheduled systems even with good compilers?

☑ Explain.

Unlike most other integer instructions, loads take two cycles to produce a result and so a compiler will need to find at least n instructions to place between a load and an instruction sourcing the loaded value. What's worse than that is that sometimes loads miss the cache. If the compiler misses the L1 cache but hits the L2 cache it might be ten cycles before a result is ready, so the compiler might be able to at least schedule away some stalls for a particular load. However, it can't do that for all loads.

In contrast a dynamically scheduled system will allow non-dependent instructions following a missing load to execute.

63 Fall 2006 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 27 October 2006, 12:40–13:30 CDT

Problem 1		(50 pts)
Problem 2		(20 pts)
Problem 3		(30 pts)
Exam Total		(100 pts)

Alias 0x1010

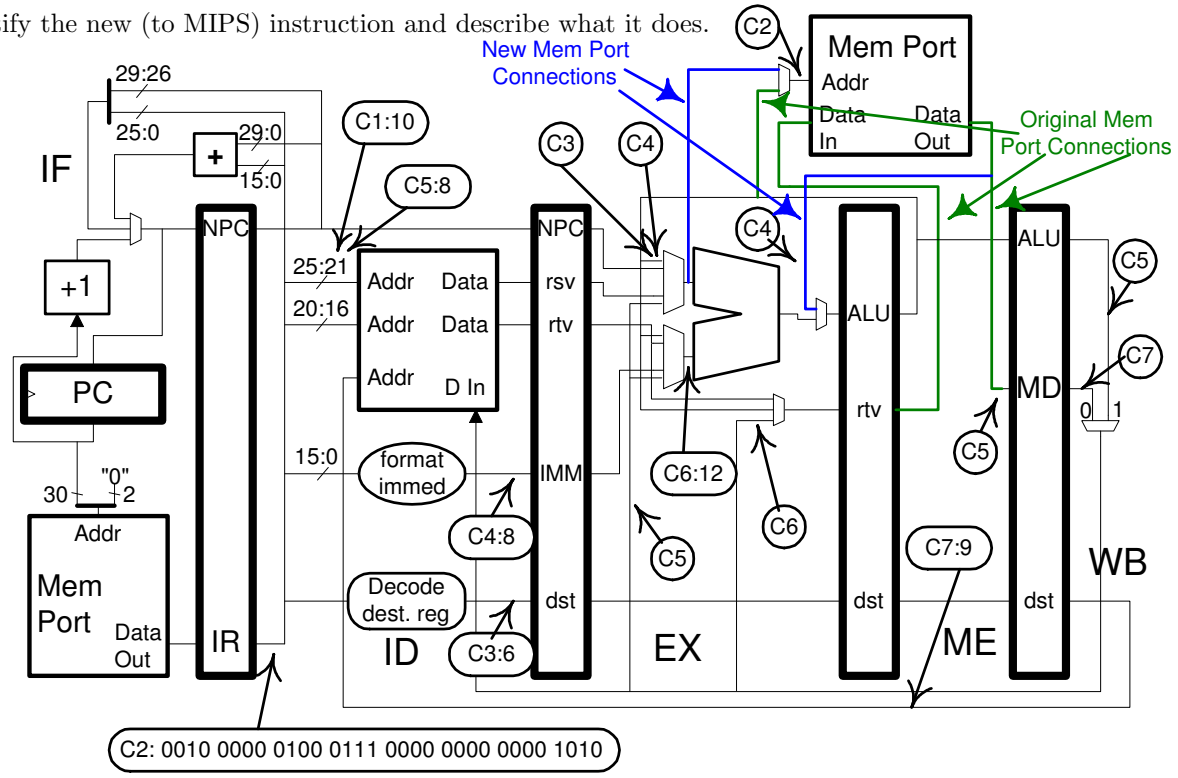
Good Luck!

Problem 1: In the MIPS implementation below the data memory port is in an unusual position that *avoids a stall* and allows the implementation of *a new (to MIPS but not CISC ISAs) instruction*. Some wires are labeled with cycle numbers and values that will then be present. For example, C1:10 indicates that at cycle 1 the wire will hold a 10. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. [50 pts]

☑ Write a program consistent with these labels.

☑ All register numbers and immediate values can be determined.

☑ Identify the new (to MIPS) instruction and describe what it does.




```
# SOLUTION.  See next page for discussion.
#
# Stage labels: eX: EX stage used for arithmetic / logical computation.
#               eA: EX stage used for effective address computation.
#               eM: EX stage used for memory access.
#               me: Memory stage not used.
#               mM: Memory stage used.
#
                    0  1  2  3  4  5  6  7  8
I1: lW   r2, 0(r10) IF ID eM me WB
I2: addI r7, r2, 4     IF ID eX me WB
I3: lW   r6, @(r7)        IF ID eM mM WB
I4: lW   r9, 8(r7)           IF ID eA mM WB
I5: sW   r6, 12(r8)             IF ID eA mM WB
                    0  1  2  3  4  5  6  7  8
```

The implementation diagram shows the data memory port moved from its home in ME to between the EX and ME stages. The connections highlighted in green are also present in the usual five-stage pipeline used in class, and so the memory port can be used by instructions in ME in the usual way. (The highlighting of differences and similarities is only present in the solution.) The new connections, highlighted in blue, allow an instruction in EX to use the memory port, with the address taken from the upper ALU input and the loaded data placed in the EX/ME.ALU latch. Each instruction is discussed below.

I1: The C2 at the memory port can be for an instruction in EX or ME, but in cycle 2 there is no listed instruction in ME and so it must be the instruction in EX, which is I1, and so I1 must be a memory instruction. The C1:10 indicates that its rs register is **r10** and because it's not using the ALU to compute its effective address the immediate must be zero (otherwise the control logic would have it use the memory port when in ME). The C3 at the ALU input indicates a dependence with the following instruction, I2, and so I1 must be some kind of load, the dependence also fixes the destination register to **r2** since that's what the rs register of I2 will turn out to be.

I2: The C2:0010 0000... provides the entire instruction. Those who happen to know that 8 is the opcode for **addi** could get the whole instruction here, for the rest of us there are hints. The C5 in WB indicates that I2 is writing back a value from the ALU, so it must be an arithmetic or logical instruction. If I2 were a type R instruction the rd register would be zero, but since only useful wires are labeled the destination register must be non-zero and so I2 is type I. From the IR value it's easy to find that **rs** is **r2**, **rt** (the destination) is **r7**, and the immediate is 10. It could be any arithmetic or logical instruction and so **addi** is possible but not certain.

I3: The C3:6 provides the destination register, **r6**. The C4 at the ALU output indicates that this instruction uses the memory port when it's in EX. The C5 in ME indicates that *it also uses the memory port when it's in ME*. How odd! This must be the new instruction since no existing instruction uses the data memory port twice. The first use of the memory port would use the **rs** register as an address, the second use would use the loaded value as the address, so I3 must be a memory indirect load.

I4: The C4:8 in ID limits I4 to a type-I instruction. The C5 in EX shows a bypass from I2 which means the **rs** register is **r7** (there is no way the bypass could be to the **rt** register in a type I ALU instruction). The C7 in WB reveals that this instruction used the memory port in ME for a load, so it must be an ordinary load instruction.

I5: C5:8 in ID gives the rs register, **r8**. It's a store since it uses the store value bypass path, C6 in EX, that connection also gives us the rt register: **r6**. The offset is provided by C6:12 in EX.

Instruction I1 is an ordinary load instruction but executes in EX instead of ME, and in doing so avoids stalling I2. The control logic presumably determined that the immediate was zero and so had it execute in EX since it didn't need the ALU. I1 could also be a new load instruction, one that does not have an offset. Unless the immediate field were used for something else there would be no benefit in that since, as shown above, an ordinary load instruction with a zero offset can use the memory port when in EX.

Instruction I3 is a memory indirect load, an instruction that MIPS does not have. And probably won't. In the original implementation the memory port gets the address in the beginning of the cycle, here it must wait for the signal to pass through two multiplexers; in the original implementation the output of the memory port goes straight to the pipeline latch, here it must go through a multiplexer, for a total of three added multiplexers for what is probably already the critical path. Also, if a load from the memory port required more than one cycle then several stages would have to be added to have two loads per instruction.

Problem 2: Answer each question below.

(*a*) Both the SPARC `subcc` and MIPS `slt` instructions below determine if the contents of r1 (g1) is less than r2 (g2).

```
! SPARC
subcc g1, g2, g3

# MIPS
slt r3, r1, r2
```

☑ [5 pts] Show where each writes the comparison result, what is written, and how the result is used for a branch.

SPARC: Comparison result is written in the CC register which consists of four bits: NZCV, N is set if result is negative, Z is set if result zero, C if a carry, and V if an overflow. Branch instructions check the CC register.

MIPS: Comparison result written to `r3`, 0 if false, 1 if true. Branch instructions check if register is zero.

☑ [5 pts] Describe two ways in which the SPARC instruction is more powerful.

Advantages:

The cc instructions do useful computation in addition to setting the condition-code register.

A cc instruction sets four condition bits so several different kinds of branches could use them, for example, one branch might be taken if result positive, another if an overflow, etc.

The use of a cc register gives the compiler (or programmer) one more general purpose register to use (between the time of the comparison and the last branch that uses it).

The control logic needed to test the four CC bits can be made faster than the logic needed in a MIPS implementation that must test a pair of 32- or 64-bit registers.

Grading Note: Several (incorrectly) gave the following advantage: instructions can be placed between the cc instruction and the branch. That's wrong because the same is true for MIPS, so it's not an advantage of SPARC over MIPS. (It is an advantage of SPARC and MIPS over those ISAs in which every arithmetic instruction sets condition code bits.)

(*b*) One advantage of variable-length ISAs is that programs are shorter (take up less memory). For each code fragment below show how CISC instructions can reduce code size using the MIPS code below as the starting point of an example: Make up CISC instruction(s) for the code below, show how they might be encoded, and indicate how much smaller the code fragments are with the CISC instructions.

☑ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
lui r1, 0x1234
ori r1, r1, 0x5678
add r2, r2, r1

# Solution


# If r1's value not used again:
#
add.rri r2, r2, 0x12345678

Coding:  ! opcode ! operand types ! rd ! rs1 ! immed32 !
Size:      8        6               5    5     32        = 56 bits = 7 bytes


# If r1's value is used again:
#
move.ri r1, 0x12345678
add.rrr r2, r2, r1

Coding:  ! opcode ! operand types ! rd ! immed32 !
Size:      8        3               5    32        = 48 bits = 6 bytes

Coding:  ! opcode ! operand types ! rd ! rs1 ! rs2 !
Size:      8        6               5    5     5   = 29 bits = 4 bytes

Total : 10 bytes.


Original MIPS code: 3 * 4 = 12 bytes
Reduction: 2 bytes (r1 needed) or 5 bytes (r1 not needed)
```

The "operand types" field specifies the types of each operand. That field is set to 6 bits for a 3-operand instruction and 3 bits for a 2-operand instruction, based on a rough guess.

Grading note: The sizes above are rounded up to the nearest byte, that's something most did not do on their solutions.

Some solutions gave size reductions in "lines of code." That's wrong because the problem is asking for the reduction in the amount of memory used by the program. *Lines of code* is a relevant (albeit imperfect) measure for things like the amount of human effort needed to write or modify a program.

☑ [5 pts] New CISC instruction(s). Encoding. Size Reduction.

```
jr r31

# Solution


# Return instruction.
#
return

Coding: ! opcode !
Size:       8      = 1 byte.

Reduction: 3 bytes.


# Jump register instruction.
#
jr r31

Coding: ! opcode ! operand type ! rs !
            8           3         5     = 16 bits = 2 bytes

Reduction: 2 bytes
```

The instruction `jr r31` can be used for procedure returns. Since returns are frequent it would make sense to have a CISC instruction just for that, it would be just one byte. That's the first solution. The second is for a CISC instruction that could use any register for the address. Either solution received full credit.

Grading Notes: In several solutions the CISC instruction encoded the full return address. That won't work because a procedure can have more than one caller, and so a single return address could not be used.

Some solutions mentioned displacement. That's not an issue because the register holds the entire address, there is no need to add the register value to anything.

Problem 3: Answer each question below.

(*a*) The SPECcpu2006 benchmark contains two suites, CINT2006 and CFP2006.

☑ [8 pts] Why are there two suites? What would be the disadvantage of combining them into one suite?

There are two suites because the characteristics of integer and FP programs are very different. If they were combined then it would be harder to find systems that were, say, good at floating point.

(*b*) A company develops a new ISA and some implementations of it. It sells the implementations, but keeps the ISA secret.

☑ [8 pts] What's wrong with that?

Can't program the implementation without the ISA definition. You'd be limited to using the software that the manufacturer supplied, and if that didn't include a compiler then you could not write your own software. (Real Programmers don't use interpreted languages.)

(*c*) Operations on packed operand data types often use saturating arithmetic.

☑ [7 pts] What is it and why is it used? Explain using a typical application for packed-operand instructions.

It's arithmetic that uses the maximum representable value for overflow. (Or minimum, for negative overflow.) It is used because in many packed operand applications overflow is likely and a maximum value would be suitable. In graphics applications, for example, the instruction might be computing a pixel intensity, so using the maximum for an overflow will produce an acceptable result.

(*d*) One reason to not use optimization is to make debugging (using a debugger) easier.

☑ [7 pts] Describe two ways optimization makes debugging more difficult. *Hint: Consider single-stepping through code and printing variable values.*

Instructions can be re-arranged and so single-stepping will jump around code that the user would expect to execute sequentially. Common sub-expression elimination is one optimization that would cause code to be rearranged.

Through dead-code elimination variables may be optimized out so one could not print their values.

Name Solution

Computer Architecture
EE 4720
Final Examination

14 December 2006, 17:30–19:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(15 pts)
Problem 6		(15 pts)
Exam Total		(100 pts)

Alias DVD-HD or Blu-ray?

Good Luck!

Problem 1: In the MIPS implementation on the next page some wires are labeled with cycle numbers and corresponding values. For example, c3:1 indicates that at cycle 3 the pointed-to wire will hold a 1. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. The ALU label shows the arithmetic operation performed at the indicated cycle.(20 pts)

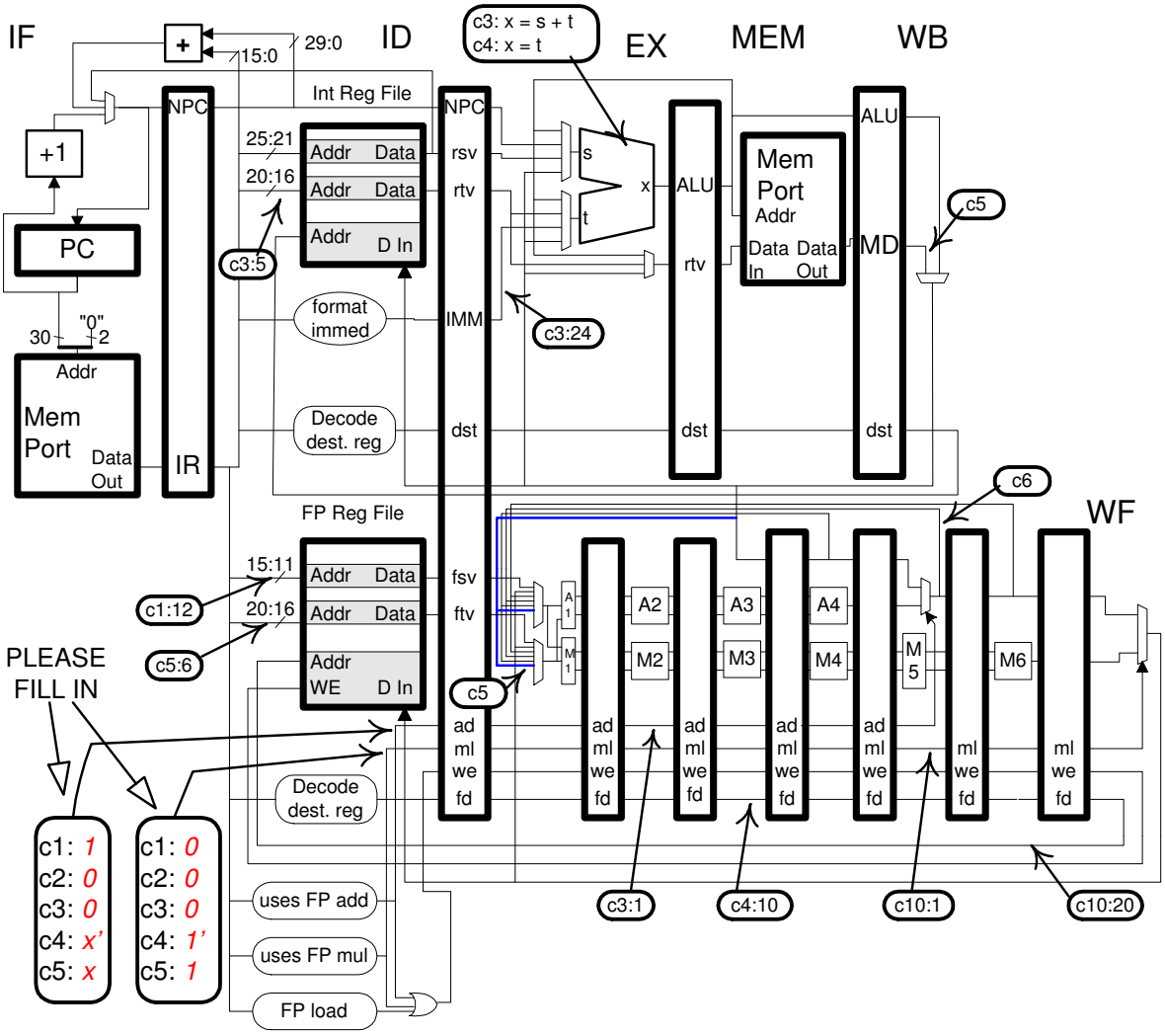
- There are no branches or other control-transfer instructions.
- There are no stalls.
- Every instruction writes the floating-point register file.
- Some instruction(s) read the integer register file.
- One instruction has only briefly been covered, make up a reasonable name for it if you don't remember it.

☑ Write a program consistent with these labels.

☑ Some registers can be determined exactly, others must be made up. **Use as many different register numbers as possible** while still being consistent with the labels.

☑ Fill in the block in the lower-left of the diagram.

Problem 1, continued:




```
# Solution Below and Above in Red
#
# Upper case shows single correct instruction or register, lower case
# shows one of several possible correct instructions or registers.

# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12
0x1000: ADD.s F10, F12, f14  IF ID A1 A2 A3 A4 _5 _6 WF
0x1004: LwC1 f16, 24(r1)        IF ID EX ME _3 _4 _5 _6 WF
0x1008: MTC1 F20, R5               IF ID EX ME _3 _4 _5 _6 WF
0x100c: mul.s f4, f16, f8             IF ID A1 A2 A3 A4 _5 _6 WF
0x1010: MUL.s f2, F10, F6                IF ID M1 M2 M3 M4 M5 M6 WF
# Cycle                      0  1  2  3  4  5  6  7  8  9  10 11 12
```

Instruction at 0x1000: The c1:12 in ID indicates that the first source register is f12. The c4:10 in stage _3 directly provides the destination register, f10. The c6 in stage _5 indicates that this can't be a multiply, and since it reads the floating point register file it can only be an add.

Instruction at 0x1004: The c3:24 in EX indicates that this starts out in the integer pipeline (using an immediate of 24), the

c5 in WB indicates that it's a load, and the c5 in stage _1 shows that it must be a floating point load, such as `lwc1` (load word co-processor 1). There is no way to determine the destination or base register.

Instruction at `0x1008`: Like the previous instruction, this starts in the integer pipeline, indicated by the c3:5 in ID. The c4:x=t in EX indicates that its result is the rt register value (the instruction performs no computation). The c10:20 in WF indicates that this instruction writes register `f20`. Since it uses an rt value it can't be a load, so it's probably an `mtc1` (move to coprocessor 1), an instruction that moves a value from the integer register file to the floating-point register file.

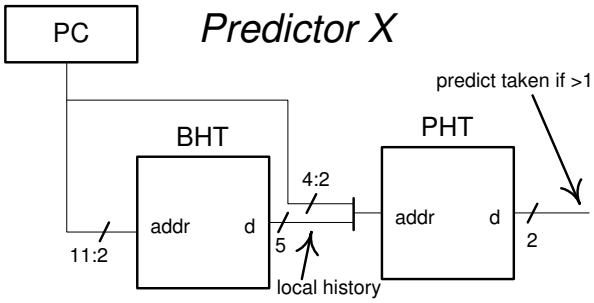
Instruction at `0x100c`: The c5 in _1 indicates that this instruction bypasses from the instruction in WB (the integer pipe) and so the second source register of this instruction must match the destination of the instruction at `0x1004`. Note that there is not enough information to determine what that register is, `f16` is an arbitrary choice.

Instruction at `0x1010`: The c10:1 in _1 indicates that this is a multiply. The c6 in _1 (or _5 from `0x1000`'s perspective) indicates a bypass with the instruction at `0x1000`, revealing the first source register, `f10`. The c5:6 in ID gives the second source register, `f6`. There is no way to determine the destination register.

AD Mux Control Signal Fill In: Values for the first fill-in are the control signals for the stage-_5 mux. A value of 0 indicates that the instruction will write a value from the integer pipeline (such as a `lwc1` or `mtc1`), a value of 1 indicates it will write a value produced by the floating-point adder (such as `add.s` or `sub.s`). For other instructions it doesn't matter what the value is. In the solution, shown in red, the x indicates either value is okay. This solution is valid when the instruction at `0x100c` is a `mul.s`; it could have been an `add.s` and if it were the value at c4 would be 1.

ML Mux Control Signal Fill In: Values for the second fill-in are the control signals for the WF-stage mux. The value should be 1 for multiply, or 0 for other instructions that write the FP register file. If an instruction doesn't write the FP register file the value doesn't matter. The solution appears in red. If the instruction at `0x100c` were an add the value at c4 would be a 0.

Problem 2: The code below runs on three systems which are identical except for the branch predictors. One system uses a bimodal predictor with a 1024-entry BHT, one uses a local history predictor with a 1024-entry BHT and a five-outcome history, and one uses Predictor X, illustrated below. (The PHT input is a concatenation of the local history and three branch PC bits.) The code below has two branches, B1 and B2, which execute in a repeating pattern as shown. There are no other branches in the code.(20 pts)



Note: In the original problem the input to Predictor X used an exclusive or rather than a concatenation.

```
LOOP:
 ..
0x1000: B1: bne r1,r2 SKIP1   N N N T T N N N T T N N N T T N N N T T ...
 ..
SKIP1:
 ..
0x1124: B2: bne r3,r4, SKIP2   N N N T T T N N N T T T N N N T T T ...
 ..
SKIP2
..
     j LOOP
```

☑ What is the accuracy of the bimodal predictor on branch B1 after warmup?

Accuracy on B1: $\boxed{\frac{2}{5}}$. Solution is straightforward.

Note: In the original exam the questions below asked about B1 rather than B2.

☑ What is the accuracy of the local predictor on branch B2 after warmup? (Do not ignore branch B1 when answering this part.)

Accuracy on B2: $\boxed{\frac{5}{6}}$. Branch B2 is predicted using six distinct local history patterns, three of which (`TNNNT`, `NNNTT` and `TTNNN`) are shared with B1. With one of the shared patterns, `NNNTT`, the next outcomes disagree: not-taken for B1 and taken for B2. Since B1 occurs slightly more often the PHT entry at address 3 (`NNNTT`) will eventually start predicting only B1 correctly. Therefore the predictions made using only five out of six patterns in B2 will be correct and the accuracy will be $\frac{5}{6}$.

☑ What is the minimum local history size for the local predictor to achieve 100% accuracy on branch B2 (without ignoring B1)?

Eight outcomes.

With eight outcomes B1 and B2 won't share patterns.

☑ What is the accuracy of Predictor X on branch B2?

Accuracy is 100%. Predictor X is similar to the local predictor except that by concatenating branch address bits with local history, two different branches (or at least those that differ in PC bits 4:2) with the same local histories will be stored in separate PHT entries. This will avoid the interference suffered by B1 and B2 in the five-outcome local history predictor.

☑ What is the minimum local history size for the Predictor X to achieve 100% accuracy on branch B2 (without ignoring B1)?

Minimum history size is three outcomes. With X one can ignore B1 because B1 and B2 will use separate entries. Then considering B2 alone, only three outcomes are needed.

☑ Explain why predictor X has lower or higher accuracy than the local predictor on branch B2.

Because it avoids interference.

☑ As indicated above, B1 is at address `0x1000` and B2 is at address `0x1124`. How would different branch addresses affect the answers above?

If bits 4:2 were the same then predictor X would perform the same as the local predictor. If bits 11:2 of the two branches were the same then the local predictor would perform more like a global predictor.

Problem 3: Consider the execution of MIPS code below. The code follows a large number of `nop` instructions. As can be seen the system below is statically scheduled. In the questions below "relatively simple" means simple compared to dynamic scheduling.

(15 pts)

```
# PC             Cycle:  0  1  2  3  4  5  6  7
0x1ff0: lh r1, 0(r2)     IF ID EX ME WB
0x1ff4: lw r4, 8(r2)     IF ID -> EX ME WB
0x1ff8: addi r6, r6, 1   IF ID -> EX ME WB
0x1ffc: xor r8, r9, r10  IF ID -> EX ME WB
0x2000: sub r11, r8, r13    IF -> ID EX ME WB
0x2004: and r14, r11, r16   IF -> ID -> EX ME WB
# PC             Cycle:  0  1  2  3  4  5  6  7
```

☑ What is the minimum fetch/decode width of a system that could produce that execution? (The x in x-way superscalar)

Minimum width: 4. It must be at least 4 because in cycle zero four instructions are being fetched at the same time.

☑ Why is the fetch/decode width above a minimum and not an exact number?

It's still possible that the system is wider, say 8-way, but is limited to aligned fetch groups and that would explain why the instruction at `0x2000` was fetched in cycle 1 rather than 0.

☑ What caused the `and` to stall in cycle 4? Could the stall be avoided?

True data dependence with the `sub`. That could not be avoided.

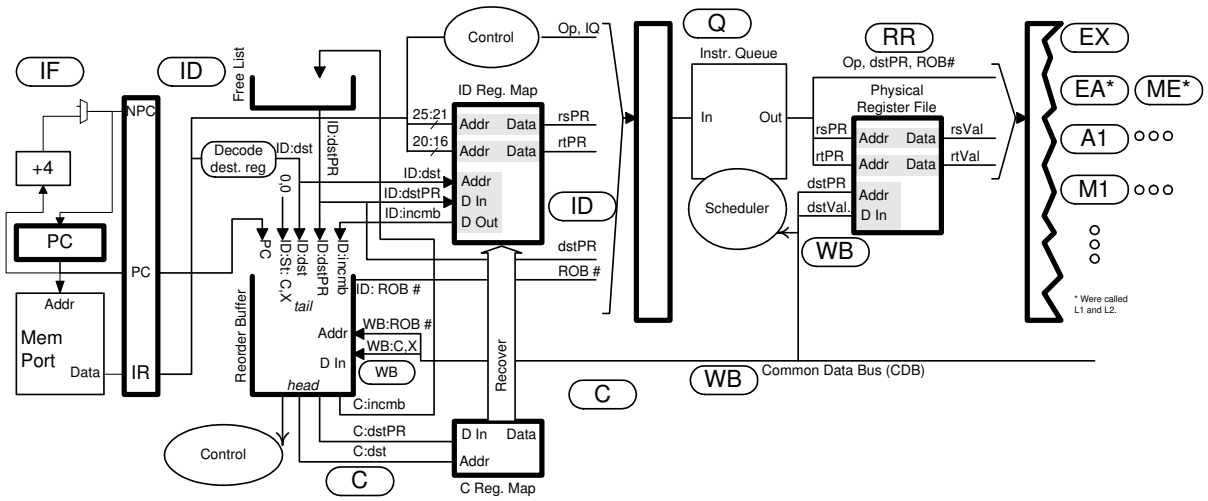
☑ What might have caused `lw` to stall? Suggest a relatively simple change to the implementation to avoid such stalls.

There may have been only one memory port in the ME stage, to be shared by all four instructions. A solution would be to include a second memory port.

☑ What might have caused `addi` and `xor` to stall? *Note: The original question also asked for a "relatively simple solution."*

If `addi` and `xor` did not stall and if new instructions moved into ID then the instructions in ID would be out of order (because of the `lw` which must stall). One could modify the control logic so that it could handle instructions being out of order in ID, the complexity would probably not be worth the effort.

Problem 4: Consider the dynamically scheduled implementation below. (15 pts)



(*a*) Where is the logic for finding the (data) dependencies that requiring bypassing most likely to be?

☑ Indicate on diagram.

Within the scheduler and instruction queue.

(*b*) Suppose due to a manufacturing error every entry in the ID register map is initialized to 12 after each reset, but the commit register map was properly initialized. Initial register values are not defined, so getting the wrong value for a register that was never written is not a problem here.

☑ Which code fragment below is more likely to encounter a problem? *Hint: It has something to do with the connection from the ID Register Map to the ROB.*

The first one.

```
# Fragment A
 add r1, r2, r3
 add r2, r1, r5
 add r1, r6, r7
 add r2, r8, r9
 nop
 ..

# Fragment B
 add r0, r2, r3
 add r0, r1, r5
 nop
 ...
```

☑ Explain what goes wrong.

Because of the flaw the first instruction gets `p12` as an incumbent for `r1` and the second instruction also gets `p12` as an incumbent for `r2`. After the third and fourth instructions commit `p12` will appear in the free list twice, and so later when they are removed from the free list the same physical register (`p12`) could be simultaneously assigned to two in-flight instructions. Each of these two instructions will expect to write its own register, instead they will be writing the same register, causing problems.

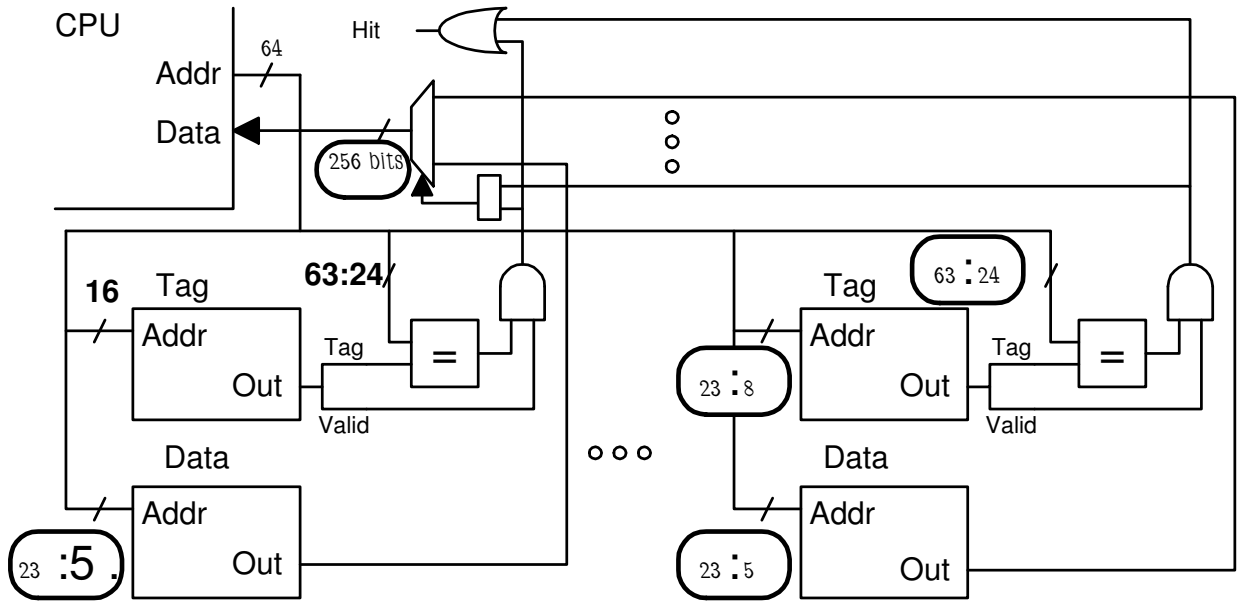
☑ Suppose millions of these defective implementations have already been manufactured. Suppose when turned on the processor starts executing code at address `0x1000`. What code could be put there to fix the problem? (It's not one of the fragments above because one of them only avoids it, later code could still trigger it.) *Hint: There's enough room for the answer below.*

If the instruction at `0x1000` raises an exception then the recovery mechanism will copy the commit map to the ID map, fixing the problem. The only tricky part is getting the exception handler in place before the first instruction executes.

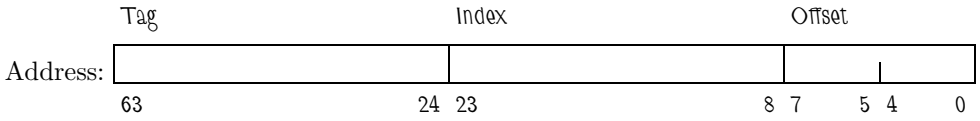
Problem 5: The diagram below is for a 64-MiB (2^{26}-character) set-associative cache on a system with the usual 8-bit characters. (15 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Associativity:

The cache is 4-way set associative.

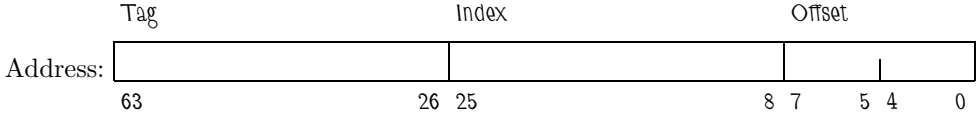
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $4 \times 2^{24-8}\,(64 - 24 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^8 = 256$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 5, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 10;  // = 2^10

for(i=0; i<ILIMIT; i++) sum += a[ i * 4 ];
```

The line size is $2^8 = 256$ characters and the size of an array element is one character. The miss on the first will bring in 256 characters, a line. The second iteration will access data on this line. The line will be "used up" at $i = \frac{256}{4} = 64$, and so for each miss there are 63 hits. The hit ratio is $\frac{63}{64}$.

(*c*) The code below also runs in the cache from part a. Find the minimum values of JLIMIT and JSTRIDE that will result in the code below having a 0% hit ratio.

☑ JSTRIDE and JLIMIT

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i, j;
int ILIMIT = 1 << 10;

int JLIMIT =                                          // FILL IN

int JSTRIDE =                                         // FILL IN

for(i=0; i<ILIMIT; i++)
 for(j=0; j<JLIMIT; j++)
  sum += a[i + j * JSTRIDE ];
```

Notice that the pattern of accesses when i=0 is the same as when i=1, etc., so the strategy is to make sure that whatever is cached when j=0 is evicted by the time j=JLIMIT.

One way of doing that is to set JSTRIDE so large that each access will use a different tag (but have the same index). Since the tag bits start at bit 24 and the array element size is a character, JSTRIDE = 1 << 24. Since the cache is four-way set associative it can only hold four lines with the same index and different tags. So with JSTRIDE = 1 << 24 the line brought in when j=0 will be evicted (assuming LRU replacement) when j=4. By setting JLIMIT=8 no data brought in on one i iteration will be present in the next one.

Problem 6: Answer each question below.

(*a*) In MIPS, SPARC, and many other RISC ISAs memory accesses are aligned and so any instruction that uses an un-aligned address, such as `0x1001` for a MIPS word, will raise an exception usually resulting in the program exiting with a Bus Error.

Perhaps due to the stress of an upcoming code freeze for a product release, a mysterious programmer at Software Company X secretly hacked the operating system of their SPARC computers so that loads and stores to un-aligned addresses would complete correctly, as though un-aligned accesses were not forbidden. There would be no more bus errors. (5 pts)

☑ How might the mysterious programmer have done it?

Modify the exception handler that is used for un-aligned accesses. If the exception were raised by a load the handler might get the data using two load instructions, say one to address `0x1000` and one to address `0x1004`, and then put together the low three bytes of the data from `0x1000` and the high byte of `0x1004`, and put that data in the destination register of the faulting load instruction. Finally, the handler would resume execution at the instruction following the load. To the program it would appear as though the load worked.

☑ Should Software Company X reward or punish the mysterious programmer? Explain.

Punish! The unaligned accesses are due to bugs in the program. The modified handler hides the bugs from the company's programmers but customers, since they don't have the hacked OS, will see them. Even if customers could be given the modified handler the programmers might still want to avoid unaligned access since it slows the program down.

(*b*) The SRAM used to implement caches is costly but in many cache designs can provide 1- or 2-cycle hit latencies. If cost were not an issue, could one use SRAM for the entire memory system and get 1- or 2-cycle latencies on *all* memory accesses?(5 pts)

☑ Explain.

No, because the time for a cache hit includes not just the time to retrieve the data from the SRAM but also the time needed for decoding the addresses and moving the data to the CPU. If SRAM were used for all memory it would have to be spread over many chips and so the part of the access latency would be chip crossing delays, contributing to an access latency larger than 1 or 2 cycles.

(*c*) Manufacturers have a great interest in having their processors score high in SPECcpu. Given this strong interest how can we be sure that benchmark selection and the run & reporting rules have not been chosen to favor a particular manufacturer? (This isn't kindergarten, so "because it's not allowed" is not a good answer.)(5 pts)

☑ Explain.

SPEC has representatives from many manufacturers. If someone proposed benchmarks or rules that favor one manufacturer's products representatives from other manufacturers can be expected to forcefully object.

64 Spring 2006 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 29 March 2006, 11:40–12:30 CST

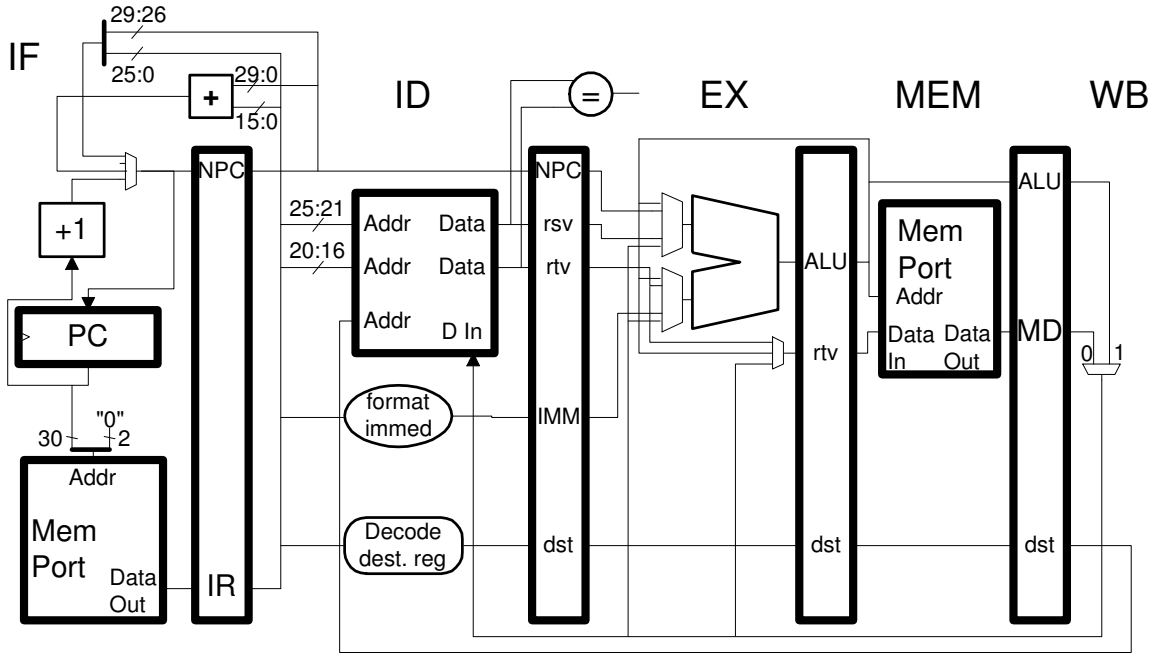
Problem 1		(10 pts)
Problem 2		(40 pts)
Problem 3		(50 pts)
Exam Total		(100 pts)

Alias Four Got In

Good Luck!

Problem 1: The MIPS code below runs on the illustrated implementation. [10 pts]

☑ Show the execution of the code below on the illustrated pipeline.




```
# Solution
# Cycle        0  1  2  3  4  5  6  7
lw r1, 0(r2)   IF ID EX ME WB
add r1, r1, 7     IF ID -> EX ME WB
sw r1, 0(r2)         IF -> ID EX ME WB
```

There is a stall in cycle 3 because the add must wait for the loaded value. (That value is bypassed in cycle 4.)

Note: To keep the test from getting too long the following parts were not included on the original exam. There should be at least one stall.

Think about the questions below for a few moments, then look at the problem on the next page.

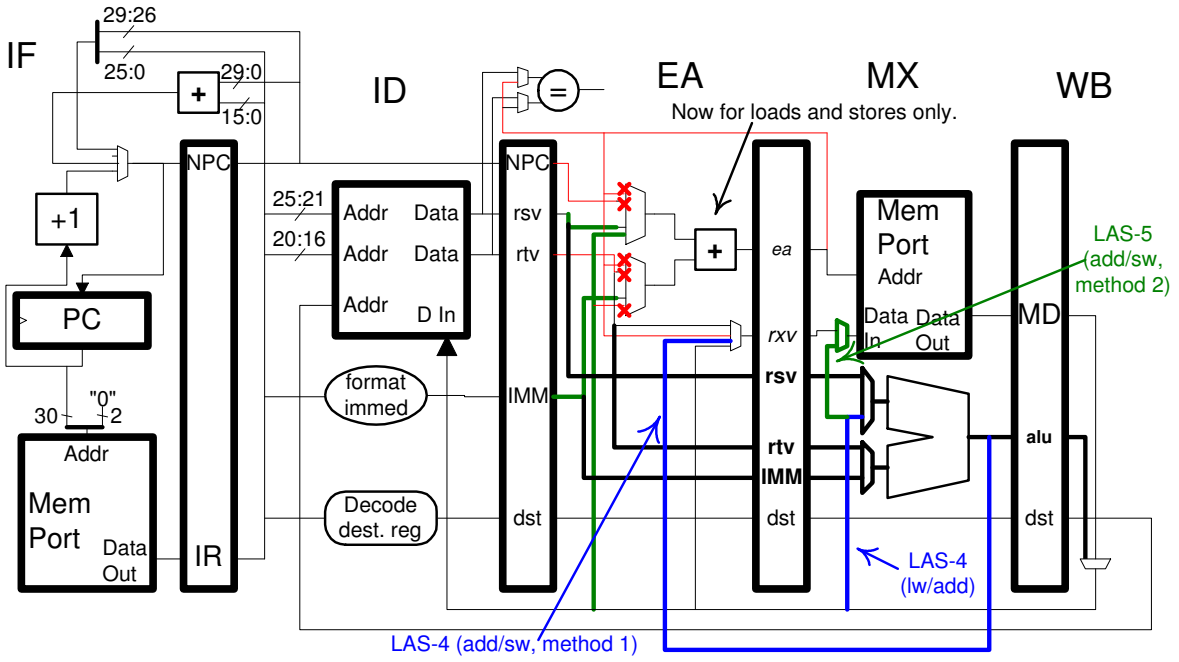
☑ Is it possible to add bypass connections to eliminate the stall and gain performance?

No, because without the stall the `add` would need the value of `r1` before the `lw` retrieved it from memory.

☑ If bypasses won't work does that mean it's impossible to eliminate the stall?

It's not impossible if the pipeline can be re-arranged. See the next problem.

Problem 2: The five-stage MIPS implementation below has an ALU in the MX (new name for MEM) stage (connections for that ALU are shown bold) and what was the ALU in the EA (new name for EX) stage is now just an adder and is to be used only for loads and stores.



(*a*) Add bypass connections needed so that the code below (same as last problem) executes as shown. Label those bypass connections: LAS-*c*, where *c* is the cycle number in which the bypass connection is used. Do not add unneeded bypass connections!

☑ [10 pts] Add bypass connections, label with LAS-*c*.

See diagram. One bypass is shown for the lw/addi dependency, two possible bypasses are shown for the addi/sw dependency; either would be correct.

```
# Cycle          0  1  2  3  4  5  6
lw r1,0(r2)      IF ID EA MX WB
addi r1, r1, 7      IF ID EA MX WB
sw r1, 0(r2)           IF ID EA MX WB
# Cycle          0  1  2  3  4  5  6
```

(*b*) Several connections to the EA-stage adder are now unneeded (but were needed when it was an ALU). (Don't add new connections here.)

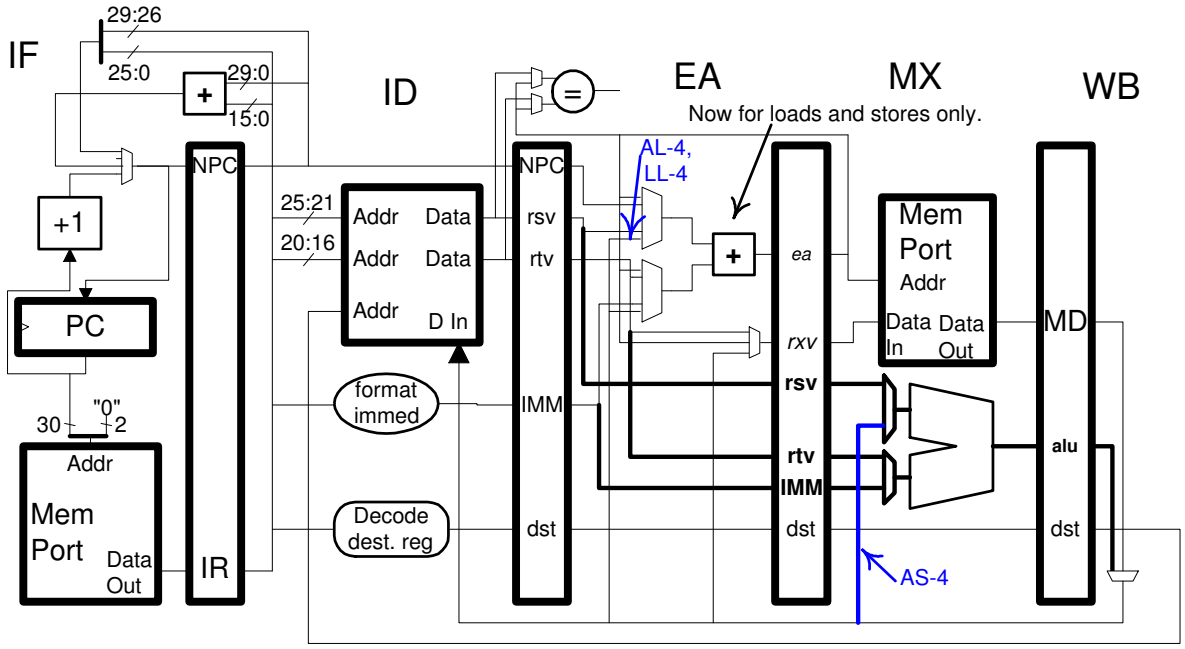
☑ [10 pts] Label unneeded EA-adder connections with an X at mux inputs.

The X's are shown in red in the diagram. The EA adder is now only used to compute effective addresses. For MIPS that's computed by adding the rs register value to an immediate. Inputs for the rt value, bypassed or from the register file, are not needed. The value bypassed from the EA stage is never needed since it's not a register value. NPC is also never used to compute an address. The table below summarizes the reasons:

Upper Multiplexer:

```
0:X Bypass from EA is prior instruction's ea value. Never needed.
1:X NPC. Not used to compute loads or store addresses.
2:  rsv. Needed.
3:  Bypass from WB. Needed to bypass newer rs value.

Lower Multiplexer:

0:X Bypass. Addresses are never computed using an rt register value.
1:X rtv. Addresses are never computed using an rt register value.
2:  IMM. Needed.
3:X Bypass. Addresses are never computed using an rt register value.
```

Problem 2, continued: The implementation below is identical to the one on the previous page. But solution on diagram is different on the two pages.



(*c*) For each code fragment below: Add reasonable bypass connections to eliminate stalls (if any). Show the execution and label the bypass connections that are used (new or existing) with a cycle number. If there is a stall circle Yes if an EX-stage ALU (in addition to the ME stage ALU) would remove the stall, if an EX-stage ALU wouldn't help circle No; if there is no stall don't circle anything. [20 pts]

☑ Add bypasses (if needed). Label bypasses used AS-*c*. Show execution. EX ALU would help: Yes No.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EA MX WB
sub r4, r1, r5      IF ID EA MX WB
```

☑ Add bypasses (if needed). Label bypasses used AL-*c*. Show execution. EX ALU would help: [Yes] No.

```
# Cycle          0  1  2  3  4  5  6
add r1, r2, r3   IF ID EA MX WB
lw r4, 4(r1)        IF ID -> EA ME WB
```

☑ Add bypasses (if needed). Label bypasses used LL-*c*. Show execution. EX ALU would help: Yes [No].

```
# Cycle         0  1  2  3  4  5  6
lw r1, 12(r2)   IF ID EA MX WB
lw r3, 16(r1)      IF ID -> EA MX WB
```

☑ Add bypasses (if needed). Label bypasses used AB-*c*. Show execution. EX ALU would help: [Yes] No.

```
# Cycle          0  1  2  3  4  5  6  7
addi r1, r1, 1   IF ID EA MX WB
beq r1,r2 TARG      IF ID ----> EA MX WB
nop
```

Problem 3: Answer each question below.

(*a*) A computer's scores on SPECint2000 are baseline, 2011; and result (peak) 2014. These baseline and result scores are close, who might be responsible and should they be proud or ashamed: [10 pts]

☑ How might the people who conducted the test be responsible for the close scores? Should they be proud or ashamed?

Rather than trying every combination of compiler switch they could, they just tried a few (because it was a sunny Friday afternoon) and so did not get as much performance as they could have. They should be ashamed.

Detailed explanation: The people that conduct the test compile and run the code. They are free to set compiler switches (and make other build choices) as long as they comply with SPEC's rules for SPECcpu2000.

If they are responsible for the close scores then it must be because of the way they set the compiler switches. For the base tests reasonable optimization options should be set, but for peak (result) scores much more optimization can be done. For the scores to be close the compiler options for the peak case were not set as well as they could be.

It would not be correct to say that the scores are close because the base scores are already close to the highest performance possible. If that were true then the testers would not be responsible, which would be avoiding the question.

☑ How might the people who wrote the compiler be responsible for the close scores? Should they be proud or ashamed?

They wrote a compiler that is very good at choosing optimizations: Just provide the `-fast` switch and the compiler will make the right choices. A human could not improve on things by specifying that this optimization should not be performed, or that optimization should be. They should be proud because they save programmers time and make the hardware and programmers look good.

Another possible answer is that there are only a few optimization choices and that there is nothing to do beyond base optimization. Specialized optimizations that other compilers performed have been omitted. In this case the compiler writers should be ashamed.

(*b*) On a SPARC system the handler for trap number 4 is located at address `0xa0001000` and is 100 instructions long. [10 pts]

☑ Show the Trap Base Register (TBR) value and Trap Table contents needed so that execution of a trap instruction will reach this handler. (Don't worry about returning.)

The trap table can be placed anywhere in system memory (as long as the address is a multiple of 16); the TBR holds that address. Picking an address above 2^{31} and a multiple of 16: `TBR = 0xe12340000`. The entry for trap 4 will be at address TBR $+ 4 \times 16 =$ `0xe12340040`. The contents of that entry is a call to the handler itself: `call 0xa0001000`.

☑ Why isn't a user program allowed to call the handler directly, for example, using the ordinary call instruction `call 0xa0001000`. (The SPARC call has a 30-bit immediate field so the target address is not too far away.)

The user program might try to call something other than the OS's "official" handlers or it might try to jump into the middle to avoid being denied access to something.

☑ What would happen if the user tried to execute the call instruction above?

The calling instruction would raise an exception. The exception handler would probably kill the program.

Problem 3, continued:

(*c*) Packed integer and BCD data types are very superficially similar. [10 pts]

☑ Describe a situation in which a packed integer data type would be useful.

Two long arrays of smaller numbers need to be added together. If each array element is an integer in the range 0-255 and eight values are packed into a 64-bit register then one packed add instruction would add eight pairs together whereas in conventional code one add instruction would add just one pair together.

☑ Describe a situation in which BCD would be useful.

When there is frequent need to show the decimal representation of a number and conversion is too slow (say, an old fashioned computer).

Rounding error is to be avoided and well behaved floating-point formats are not available or not trusted.

(*d*) Describe the contents of a typical bundle in a VLIW ISA. [10 pts]

☑ Bundle contents.

Several instructions (three is common), information on dependencies between the instructions within the bundle and between bundles, and possibly other information about the instructions.

(*e*) Which is it more important to do a good job on, the ISA or the implementation? Explain. [10 pts]

☑ Good job on ISA or implementation? Explain.

The ISA, because it's long-lasting and so if a mistake is made the only options are to live with it or to discard the ISA and all of its implementations (risking loss of customers).

Name Solution

Computer Architecture

EE 4720

Final Examination

8 May 2006, 10:00–12:00 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias DVD-HD or Blu-ray?

Good Luck!

Problem 1: The execution of code on a dynamically scheduled scalar MIPS implementation using Method 3 (the only one covered in class) is shown below. Beneath the diagram are signal labels, for example, `ID:dst`. (25 pts)

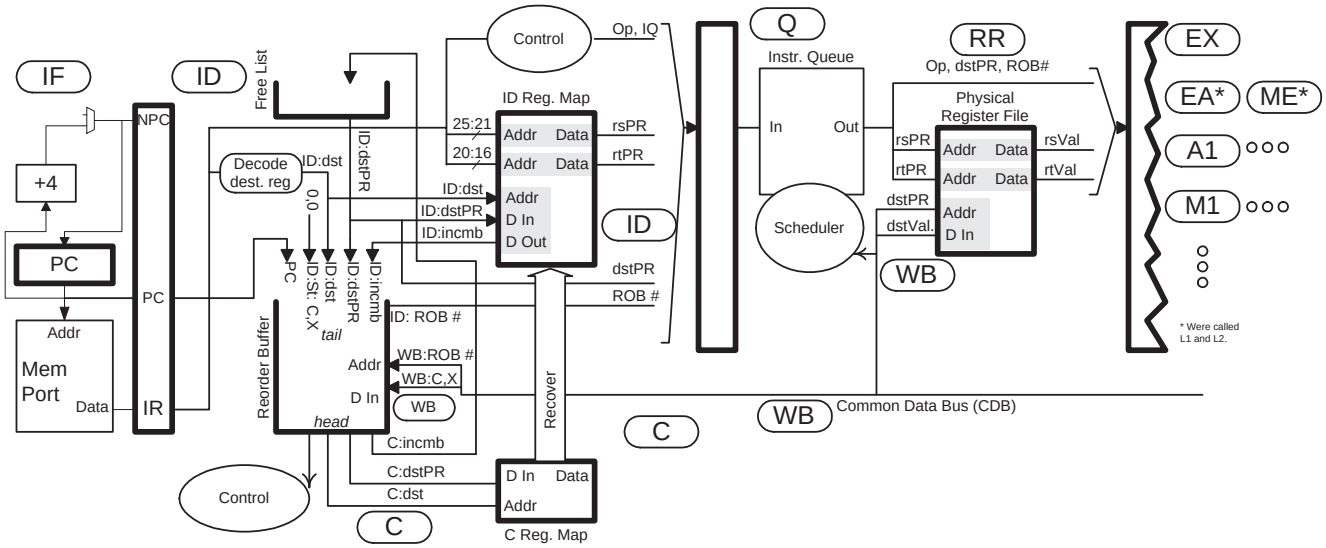
(*a*) On the next page show the values of the signals at the cycles they are used.

- Some values are shown, they are needed to determine other values.
- All values can be determined.
- Ignore the distinction between integer and FP registers.
- The free list contents at cycle zero is shown on the lower right-hand side of the figure on the next page.

Solution appears on next page in blue. The values of `ID:rsPR` in cycles 1 and 3 (appearing in the problem, not just the solution) tell us that register `f2` is initially mapped to physical register 56, `f4` is initially mapped to physical register 63, `r5` is initially mapped to physical register 30, and `r6` is initially mapped to physical register 54.

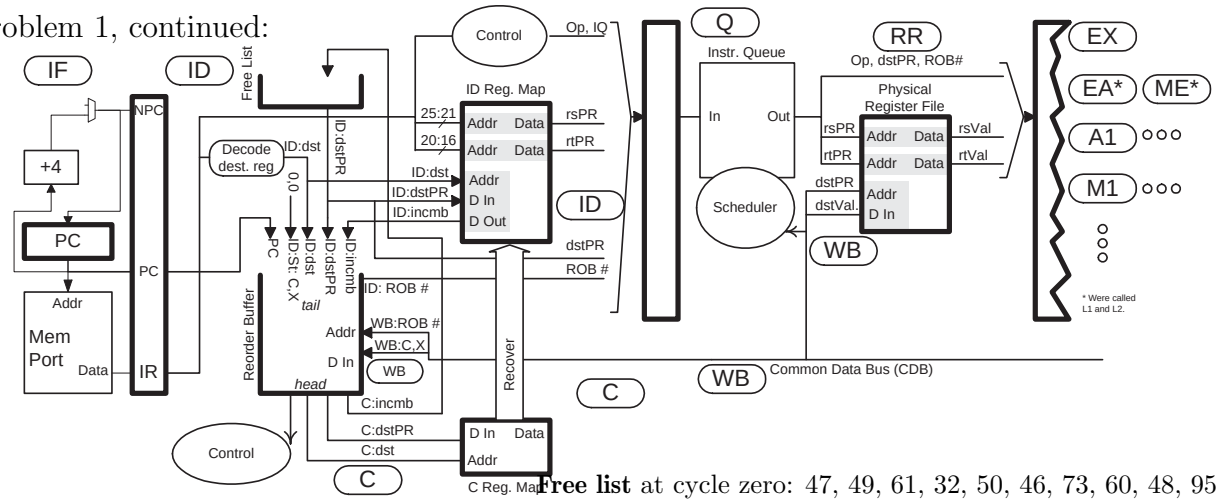
Because they are not written in the code fragment the mappings for `f4` and `r6` don't change. Registers `f2` and `r5` get new mappings, the physical registers are taken from the free list.

The `ID:dst` row shows an "r" or "f" in addition to the register number. In a real implementation there might be separate maps for integer and floating point registers. If not, a bit might be used to indicate whether the register is floating point.



Tables for answer on next page.

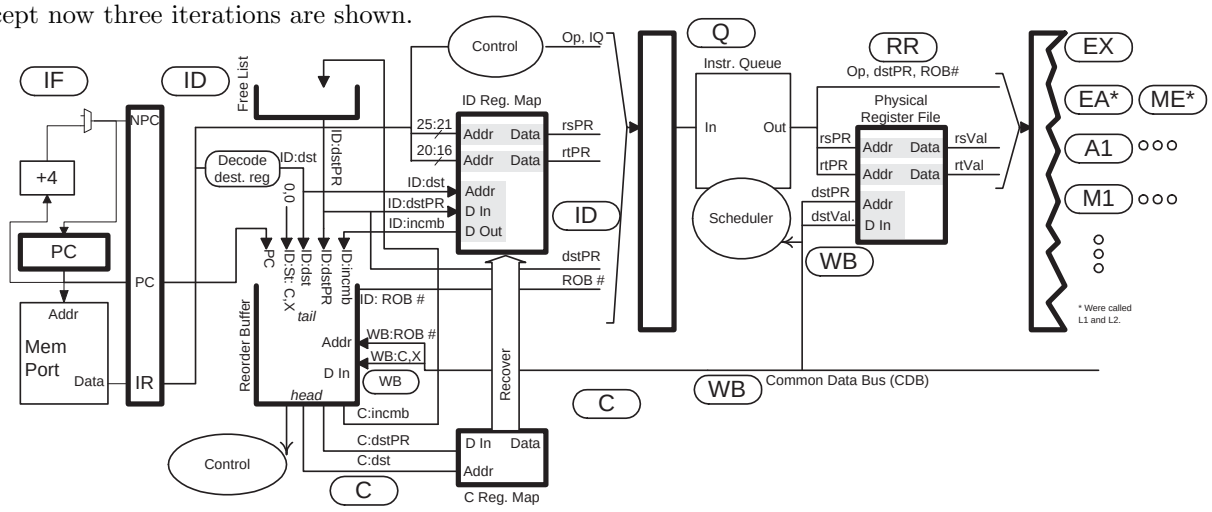
Problem 1, continued:



Free list at cycle zero: 47, 49, 61, 32, 50, 46, 73, 60, 48, 95

```
LOOP:              # First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
LOOP:              # Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:rsPR               56 30 30 47 49 49

ID:rtPR               63    54 63    54

ID:dst                f2    r5 f2    r5

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

ID:dstPR              47    49 61    32

ID:incmb              56    30 47    49

RR:rsPR                     56 30 30    47 49 49

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

RR:rtPR                     63    54    63    54

WB:dstPR                                  49 47       32 61

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

C:incmb                                          56    30    47    49

C:dstPR                                          47    49    61    32

C:dst                                            f2    r5    f2    r5

# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
```

Problem 1, continued: The diagram below is for the same code on the same system as the previous part, except now three iterations are shown.




```
LOOP:            #  First Iteration
 add.s  f2, f2, f4  IF ID Q  RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP         IF ID Q  RR B  WB          C
 sub r5, r5, r6           IF ID Q  RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Second Iteration
 add.s  f2, f2, f4           IF ID Q     RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                  IF ID Q     RR B  WB          C
 sub r5, r5, r6                    IF ID Q     RR EX WB          C
# Cycle:            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:            #  Third Iteration
 add.s  f2, f2, f4                    IF ID Q        RR A1 A2 A3 A4 WB C
 bgtz r5  LOOP                           IF ID Q  RR B  WB                C
 sub r5, r5, r6                             IF ID Q     RR EX WB             C
```

(*b*) What is the CPI for a large number of iterations? *Hint: It's not 1.*

☑ CPI?

The quick, common sense way to solve this problem is to note that each `add.s` will start right after the completion of the `add.s` in the previous iteration and the `add.s` will commit in the cycle after `WB` (since the `sub` and `bgtz` finish much faster), and so execution will proceed at a rate of four cycles per iteration or a CPI of $\frac{4}{3}$.

The regular way to determine CPI is to find a repeating pattern. Normally, we look for two cycles in which the first instruction of the loop is in `IF` and in which the state of the system is identical. That doesn't happen in the three iterations appearing above, and it won't happen for a while because at each iteration there are more instructions in flight.

An alternative way to find CPI is to look at the state of the system when the first instruction is in commit. In each case `add.s` in the following iteration is in `A2`. Assuming nothing interferes with the `add.s` in the next iteration starting execution on time, the processor will sustain a commit rate of four cycles per iteration. (Note that with the regular technique there is no need to make such assumptions.)

Therefore $\boxed{\text{CPI} = \frac{17-13}{3} = \frac{4}{3}}$.

Problem 1, continued:

(*c*) Suppose the ROB can hold 256 entries and there are an unlimited number of physical registers. Assuming a very large number of iterations, at about what cycle will fetch stall?

☑ Fetch will stall at cycle ≈

Instructions are being fetched at 1 instruction per cycle and committed at a rate of $\frac{3}{4}$ IPC. Let $n(c)$ denote the number of instructions in the reorder buffer at cycle c, then $n(c) = c - \frac{3}{4}(c-9)$ for $c \geq 9$. (Note that no instructions are committed until cycle 9.) Solving for c yields $c = 4n(c) - 27$, substituting $n(c) \to 256$ gives $\boxed{c = 997 \text{ cycles}}$.

(*d*) Would the code above run faster on a four-way superscalar dynamically scheduled system with the same clock frequency and pipeline depth? Explain.

☑ Circle One: Faster $\boxed{\text{Not Faster}}$.

☑ Because

Execution above is limited by the execution of the `add.s`. A superscalar system would fetch instructions faster but since the next `add.s` starts as soon as the previous one finishes, that won't help.

Problem 2: The code fragments below run on three systems which are identical except for the branch predictor: One uses a **bimodal** predictor with a 2^{14}-entry BHT, one uses a **local predictor** with an 8-outcome local history and a 2^{14}-entry BHT, and one uses a **global predictor** with an 8-outcome global history. (25 pts)

(*a*) Provide the information requested below.

- All accuracies are after warmup.
- For the warmup time show the approximate number of times the predicted branch needs to be executed before the predictor reaches its warmup accuracy. Assume the 2-bit counters start out at 0 or 3, whichever is worse.

```
BIG:addi r3, r0, 3
LP: bne r3, r0 LP  # Iterates four times.
    addi r3, r3, -1

    lw r1,0(r2)
C2: beq r1, 0 SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

☑ Bimodal: accuracy on C2: 60%

☑ Bimodal: warmup time on C2: 8 executions

☑ Local: accuracy of C2: 100%

☑ Local: warmup time on C2: $\approx 8 + 2 \times 5 = 18$ executions

☑ Local: smallest history size needed for 100% accuracy on C2: 5 (If LP ignored would be 4 outcomes.)

☑ Global: accuracy on C2: 80%

☑ Global: warmup time on C2: $\approx 1 + 6 + 2 = 9$ executions

☑ Global: smallest history size needed for 100% accuracy on C2: $4 \times 5 = 20$

See next page for discussion of solution.

Problem 2, continued:

The code below is repeated from the previous page with stuff added to show how to compute bimodal prediction accuracy.

```
BIG:addi r3, r0, 3
LP: bne r3, r0 LP  # Iterates four times.
    addi r3, r3, -1

    lw r1,0(r2)
# Comments show solution work for bimodal accuracy.
# Execution Num    # 0  1  2  3  4   5  6  7  8  9   10 11 12 13 14
C2: beq r1, 0 SK   # T  N  T  T  N   T  N  T  T  N   T  N  T  T  N ...
# Counter:          0  1  0  1  2  1   2  1  2  3  2   3  2  3  3  2  ...
# Prediction:       N  N  N  N  T  N   T  N  T  T  T   T  T  T  T  T
# Correct            x     x  x  x   x  x  x     x      x        x
    nop
    nop
SK: j BIG
    addi r1, r1, 4

# Global history, C2 outcomes in upper case, LP outcomes in lowercase.
# T tttn N tttn T tttn T tttn N tttn  T tttn N tttn T tttn T tttn N tttn  ...
```

General: Branch C2 has an outcome pattern of length 5: TNTTN.

A branch is warmed up when the 2-bit counters it uses reach the correct value. Bimodal is easiest since it uses just one counter. If the branch is highly biased it will take two executions to warm up a counter. Branch C2 is biased, but not highly so it will take more. More on bimodal further below. For the other predictors you need to compute how many counters they use (by finding the number of local or global history patterns used to predict them). See the discussion of those predictors, below.

Bimodal: The code above shows the 2-bit counter used by the bimodal predictor to predict branch C2. A repeating pattern is established after execution 9 (see the numbers in the diagram), in which the counter value is 2, it is also 2 after 14, and so the pattern will repeat. With 3 out of 5 correct the accuracy is 60%.

Bimodal just uses one counter. Since C2 is not highly biased it will take more than 2 executions to warm it up, from the diagram above it can be seen that it takes about 8.

Local: The 5-outcome pattern easily fits in the 8-outcome history so prediction accuracy is 100%. After 8 executions (the length of the local history) there will be 5 different local histories for C2: **TNTTNTNT**, **NTTNTNTT**, **TTNTNTTN**, **TNTNTTNT**, **NTNTTNTN**. At worst, each will take two executions to warm up, so the warmup time is $8 + 5 \times 2 = 18$ executions.

If ignoring the LP branch the smallest history size would be 4. Two would not be enough because **NT** can be followed by an **N** or **T**; three is not enough because **TNT** can be followed by an **N** or **T**. If the LP branch is considered then 4 is not enough because patterns **TTNT** and **TNTT** occur with both branches, LP and C2, and the subsequent outcomes differ. Five outcomes are sufficient.

Global: The first step in solving this is determining the global histories seen when making the prediction. Between each outcome of C2 there are four executions of LP (SK doesn't count because it's a jump, not a branch). The global history is shown in the bottom of the code above, the history used when predicting C2 is the eight outcomes just before an upper-case letter, for example, **ttn N tttn** and **ttn T tttn**. In fact, these are the only two distinct global histories see when predicting C2. Pattern **ttn N tttn** is seen twice, in both cases before a taken branch, so its predictions will always be correct. Pattern **ttn T tttn** is seen three times, once before a **T** outcome and twice before an **N** outcome, and so after warmup it will predict **N** and be correct $\frac{2}{3}$ of the time. The overall prediction accuracy is $\frac{1}{5}\left(2 \times 1 + 3 \times \frac{2}{3}\right) = \frac{4}{5}$.

A hand analysis shows that the counter for **ttn T tttn** warms up in 6 executions (assuming a worst-case start value of 3), being highly biased, the counter for **ttn N tttn** warms up in 2 executions. The patterns above will not be seen until after the first iteration, so the total warmup time is $1 + 6 + 2 = 9$ executions.

For 100% accuracy the global history must hold four outcomes of C2 (see the discussion for the local predictor). It takes 5 global outcomes to capture one C2 outcome, so for four C2 outcomes the global history length must be 20 outcomes.

Problem 2, continued:

(*b*) The code below is similar to the code on the previous page except the four-iteration loop has been replaced by two random branches. Each random branch will be taken with probability .5 and the outcome is independent of everything, including the other random branch.

```
BIG:lb r3, 0(r4)
    beq r3, r0  S2  # Random.
    lb r3, 1(r4)
S2: beq r3, r0  S3  # Random.
    addi r4, r4, 2
S3: nop

    lw r1,0(r2)
C2: beq r1, 0  SK   # T N T T N  T N T T N  T N T T N  T N T T N  ...
    nop
    nop
SK: j BIG
    addi r1, r1, 4
```

```
# Solution - Global History.
# Upper case is C2, r is first random branch, s is second random branch;
# r and s can be either taken or not-taken.
# T r s N r s T r s T r s N r s  r s T r s N r s T r s T r s N r s ..
```

☑ Global: accuracy on C2: 80%

☑ Explain using GHR values.

☑ Global: warmup time on C2: $\approx 3 + 2 \times 3 \times 2^6$ executions

☑ Explain using GHR values.

☑ Global: smallest history size needed for 100% accuracy on C2: $4 \times 3 = 12$

☑ Explain using GHR values.

This loop brings good news and bad news. The good news is that an eight-outcome global history will hold two C2 outcomes. The bad news is that it doesn't help, and warmup time is worse.

Accuracy: The global history patterns when predicting C2 are: **rsTrsTrs**, **rsNrsTrs**, and **rsTrsNrs**, where each **r** and **s** can be either taken or not taken. A pattern of the form **rsTrsTrs** occurs once and is always followed by an **N** so it predicts at 100%. Form **rsNrsTrs** occurs twice and is followed by an **N** or **T**, its accuracy will be discussed further below. Form **rsTrsNrs** occurs twice and is always followed by a **T**, so it predicts at 100%.

As the alert reader guessed, **rsTrsNrs** is being called a "form" because it represents $2^6 = 64$ patterns, one for each distinct outcome of the random branches. For forms **rsTrsNrs** and **rsTrsTrs** the only impact of this diversity is increased warmup time. Each pattern still takes at worst 2 outcomes to warm up, but each form represents 64 patterns.

For pattern **rsNrsTrs** there is also an impact on accuracy. If **r** and **s** were always taken, then the counter for **rsNrsTrs** would see outcomes **T N T N T ...**, and so the prediction accuracy would be 50% if the initial value of the counter were 0, 2, or 3; if the counter were 1 the accuracy would be 0%. Since **r** and **s** are random each counter sees a random subset of **T N T N T N**, since that subset is balanced between **T**'s and **N**'s, it is equally likely that the counters predict taken or not taken. (Actually, when

the next C2 outcome is `T` there is a tiny bias towards predicting `N`, and vise versa.) So the prediction is random, uniform (between `T` and `N`), so the prediction accuracy for form `rsNrsTrs` will be 50%. The overall prediction accuracy is then still 80%.

As stated above, the worst-case warmup time for the highly biased patterns is 2 outcomes each. Assume the same is true for `rsNrsTrs`, though it will actually be larger. If the random branches uncharacteristically did not repeat a pattern, the total warmup time for all three forms would be $3 + 3 \times 2^6 \times 2$, but that's unlikely. Since branches are random there is no precise warmup time, instead the solution will be to find the number of executions at which the probability that the 2-bit counter used for a prediction is warmed up is .05. Let x denote the number of times form `rsNrsTrs` appears, and suppose without loss of generality that we are predicting `ttNttTtt` (the $x + 1$ time the form appears). The probability that `ttNttTtt` is seen a particular time is $\frac{1}{64}$, the probability it's not seen a particular time is $1 - \frac{1}{64}$ and the probability it's not been seen in the past x executions is $\left(1 - \frac{1}{64}\right)^x$. The probability that it's been seen exactly once is $\left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64} x$. The probability that it's been seen less than 2 times is the sum, $\left(1 - \frac{1}{64}\right)^x + \left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64} x$. So to find the warmup time for this pattern solve $\left(1 - \frac{1}{64}\right)^x + \left(1 - \frac{1}{64}\right)^{x-1} \frac{1}{64} x = 0.05$ for x. Using a symbolic math program (Mathematica) that yields $x = 301.7$. So the total warmup time is 3×301.7. *Grading note: no one got it quite that close.*

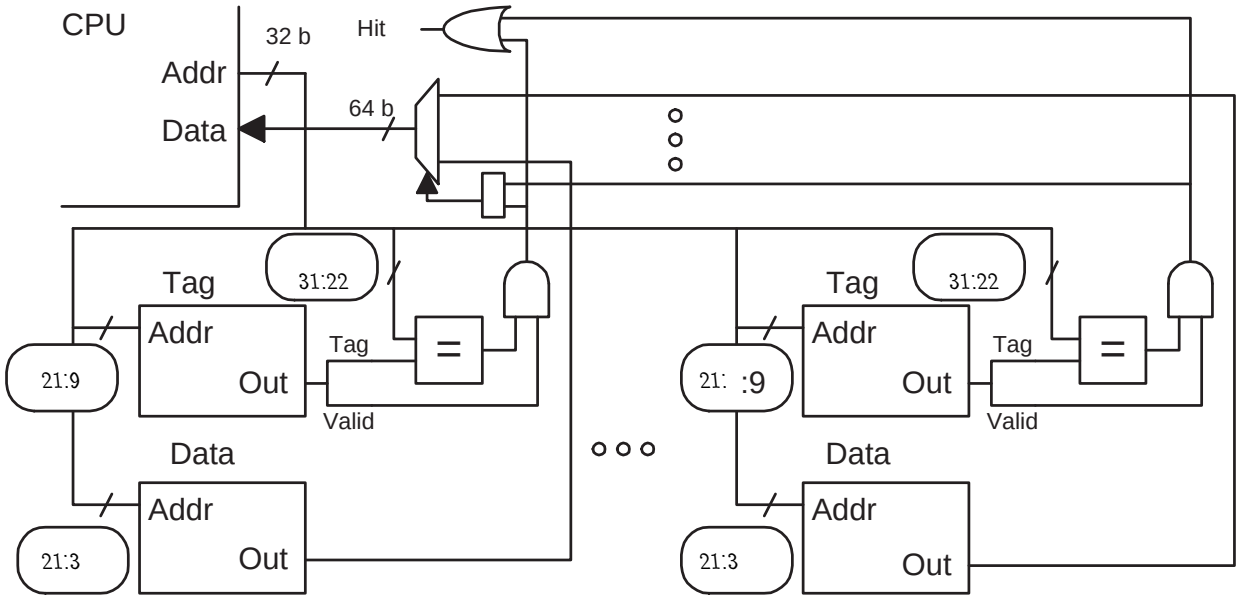
An acceptable answer to the problem would be a warmup time of $3 \times 2 \times 2^6$ executions, which is the boxed answer before the explanation.

The minimum history size has to be large enough to hold 4 C2 outcomes. It takes three global outcomes to hold one C2 outcome, so the minimum history size is 12 outcomes.

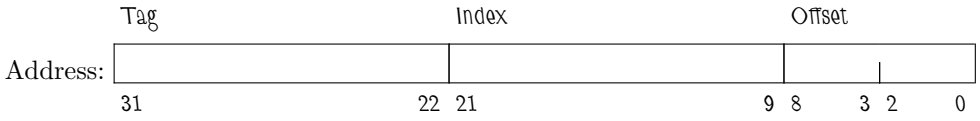
Problem 3: The diagram below is for a 64-MiB (2^{26}-character) 16-way set-associative cache on a system with the usual 8-bit characters. (20 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $16 \times 2^{22-9}\,(32 - 22 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^9 = 512$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Problem 3, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below? Explain

```
double sum = 0.0;
char *a = 0x2000000;    // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;  // = 2^27

for(i=0; i<ILIMIT; i++) sum += a[ i ];
for(i=0; i<ILIMIT; i++) sum += a[ i ];
```

The line size is $2^9 = 512$ characters and the size of an array element is one character. Therefore each miss will be followed by 511 hits. The hit ratio for the first loop will be $\frac{511}{512}$. Will `a[0]` still be cached when the second loop starts? The cache capacity is given as 2^{26} characters and the first loop accesses 2^{27} characters, so they can't all fit. The cache will replace the least-recently used line within a set, and so `a[0]` won't be cached when the second loop starts. In fact, nothing brought in by the first loop is used in the second loop so the overall $\boxed{\text{hit ratio is } \frac{511}{512}}$.

(*c*) The slightly different code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

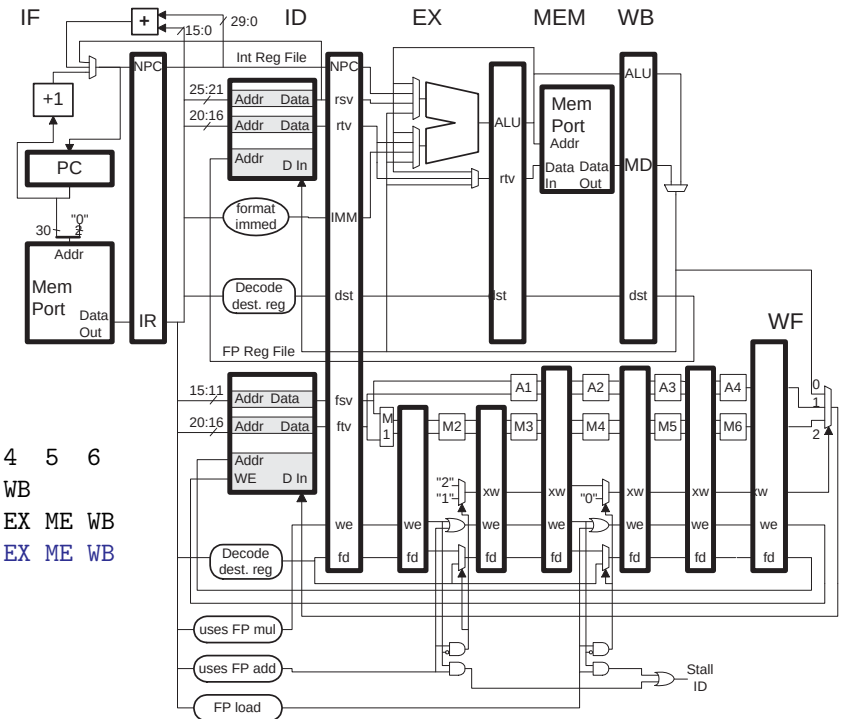
☑ Find the hit ratio. Very briefly explain.

```
double sum = 0.0;
char *a = 0x2000000;  // sizeof(char) = 1 character.
int i;
int ILIMIT = 1 << 27;

for(i=0;         i<ILIMIT; i++) sum += a[ i ];
for(i=ILIMIT-1; i>=0;     i--) sum += a[ i ];
```

The only difference here is that the second loop iterates backward. The last element accessed by the first loop is $\texttt{a}[2^{27}-1]$, which is also the first element accessed by the second loop and so the second loop will start out with a hit (and 511 more while it consumes the line). Since the cache can hold half the array, half of the execution of the second loop will have a 100% hit ratio. The overall $\boxed{\text{hit ratio is } \frac{3}{4}\frac{511}{512} + \frac{1}{4}\frac{512}{512}}$.

Problem 4: Answer each question below.

(*a*) The execution of some code fragments on the illustrated implementation appear below. Explain why each execution is impossible and show a corrected pipeline execution diagram. (7 pts)




```
# Cycle          0  1  2  3  4  5  6
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r3      IF -> ID EX ME WB
Fix:                IF ID -> EX ME WB
```

☑ Fix.

☑ Was impossible because:

The `add` did need to stall so the `r2` value could be bypassed from the `lw`. However there is no way the control logic could know this in cycle 2 because the `add` had not yet arrived.

```
# Cycle          0  1  2  3  4  5  6  7
lw r2, 0(r4)     IF ID EX ME WB
add r1, r2, r3      IF ID -> EX ME WB
xor r5, r6, r7         IF ID -> EX ME WB
Fix:                   IF -> ID EX ME WB
```

☑ Fix.

☑ Was impossible because:

In cycle 3 the `ID` stage holds two instructions, `add` and `xor`. It only has room for one.

```
add.d f0, f2, f4  IF ID A1 A2 A3 A4 ME WB
Fix:              IF ID A1 A2 A3 A4 WF
```

☑ Fix.

☑ Was impossible because:

Floating point instructions do not pass through the `ME` stage, and they use `WF` for writeback.

(*b*) For each feature below indicate whether it is usually a feature of the ISA or the implementation, and explain why it's not a feature of the other. (7 pts)

Feature: *The opcode can be found in bits 31:26.*

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA should define everything you need to know to write a program (or to write a compiler). That would have to include instruction coding, including the position of the opcode field. If it were in the implementation then a compiler would have no idea where to place the opcode and different implementations could expect the opcode bits in different positions, so there would be no compatibility.

Feature: *The* `add` *in the code below will stall for one cycle.*

```
lw  r1, 0(r2)
add r3, r1, r4
```

☑ ISA or [Implementation]?

☑ Why not the other?

The ISA in principle could specify that the `add` should stall, but that would overly constrain implementations. If an implementation technique were developed that would allow the `add` execute without a stall it could not be used in the implementation of this ISA.

Feature: *Two consecutive delayed (as in MIPS) branches will yield unpredictable results.*

```
bneq r1, r2 DEST1
beq r3, r4 DEST2
addi r5, r6, r7
```

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA specifies what programs do. Normally, an ISA would either say such pairs are illegal, in which case an implementation might raise an exception on such code, or it might say such pairs are fine (just execute one instruction at `DEST1` then continue at `DEST2`), in which case the implementation must handle them properly. Leaving it up to the implementation would break compatibility because each might handle such pairs differently.

Feature: *Integer addition overflow raises exception.*

☑ [ISA] or Implementation?

☑ Why not the other?

The ISA specifies the behavior of the program, including whether instructions raise exceptions. If it were up to the implementation then code would not be portable.

Note that the question is about whether overflow raises an exception, it is not about what the exception handler should do. The handler is part of the operating system, not the ISA or implementation.

(*c*) Consider the use of a packed-operand 8-bit add (of the type described in class) to speed up the code fragments below. For each fragment indicate whether it's certainly feasible, feasible with certain assumptions, or not feasible. (6 pts)

```
extern char *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☑ Circle One: No. Yes! Yes, assuming ...

☑ Explain.

Yes assuming that saturating arithmetic is okay. That is, if the program expects $120 + 10 = -126$ (because of overflow), then saturating arithmetic would break the program. If, on the other hand the program works correctly with $120 + 10 = 127$ (because of saturation), then packed operand instructions should work fine.

```
extern int *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

☑ Circle One: No. Yes! Yes, assuming ...

☑ Explain.

Assuming that despite the use of integers, array elements are in the range $[-128, 127]$ and a sum is also in the range or that saturation is okay.

```
extern char *a, *b, *c;
for(i=1; i<1024; i++) a[i] = a[i] + a[i-1];
```

☑ Circle One: No. Yes! Yes, assuming ...

☑ Explain.

To compute `a[i]` element `a[i-1]` is needed so there is no easy way to do this in parallel with packed operand instructions.

(*d*) MIPS and other RISC ISAs typically have instructions using displacement addressing but lack register deferred addressing. (See the examples below.)

```
lw r1, 4(r2)   # Displacement addressing.
lw r1, (r2)    # Register deferred addressing.
```

Given that RISC and CISC ISAs have displacement addressing:

☑ (5 pts) Why is register deferred addressing helpful for CISC but not helpful for RISC?

Register deferred addressing lacks the displacement that is included in the displacement addressing already present in the two ISAs, they are otherwise identical. Because CISC instructions have variable length, an instruction with register deferred addressing can be shorter than one with displacement addressing, making programs smaller. In RISC all instructions are the same size so there is no advantage in omitting the displacement.

(*e*) Dead-code elimination (DCE) is a common compiler optimization.

☑ (5 pts) What is it? Illustrate using an example.

The removal of code that's either never executed or that produces values that are never used. See the example below.

```
a=1; // This line would be removed by DCE.
a=2;
```

65 Fall 2005 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 24 October 2005, 12:40–13:30 CDT

Problem 1		(50 pts)
Problem 2		(50 pts)
Exam Total		(100 pts)

Alias New Orleans Saints

Good Luck!

Problem 1: The partially completed routine on the next page is called with the address of a string which may contain characters that look like letters, for example, "`g0!1y`." The routine should replace those characters with the letters they look like; the example would be converted to "`golly`."

The string DMAP (see the next page) specifies the look-alikes. It specifies a letter (always lower case) followed by one or more look-alike characters followed by a comma (possibly followed by another group). For example, "`o0,l1!,`" indicates that the letter oh's look-alike is the digit zero and the letter el's look-alikes are the digit 1 and an exclamation point.

The routine on the next page is almost finished. The part at the end uses a look-up table to translate the string, and the part at the beginning has started setting up the look-up table. Write the code that finishes setting up the look-up table based on DMAP. [50 pts]

☑ Write the code that sets up the look-up table.

☑ The code must be reasonably efficient.

☑ The only synthetic instructions that can be used are `nop` and `la`.

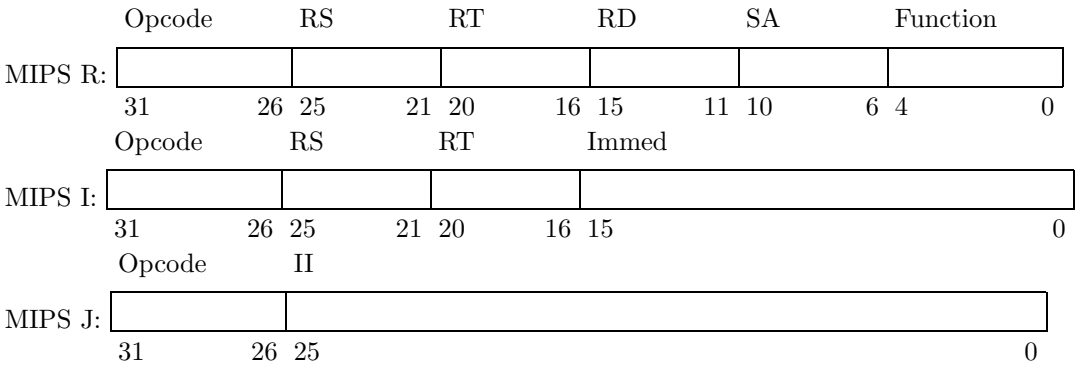
Solution on next page.

Grading Notes: To check for a comma in, say, `t4` many solutions used a `sub t3,` $t4$`,t5` followed by something like `beq` $t3$`,0, COMMA` (remember that `t5` held the comma character). The easier way to do it is just `beq` $t4$`,t5, COMMA`.

Use the next page for the solution.

```
        ## Register Usage
        #
        # $a0: Procedure call argument. Address of string to translate.
        # There are no return values.

DMAP:   .asciiz "o0,l1!,c([,t+,s$,"  # Translate 0->o, 1->l, !->l, (->c, etc.

LUT:    .space 256

        la $t0, LUT
        addi $t1, $0, 255

        # First, initialize look-up table so every character is mapped to itself.
        #
LOOP0:  add $t3, $t0, $t1         # Compute address of LUT entry.
        sb $t1, 0($t3)            # Write default entry. (A->A, B->B, etc.)
        bne $t1, $0  LOOP0
        addi $t1, $t1, -1

        # Next, set up look-up table based on DMAP string. (Finish in solution.)
        #
        la $t0, DMAP
        la $t1, LUT
        addi $t5, $0, 44   # ',' Comma character separates groups.

        # Start solution here. (Can be done in 9 insn.)

        # Solution Starts Here -- Solution Starts Here -- Solution Starts Here
LOOP1:
        lb $t4, 0($t0)
        beq $t4, $0 EXIT
        addi $t0, $t0, 1

LOOP2:
        lb $t3, 0($t0)
        beq $t3, $t5 LOOP1
        addi $t0, $t0, 1
        add $t3, $t3, $t1
        j LOOP2
        sb $t4, 0($t3)
EXIT:
        # Solution Ends Here -- Solution Ends Here -- Solution Ends Here


        # Use the look-up table to translate the string.
        la $t1, LUT
LOOP3:  lb $t2, 0($a0)       # Load character of string.
        beq $t2, $0, EXIT2
        add $t2, $t2, $t1    # Compute address of look-up table entry.
        lb $t2, 0($t2)       # Load translated character.
        sb $t2, 0($a0)       # Write translated character to string.
        j LOOP3
        addi $a0, $a0, 1

EXIT2:  jr $ra
        nop
```

Problem 2: Answer each question below. The instruction format descriptions are provided for reference.



(*a*) A proposed new MIPS branch instruction compares a register value to a constant to determine if the branch should be taken. For example, the branch below is taken if the contents of `t1` is 123. [10 pts]

```
beqi $t1, 123,  TARGET
nop
```

☑ Why isn't it feasible to code such an instruction using any of the existing MIPS formats?

Because that instruction uses two constants, the value to compare (`123`) and the branch target (`TARGET`). Since none of the three MIPS formats has two immediate fields it can't be coded. One could split the immediate field in Format I in half, use 8 bits for the value and 8 bits for the displacement, but then it would not be correct to say `beqi` used Format I. This is more than being picky about words because the fields in the format correspond to data paths in the implementation and so adding or modifying fields means adding or modifying hardware in the implementation.

☑ How could a conditional control transfer instruction that compared a register to an immediate be coded using the MIPS formats? (The instruction would be different than `beqi`.)

Put the branch target in a register. For example, `beqir t1, 123, t2`, in which the contents of `t1` is compared to `123`, if they are equal branch to the address in register `t2` (after the delay slot). Like `jr` and `jalr` the target address is in a register. This instruction would be coded in Format I: `beqir rs, immed, rt`.

☑ Show the instruction in assembly language.

See above.

(*b*) Show the results of addition for the 32-bit data types shown below. [10 pts]

☑ Unsigned Integer:

```
  0x0999
+ 0x0109
  0x0aa2
```

☑ BCD:

```
  0x0999
+ 0x0109
  0x1108
```

☑ Packed 4-bit integers with saturating arithmetic.:

```
  0x0999
+ 0x0109
  0x0a9f
```

(*c*) The MIPS ISA specifies that the sa (shift amount) field in the `add` instruction must be zero (an `add` instruction with a non-zero sa field value should cause an execution error). [10 pts]

☑ Describe a difficulty that might have arisen if the MIPS ISA had specified that implementations should ignore the sa field in an `add` instruction (and so the `sa` field could contain any value).

If implementations ignored the field then programmers would be free to put any value in the field without changing what the `add` does. Because programmers might do this a later version of the ISA could not use the field for an extended opcode or for an improved add instruction because existing code expects the `add` to be an add, regardless of the value in the field. As an extended opcode field, a non-zero value in sa wold specify completely different instructions, say a 1 in sa would specify a andn (and-not) instruction. An improved add instruction might use the sa field as a shift amount to be applied to the second operand (called a scaled add).

☑ Describe a difficulty that might have arisen if the MIPS ISA said nothing about the `sa` field in an add instruction.

The same difficulty as the previous case. If the ISA says nothing then some implementors would assume ignoring the field is okay and programmers for those implementations might put values in the field.

☑ Describe a difficulty that might have arisen if the MIPS ISA did specify that sa must be zero but did not say what should happen if the field were non-zero.

Not a very good ISA, since it's not fully describing the behavior of the `add` instruction. If some implementors chose to ignore non-zero values then there would be the same problem as the previous cases.

(*d*) Consider a benchmark suite that's similar to SPECcpu in that the tester is responsible for compiling the programs but that unlike SPEC, specifies that the tester should not use any optimizations when compiling the benchmarks. [10 pts]

☑ Compared to SPECcpu base and peak (result) scores, how useful would such results be? Explain.

Not very useful since there is no reason not to optimize code. A machine that is outperformed by its competitors on optimized code but outperforms its competitors on unoptimized code might be doing something clever, but also unnecessary since optimizing code is no big deal.

(*e*) Using a new compiler optimization a program's dynamic instruction count is cut in half. The values for CPI, IC, and ϕ are available for a run of the program compiled without the new optimization. Other than the instruction count, nothing has been measured for the program using the new optimization. How useful is the CPU performance equation for estimating the run time with the new optimization on the same system in this situation? Explain. [10 pts]

The CPU performance equation, $T = \frac{1}{\phi} \times \text{IC} \times \text{CPI}$, would not be very useful in estimating the run time. After the optimization we know IC (because that's given: half the previous value) and ϕ (its the same machine so it would be the same), but we don't know CPI. Since for a given machine CPI depends upon the instruction mix and arrangement we would expect them to change by a substantial amount with the new optimization because the new optimization had such a big effect on instruction count.

Name Solution

Computer Architecture
EE 4720
Final Examination

13 December 2005, 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(17 pts)
Problem 4		(15 pts)
Problem 5		(33 pts)
Exam Total		(100 pts)

Alias Out-of-order graduation?

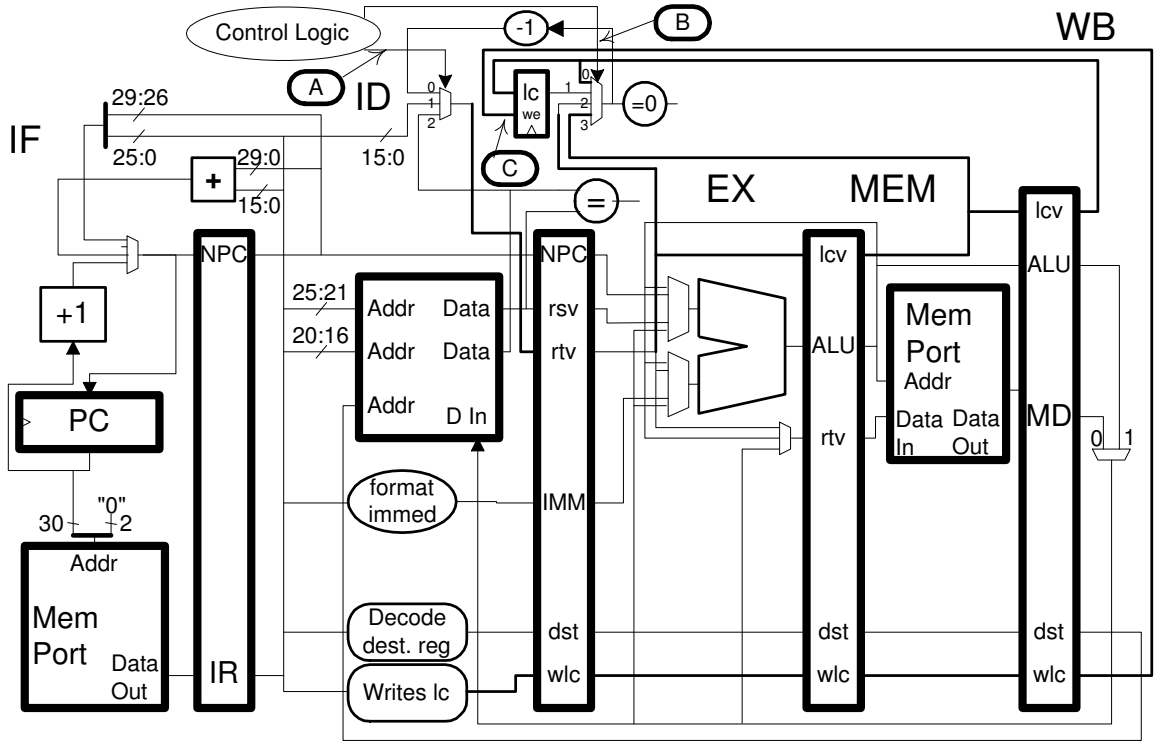
Good Luck!

Problem 1: The implementation of a version of MIPS with loop count instructions is shown below, these are the same instructions used in Homework 5. Instruction `mtlc rt` moves the contents of the `rt` register into a special loop count register, `lc`, and instruction `mtlci immed` moves a 16-bit immediate, `immed`, into `lc`. The loop-counted branch `bclz` is taken unless the `lc` register is zero, it also decrements `lc`. (In the diagram the upper input to the `lc` register is the data input and the lower input, `we`, is a write enable.)

Three wires in the implementation are labeled (A , B , and C). Show the values present on those wires for each cycle of the illustrated execution of the program in the space provided. Leave a value blank if the value has no effect, assume that instructions before and after the illustrated code are nops. Note that two iterations of the loop are shown. (15 pts)

```
 # Cycle            0    1    2    3    4    5    6    7    8
 mtlci 100          IF   ID   EX   ME   WB
LOOP:
 bclz LOOP               IF   ID   EX   ME   WB
 add r2, r2, r3               IF   ID   EX   ME   WB
# (2nd iteration)
 bclz LOOP                         IF   ID   EX   ME   WB
 add r2, r2, r3                         IF   ID   EX   ME   WB
 # Cycle            0    1    2    3    4    5    6    7    8

 A:                      1    0    2    0    2    0*   2*   0*   <- SOLUTION
 B:                           2         3         3*        3*   <- SOLUTION
 C:                 0    0    0    0    1    1    0    1    0    <- SOLUTION

 # Cycle            0    1    2    3    4    5    6    7    8
```

See next page for discussion of solution.



Discussion of solution on previous page.

The loop count branch examines the `lc` register (or a bypassed value) when it is in the ID stage. (If it examined it in a later stage the first branch would have to stall.) Control signal B selects whether the `lc` register itself is used, 1, or a value in EX, 2, in MEM, 3, or in WB, 0. Register `lc` is updated by `bclz`, `mtlc`, and `mtlci`, but to maintain precise exceptions the update is done in the WB stage. The new value for the `lc` register passes through the existing `ID/EX.rtv` latch, then goes through two new latches, `EX/MEM.lcv` and `MEM/WB.lcv`. Wire A controls an ID-stage mux that selects which value to send to the `ID/EX.rtv` latch. A 2 on A selects the rt value, this would be used for `mtlc` and also for existing type R and store instructions. A 1 selects the immediate, for `mtlci`, and a 0 selects the decremented `lc` value, for `bclz`. Signal C is the write-enable for the `lc` register.

For the `mtlci` instruction A is set to 1 in cycle 1 to select the immediate. The value for B in cycle 1 is left blank because the value of `lc` is not being used. When `mtlci` reaches writeback, in cycle 4, C is set to 1, writing the immediate to lc.

Signal C is set to 1 when the `bclz` instructions reach writeback, cycles 5 and 7 and should be 0 at other times to avoid overwriting a good value. That is, in a correct solution a blank could not appear anywhere for C.

When `bclz` is in ID signal A is 0 to select the decremented value for writeback, that's in cycles 2 and 4. In cycle 2 B is set to 2 to bypass the `lc` value from EX (written by `mtlci`); in cycle 4 B is set to 3 to bypass the `lc` value from MEM (written by the previous `bclz`). If the loop proceeds for another two iterations B would be set to 3 in cycles 6 and 8. All other values for B should be blank because the output of the mux will be ignored in those cycles.

The `add` instruction is in ID in cycles 3 and 5 at which time A is set to 2 to pass the rt register value. If the loop continues A is 2 in cycle 7 and 0 in cycles 6 and 8.

Problem 2: Complete the pipeline execution diagrams for the different MIPS implementations below. Pay attention to the type of implementations in each part, they're not all the same. Don't forget to **check for dependencies.**

☑ (5 pts) Scalar (not superscalar), statically scheduled, as illustrated in the previous problem.

```
# SOLUTION
#
# Cycle           0  1  2  3  4  5  6  7
 add r1, r2, r3   IF ID EX ME WB
 lw r4, 6(r1)        IF ID EX ME WB
 xor r7, r4, r8         IF ID -> EX ME WB
# Cycle           0  1  2  3  4  5  6  7
```

In all of the parts below the floating-point multiply unit has four stages and is fully pipelined, the stage labels are *M1-M4*. *The floating-point add unit is two stages and fully pipelined, the stage labels are* *A1 and A2*.

☑ (5 pts) Scalar, statically scheduled, full set of bypass paths.

```
# SOLUTION
#
# Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12 13
 mul.d f2, f4, f6      IF ID M1 M2 M3 M4 WF
 add.d f8, f2, f10        IF ID -------> A1 A2 WF
 mul.d f2, f4, f14           IF -------> ID M1 M2 M3 M4 WF
 sub.d f12, f2, f10                      IF ID -------> A1 A2 WF
# Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Common mistake:

Many solutions showed a MEM stage. The MEM stage is only used in the integer pipeline.

Problem 2, continued:

☑ (5 pts) Two-way superscalar, statically scheduled, full set of bypass paths. No alignment restriction on instruction fetches.

```
# SOLUTION
#
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
mul.d f2, f4, f6      IF ID M1 M2 M3 M4 WF
add.d f8, f2, f10     IF ID ----------> A1 A2 WF
mul.d f2, f4, f14        IF ----------> ID M1 M2 M3 M4 WF
sub.d f12, f2, f10       IF ----------> ID ----------> A1 A2 WF
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Common Mistake:

Some did not show a stall for the second multiply. There is room in ID for the second multiply in cycles 2-5, but it would require more complex control logic to accommodate it (since the instructions in ID would be out of order) and so the default behavior in this class is a stall.

☑ (5 pts) Two-way superscalar, dynamically scheduled, with a full set of bypass paths. No alignment restriction on instruction fetches.

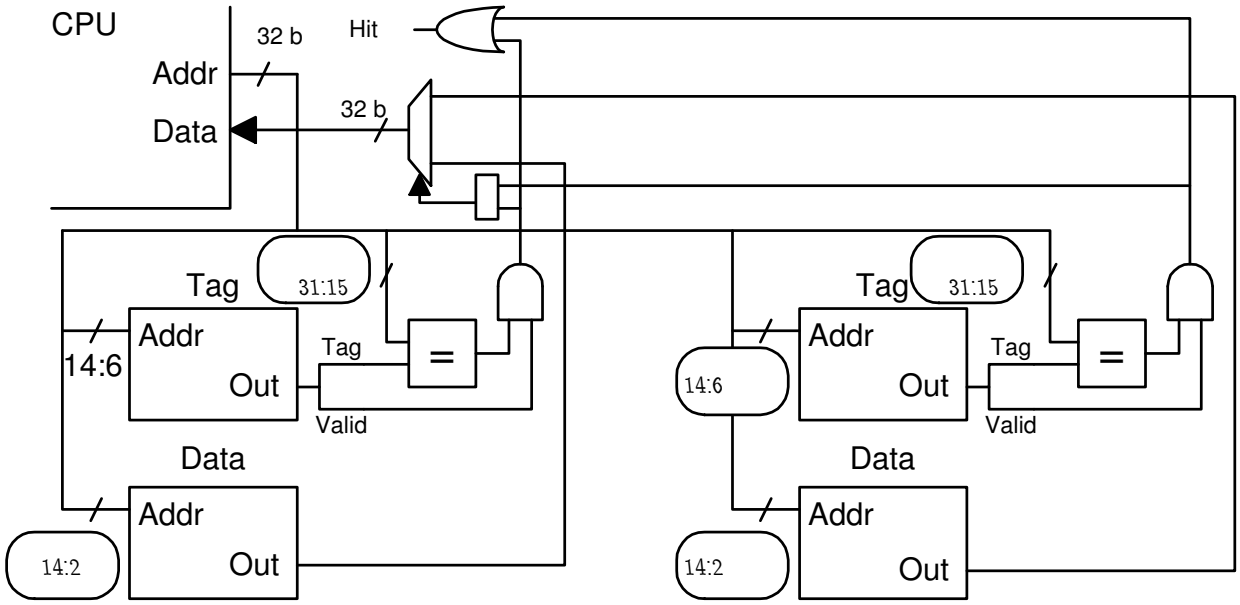
```
# SOLUTION
#
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
mul.d f2, f4, f6      IF ID Q  RR M1 M2 M3 M4 WB C
add.d f8, f2, f10     IF ID Q              RR A1 A2 WB C
mul.d f2, f4, f14        IF ID Q  RR M1 M2 M3 M4 WB    C
sub.d f12, f2, f10       IF ID Q              RR A1 A2 WB  C
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Not-too-common-but-more-common-that-it-should-have-been-mistake: Some solutions showed the second multiply waiting in one way or another for the add. There is no reason for the wait, this is a dynamically scheduled processor.

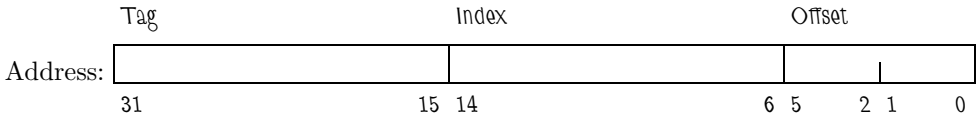
Problem 3: The diagram below is for a cache on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Cache Capacity (Indicate Unit!!):
Capacity is 2×2^{15} characters (64 kiB).

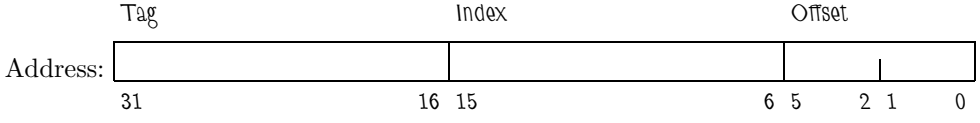
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $2 \times 2^{15-6}\,(32 - 15 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^6 = 64$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 3, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ (5 pts) What is the hit ratio running the code below?

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j;

for(j=0; j<3; j++)
  for(i=0; i<32; i++)
    sum += a[ i ];
```

The code above sequentially accesses elements of size 8 characters and so there are $64/8 = 8$ elements per line. On the first `j` iteration the first access to an element on a line will miss and the others will hit, for a hit ratio of $\frac{7}{8}$. Only four lines will be used which is much smaller than the cache capacity and so on the second and third `j` iterations all access will hit. The overall hit ratio is $\boxed{\frac{7+8+8}{8+8+8} = \frac{23}{24} = 0.95833333}$.

(*c*) The slightly different code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ (5 pts) Set `ILIMIT` to the smallest value that will fill the cache.

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i;

int ILIMIT = 1 << 13; // 1 << 13 == 8192     <- SOLUTION


for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

The cache capacity was found to be 2^{16} characters. The code above starts on a nice round address (many low bits are zero) and accesses data sequentially and therefore does not waste any space on a line. Each array element is 2^3 characters and so the cache will be exactly filled if $\frac{2^{16}}{2^3} = 2^{13} = 8192$ elements are accessed. (For those rusty on their C operators, `<<` is the left-shift operator.)

Problem 4: Answer each question below.

(*a*) An n-way superscalar processor need not have n copies of everything that an ordinary scalar implementation has.

☑ (5 pts) Which of the following is it most important that an n-way superscalar processor **does** have n of (explain):

- Mem-stage memory ports.
- Write ports to the register file.
- Floating-point adders.

Most important: Write ports to the register file.

Suppose a 2-way statically scheduled superscalar processor had one memory port. Then it would stall whenever a fetch group had two memory instructions. Similarly, such a system with two write ports to the register file would stall whenever a fetch group had two instructions that write the register file. Since more instructions write the register file (in fact, most of them) than use the MEM-stage memory ports it's better to have two register write ports. The same argument can be made for the floating-point adder.

The answer would be the same for a dynamically scheduled system but the argument would be slightly different.

(*b*) In the dynamically scheduled MIPS implementation covered in class:(5 pts)

☑ When is a physical register allocated (removed from the free list and assigned to an instruction)?

When an instruction is in ID.

☑ When is the physical register put back in the free list. Show an example that includes register numbers.

When the next instruction that writes the same architected register commits. In the example below a physical register (say `p123`) is assigned to the `add` when it is in ID, in cycle 1. The `xor` is the next instruction that writes the same architected register as the `add`, `r1`. When `xor` commits, in cycle 8, register `p123` is put back in the free list.

```
# Cycle           0  1  2  3  4  5  6  7  8
 add r1, r2, r3   IF ID Q  RR EX WB C
 sub r9, r1, r10     IF ID Q  RR EX WB C
 xor r1, r4, r6         IF ID Q  RR EX WB C
# Cycle           0  1  2  3  4  5  6  7  8
```

(*c*) Describe the contents of an Itanium (IA-64) bundle or a bundle in a typical VLIW ISA.

☑ (5 pts) Bundle Contents

In the Itanium ISA a bundle contains three 41-bit instructions and 5 template bits. The template bits specify whether the instructions are dependent and also the functional units that they will use.

For a typical ISA a bundle contains several instructions (three is a popular number) along with some dependency and possible other information about the instructions.

Problem 5: Answer each question below.

(*a*) When a company wants to benchmark a new computer using SPECcpu it gets the benchmark suite from SPEC, compiles the code, runs the benchmarks, and proudly publishes the results. (The last step is optional.)

Below are two variations on this procedure, for each explain how the results might differ **from those obtained using the usual SPEC procedure**, and explain how useful the results would be to the typical user of SPECcpu results. (7 pts)

The key difference between the scenarios below and the standard testing procedure is in who chooses the compiler and in who runs the compiler; normally the company does both. The choice of compiler is important because the full potential of the processor may only be realizable with specially tailored compiler optimizations and code generation. It is in the company's interest to develop such a compiler and its development should go hand-in-hand with developing the processor implementation used in the new computer. SPEC's interest is in having useful benchmark results so if they did choose a compiler they would try to pick one suitable for the computer, but they lack the company's expertise and they certainly would not take the trouble to write one. (In reality they would ask the company to provide one, but in this problem that doesn't happen.) So, if SPEC were to choose the compiler it may not choose one best suited to the processor and so the results may not be as good.

Users who compile their own code would probably use the company's compiler to get the best results and so procedure (1) would not be very realistic.

In procedure (2) the company's compiler is used but SPEC is doing the compilation. The company may have had well-motivated experts doing the compilation who would have gotten better results than SPEC's not-as-expert and not-as-motivated people. The base scores might not differ because only basic optimizations should be used and so SPEC and the company would compile identically. Whether the peak ("result") results would be better than procedure (1) depends on how important the compiler is. (Either answer would be correct if properly justified.) The results would certainly be more useful than (1) for users since if more closely reflects how they would develop code.

(1) SPEC gives a compiler (in addition to the usual material) to the company and allows the company to compile and run the suite following the usual SPECcpu rules (but using SPEC's compiler).

☑ Compared to actual SPEC rules and to (2), how might the results differ? Explain.

Not as high as actual SPEC or (2) because the company can produce a much better compiler. (See discussion above.)

☑ Compared to actual SPEC rules and to (2), how useful would results be? Explain.

Not as useful as actual SPEC or (2) because real users would use company's compiler. (See discussion above.)

(2) The company gives the new computer and a compiler to SPEC and SPEC compiles and runs the benchmark suite using the usual SPECcpu rules.

☑ Compared to actual SPEC rules and to (1), how might the results differ? Explain.

The results would be better than (1) because a better compiler is used but not as good as actual SPEC because tuning not as good as the company's.(See discussion above.)

☑ Compared to actual SPEC rules and to (1), how useful would results be? Explain.

Results would be more useful than both (1) and actual SPEC because they were obtained by skilled people but not super-motivated experts, so one might expect to get similar performance with reasonable skill and expertise.(See discussion above.)

(*b*) There is much less advantage in unrolling one of the loops below when the code is to run on a two-way statically scheduled superscalar MIPS implementation. *Note: The original problem did not mention the code was to run on a superscalar implementation.*(6 pts)

☑ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_A:
 lw r1, 0(r2)
 lw r1, 0(r1)
 add r3, r1, r3
 addi r2, r2, 4
 bneq r9, r0 LOOP_A
 addi r9, r9, -1
```

Unrolling loops provides more flexibility to schedule away stalls. In the code above the stalls are between the first two loads and the load and the add. These can be scheduled away on a scalar implementation but not on a 2-way superscalar.

Unrolling only works if the only dependencies between iterations are carried by things like steadily increasing loop indices, such as those carried by `r2` and `r9` in the loop above. To unroll the loop twice two iterations of the original loop are made one iteration of the new loop; that's shown below. On a scalar system all stalls are eliminated, the 2-way superscalar system would still have stalls so it would have to be unrolled four times.

```
# SOLUTION
 addi r9, r0, 1024
LOOP_AU:
 lw r1, 0(r2)
 lw r10, 4(r2)
 lw r1, 0(r1)
 lw r10, 0(r10)
 add r3, r1, r3
 add r13, r10, r13
 addi r2, r2, 8
 bneq r9, r0 LOOP_AU
 addi r9, r9, -2
 add r3, r3, r13
```

☑ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_B:
 lw r1, 0(r2)
 add r3, r3, r1
 lw r2, 4(r2)
 bneq r9, r0  LOOP_B
 addi r9, r9, -1
```

There is not much benefit in unrolling the loop above because `r2` used in the beginning of an iteration is produced by a `lw` from the previous iteration so there is no way two iterations could overlap.

(*c*) Which are smaller, RISC programs or CISC programs. Provide some instruction examples to illustrate your answer. (5 pts)

☑ Example and explanation.

CISC programs are smaller because their instructions, being variable size, do not waste space. Also, instructions can do more such as accessing operands from memory, doing arithmetic, and storing the result back in memory. In the example below one CISC instruction does the work of four RISC instructions. Three of the RISC instructions waste 16-bits on an unused immediate. In additional the RISC code uses four opcodes and uses space for 9 register numbers, while the CISC code uses one opcode and space for three registers.

BTW, the size advantage of CISC is outweighed by the disadvantage of more complex implementations so don't get the wrong idea.

```
# SOLUTION EXAMPLE

# CISC
 add (r1), (r2), (r3)

# RISC
 lw r4, 0(r2)   # Load instruction wastes 16 bits on an immediate thats zero.
 lw r5, 0(r3)   # Ditto
 add r6, r4, r5
 sw r6, 0(r1)   # Also wastes 16 bits.
```

(*d*) The cost of implementing a rarely used instruction is deemed too high and so is omitted from an implementation. How can an operating system enable the implementation to run code that uses the rare instruction without modifying the code or examining it in advance. (5 pts)

The rarely used instruction will raise an illegal instruction exception (since it's not implemented). The operating system will have the handler check the opcode and if it's the rarely used instruction the handler will do exactly what the instruction would (reading the same registers, computing the result, and writing the result in the correct register) using implemented instructions.

(*e*) BCD data types were once popular ISA features. (5 pts)

☑ Were BCD data types absolutely necessary?

No, could use integer data types but that would be inconvenient. When it comes to computers—and many other areas—very few things are absolutely necessary.

☑ What were the reasons for including BCD data types in an ISA?

It eliminated the need to convert to decimal, as when preparing human-readable output. It also allowed for the exact representation of fixed point decimal numbers and eliminated rounding errors in many calculations.

☑ Why might a BCD data type be considered old fashioned and so not worthy of adding to an ISA now?

Computers are much faster and so decimal conversion is very fast. With floating-point standards, in particular IEEE 754, rounding errors are predictable and are minimized in calculations using fixed point decimal numbers.

(*f*) MIPS-I is big endian but more recent MIPS versions can run in either big-endian or little-endian modes.

☑ (5 pts) Complete (possibly modifying) the MIPS program below so that it writes `v0` with a 0 if it is running on a system using big endian byte order or a 1 if running on a system using little-endian byte order.

```
# At finish: $v0: 0, if big endian;  1, if little endian.
lui $t1, 0x1122
ori $t1, $t1, 0x3344
sw $t1, 0($t2)
```

```
# SOLUTION
# At finish: $v0: 0, if big endian;  1, if little endian.
ori $t1, $0, 0x1
sw $t1, 0($t2)
lb $v0, 0($t2)
```

Note that the operands for `ori` were changed and `lui` was eliminated. Byte order **does not** effect non-memory instructions, including `lui` and `ori`. The `ori` sets `t1` to `1` which is written to memory. On a little-endian system the least-significant 8 bits of `t1`, `0x01`, is written to address `t1`, the next 8 bits, `0x00`, are written to `t1+1`, etc. On a big-endian system the most-significant 8 bits, `0x00`, is written to `t1`. If the `lb` reads a `0x01` it must be a little endian system.

Grading Note: Many solutions were correct but most were more complicated than they needed to be, some using branches.

66 Spring 2005 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 1 April 2005, 11:40–12:30 CST

Problem 1		(25 pts)
Problem 2		(30 pts)
Problem 3		(25 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Yes

Good Luck!

Problem 1: In the diagram below some wires are labeled with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly.

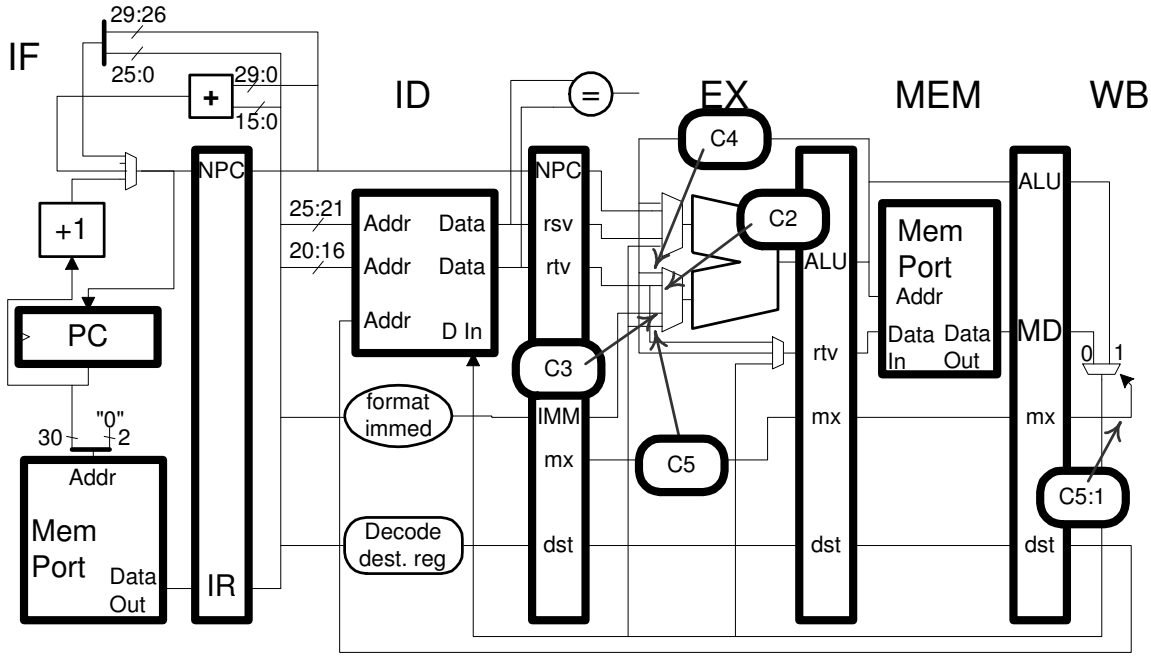
- There are no branches or other control-transfer instructions.

☑ [15 pts] Write a program consistent with these labels.

☑ All register numbers must be made up; **use as many different register numbers as possible** while still being consistent with the labels.

☑ [10 pts] Why would there be no solution to the problem if the C5:1 label in WB were changed to C5:0?

If the label were changed then the second instruction would have to be a load. The third instruction bypasses the result of the second instruction without a stall, and there's no way that can be done if the second instruction is a load.




```
# Solution
Cycle                0  1  2  3  4  5  6  7  8  9
add r1, r2, r3       IF ID EX ME WB
addi r4, r5, 6          IF ID EX ME WB
sub r6, r7, r4             IF ID EX ME WB
or  r8, r9, r4                IF ID EX ME WB
Cycle                0  1  2  3  4  5  6  7  8  9
```

Problem 2: In the diagram on the next page some wires are labeled with cycle numbers and corresponding values. For example, C1:8 indicates that at cycle 1 the pointed-to wire will hold an 8. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly.

- Unlike other problems of this type all registers are different and there are no dependencies.
- The code contains at least one floating-point add and one floating-point multiply.

☑ [10 pts] Write a program consistent with these labels; if a register number cannot be determined use rX or fX.

☑ [5 pts] Complete the pipeline execution diagram.

☑ [5 pts] Fill in the box on the lower-left of the diagram.

☑ [5 pts] Your answer should have a FP multiply instruction. Explain how you knew it was a multiply. (Don't answer "because I already found the add.")

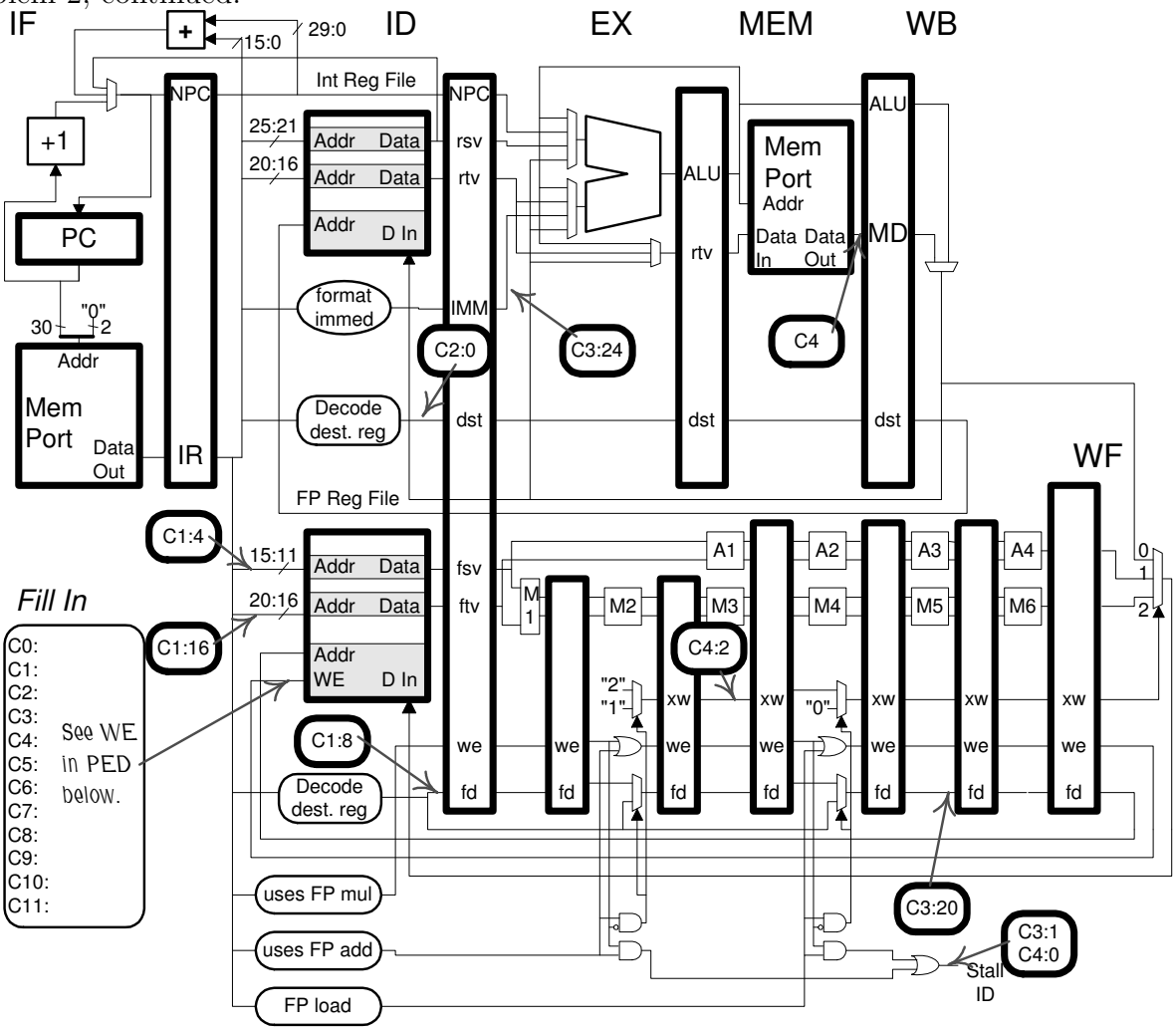
The C4:2 in A1/M3 indicates that in cycle 4 there is a multiply in M3. Working backward, that multiply had to be in ID in cycle 1, and so it's the first instruction.

☑ [5 pts] Your answer should have a FP add instruction. Explain how you knew it was an add. (Don't answer, "because I already found the multiply.")

The C3:1 in the lower right indicates a stall in cycle 3. Such a stall could be because the decoding instruction (the third one, since it's cycle 3) is an add or a load. If it were an add then it would indeed have a structural hazard (thus requiring a stall) with the first instruction, a multiply. If it were a load then the only instruction it could have a structural hazard with is the second one, but there is no instruction type that would need the WF register at the same time, so it has to be an add.

The second instruction is a FP load. We know that because it's using the Data Out of the memory port, and when it reaches WB/F the WB register is zero (from the C2:0) and the WF register is 20 (from the C3:20). Many realized it was a load, few got the correct destination register.

Problem 2, continued:




```
# Solution
Cycle                0  1  2  3  4  5  6  7  8  9
mul.d f8, f4, f16    IF ID M1 M2 M3 M4 M5 M6 WF
lwc1 f20, 24(rX)        IF ID EX ME WF
add.d fX, fX, fX           IF ID -> A1 A2 A3 A4 WF
WE:                  0  0  0  0  0  1  0  0  1  1  0  0  <- Values for Fill-In box.
Cycle                0  1  2  3  4  5  6  7  8  9  10 11
```

Be sure to complete all parts, including the justification for add and multiply.

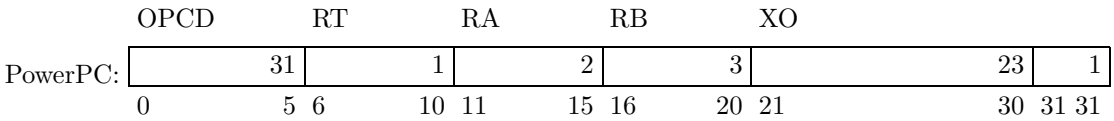
Problem 3: Answer each question below.

(*a*) The encoding of the PowerPC `lwzx r1, r2, r3` instruction is illustrated below, followed by the Type-R MIPS instruction format. [6 pts]

☑ If a field in the illustrated PowerPC instruction format has a counterpart in the MIPS R format draw an arrow between the fields. For example, an arrow should be drawn from OPCD to Opcode. Pay close attention to the register fields.

☑ If fields connected by an arrow are different sizes explain the significance of the difference. (What advantage or disadvantage would the format with the larger field have, if any.)

The smaller function field means that Type R can specify fewer functions than the PowerPC field. Having more opcode space is an advantage if those opcodes are needed.



OPCD ⟶ Opcode, RT ⟶ RD, RA ⟶ RS, RB ⟶ RT, XO ⟶ Function,

(*b*) Suppose that your favorite program is both in the SPEC CPU2000 suite and in the suite used by a popular personal computing magazine and that the input data used for testing the program are also identical. You are considering buying two computers, A, and B. The spec testing shows that your favorite program is faster on A but the magazine's test shows that it is faster on B. The exact same model of machine A is used for running the spec test and the magazine's test, likewise for B. [6 pts]

☑ Provide a possible reason for the discrepancy, the reason should help explain why SPEC is used by the CPU research and development community. *Note: In the original exam the text after discrepancy was offered as a hint.*

Short answer: unlike the magazine's, the spec test would be specially compiled for each machine and might have taken advantage of some unique features of A.

When doing the spec testing the program can be compiled just for the tested machine. Personal computer magazines test programs in the form they are sold to customers, which is already compiled (and so in binary form). Therefore the magazine would test the same binary on both machines whereas those doing the spec testing would specially compile it for each machine, possible taking advantage of special features of A.

Grading Notes: Most missed the key point, that spec benchmarks are compiled for each test whereas end-user benchmark suites, as used by magazines, use just one binary.

Another common misconception is that spec does the testing. They don't, they provide the suite and specify the rules, but they don't actually test.

(*c*) SPEC members and associates are drawn from computer manufacturers, other industries, academia, and elsewhere. [6 pts]

☑ Would you trust SPEC benchmark results more (or less) if computer manufacturers were not allowed to join SPEC? Explain.

Both answers would be considered correct (that includes the reasoning):

I would trust them less because manufacturers have a very strong incentive to keep each other honest.

I would trust them more because manufacturers might want to make it look like future machines are faster by putting in benchmarks which would be faster on future processors but which are not the kinds of programs that many users would run.

(*d*) In class we saw that the optimized π program (reproduced below) ran faster than the unoptimized version (good) but also had higher CPI than the unoptimized version (bad). *Note: the words "than the unoptimized version" was not included in the original exam.* [7 pts]

☑ Why was the CPI higher?

Hint: It has to do with something you learned early in grade school.

```
double i, sum = 0;
for(i=1; i<5000; )
  {
    sum = sum + 4.0 / i;
    i += 2;
    sum = sum - 4.0 / i;
    i += 2;
  }
```

Optimization eliminated many instructions though not divides. Since divides take the longest the CPI went up, performance improved because there were much fewer instructions.

Grading Note: A surprising number of people answered that the optimizations included loop unrolling. The optimized example used in class did not, perhaps because there would be no performance improvement because execution time is dominated by the division dependencies and loop unrolling won't help that. What the optimizer did do is eliminate unneeded loads and stores and other instructions (reducing instruction count) and scheduled (re-arranged) the code to reduce stalls.

Problem 4: Answer each question below.

(*a*) MMX and VIS are examples of ISA extensions that add packed-operand data types and instructions.[7 pts]

☑ How do packed-operand data types and instructions improve performance?

They allow simultaneous operations on several pairs of operands, a set of operands are packed together in a register and operated on by special instructions.

☑ Show a code example that can use such instructions and explain how they would be used.

```
 // Before
 extern char *a, *b, *c;
 for(i=0; i<1024; i++) c[i] = a[i] + b[i];

 // Before, assembler.
LOOP:   // Iterates 1024 times.
 lb t0, 0(t1)
 lb t2, 0(t3)
 add t4, t0, t2
 sb t4, 0(t5)
 addi t1, t1, 1
 addi t3, t3, 1
 bne t5, t6, LOOP
 addi t5, t5, 1

 // After, assembler.
LOOP:   // Iterates 256 times.
 lw t0, 0(t1)
 lw t2, 0(t3)
 add.pack8 t4, t0, t2  // Add assuming 8-bit quantities packed into register.
 sw t4, 0(t5)
 addi t1, t1, 4
 addi t3, t3, 4
 bne t5, t6, LOOP
 addi t5, t5, 4
```

(*b*) Name an advantage and a disadvantage that RISC's fixed-size instructions have over CISC's variable length instructions.[6 pts]

☑ Advantage:

Branch displacements can be in instructions rather than bytes, allowing for further jumps for the same instruction size. Simplified fetch and decoding.

☑ Disadvantage:

Some instructions are larger than they have to be. Another disadvantage, what could be done with one instruction in CISC, say loading a 32-bit constant, needs at least two instructions in RISC because of space limitations.

(*c*) A trap instruction is something like a subroutine call to the operating system. [7 pts]

☑ So why couldn't it just specify the address of the trap handler?

Because then it might bypass protections that the operating system tries to impose. For example, suppose the address of a file read system call were 0x1000, and that the first part of this routine checked if the user had permission to read that file, followed at address 0x1800 by the code that does the read. The user might trap to address 0x1800 and bypass the protection.

Grading Note: Many answered that a user program would not be allowed to execute in the system area. That's not an issue here because the trap instruction already changes the processor from user to system mode.

☑ What does the trap instruction specify (in SPARC but not MIPS) in place of an address and how does execution actually reach the trap handler.

It specifies an index into (entry number in) the trap table. The hardware will combine this index with the contents of the trap base register (address of the beginning of the trap table) obtaining the address of the handler itself (within the trap table). This address will make it to the PC, and so execution will proceed in the handler.

Name Solution

Computer Architecture
EE 4720
Final Examination

10 May 2005, 15:00–17:00 CDT

Problem 1		(25 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias Resolution Time

Good Luck!

Problem 1: (25 pts) The `madd.s fd, fr, fs, ft` instruction writes floating-point register `fd` with (`fr` × `fs`) + `ft`. The `fr` field is in bits 25:21, the other fields are in their usual places. The instruction is used in the code below:

```
# With ordinary instructions:
 mul.s f1, f2, f3
 add.s f1, f1, f4

# With a madd.s instruction:
 madd.s f1, f2, f3, f4
```

(*a*) Add datapath connections (including any connections to the register file) to the implementation on the next page (also shown below) to implement the `madd.s` instruction. The code below should execute as shown (pay attention to `f4`).

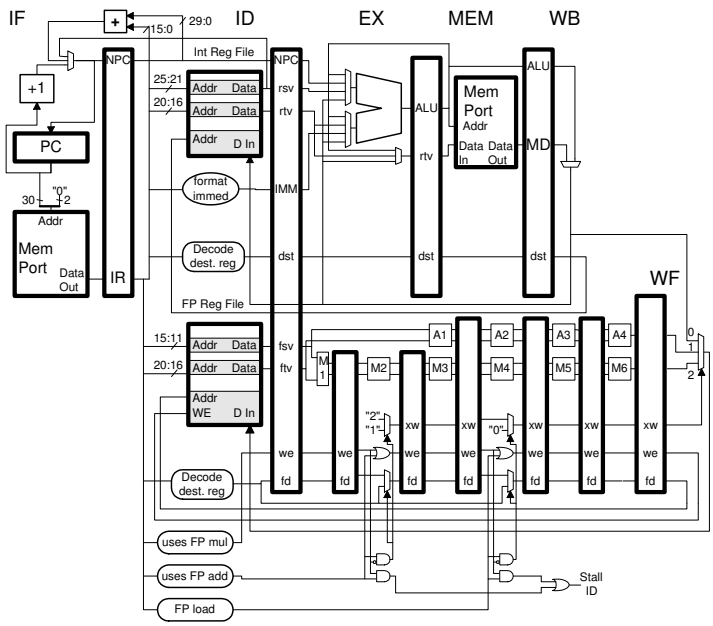
```
 madd.s f1, f2, f3, f4 IF ID M1 M2 M3 M4 M5 M6 A1 A2 A3 A4 WF
 lwc1 f4, 0(r2)           IF ID EX ME WF
 add.s  f5, f6, f7           IF ID A1 A2 A3 A4 WF
```

- Use the existing multiplier and adder.
- It should still be possible to execute ordinary floating point multiply and add instructions.

(*b*) Modify the logic so that `xw`, `we`, and `fd` work correctly for the `madd.s` instruction (and continue to work correctly for existing instructions).

(*c*) Without a `madd.s` instruction there is no possible structural hazard on `WF` in the implementation below for multiply because multiply is the longest latency instruction implemented. With `madd.s` that's no longer true; add control logic to detect this new structural hazard and generate the Stall ID signal.

USE NEXT PAGE FOR SOLUTION.



USE NEXT PAGE FOR SOLUTION.

Problem 1, continued:

☑ Datapath for `madd.s`, don't break `add.s` or `mul.s`.

☑ Code on previous page must run as shown.

☑ Modify `xw`, `we`, and `fd` for `madd.s`, don't break other instructions.

☑ Detect and handle new structural hazard when `mul.s` in ID.

Solution on next page.

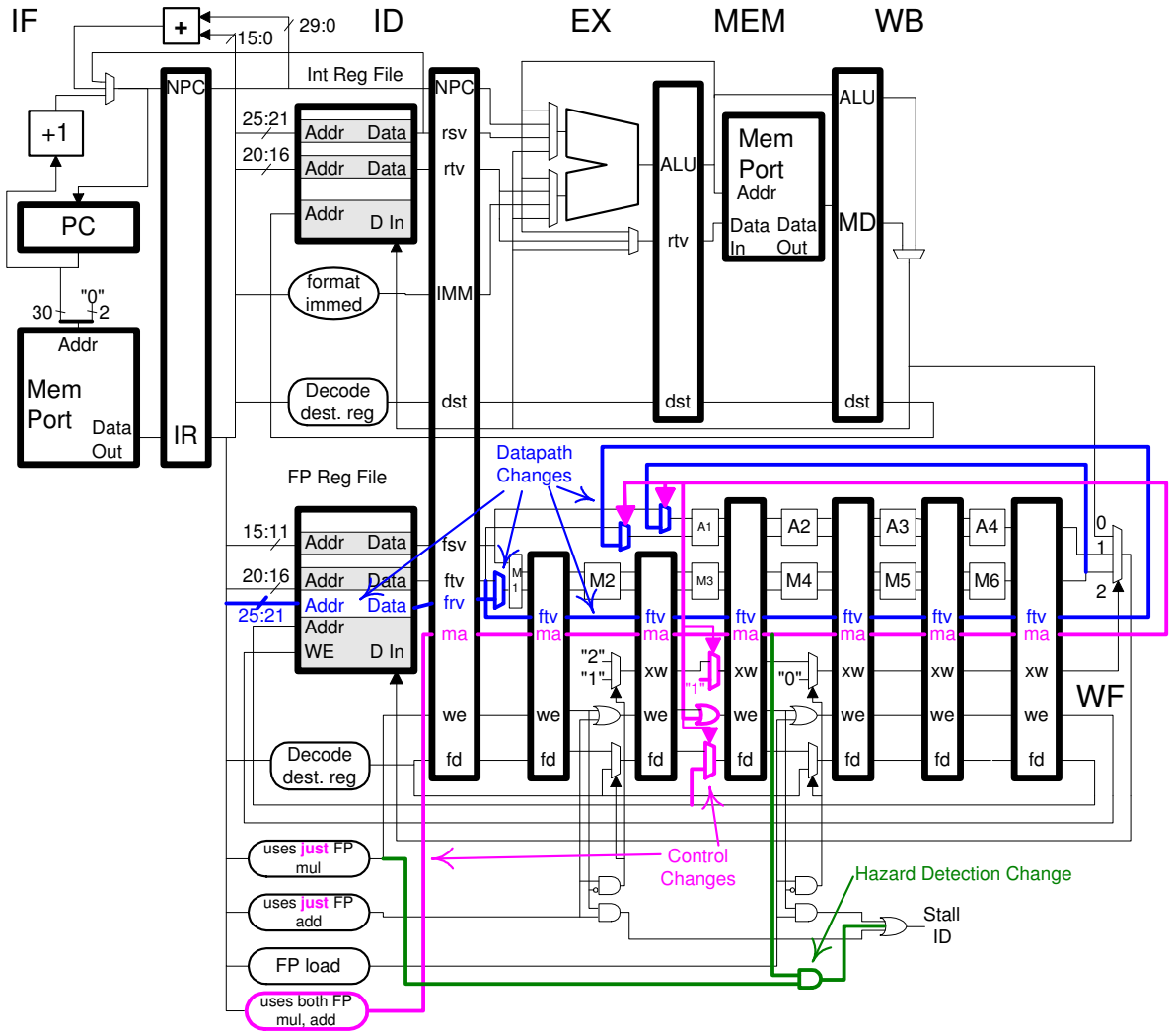
Solution shown below.

Part(a): Datapath changes are shown in blue. A third read port was added to the floating-point register file to retrieve the fr value (`frv`). For the `madd.s` instruction the `frv` is used for the multiply, a multiplexor is added in M1 to handle that. Register value `ftv` is sent down the pipeline in a new set of latches for use by the add in a second pass through the pipeline. The first add stage has new multiplexors to select `ftv` and multiply result from the WF stage.

In the code sample on the previous page a `lwc1` writes `f4`, in the illustrated implementation that's no problem. There would be incorrect results if an implementation did not read the `ft` register until the cycle before it was needed, when `madd.s` reaches M6. That's what "pay attention to f4" was warning about.

Control logic changes are shown in purple. A new logic block, uses both PF mul, add, detects multiply add instructions. The existing multiply- and add-unit detection blocks now (and perhaps always did) only assert their outputs if an instruction just does a multiply or just does an add. A `ma` (multiply-add) signal is sent down the pipeline and used to inject the product, `ftv`, `fd` (the destination register number), the output mux signal (`xw`), and write enable (`we`) into the A1 stage when `madd.s` reaches WF the first time. After that the `madd.s` looks like an ordinary FP add to the pipeline.

The logic to detect the new hazard appears in green. If a `madd.s` instruction is in M4 (it's first trip) and an ordinary multiply is in ID a stall signal is generated.



Problem 2: (15 pts) The execution of a MIPS program on a dynamically scheduled system using method 3 appears below. Complete the ID Register Map, Commit Register Map, and Physical Register File tables for registers `f2` and `f4`.

- The initial value of `f2` is 2.0, the `mul.d` writes 2.1, `ldc1` writes 2.2, and `sub.d` writes 2.3. The initial value of `f4` is 4.0, the `add.d` writes 4.1. Make up physical register numbers as needed.

☑ As always, show table contents.

☑ Show where registers are removed from and placed back in the free list.

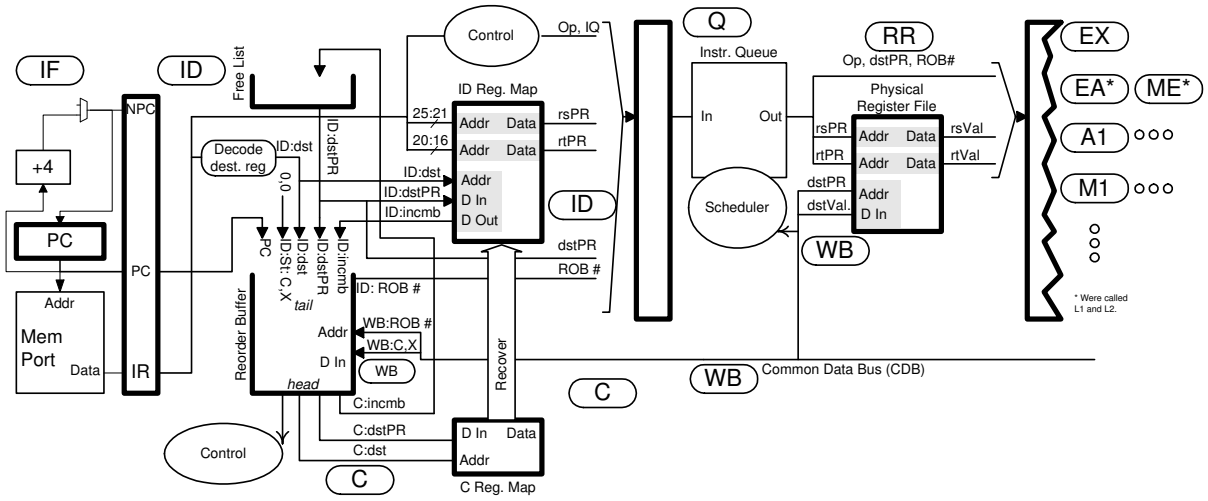
☑ Show initial values.

The solution shown below, there is nothing interesting about this problem.

```
 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
 mul.d f2, f4, f6   IF   ID   Q    RR   M1   M2   M3   M4   WF   C
 add.d f4, f2, f10       IF   ID   Q                   RR   A1   A2   A3   WF   C
 ldc1 f2,0(r1)                IF   ID   Q    EA   ME   WF                             C
 sub.d f2, f2, f8                  IF   ID   Q    RR   A1   A2   A3   WF                  C

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
ID Register Map

 # Solution
 F2       12             9         71   99
 F4       18                  51

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Commit Register Map

 # Solution
 F2       12                                                       9                   71   99
 F4       18                                                                      51

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
Physical Register File

 # Solution
 12          2.0                                                  ]

 18          4.0                                                                 ]

  9                       [                              2.1                         ]

 51                            [                                        4.1

 71                                 [                   2.2                              ]

 99                                      [                        2.3

 # Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12   13   14
```

HARDWARE SHOWN ON NEXT PAGE.

Problem 2, continued: Dynamically scheduled processor shown for reference.



Problem 3: (20 pts) The code below runs on three systems, one using a bimodal predictor (B) with a 2^{10}-entry BHT, a local predictor (L) with a 2^{10}-entry BHT and a 16-bit local history, and a global predictor (G) with a 16-bit GHR. The outcomes of B2 are shown but not B1's so **pay attention to where B1's outcomes are located**.

```
LOOP:
SHORT:
# B1 Iterates 10 times
B1: bne r3,r4 SHORT
addi r4, r4,1
...
B2: beq r1, r2 SKIP     N    N    T    N    T    N    N    T    N    T ...
...
SKIP:
j LOOP
nop
```

(*a*) Find the prediction accuracy of each predictor on each branch after warmup. Briefly explain.

☑ Accuracy predicting B1 on B: 90%

☑ Accuracy predicting B1 on L: 100%

☑ Accuracy predicting B1 on G: 100%

☑ Accuracy predicting B2 on B: 60%

☑ Accuracy predicting B2 on L: 100%

☑ Accuracy predicting B2 on G: 80%

(*b*) Determine the number of table entries used by branch B2 on each predictor:

☑ Entries for B2 on B: 1 BHT.

☑ Entries for B2 on L: 1 BHT + 5 PHT.

☑ Entries for B2 on G: 1 BHT + 2 PHT.

(*c*) Suppose the history sizes were reduced.

☑ What is the smallest local history size for which the accuracy on B2 will be unchanged (from the previous answer) when using the local predictor. Briefly explain.

Four bits. With three bits a local history of NTN would appear before both a T and N outcome so the PHT entry counter will mis-predict the T, the N, or even both. With four bits all outcomes can be distinguished.

☑ What is the smallest global history size for which the accuracy on B2 will be unchanged when using the global predictor. Briefly explain.

Eleven bits. For the global history size of 16 used above the GHR holds (global predictor can "see") just one previous outcome of B2. At 11 bits the GHR still holds that previous outcome, at 10 bits is does not and so prediction accuracy will be reduced.

Branch B1 is a simple ten-iteration loop, so there will be nine taken outcomes followed by a not-taken. The diagrams below show the counter values, predictions, and prediction outcomes using a bimodal predictor on branches B1 and B2. Prediction accuracy is computed over a range of cycles that will repeat. For B1 that is cycles 5-10, for B2, that is 0-5 (and 5-10).

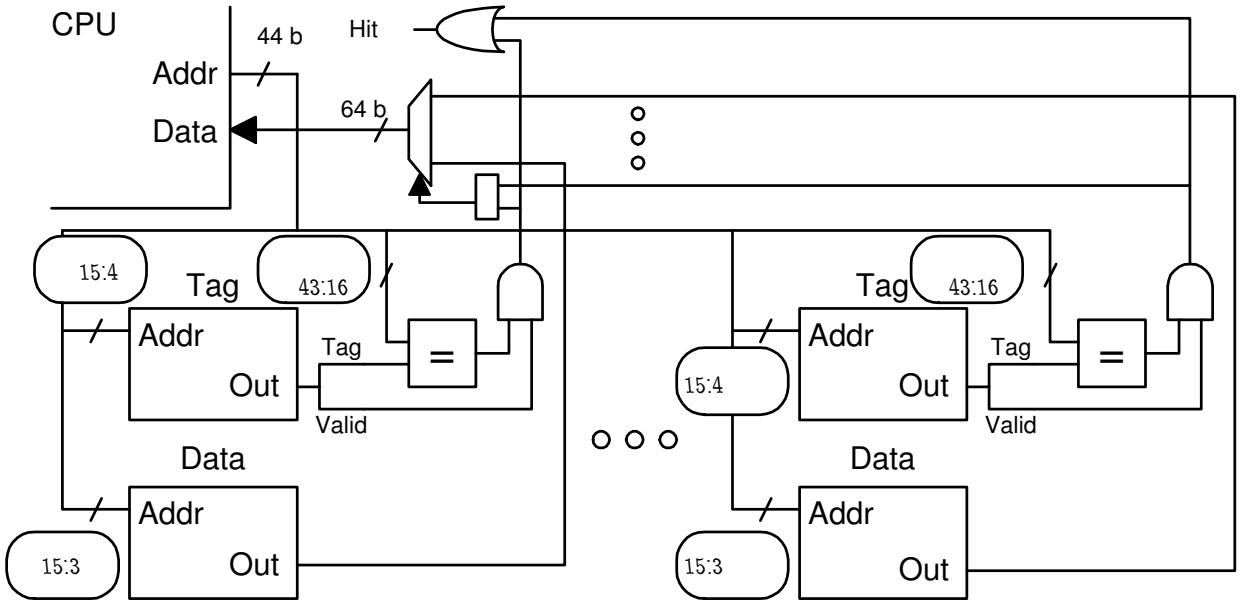
```
# Analysis for computing bimodal prediction accuracy on B1.

     # B1 Iteration  0   1    2    3    4    5    6    7    8    9      10
     # Counter       0  1   2   3    3     2   3    3    3    3       2
     # Prediction     n   n   t   t      t   t    t    t    t    t      t
     # Mispredict       x   x              x                          x
B1: bne r3,r4 SHORT    T   T   T    T ... N   T    T    T    T ...    N
addi r4, r4,1
...


# Analysis for computing bimodal prediction accuracy on B2.

     # B2 Iteration   0   1    2    3    4    5    6    7    8    9    10
        # Counter     0   0    0    1    0    1    0    0    1    0    1
     # Prediction      n   n    n    n     n   n    n    n    n    n    n
     # Mispredict                 x         x              x         x
B2: beq r1, r2 SKIP     N    N    T    N    T    N    N    T    N    T ...
```

The local predictor will predict both with 100% accuracy because their patterns are each less than 16 outcomes. The global predictor will predict B1 with 100% accuracy because the GHR will hold more than the last 9 iterations of B1, which is enough to predict with 100% accuracy. For B2, the GHR will hold only one previous outcome of B2 (the other 15 bits are outcomes of B1 which do not help in predicting B2). Therefore two PHT entries are used with the global predictor predicting B2, one for the previous outcome being taken and one for the previous outcome being not taken. After warmup the not-taken entry will predict taken (because that's the outcome 2 out of 3 times it is used, per iteration) and be wrong once. The taken entry will predict not taken and always be right. Therefore the prediction accuracy will be 80%.

Problem 4: The diagram below is for a 256-kiB (2^{18} byte) 4-way set-associative cache with a line size of 16 characters on a system with 8-bit characters. (20 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram. *Pay attention to the width of the CPU address port.*

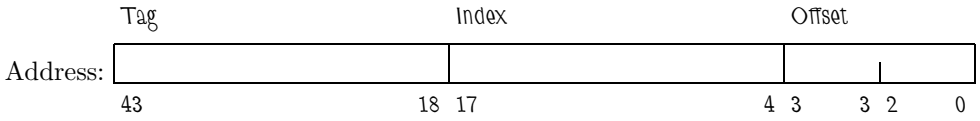


☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $4 \times 2^{16-4}\,(44 - 16 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 4, continued:

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ What is the hit ratio running the code below?

On the first `j` iteration the hit ratio is 0.5 because the line size is only 16 characters and a data element is 8 characters. The number of `i` iterations is large enough so that the array does not fit in the cache and so on the second `j` iteration the hit ratio will also be 0.5, so the overall hit ratio is, of course, 0.5.

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j, ILIMIT = 0x1000000;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

The code in the problems below is to be run on a cache of unknown configuration, though it will be one of the types discussed in class (direct mapped or set-associative). In all cases the cache is empty when the program starts. Routine `get_miss_count()` *returns the number of cache misses at the time of the call (and does not cause any misses).*

(*c*) Complete the code so that `line_size` is assigned the correct line size based on the number of misses encountered and the nature of the code. Assume that all misses are due to access to `a`.

```
int ILIMIT = 0x10000;
char *a;
int miss_count_before = get_miss_count();
for(i=0; i<ILIMIT; i++) sum += a[i];
int miss_count_during = get_miss_count() - miss_count_before;

int line_size =  ILIMIT / miss_count_during;          // SOLUTION
```

There will be one miss for each line used to cache the array. The total size of the array accessed is `ILIMIT` characters so the line size is that divided by the number of misses.

(*d*) Complete the code so that `associativity` is aptly assigned. Assume that the cache is smaller than 256 MiB (2^{28})(but the exact size is not known), that the associativity is no larger than 64, and that there is a 64-bit address space.

```
int ISHIFT =  28;                                     // SOLUTION
int JSHIFT =  28;                                     // SOLUTION
char *a;  // Pointer to a really large array.
int miss_count_before = get_miss_count();
for(i=0; i<64; i++) sum += a[ i << ISHIFT ];
for(j= 63; j>=0; j--            )                     // SOLUTION
   sum += a[ j << JSHIFT ];

int miss_count_during = get_miss_count() - miss_count_before;
int associativity =  128 - miss_count_during;         // SOLUTION
```

Use the `i` loop to load 64 lines with different tags and the same index by setting `ISHIFT` to 28 (since the cache size is no more than 2^{28} bytes). The `i` loop will miss 64 times, once for each iteration. Have the `j` loop check for the lines, starting with the most recent one (`j=63`) which should be a hit. Then check `j=62`, if the cache is direct mapped it will be a miss, otherwise a hit, check `j=61`, if the cache is less than 3-way it will be a miss, etc.

Problem 5: Answer each question below.

(*a*) Suppose in a five-stage statically scheduled MIPS implementation (like the one in class) an instruction could raise an exception in the WB stage. Explain why that would make precise exceptions for that instruction impossible. Use a code example to explain what should happen for a precise exception and why its impossible (or very difficult) if the exception is raised in WB. (5 pts)

If the instruction **after** the faulting instruction were a store it could not be stopped from writing to memory, and so the exception could not be precise. (In a precise exception the handler should start at a point in the program just before the faulting instruction.) In the example below the `jalr` instruction uses an illegal destination register (it can't be the same as the source). If that illegal register were detected in WB rather than ID, then it would be too late to stop the `sw`. When the handler started it would see the effect of the `sw`, which means the `jmpl`'s exception was not precise.

```
jalr r1, r1   IF ID EX ME WB
sw r3, 0(r4)     IF ID EX ME WB
```

(*b*) Arrange the ISA families below in order by code (program) size, the one for which programs are smallest should be first. Starting at the second ISA in the arranged list, explain why code size is larger than the ISA above. (That is, provide three reasons why code size is larger.) (5 pts)

ISA families (in alphabetical order): CISC RISC Stack VLIW

☑ ISA families in code size order.

☑ Reasons for size differences.

Smallest: Stack. Unlike CISC and the other ISAs, Stack instructions refer to the stack and so register operands are not used in arithmetic instructions, making them small. In those cases where operands are not in the right place in the stack rearrangement instructions are needed, but even so stack programs are smaller than programs in other ISAs.

CISC. Unlike RISC instructions, CISC instructions are variable size and so they are no larger than they have to be. All RISC instructions are the same size, so some have unused space.

RISC. Unlike VLIW, a RISC program consists only of RISC instructions and other than four-byte (usually) alignment there is little restriction on where instructions can be placed. In VLIW they must be arranged into bundles with restrictions on placement, for example, it might not be possible to put a floating-point instruction in the first slot of a bundle. Because of these restrictions slots may occasionally be filled with nops. The bundles also include template information. In Itanium three RISC-like instructions take 128 bits, a RISC ISA could fit four instructions in the same space. Branches must be to the beginning of a bundle, further wasting space.

Largest programs: VLIW.

(*c*) Deciding on a line size for a cache is a tough decision. (5 pts)

☑ Describe the behavior of programs that run better on caches with smaller line sizes.

The programs should have low spatial locality, meaning if they access an address they probably won't be accessing something nearby anytime soon. Small line sizes help because less space is wasted bringing in data near something which is accessed.

☑ Describe the behavior of programs that run better on caches with larger line sizes.

The programs should have high spatial locality. For example, they might access data sequentially.

(*d*) Answer the following question on cost.(5 pts)

☑ Name two parts of an n-way superscalar processor that cost about n times as much as a comparable scalar processor.

The fetch hardware. The integer ALUs. In each case no more than n of them are needed.

☑ Name two parts of an n-way superscalar processor that cost about n^2 times as much as a comparable scalar processor.

The control logic for detecting dependencies because the source of each of n instructions being decoded must be compared with the destinations of other instructions in the same stage, and n instructions in each of the next few stages. That's proportional to n^2 comparisons.

The bypass connections. It should be possible to bypass results to any of the n instructions in EX; each ALU input will have a multiplexor connecting to n pipeline latches in each of the next two stages (for the five stage design used in class).

67 Fall 2004 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination 1.0.4

Friday, 22 October 2004, 10:40–11:30 CDT

Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(25 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias Titan at last!!!

Good Luck!

Problem 1: The routine below is to do a binary search on an array and return 1 if the item is found, 0 otherwise. Complete the MIPS program using the C++ code as a guide. [30 pts]

☑ The only allowable synthetic instruction is `nop`.

☑ Fill as many delay slots as possible. The order of the assembly code **does not** have to match the order of the C code.

```
#  unsigned int delta = size;  int pos = 0;
#  while( true ) {
#      delta = delta >> 1;
#      int next_pos = pos + delta;
#      int e = array[next_pos];
#      if( e == target ) return 1;
#      if( !delta ) return 0;
#      if( e < target ) pos = next_pos;
#    }
        # CALL VALUE: $a0,  array: Address of array. Each element is a WORD-SIZED int.
        # CALL VALUE: $a1,  size: Number of elements in array. A power of 2.
        # CALL VALUE: $a2,  target: Value to find.
        # RETURN VALUE: $v0,  1, if target in array; 0, if not in array.
        # Can modify registers $a0-a3, $t0-$t9

# Solution
#
# Common problems highlighted.  Solution below shows correct code.
#
bsearch:
        # $a1 is delta
        addi $t0, $0, 0     # pos = 0;
LOOP:
        srl $a1, $a1, 1     # delta = delta >> 1;
        add $t1, $t0, $a1   # next_pos = pos + delta
                            # COMMON PROBLEM: Index not scaled.
        sll $t2, $t1, 2     # Scale array index since elements are four bytes.
        add $t2, $a0, $t2   # t2 = &array[next_pos]
        lw $t2, 0($t2)      # t2 = e, array[next_pos]
        bne $t2, $a2  SKIP1
        nop
        jr $ra              # Element found, return 1.
        addi $v0, $0, 1
SKIP1:
        bne $a1, $0  SKIP2
        nop                 # Delta zero, return 0.
        jr $ra
        addi $v0, $0, 0     # COMMON PROBLEM: Jump delay slots ignored.
SKIP2:
        # if ( element < target ) pos = next_pos
        #
        slt $t3, $t2, $a2
        beq $t3, $0  LOOP
        nop
        j LOOP
        addi $t0, $t1, 0
```

Problem 2: In MIPS and many other ISAs floating-point instructions read and write a FP register file while most other instructions read and write general-purpose (integer) registers. Suppose MIPS-I were modified so that FP instructions used the same registers as the integer instructions. The registers are still 32 bits and can be used in pairs by instructions operating on double-precision operands. The new version of MIPS is not compatible with MIPS-I, don't worry about that.

(*a*) For each MIPS-I instruction below indicate whether it will be retained in the new ISA, "KEEP", or will not be needed, "OMIT." Also add any new instructions that might be needed. If an instruction is kept but operates differently describe the difference. [15 pts]

☑ Indicate KEEP or OMIT.

☑ If omitted show an existing instruction that would be used in its place.

☑ If kept, explain how it would operate differently.

lwc1: OMIT, use lw instead.

ldc1: KEEP, but writes GPRs (call it ld now).

cvt.w.d: KEEP (but operates on the integer registers.) Still need to convert from integer to FP.

cvt.d.w: KEEP (but operates on the integer registers.) Still need to convert from FP to integer.

mfc1: OMIT. No longer an FP reg file to move values from.

mtc1: OMIT. No longer an FP reg file to move values to.

add.s: Keep, but operates on integer registers.

add.d: KEEP, but operates on integer register pairs.

c.lt.s: KEEP, write result (0 or 1) to register.

c.le.d: KEEP, write result (0 or 1) to register.

bc1t: OMIT. Use bne or beq.

Hint: In `bc1t` *b is for branch, t is for true.*

Grading Notes:

Many suggested eliminating `c.lt.s` and using `slt` instead. This will almost work because if one IEEE 754 number is larger than another then its binary encoding interpreted as a two's complement number is larger than the other and so an integer comparison can be performed. Perhaps many had that in mind when they suggested using `slt` (I hope). The problem is that IEEE 754 has several non-number encodings and the comparison instruction is supposed to raise an exception if an attempt is made to compare them, slt won't do that.

The original exam had c.gt.d. There is no such instruction, ooops.

(*b*) If your answer for `bc1t` above was "OMIT" describe below how the instruction can be kept (possibly modified). If your answer was "KEEP" explain how it can be omitted. This will affect other instructions, list them and explain how they are affected.

Instruction `bc1t` branches if the FP condition code register is true. The instruction was omitted above because the comparisons write a general purpose register.

In this part keep `bc1t` and modify the comparison instructions `c.lt.d`, etc. so that they write the FP condition code register instead of an integer register, as they did in the previous part.

Problem 3: Using an interesting technique the program below returns the value of register X, where X is a number in register `$a0`. For example, if `$a0` held an 8 the program would return the value of `$8` (or `$t0` using the register name) in register `$v0`. Consider:

```
jal copyreg
addi $a0, $0, 8
# After copyreg returns $v0 has contents of $t0.

addi $v0, $t0, 0   # $t0 is $8, register number 8
# At this point $v0 has contents of $t0.
```

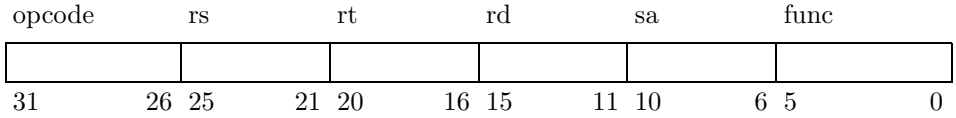
From the code above it looks like the `copyreg` routine is doing things the hard way. The one advantage is that it can copy a source register known only at run time or that might change. In contrast, the `addi` instruction always copies `$t0`. *Note: In the original exam the description and example above were not included.*

(*a*) Modify the program so that it takes a second call value, `a1`, which holds a second register number. The return value is put in that register. For example, if `a0` holds a 12 and `a1` holds a 3 then the routine copies the value in `r12` to `r3`. [17 pts]

☑ Show additions and changes. The added code must use a similar technique to the existing code.

Hint: Can be solved with two added instructions and a few minor modifications.

```
        # Solution.

        # CALL VALUE:   $a0   A register number. (Except v0)
        # CALL VALUE:   $a1   A register number.
        # RETURN VALUE:       Return value in register specified by $a1

copyreg:
        la $v0, template    # Put address of label "template" into $v0.
        lw $v0, 0($v0)      # Load template instruction into $v0.
        sll $a0, $a0, 21    # Shift register number to rs position.
        add $a0, $v0, $a0   # Insert source register number into instruction.

        # Solution (Next two instructions, and modification of template.)

        sll $a1, $a1, 11   # Shift register number to rd position.
        add $a0, $a0, $a0  # Insert destination register into instruction.

        la $v0, mod        # Load address of last instruction in this routine.
        sw $a0, 0($v0)     # Store created instruction at end of this routine.
        jr $ra             # Return ...
mod:
        add $v0, $0, $0    # ... and move registers.

template:
        # add $v0, $0, $0  Original instruction had dest of v0.
        add $0, $0, $0      # In solution make that a $0 since it's modified.
```



Format R: 31 26 25 21 20 16 15 11 10 6 5 0

Problem 3, continued:

(*b*) Suppose one were considering adding a new machine instruction, `copyreg`, that does the same thing as the routine above (before modification for part 1). Provide arguments for and against adding the new instruction. The arguments can include made-up data as long as it does not conflict with what has been given in this problem or in class. At least one of the arguments must refer to features of the code from part 1. [8 pts]

☑ Provide an argument in favor of adding the instruction.

Programs needing to copy a register known only at run time would work much faster if they had a machine instruction to do it.

Grading Note: Though the routine above performs a load and a store, that's not the way an implementation would do it.

☑ Provide an argument against adding the instruction. Do not simply reverse an assumption made above.

Instructions like this are not needed very often.

Problem 4: Answer each question below.

(*a*) A company is developing an implementation of an ISA. To guide their design they are using a **SUBSET** of the SPECcpu benchmark programs. [7 pts]

Grading Note: A surprising number of students seemed to miss the word subset and answered the question as though it were about the full set of SPEC CPU benchmarks, not a subset.

☑ Describe a situation in which that's a foolish thing to do and explain why it's foolish.

It is foolish if the intended customers run a wide range of programs similar to the full set of benchmarks. By omitting certain programs the company will not be designing their implementation to run those programs fast.

☑ Describe a situation in which that's a smart thing to do.

It's smart if the subset matches what the intended customers to run. This way time is not wasted optimizing a design for programs their customers won't be running.

(*b*) Compiler optimizations are important. [8 pts]

☑ Describe a compiler optimization for which the compiler needs no knowledge of the ISA.

☑ Show an example.

Common subexpression elimination.

```
// Before
if( a+b < c ) x = a + b;

// After (optimization shown in C, but actually done in compiler intermediate
//        representation.)
int s = a + b;
if ( s < c ) x = s;
```

☑ Describe a compiler optimization for which the compiler does need to know the target ISA.

Register assignment. Compiler needs to know how many registers there are, not to mention what are the instructions which need register values in the first place.

(*c*) A goal in the design of MIPS and other RISC ISAs is to minimize the number of registers with specialized purposes. In MIPS `r31`, a.k.a. `$ra`, does have a specialized purpose. [7 pts]

☑ Why was it necessary to make an exception in this case?

The `jal` instruction stores the return address in register `ra`. It is coded in format J which only has an opcode and immediate (`ii`) field, there is no room for a register number. Therefore if there was to be a jump and link with a $26 + 2$-bit region it would have to save the return address in a fixed register.

(*d*) The target in MIPS branch instructions is specified by indicating the number of instructions to skip. [8 pts]

☑ Why couldn't such an approach be used in CISC ISAs?

Because CISC instructions are variable size and so computing a target address using a number of instructions to skip cannot be done by just multiplying that number by the instruction size. The implementation would have to know the sizes of the intervening instructions which would be extremely impractical. Instead branch targets typically specify the number of bytes to skip.

☑ Specifying the number of instructions to skip saves two bits over what is needed for CISC targets. Why is saving two bits not as important in the design of CISC instructions?

Because instructions are variable length their size can be increased to accommodate the size of any needed operands. In the case of a branch the instruction encoding can be made as large as needed to hold the displacement, even if the displacement is two bits larger (which isn't very much anyway) than in most RISC ISAs.

Name Solution

Computer Architecture
EE 4720
Final Examination

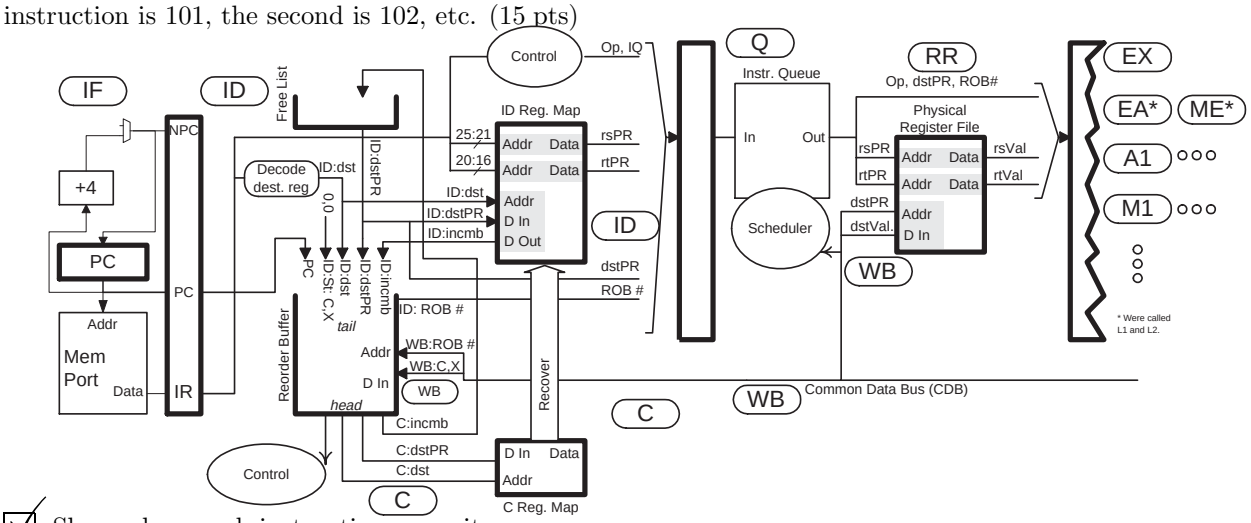
6 December 2004, 7:30–9:30 CST

Problem 1		(15 pts)
Problem 2		(25 pts)
Problem 3		(25 pts)
Problem 4		(10 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias FF MV̄DCCXX

Good Luck!

Problem 1: The MIPS code below executes as shown on the illustrated dynamically scheduled scalar implementation. There are no exceptions or recoveries; the result (value to be written in `r2`) of the first instruction is 101, the second is 102, etc. (15 pts)



☑ Show where each instruction commits.
☑ The initial value of `r1` is 111 and the initial value of `r2` is 222. Fill in the tables to show these values.
☑ Complete the tables for the execution of the code. Take into account the results (see first paragraph).
☑ Show where registers are removed from and put back in the free list.

```
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
lw  r2, 0(r4)        IF  ID  Q   RR  EA  ME  WB  C
add r1, r2, r3           IF  ID  Q       RR  EX  WB  C
sw r1, 0(r4)                 IF  ID  Q   RR  EA      ME  WB  C
xor r2, r5, r3                   IF  ID  Q   RR  EX  WB          C
and r2, r2, r6                       IF  ID  Q       RR  EX  WB      C
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
# ID Map

Reg   Initial PR
r1  17                       66
r2  43                   63          83  88

# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
# Commit Map

Reg   Initial PR
r1  17                                           66
r2  43                                       63              83  88

# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
Physical Register File

PR    Initial State
17    111 r1                                         ]
43    222 r2                                     ]
63                       [r2                 101                 ]
66                           [r1                 102
83                                   [r2             104             ]
88                                       [r2                 105
# Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12
```

The solution is shown on the previous page. This problem is uninteresting and so it would be a good first study problem for those reading this in the future.

Common Mistake: Allocating a register for `sw`. The `sw` instruction does not write a register and so it is not allocated a physical register.

Common Mistake: Freeing the wrong register. When an instruction commits the correct physical register to free is **not** the one assigned to the instruction but the incumbent, the previous one assigned to the same register. For example when the `add`, which was assigned physical register 66, commits it is physical register 17 which gets tossed back into the free list. The incumbent can be found in the commit (or ID) map to the left of the register assigned to the instruction.

Problem 2: An extended version of MIPS, called *MMMIPS*, includes memory-to-memory (MM) arithmetic instructions that can read the first source operand from memory and write a result to memory; the second source operand is always an immediate and the MM's are encoded in format I. Their mnemonics end with `.mm`, `.mr`, or `.rm` and they operate as described in the comments below. (25 pts)

```
lw     r1, 2(r3)        #  r1 = Mem[r3+2]  (Existing instruction, for your reference.)
add.mm (r4),(r5), 3     #  Mem[r4] = Mem[r5] + 3
sub.rm  r6, (r7), 3     #  r6 = Mem[r7] + 3
or.mr  (r8), r9,  3     #  Mem[r8] = r9 + 3
```

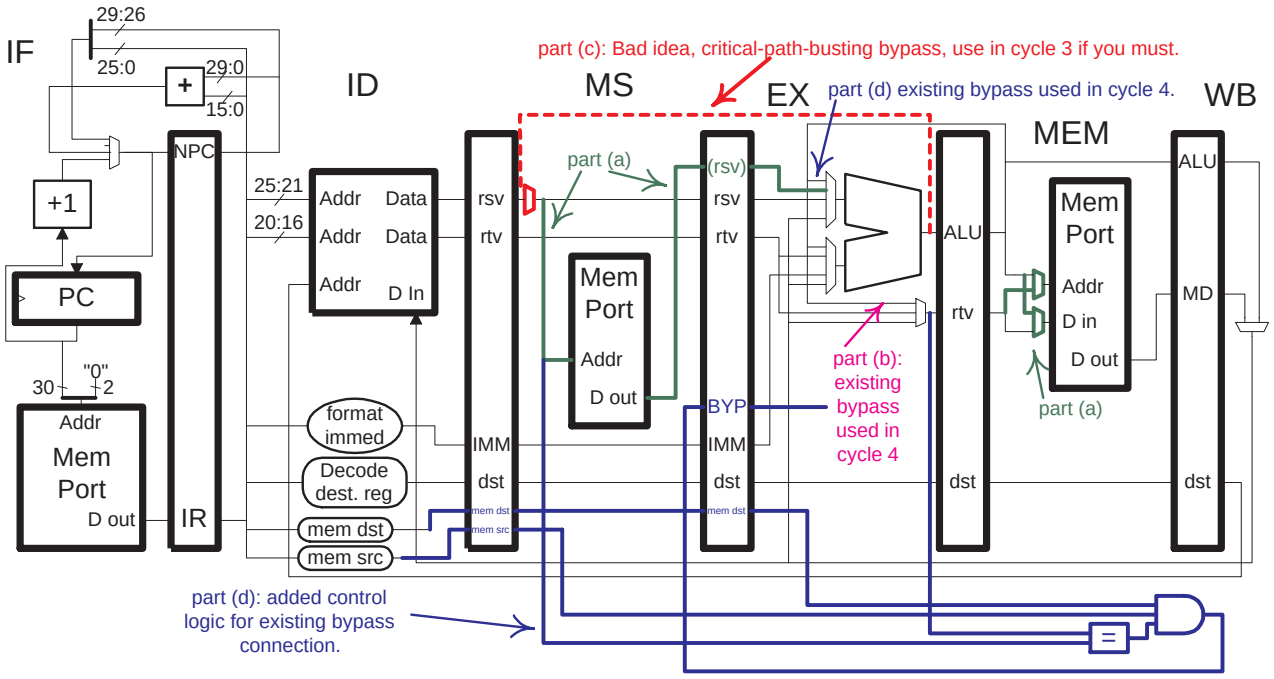
(*a*) Shown below is a partially completed implementation of MMMIPS. It includes a new stage, MS (the S is for source), that has a memory port for the source operand, shown unconnected. For use in a later part of this problem, boxes mem src and mem dst identify an instruction as having a memory source operand (output `1`) or a memory destination operand (output `1`) (an instruction can have both). An output of `0` means the respective operand is not from or to memory.

☑ Connect the MS-stage memory port for the MM instructions.

☑ Modify connections to the MEM-stage memory port for the MM instructions.

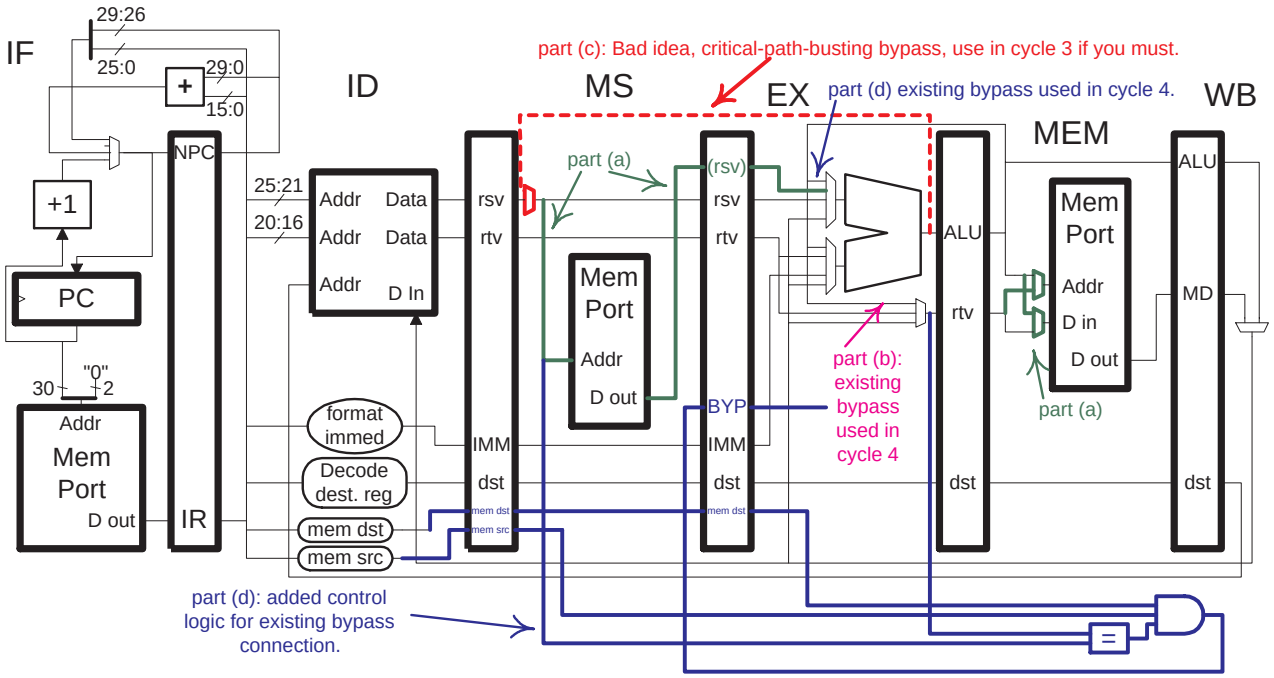
☑ Make sure existing MIPS load and store instructions continue to work correctly.

☑ For this part don't add bypassing or control logic.



Solution to this part appears above in green along with solution to other parts.

For the problems below use either the diagram below or the one on the previous page.



(*b*) Add a bypass connection so that the code below can execute without a stall or show which existing bypass connection can be used.

☑ Add or identify the connection.

☑ Show which cycle the bypass connection is used.

```
# Cycle              0  1  2  3  4  5  6
add     r1, r2, r3   IF ID MS EX ME WB
sub.mr (r1), r4, 5      IF ID MS EX ME WB
```

The value can use the exiting MEM-to-EX bypass connection, the bypassed value goes through the mux to EX/MEM.rtv. The value would be bypassed in cycle 4 for use in the memory write in cycle 5.

(*c*) Can a bypass connection be added for the code below? If yes, should it be added? Explain.

☑ Show how and/or explain why not.

```
# Cycle              0  1  2  3  4  5  6
add    r1, r2, r3    IF ID MS EX ME WB
sub.rm r6, (r1), 5      IF ID MS EX ME WB
```

Yes it can be added, but it shouldn't because of critical path impact. Added connection is shown above in red.

The subtract needs the address in the beginning of cycle 3, while it's in MS, but the value is computed at the end of cycle 3. If a bypass connection were provided then the cycle time would have to be long enough for both an add and a memory access, substantially lowering the clock frequency.

(*d*) Show the bypass connections (if any) and control logic so that the code below executes without a stall (and without causing other instructions to execute incorrectly, of course). The control logic should deliver a `BYPASS` signal to the EX stage at the right time, it should be `1` if a bypass of the type needed below is necessary. Do not connect it to anything.

☑ Show the control logic generating `BYPASS`.

☑ Add the bypass connection or show which existing one would be used.

```
# r1 = 0x1000, r5 = 0x1000
# Cycle              0  1  2  3  4  5  6
add.mr (r1), r2, 3   IF ID MS EX ME WB
sub.rm  r4, (r5), 6     IF ID MS EX ME WB
```

The bypass is only necessary if the two addresses are the same and so the control logic must compare the addresses as well as checking if the earlier instruction writes a memory location and the later one reads one. The control logic and existing bypass path are shown above in blue.

Problem 3: Answer each question below. Be sure to check each code fragment for dependencies. (25 pts)

(*a*) Show a pipeline execution diagram for the code fragment below running on the usual statically scheduled and bypassed scalar MIPS implementation. *Note: Bypassing and static scheduling not mentioned in the original exam.*

```
# Cycle         0    1    2    3    4    5   6   7   8

lw  r1, 0(r2)

add r3, r1, r4
```

```
# Solution:
# Cycle         0  1  2  3  4  5  6
lw  r1, 0(r2)   IF ID EX ME WB
add r3, r1, r4     IF ID -> EX ME WB
```

☑ Solve this easy problem.

Grading Note: The original test omitted the kind of implementation. Some did it on an unbypassed statically scheduled, a few did it on a dynamically scheduled system, and one on a superscalar statically scheduled implementation.

(*b*) The `sub.s` instruction below stalls in IF.

```
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11
add.s f4, f5, f6  IF ID A1 A2 A3 A4 WF
sub.s f7, f4, f8     IF -------> ID A1 A2 A3 A4 WF
add.s f9, f10, f11                IF ID A1 A2 A3 A4 WF
```

☑ What's wrong about the stall?

☑ Show correct execution.

It's stalling because of the dependency through `f4`, but how could the implementation know that in cycle 1 to 4 if `sub.s` does not yet been decoded. The correct execution appears below.

```
# Solution
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11
add.s f4, f5, f6  IF ID A1 A2 A3 A4 WF
sub.s f7, f4, f8     IF ID -------> A1 A2 A3 A4 WF
add.s f9, f10, f11      IF -------> ID A1 A2 A3 A4 WF
```

Problem 3, continued:

(*c*) The code below executes on a dynamically scheduled machine of the type used in the first problem and in class. The `mul.s` stalls in ID.

```
# ROB initially empty.
# Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
add.s f4, f5, f6  IF ID Q  RR A1 A2 A3 A4 WF C
sub.s f7, f4, f8     IF ID Q           RR A1 A2 A3 A4 WF C
mul.s f9, f10, f11      IF ID -------> Q  RR M1 M2 M3 M4 M5 M6 WF C
```

☑ What's wrong about the stall?

☑ Show correct execution.

This is a dynamically scheduled system and so `sub.s` waiting for an operand does not stall `mul.s`. Instruction `sub.s` moves out of ID and into the instruction Q, and so `mul.s` is not blocked.

☑ If the "`ROB initially empty.`" comment was not there then the execution above would be possible, albeit misleading. How would the stall be possible?

Dynamically scheduled systems stall when some resource is exhausted. The comment indicates that the three instructions have the entire processor to themselves and so it's unlikely that it has run out of anything. But without the comment there might be lots more instructions present and so the processor might have run out of something, for example, ROB slots or physical registers. If it runs out it must stall the instruction in ID. Unlike waiting in the instruction queue, this is a real stall of the fetch/decode pipeline that will block later instructions.

(*d*) Show a pipeline execution diagram for the code below executing on a 2-way statically scheduled superscalar MIPS implementation. *Note: In the original exam "statically scheduled" was omitted so an answer for either type of system would be correct.*

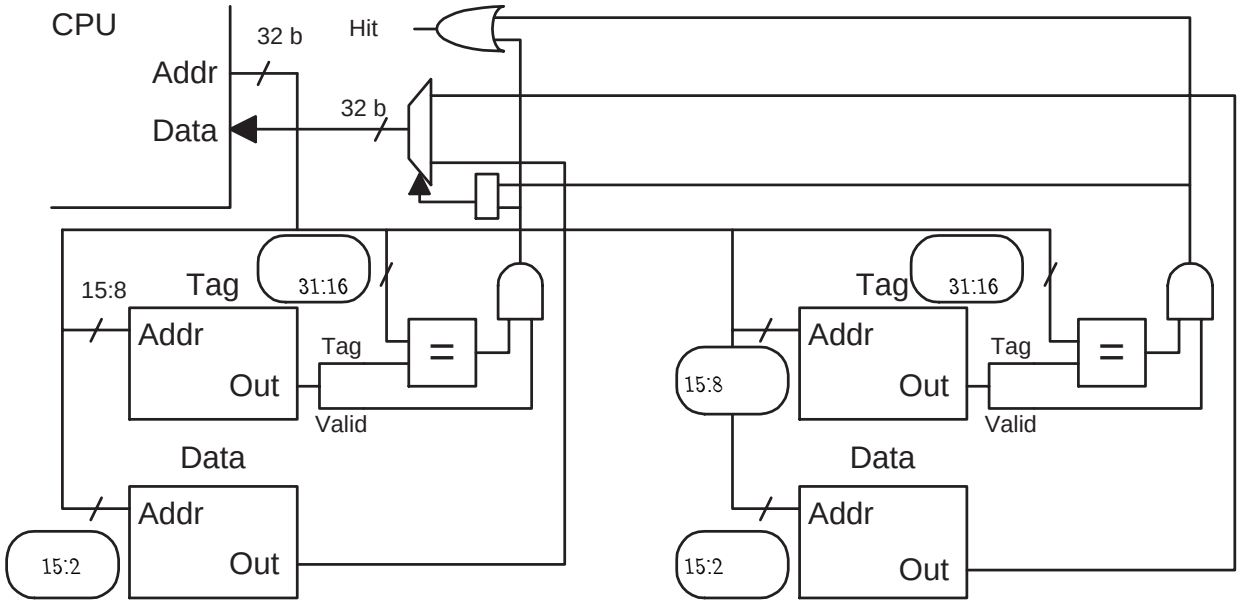
☑ Execution must be for decode logic of ordinary complexity. (See next item.)

☑ Execution must allow precise exceptions.

```
 # solution
 # Cycle         0  1  2  3  4  5  6  7
0x1000:
 lw  r4, 0(r5)   IF ID EX ME WB
 add r1, r2, r3  IF ID EX ME WB
 lh  r6, 0(r4)      IF ID -> EX ME WB
 xor r7, r1, r9     IF ID -> EX ME WB
 sll r10, r11, 12      IF -> ID EX ME WB
 srl r14, r15, 16      IF -> ID EX ME WB
```

Problem 4: (10 pts) The diagram below is for a two-way set-associative cache on a system with 8-bit characters. Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

☑ Cache Capacity (Indicate Unit!!):
Capacity is 2×2^{16} characters (128 kiB).

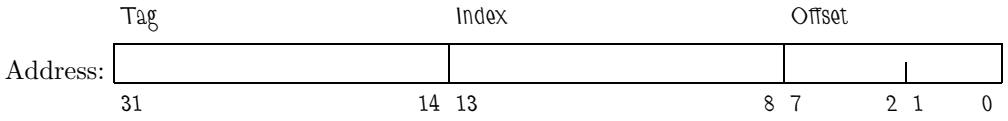
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $2 \times 2^{16-8}(32 - 16 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is 2^8 characters.

☑ Show the bit categorization for an eight-way cache with the same capacity and line size.



Problem 5: Answer each question below.

(*a*) The register renaming technique used in the dynamically scheduled processor (what the register maps do) solves a problem encountered when instructions execute out of order.(5 pts)

☑ What is the problem and how does it solve it?

☑ Provide an example showing what would go wrong without renaming.

The problem is that because instructions write back their results out of order and also read source registers out of order an instruction might read the wrong register value because (1) a later instruction already wrote the same register, overwriting the value that was supposed to be read, or (2) earlier instructions wrote the registers out of order. See the examples below. The solution is to assign a different physical register to each instruction that writes a destination so that there is no ambiguity. The ID register map is used to find which physical register is assigned to a given architected (from the ISA) register. Since the register map is read and updated in program order it will be correct.

```
# Case (1):  add reads wrong value of r2.
add  r1, r2, r3              ..RR..  # Ooops, got wrong r2.
sub  r2, r4, r5      ..WB..

# Case (2): xor reads wrong r1.

add  r1, r2, r3                ..WB..
...
or   r1, r6, r7      ..WB..
xor  r8, r1, r9                        ..RR..  # Ooops, got wrong r1.
```

(*b*) When it comes to caches one line size does not fit all.(5 pts)

☑ When is it better to have larger line sizes (while holding cache size constant)? Explain how the larger lines help.

When programs do alot of sequential access. The larger the line the more data is brought in on a cache miss. Since access is sequential that data will soon be accessed if the missing instruction is part of a piece of code accessing data sequentially.

☑ When is it better to have smaller line sizes (while holding cache size constant)? Explain how larger lines would hurt.

Suppose there is a miss on a system with large lines by a program doing no sequential access and with little spatial locality of any kind. The accessed data will be read but the other data on the line will not, wasting cache capacity. It would not hurt to have smaller line sizes since only the part which is first accessed is used, it would in fact help to have smaller lines because there would be more of them, increasing the hit ratio in many cases.

(*c*) Why would the SPEC CPU benchmark suite be less useful if it contained one integer program and one floating-point program? (5 pts)

☑ Why not just one of each?

Different programs have different characteristics making it difficult to choose one typical program. It might happen that the chosen program on a particular implementation does particularly well but that other programs don't do as well on that implementation. So if the suite had just one program it would not do a good job of predicting overall performance on a mix of programs.

(*d*) How would you answer a critic who said that the SPEC CPU benchmarks were rigged to make rich and powerful company I's products look good? (5 pts)

☑ They are not rigged because ...

... the benchmarks are chosen by SPEC members. Anyone can join SPEC and that includes I's competitors. If I were pushing benchmarks that favored its products the other members of SPEC, with their own hardware to sell, would object and the benchmarks would not be included in the suite.

Grading Note: The question is asking about the benchmarks themselves, some answered as though it were asking about the testing procedures.

Some answered that they are not rigged because SPEC is a non-profit who's mission is to provide fair tests. That does not make them immune to being influenced. The Washington, DC phone book would be alot lighter if one removed all non-profits (PACs, etc) that were not as impartial as they claim to be. What's important for SPEC is that its members have competing interests and so they will keep each other honest.

(*e*) What is the difference between a hardware interrupt, an exception, and a trap (as defined in class)? (5 pts)

☑ A hw interrupt's unique feature.

It has nothing to do with what's executing, it's triggered by a signal on a special processor port.

☑ An exception's unique feature.

It's tied to a particular instruction in which there was some problem with execution.

☑ A trap's unique feature.

It is an instruction, inserted to call a handler routine.

68 Spring 2004 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Monday, 29 March 2004, 13:40–14:30 CST

Problem 1		(40 pts)
Problem 2		(25 pts)
Problem 3		(35 pts)
Exam Total		(100 pts)

Alias Hazardous Data Dependency

Good Luck!

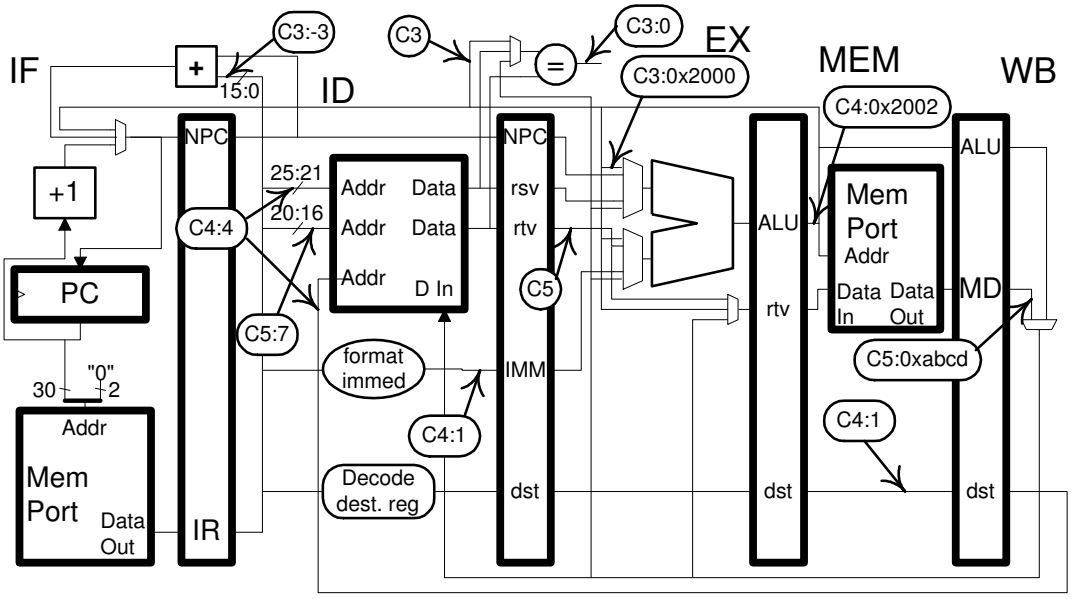
Problem 1: In the diagram below some wires are labeled with cycle numbers and values that will then be present. For example, C3:0 indicates that at cycle 3 the pointed-to wire will hold a 0. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. If a value on any labeled wire is changed the code would execute incorrectly. There are no stalls during the execution of the code. The first instruction (`or`) is shown (but don't forget to add the registers). [40 pts]

☑ Write a program consistent with these labels.

☑ Show the address of every instruction.

☑ Show every register number that can be determined and use `r10`, `r11`, etc. for other register numbers.

☑ Show the exact instruction (they can all be determined). For example, not just a load, but a load ____.




```
# Solution:

#                Cycle:  0   1   2   3   4   5   6   7   8
0x1000: or  r4, r10, r7   IF  ID  EX  ME  WB
0x1004: lhu r1, 2(r4)         IF  ID  EX  ME  WB
0x1008: bne r4, r5 TARG           IF  ID  EX  ME  WB
0x100c: sb  r11, 1(r4)                IF  ID  EX  ME  WB
0x1000: or  r4, r10, r7                   IF  ID  EX  ME  WB
#                Cycle:  0   1   2   3   4   5   6   7   8
```

Detailed discussion on next page.

Register numbers can be determined from boxes such as C4:4 and by the use of bypass paths (which tell you that the two registers are the same) such as C3:0x2000 (this indicates that **rs** register number of the second instruction is the same as the destination of the first; the **0x2000** provides additional information).

Each instruction can be determined from a variety of clues. The first instruction is given (**or**), the others are determined as follows:

lhu: The use of MD in cycle 5 tells us it's some kind of load. The effective address, C5:0x2002, is a multiple of two but not a multiple of 4, so it cannot be a load word. The loaded data, **0xabcd**, spans 16 bits so it can't be a load byte. It can't be an **lh** because the bit at position 15 is 1 and would have been sign extended (the data would be **0xffffabcd**) if it were. Therefore it must be a **lhu** (load half unsigned).

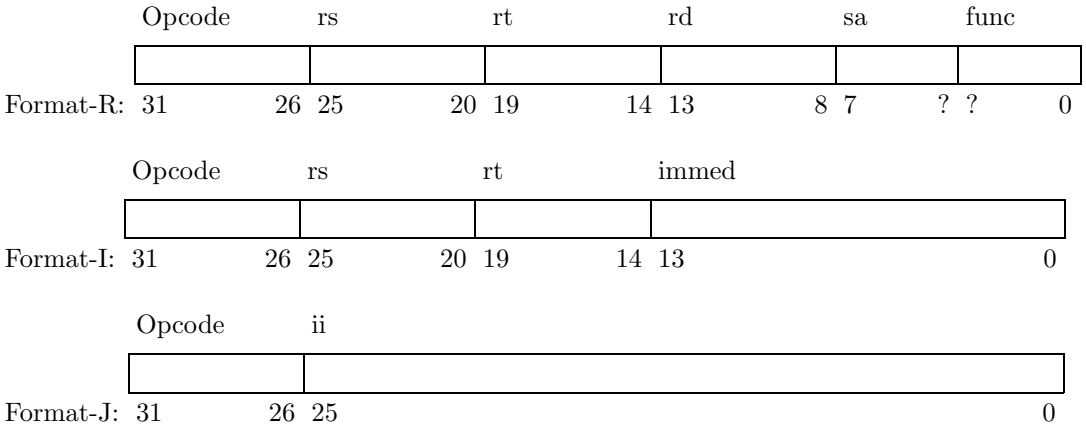
bne: The C3:-3 tells us it is a branch and that the branch is taken. (Because the problem description states that labeled wires are being used.) The C3:0 tells us it is either **bne** or **beq** (it can't be **bgtz**, etc. because those branches do not test equality or use the **rt** register value). The C3:0 also tells us that the two operands are not equal, since the branch is taken it must be a **bne** (branch not equal).

sb: This instruction uses both an immediate (C4:1) and the **rt** register value (C5) beyond the ID stage, so it must be a store. (Branches use the **rt** register value and an immediate in ID, not beyond it.) The store address is **1(r4)**, while the **lhu** is **2(r4)**, since **r4** does not change the store address is not a multiple of 2 and so it can only be a **sb** (store byte).

or: The C3:-3 tells us that the branch is branching up three instructions (the delay slot instruction, **sb**, is zero), so we know the branch target is the first instruction, an **or**. So the last instruction is the same as the first.

Problem 2: Consider a new ISA, F-MIPS, similar to MIPS-I except that it has 64 rather than 32 general purpose registers. F-MIPS has R, I, and J instruction formats like MIPS-I but with modifications to handle the larger number of registers. A goal of F-MIPS is to have all of MIPS-I instructions.

Possible instruction formats are shown below. Some details of Format R are omitted and are the subject of the first question.



☑ [6 pts] Describe a problem with Format-R F-MIPS instructions using the Format R shown above.

Field sa or func (or both) would have to be shrunk. If sa were shrunk there would be no way to specify all shifts. If func were shrunk it might not be possible to code all instructions.

☑ [7 pts] Fix the problem in Format R (show the changes in the illustration above). Explain the impact (this is important) on the coding of F-MIPS instructions.

Keep the func field six bits and change the sa field to a reserved-for-future-use field. The immediate shift instructions can be encoded in Format I or the rs field can be used for the shift amount.

☑ [6 pts] Explain an impact on typical F-MIPS Format-I instructions (compared to their MIPS-I counterparts) that would not apply to Format-R instructions.

The immediate is smaller so branches can't branch as far and code that would require one immediate instruction in MIPS-I might require two instructions in F-MIPS.

☑ [6 pts] Why is something going to have to be done about `lui`? Describe a new version of `lui`, possibly using a new format, that would fix the problem. *Hint: That's load upper immediate.*

There should be some way to load a 32-bit immediate in two instructions, with the immediate now at 14-bits that can't be done with two Format I instructions. Code frequently needs to load a 32-bit immediate and so a solution that requires three or more instructions would have a significant impact on performance and code size. A two-instruction solution would be to have a new format, say Format Ib in which the immediate field extends into rt and rs is used for the destination.

Problem 3: Answer each question below.

(*a*) A company has to choose between developing two new implementations of their ISA. Implementation A would have a peak (result) score of 2200 and a base (baseline) score of 2000 on the SPEC CINT2000 benchmarks. Implementation B would have a peak (result) score of 2150 and a base score of 2100 on the benchmarks.

☑ [0 pts] Which implementation should the company choose? *Hint: Either answer is correct.*

Implementation A. *Implementation B.* (Both answers are discussed below.)

☑ [9 pts] Why? Your reason should say something about the difference between the peak and base scores and about the company's customers.

The base numbers are obtained by compiling the benchmarks with ordinary optimizations, the peak numbers are obtained by compiling the benchmarks with great skill and effort to get the best results.

Implementation A: This implementation gives the better peak performance, which is appropriate for our customers since they compile the code for each implementation and their conscientious programmers are experts at optimization.

Implementation B: This implementation gives the better base performance. Our customers do want high performance but not enough to choose programmers skilled in performance tuning over those who can get code out the door quickly.

(*b*) Should a BCD data type be added to a modern general-purpose ISA?

☑ [8 pts] Explain why or why not, using the criteria discussed in class for adding data types to an ISA. (Discuss specific features of BCD, don't give an answer that could apply to any data type.)

BCD represents numbers as a string of decimal digits, encoding each digit using four bits. Any fixed point decimal number (within range) can be represented exactly in BCD, such as 0.3. Such numbers cannot be represented exactly in fixed- or floating-point binary. For example, 0.3 is $0.0100110011001100\overline{1100}_2$.

Due in part to the inexact representation, certain computations on certain arithmetic hardware would produce results that are off by a small amount. The problem might occur with financial calculations and of course is considered unacceptable. An early solution to that problem was to include a BCD data type and have BCD arithmetic instructions.

In the IEEE 754 floating point standard rounding is precisely specified and the default rounding mode was chosen so rounding results in the expected answer. Most modern systems implement this standard and so financial computations can use floating-point arithmetic without fear of rounding errors.

Note: Both answers below were marked correct.

No, because few modern computer languages use BCD and so there is no need to support it in hardware. Compilers for languages that still use BCD can use software to perform BCD computations, so it will still be possible to run old code.

Yes, because there is still alot of COBOL code (smoking as killed off or incapacitated most, but not all, COBOL programmers) and there are other languages that still use BCD and their code must be supported.

(*c*) Re-write the SPARC code fragment below in MIPS-I. Use as few instructions as possible. [9 pts]

The `subcc` instructions performs a subtraction and sets the condition-code (CC) register bits based on the result. It also writes an ordinary destination register, in this case `r0`. The branch (and a few other) instructions read the CC register to determine if the branch (or other action) should be taken. The `be` instruction means branch if the result of the last cc instruction is equal to zero (assuming a subcc that can be interpreted as meaning branch if the operands are equal). Since the `subcc` writes the zero register there is no need for the MIPS code to include a subtract instruction, it can just compare the two registers.

```
! Notes: r0 is the zero register;  destination is rightmost register;
!        be means branch if equal.  Use the same register names.

! Note: Comments in SPARC code are part of solution.

subcc %r1, %r2, %r0   ! r0 = r1 - r2, set CC register.
add   %r3, %r4, %r5
be    TARG            ! Branch if result (from last cc insn) is zero.
xor   %r6, 10, %r6    ! Delay slot instruction.



# Solution shown below.

add r5, r3, r4
beq r1, r2, TARG
xori r6, r6, 10
```

(*d*) The code below includes a hypothetical MIPS predicated instruction. Re-write the code using real MIPS instructions. [9 pts]

The `add` is the predicated instruction. It is executed if `r1` is nonzero. In the equivalent code a branch instruction checks if `r1` is nonzero, if not the `add` instruction is skipped.

```
      sub r3, r6, r7
 (r1) add r2, r3, r4
      xor r5, r8, r7


# Solution below.

      beq r1, r0 SKIP
      sub r3, r6, r7
      add r2, r3, r4
SKIP:
      xor r5, r8, r7
```

Name Solution

Computer Architecture
EE 4720
Final Examination

13 May 2004, 17:30–19:30 CDT

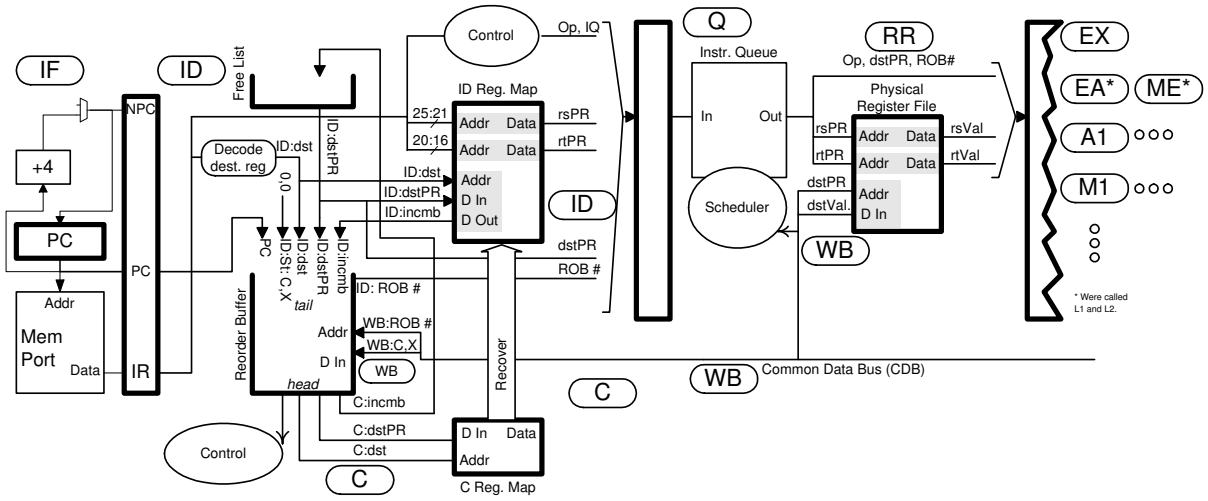
Problem 1		(19 pts)
Problem 2		(18 pts)
Problem 3		(19 pts)
Problem 4		(19 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias 17 December 1903

Good Luck!

Problem 1: The MIPS code below executes on the illustrated implementation. (19 pts)

- The implementation makes backup copies (not illustrated) of the ID map for each predicted branch instruction.
- The implementation is scalar, but there can be any number of writebacks per cycle.



The pipeline execution diagram on the next page shows the complete execution of the first instruction, a branch, and partial execution of the others. (Because of instructions before it, the branch enters `RR` after a wait of four cycles and commits after waiting another two.) The branch is mispredicted and some instructions will have to be squashed.

(*a*) Complete the pipeline execution diagram.

☑ Show where instructions are squashed.

☑ Show correct path instructions.

☑ Show each instruction until it is squashed or commits.

(*b*) Complete the tables:

☑ Physical register file. Assume the result of the load is 101, `xor` is 102, and `add` is 103. You can make up your own physical register numbers!

☑ Show where registers are removed from and put back in the free list.

☑ Complete the ID map. Don't forget to include the effect of branch misprediction recovery.

☑ Complete the commit map.

Problem 1, continued: Continued from previous page.

```
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 bneq r1, r2 SKIP     IF  ID  Q                   RR  B   WB          C

 lw r3, 0(r4)             IF  ID  Q   RR  EA              ME  WB          C

 xor r3, r8, r6               IF  ID  Q   RR  EX  WB        x

SKIP:

 add r3, r3, r7                   IF  ID  Q   RR  EX  WB    x IF  ID  Q   RR  EX  WB  C
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # ID Map

 Reg   Initial PR
 r1    19
 r2    20
 r3    27                     10  4   15                  10      4
 r4     2
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 # Commit Map

 Reg   Initial PR
 r1    19
 r2    20
 r3    27                                                                  10          4
 r4     2
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 Physical Register file

 PR    Initial State
  2    [r4
  4    ]                      [r3             102     x]      [               103
 10    ]                  [r3                                 101                     ]
 15    ]                          [r3             103 x]
 19    [r1
 20    [r2
 27    [r3                                                                 ]
 # Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Branch Misprediction Recovery: The branch resolves (taken or not taken is determined) in cycle 8 but the misprediction recovery does not start until WB. To recover from the misprediction the wrong-path instructions **xor** and **add** are squashed and the ID register map is returned to the state it was when the branch was decoded (register **r4** mapped to 10). In the next cycle fetch proceeds on the correct path, fetching the **add** for a second time. The first time around, on the wrong path, the **rs** source for the add was mapped to physical register 4; the second time around it maps to physical register 10. Note that the add is fetched twice even though it is on both the taken and not-taken paths. It would be possible but probably not worth the trouble to build a processor that would not have to fetch the **add** a second time in this situation.

Before recovery starts in cycle 10 the **xor** and **add** execute normally, unaware of the squash which they are fated to suffer.

Note: WB occurs as soon as the result is computed (after EX, ME, etc), it never stalls. In hand-solved problems there can be any number of WB per cycle (to reduce tedium).

A common mistake was to show **add** being fetched once, it is fetched twice as explained above.

Problem 2: Answer each question about the illustrated MIPS implementation. (18 pts)

(*a*) Complete a pipeline execution diagram for the code below running on the illustrated implementation. Assume that any needed bypass connection is available and please check for dependencies.

```
# Solution
# Cycle           0  1  2  3  4  5  6  7  8  9  10
add.d f6, f2, f4  IF ID A1 A2 A3 A4 WF
add.d f4, f6, f2     IF ID -------> A1 A2 A3 A4 WF
```

Note that the stall is in ID, an IF stall for the second `add.d` is impossible because there is no way to detect the dependency while the second `add.d` is still being fetched.

(*b*) In the diagram below add the bypass connection(s) needed for the code above. **Do not** add bypass connections that are not needed. (Don't worry if it's a tight squeeze in the diagram, but be legible.)

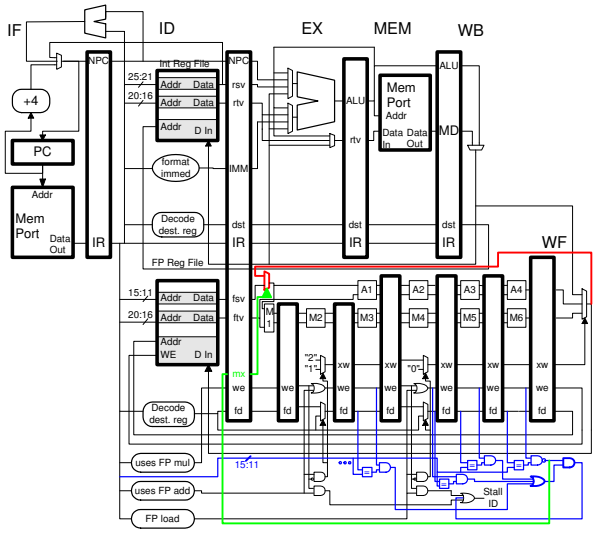
Solution shown in **red bold**. The output of the added multiplexor connects to both A1 and M1, though the problem would be correctly solved if it just connected to A1. By connecting it to both it can be used to bypass results to instructions that use the multiply unit. Similarly, the input of the added multiplexor comes from the output of the WF mux, though it could have come from the adder output in WF.

(*c*) Add control logic that generates the stall signal encountered in the code above.

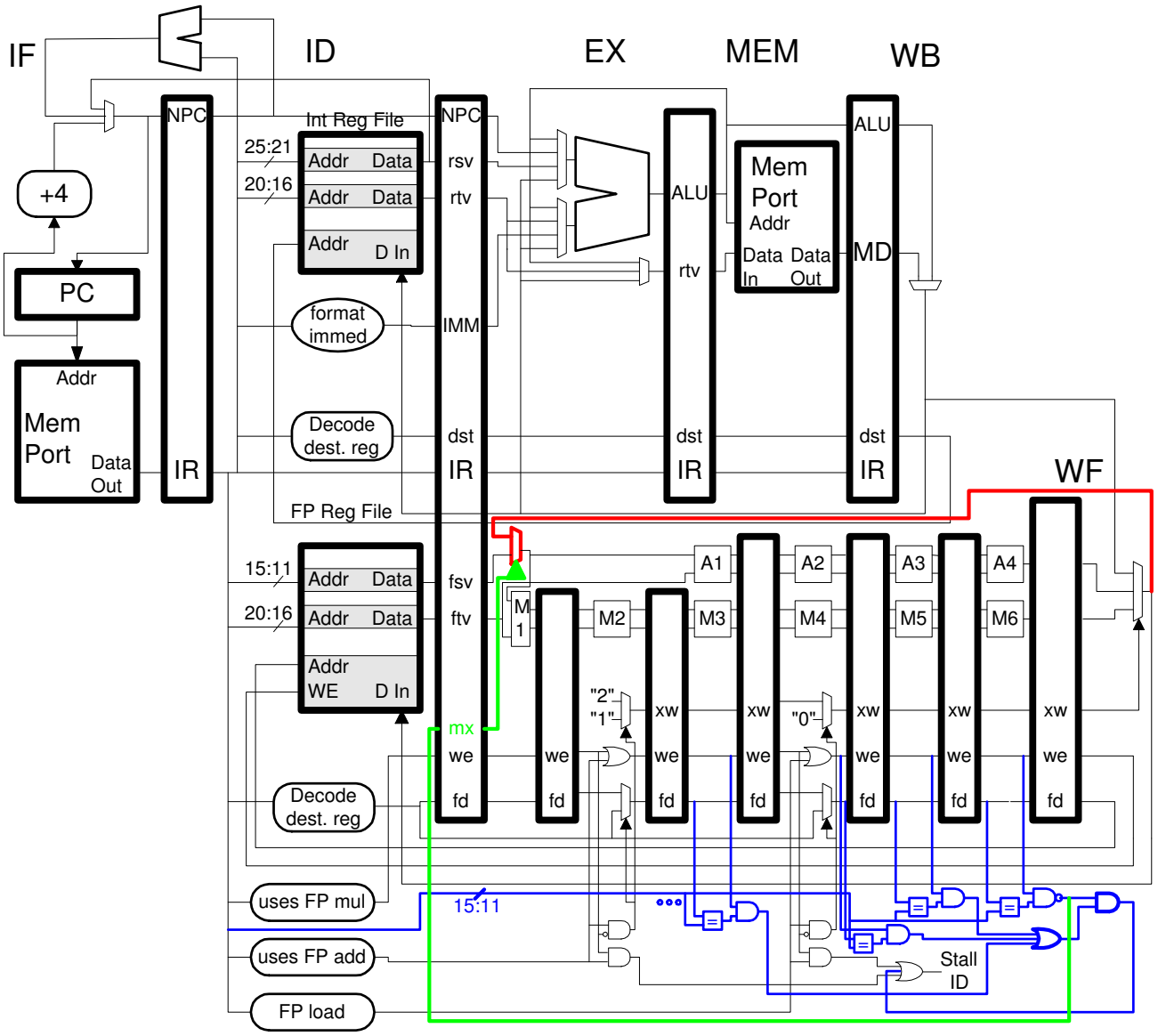
Solution shown in **blue bold**. To generate a stall the hardware needs to detect a match between registers to be written back, in the `fd` pipeline latches, and the `fs` register number. If there is a match between `fs` and the `fd` in A4, the value can be bypassed and there is no need to stall. If there is no such match but there is a match between `fs` and an instruction in A1, A2, or A3 then a stall signal is generated. A match between `fs` and `fd` is detected by comparing register numbers using a $\boxed{=}$ and checking whether the respective instruction writes anything at all, by looking at the `WE` (write enable) bit.

(*d*) Add control logic for the bypass multiplexor added in a previous part.

Solution shown in **green bold**. The logic used to generate the stall signal already checks if there is a match between `fs` and the `fd` of the instruction in the stage before writeback. That signal is used as the multiplexor control signal. Note that when the mux control signal is zero it will bypass otherwise it will use `fsv`.



*** Larger implementation diagram on next page.


IF
ID
EX
MEM
WB
WF
NPC
+4
PC
Addr
Mem
Port
Data
Out
IR
Int Reg File
25:21
20:16
Addr
Data
Addr
D In
format
immed
Decode
dest. reg
FP Reg File
15:11
Addr
Data
Addr
WE
D In
Decode
dest. reg
uses FP mul
uses FP add
FP load
rsv
rtv
IMM
dst
fsv
ftv
mx
we
fd
ALU
Mem
Port
Addr
Data
In
Data
Out
MD
A1
A2
A3
A4
M1
M2
M3
M4
M5
M6
"2"
"1"
"0"
xw
15:11
Stall
ID

Problem 3: The following questions refer to the assembly language code below. Branch outcomes are shown. (19 pts)

```
LOOP:
 0x1000:
B1: beq r1,r2 SKIP1     T  T  T  N  N  T  T  T  N  N  T  T  T  N  N
 nop                                                              !
 add r5, r6, r7                                                   A
SKIP1:                                                            !
 # **Ten** arithmetic instructions only until next branch, no other instructions.
 #                                                                !
B2: beq r3,r4 SKIP2      T  N  T  N  T  N  T  N  T  N  T  N  T  N  T  N
 nop
 add r8, r9, r10
SKIP2:
 add r11, r12, r13
 j LOOP
 nop
```

(*a*) Suppose the code above runs on a system using a bimodal branch predictor with a 256-entry branch history table. What would be the best prediction accuracy of branch `B1` and `B2` after warmup. *Hint: "Best" applies to* `B2` *but not* `B1`.

☑ Accuracy of `B1`. Just 40%.

☑ Accuracy of `B2`. At best, 50%.

☑ Why is there a "best" accuracy?

For `B2`, if its entry in the `BHT` starts out at 1 it will mispredict `B2` consistently, for an accuracy of 0%.

(*b*) Suppose the code above runs on a system using a local predictor with a 256-entry BHT. What is the smallest local history size that would allow the two branches above to be predicted with 100% accuracy?

☑ Smallest size. Explain

The smallest local history size is three bits. If it were two bits then for `B2` it could not distinguish the third consecutive taken from the first not taken because both would have a local history of `TT`. With three bits the local history would be `NTT` for the third taken and `TTT` for the first not-taken. Branch `B1` would only need one bit (but in a real system that would interfere with other branches), the local history would be the larger of the two, 3. Note that there would be no interference.

Problem 3, continued:

(*c*) Suppose the code runs on a system using a gshare branch predictor with a 10-bit GHR. Show the contents of the GHR at time `A` (see the right-hand side of the diagram).

☑ Gshare GHR contents:

The global history register would hold: **NTNNTTTNTT**, with the most recent outcome the rightmost character.

(*d*) Suppose the code runs on a PPC970 (or a POWER4) processor. Show the contents of what it uses for a GHR at time `A`. State any assumptions. *Note: This part based on a homework assigned Spring 2004, and would not be asked other semesters.*

☑ PPC970 GHR contents:

The PPC970 predictor GHR inserts one bit per fetch group, which has as many as eight instructions and must be contiguous. A fetch group that does not contain a branch is treated like one having a not-taken branch.

To solve the problem one must figure out where the fetch groups are. Start at B2. That fetch group would end at B2+1 (the **nop**) and start with one of the arithmetic instructions, B2-6. The table below shows the last ten fetch groups (before A). The GHR would be **TnTtTntTnT**, as in the homework, an uppercase T or N is for a branch instruction, lower-case t is for the jump and lower-case n is for a group of instructions without a control transfer.

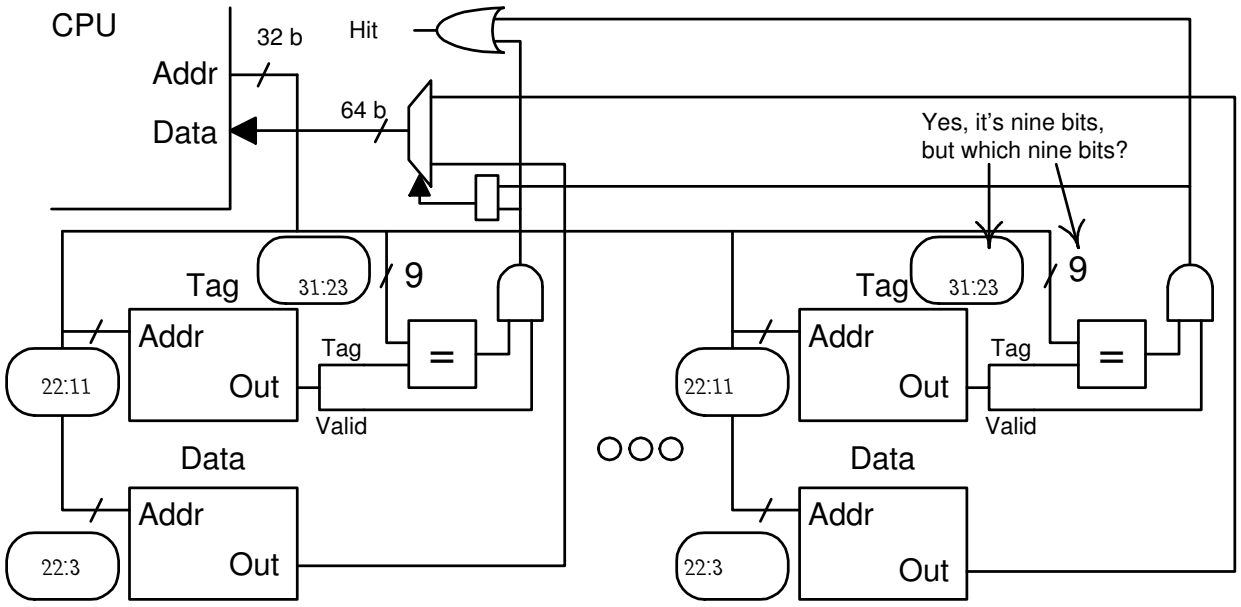
Fetch Groups:

Instructions	Branch/Jump/none	Outcome
B1 to B1+1	B1 Taken:	T
SKIP1 to B2-6	No cti.	n
B2-6 to B2+1	B2 Taken:	T
SKIP2 TO SKIP+2	Jump.	t
B1 to B1+1	B1 Taken:	T
SKIP1 TO SKIP2-6	NO cti.	n
SKIP2-5 to SKIP2+2	B2 not taken, jump:	t
B1 to B1+1	B1 Taken:	T
SKIP1 to B2-6	No cti.	n
B2-6 to B2+1	B2 Taken:	T

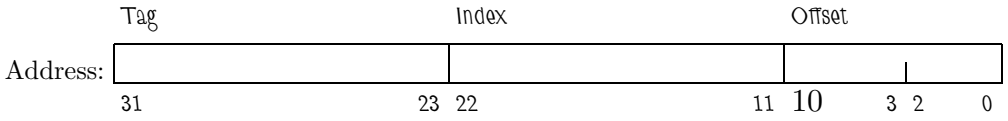
Problem 4: (19 pts) The diagram below is for a four-way set-associative cache on a system with 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Cache Capacity (Indicate Unit!!):

Capacity is 4×2^{23} characters (32 MiB).

☑ Memory Needed to Implement (Indicate Unit!!):

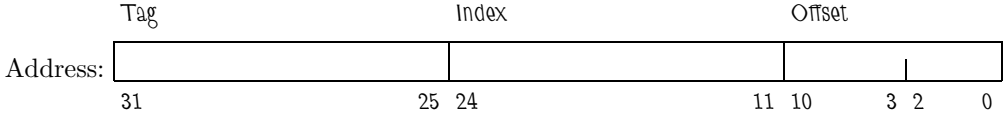
It's the cache capacity plus $4 \times 2^{23-11}(32 - 23 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is 2^{11} characters.

Note: Too many gave "bits" for the unit (but meant characters).

☑ Show the bit categorization for a **direct mapped** cache with the same capacity and line size.



Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

The array element size is two characters (sizeof(short int). The line size is $2^{11} = 2048$ characters and so 1024 consecutive `i` iterations will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $1023/1024$. The total amount of memory accessed is 1024 characters, which is much smaller than the 32 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{1023}{1024}+1\right) = \frac{2047}{2048}.}$

(*c*) Choose values for the address of `b`, `ILIMIT`, and `ISTRIDE` so that the cache is completely filled with the minimum number of accesses. *Note: The original exam read "minimum number of misses," is easier to do.* The address of `b` must be greater than `a` and as small as possible.

```
short int *a = 0x1000000;  // sizeof(short int) = 2 characters
int sum, i;


short int *b =


int ILIMIT =


int ISTRIDE =

  for(i=0; i < ILIMIT; i++)
    sum += a[ i * ISTRIDE ] + b[ i * ISTRIDE ];
```

The cache is four-way set associative with a tag field that starts at bit 22, which would be the fifth hexadecimal digit in the array `a`'s starting address. To fill the array with the fewest accesses each array access must be to a different line, that is, no line can be accessed twice. This can be achieved by setting `ISTRIDE` to half the line size. That is, `ISTRIDE = 1 << 10;` (that is, 2^{10}). Half because the array element size is two and so, for example, `a[0]` and `a[1]` will be on different lines and the indices of those lines will differ by 1. We would like `a` and `b` to access different lines and obviously `a` would load one half of the cache and `b` would load the other half. Therefore `b` should be set to `a`'s address plus half the cache capacity and so `b = a + ( 1 << 24 );` which is `b = 0x200000;` (that is, $b = a + \frac{4\times 2^{23}}{2} = a + 2 \times 2^{23} = a + 2^{24}$). There are many ways to determine what `ILIMIT` should be. Using the approach taken so far, `ILIMIT` should be set so that `a[0]` to `a[ILIMIT*ISTRIDE]` is half the cache capacity. Solve `2*ILIMIT*ISTRIDE==1<<24` or in math notation $2i_{\text{ilimit}}i_{\text{stride}} = 2^{24}$, rearranging: $i_{\text{ilimit}} = \frac{2^{24}}{2i_{\text{stride}}}$, substituting: $i_{\text{ilimit}} = \frac{2^{24}}{2^{2^{10}}} = 2^{13}$. So `ILIMIT = 1 << 13;`.

Problem 5: Answer each question below.

(*a*) One way to improve the performance of an implementation is to redesign the processor so that it uses more stages. Increasing the number of stages beyond five will yield a big improvement in performance, but at some point adding stages will have little effect.(5 pts)

☑ Give a reason for this limit having to do with the pipeline latches.

Signals must remain stable a certain amount of time at the input to a pipeline latch. In a five stage implementation the total time to execute an instruction includes five of these delays. With more stages there will be more delay, and eventually this will be a majority of the time needed to execute an instruction. One place this delay is seen is in misprediction recovery.

☑ Give a reason for this limit having to do with program characteristics.

If a functional unit is split across multiple stages then an instruction closely following and dependent on one using the unit must stall on a statically scheduled implementation. Adding stages will only add stalls in such cases. This limiter of performance gain is more severe for programs with more close dependencies. (If there are few close dependencies then there would be fewer stalls.) The limit also applies to dynamically scheduled systems in which the rate at which instructions commit would not increase linearly with higher clock frequency when there are enough dependencies.

(*b*) An instruction does not raise precise exceptions (because the ISA says it does not have to). (5 pts)

☑ Name something its handler cannot do (but could do if the exception were precise).

It cannot resume execution at the instruction immediately following the faulting instruction, nor can it re-execute the instruction itself. This is because if exceptions are not precise the handler might be called when the program executes several instructions past the faulting instruction.

(*c*) The code below uses an indirect load. Re-write it using MIPS instructions. (5 pts)

```
lw r1, @(r2)   # Load using indirect addressing.

# Solution.
lw r1, 0(r2)
lw r1, 0(r1)
```

(*d*) Every integer instruction that reads a GPR uses the `rs` and `rt` fields (or just one of them). What would be the disadvantage of a modified ISA in which some instructions use other fields to specify GPR source registers? (5 pts)

Multiplexors would be needed at the register file read ports to select the correct field for the instruction being decoded.

(*e*) Consider two options for the design of a load/store unit (LSU) for a dynamically scheduled system. The aggressive option would execute the code below quickly, while the conservative option would execute it more slowly. *Hint: the conservative option is how the LSU was described in class. Note: The original problem used* `r1` *for the address, which was not intended, but the answer with* `r1` *is similar.*

(5 pts)

```
div.d f2, f4, f6
sw r1, 0(r9)
sh r2, 0(r9)
sb r3, 0(r9)
lw r4, 0(r9)
```

☑ What is it about that code that requires special treatment?

The value to load is spread across three stores, bits 31:24 from the store byte, bits 23:16 from the `sh`, and bits 15:0 from the `sw` (assuming big-endian byte ordering).

☑ Describe how the aggressive option would execute the code.

An aggressive LSU would combine the data from the three stores when bypassing to the load.

☑ Give a brief argument for the conservative option, feel free to include made-up data (for the purposes of this test only!).

Studies show that only 1% of loads [warning, made up data] get their data from multiple stores (as in the example above). The cost of implementing an LSU that can handle such rare cases is high and so it is not worth implementing. The conservative solution would be cheaper and would be almost as fast.

69 Fall 2003 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 22 October 2003, 10:40–11:30 CDT

Problem 1		(25 pts)
Problem 2		(15 pts)
Problem 3		(18 pts)
Problem 4		(18 pts)
Problem 5		(24 pts)
Exam Total		(100 pts)

Alias 5d 5d 5d 5d 5d

Good Luck!

Problem 1: The routine below is called with an unsigned integer in register `$a0` and the address of some allocated memory in register `$a1`. When it returns the memory at should `$a1` contain the hexadecimal representation of `$a0` as a null-terminated (C format) string. Complete the routine, follow the guidelines in the comments. (For **partial credit** write a routine that converts a string holding a hexadecimal number to an integer.) [25 pts]

A solution including test code can be found at `http://www.ece.lsu.edu/ee4720/2003f/mtconv.html`

```
################################################################################
# utoh: Convert unsigned integer to hexadecimal string.
#
        # $a0: Call Value: Unsigned integer to convert.
        # $a1: Call Value: Address of allocated memory.
        #      Write converted string to this address, assume there is enough.
        #      Sample strings: "1F3", "1", "0" written at $a1.

        # [ ] String should not have leading zeros. (Good: "123", Bad "00123".)
        # [ ] Fill as many delay slots as possible.
        # [ ] Registers $a0-$a3 and $t0-$t7 can be modified.
        # The ASCII value of '0' is 48,  the ASCII value of 'A' is 65


        ## Step 1: Count number of leading zeros in hexadecimal representation
        # of $a0 (actually $a0  1 ) and use count to set $t3 to the
        # address of what will be the last character in the string.
        #
utoh:   lui $t1, 0xf000     # Mask used for extracting digits (starting at MSD).
        addi $t3, $a1, 7    # Init $t3 to address of last char of 8-digit number.
        ori $t5, $a0, 1     # OR in a 1 so that when a0 = 0 output is not "".

ILOOP:  and $t0, $t5, $t1   # Extract a digit.
        slti $t4, $t0, 1    # Set $t4 to 1 if digit is zero.
        sub $t3, $t3, $t4   # Adjust end-of-string pointer.
        bne $t4, $0, ILOOP  # Loop if digit is zero.
        srl $t1, $t1, 4     # Shift the mask to the next digit.

        ## Step 2: Convert the number to a string.
        #
        sb $0, 1($t3)       # Null-terminate the yet-to-be-written string.
LOOP:   andi $t0, $a0, 0xf  # Extract LSD (least significant digit).
        slti $t1, $t0, 10   # Check if it will be 0-9 or A-F
        bne $t1, $0, SKIP
        srl $a0, $a0, 4     # Shift in next digit (for next iteration).
        addi $t0, $t0, 7    # Add 7 to digit if it is a letter.
SKIP:   addi $t0, $t0, 48 # Add 48 to digit. (Seven also added if it's a letter.)
        sb $t0, 0($t3)      # Store the ASCII value in string.
        bne $t3, $a1, LOOP  # Loop if we have not written the first character.
        addi $t3, $t3, -1

EXIT:   jr $ra
        nop
```

Problem 2: Code similar to the histogram program presented in class appears below. MIPS instruction formats are shown below for reference. [15 pts]

(*a*) Design three new MIPS instructions to reduce the size of the program fragment below

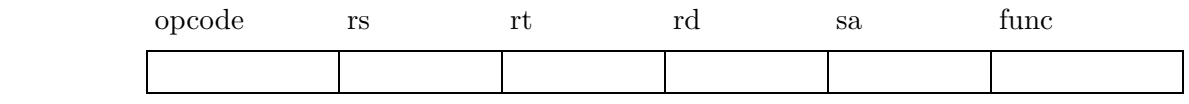
- Each new instruction should replace at least two related instructions in the program below. (Do not combine two *unrelated* instructions, such as `j` and `sw`.)

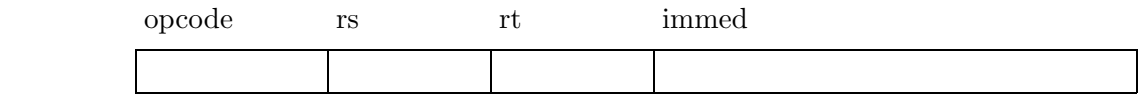
☑ Show the coding for the new instruction, making up the opcode and other field values as necessary. The coding should use one of MIPS' existing formats and should fit as naturally as possible.

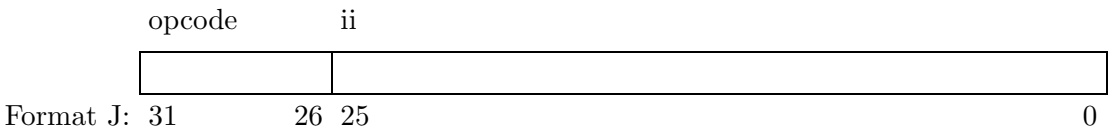
- Do not worry whether the instruction is appropriate for a RISC ISA.

The solution is on the next page.

```
 addi $t7, $0, 26
LOOP:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
 beq $t1, $0, DONE
 addi $t1, $t1, -65
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
 addi $t4, $t4, 1
 j LOOP
 sw $t4, 0($t3)
```







(*b*) Show an instruction that can be created by combining several instructions from the program but that would be impossible to code. Explain why it would be impossible to code.

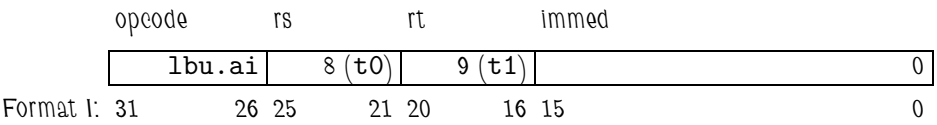
The solution is on the next page.

Solution to part (a).

Combine `lbu` and `addi` to form an autoincrement add, `lbu.ai`. The coding for `lbu` provides all the information needed by `lbu.ai` so the only difference is in the opcode field. (The amount to autoincrement by is the size of the item loaded, in this case one character, it would be two characters for a `lhu.ai`, etc.)

MIPS does not include an autoincrement load because a non-superscalar implementation would either have to write back to two registers simultaneously (an added cost) or else would be subject to stalls. (An n-way superscalar implementation might have to write back to $2n$ register per cycle.) At least one RISC ISA, PA-RISC, has an autoincrement add despite the difficulties.

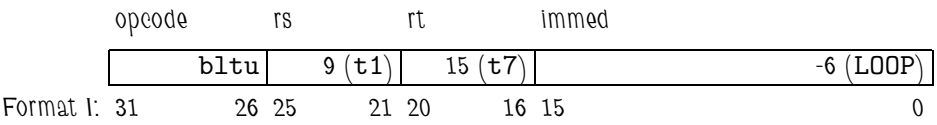
```
Combine:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
Into:
 lbu $t1, 0($t0)+
```



The `sltu` and `beq` can be combined to create a `bltu` instruction. As with the `lbu.ai` instruction, no new fields are needed, the coding would be the same as the `beq` instruction except for the opcode.

(MIPS does not include such an instruction because the time needed to compare to values would be a bit too long. Some other RISC ISAs do include branch instructions that can compare if one register is less than another, for example, SPARC V9.)

```
Combine:
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
Into:
 bltu $t1, $t2, LOOP
```

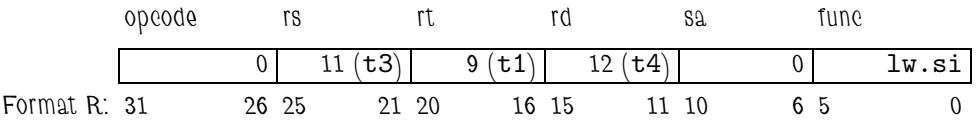


The `sll` and `add` could be combined to make a scaled add, in which the second source operand is shifted by two. But why stop there, the `lw` could also be included to create a scaled indexed load, `lw.si`. The scaled indexed load has three register arguments and so format R must be used. This means that it cannot use a 16-bit displacement. If the scaled indexed load is used only to load word-sized array elements then there would be no need to include the shift amount in the instruction (for the same reason there is no need to include the increment amount in an autoincrement instruction). The sa field might be used for small displacements or it might be used for other shift amounts (for example, loading a word from an array of 64-character structures). Since neither is needed here, the field will be unused and set to zero.

If `lw.si` is used then a value for `t3` is not written and so something must be done for the `sw`. The natural thing to do would be to add a `sw.si` instruction.

(MIPS does not include an instruction like `lw.si` because of implementation complexity. At least one other RISC ISA does, PA-RISC.)

```
Combine:
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
Into:
 lw.si $t4, ($t3,$t1)
```



Solution to part (b):

An instruction is impossible to code if there is not enough room in the format for all the registers and constant needed. One handy instruction that can't be coded is `bgei`, a branch that compares a register's contents to an immediate. Since the immediate field is also needed for the displacement that would be impossible.

It is possible to add a new format to MIPS in which this instruction can be coded, but the problem stated existing formats must be used. Though it is possible to add a new format, doing so would increase the complexity of implementations because additional paths must be added to move the immediate values to where they are needed.

```
Combine:
 addi $t7, $0, 26
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
Into:
 bgei $t1, 26, LOOP
```

Problem 3: Answer each question below.

(*a*) In MIPS (and similar ISAs) there is a `lb`, `lbu` (load byte unsigned), and a `sb` but there is no `sbu` (store byte unsigned). Why not? [6 pts]

Because an unsigned variant is only needed when writing a value to a location that's larger than the value. For `lb` an 8-bit value is written to a 32-bit register, and so the instruction must specify what to do with the other 24 bits (sign extend for `lb`, set to zero for `lbu`). For `sb` an 8-bit value is written to an 8-bit memory location and so there are no extra bits to set.

(*b*) Explain why the MIPS instruction below won't work: [6 pts]

```
add.d $f1, $f2, $f3
```

The instruction uses double-precision floating-point operands, these can only be retrieved from even-numbered registers.

(*c*) Show the result of each add instruction below. The instructions execute on a machine with 32-bit registers that is capable of BCD, and packed integer, and ordinary integer arithmetic. The packed integer operations all use saturating unsigned arithmetic. [6 pts]

```
# r1 = 0x8080888
# r2 = 0x1020999

                                Solution Shown Below
# Ordinary Integer Add
add r3, r1, r2            r3 =  0x90a1221  = 151654945 (decimal)

# BCD Add
add.bcd r3, r1, r2        r3 =  0x9101887  = 9101887 (decimal)

# Packed Integer (4 bits per int.)
add.p4 r3, r1, r2         r3 =  0x90a0fff  = { 9, 0, 10, 0, 15, 15, 15 } (decimal)

# Packed Integer (8 bits per int.)
add.p8 r3, r1, r2         r3 =  0x90a11ff  = { 9, 10, 17, 255 } (decimal)
```

Problem 4: Answer each question below.

(*a*) A company compiles and runs the SPECint2000 benchmarks on its new system, complying with all rules except one: it refuses to divulge the steps it used to compile the programs. Nevermind that it's against the rules, is it in the company's interest to keep this information secret? Explain. [6 pts]

No, because customers would want to know how to compile their programs to run quickly, and it would only help the company it they succeed.

(*b*) A program is compiled two ways, one for ISA A, implementation x, the other also for ISA A, but for implementation y. Explain what would be common to the two executables and what would be different. Provide an example. [6 pts]

The kinds of instructions would be the same since it's the same ISA. The choice of instructions in a particular place and their order would be different.

Grading Note: An executable is the compiled and linked program. Many answered the question as though an executable were a system running a program.

(*c*) Explain the dead-code elimination (DCE) optimization using an example. [6 pts]

A compiler performing the DCE optimization ignores code that writes register values that are never used.

```
Before:
 a = 3;
 a = b + 1;
After:
 a = b + 1;
```

Problem 5: Answer each question below.

(*a*) CISC ISAs have variable length instructions. What advantages over RISC ISAs does that enable? Name at least two and briefly explain each advantage. [6 pts]

Smaller code size because when space isn't needed it's not there.

Larger immediates, because the instruction can be made as large as necessary to fit the immediates.

Grading Note: Many gave advantages of CISC that were unrelated to variable instruction size, such as arithmetic instructions being able to load operands from memory. To get maximum credit, please answer the question that was asked.

(*b*) RISC ISAs have fixed length instructions. What advantages over CISC ISAs does that enable? Name at least one and briefly explain each advantage. [6 pts]

Branch displacements can go further since they can specify the number of instructions to skip rather than the number of characters to skip. The instruction fetch mechanism is easier to design.

(*c*) Why are programs written in a stack ISA small? Be brief but as specific as possible. [6 pts]

Because there is no need to provide space for register operands in many instructions.

(*d*) Listed below are several behaviors or features. For each, explain whether it is usually an ISA feature, an implementation feature, or an ABI (application binary interface) feature. **Briefly explain why.** If it can fit into more than one category (for example, ISA or implementation) say so and explain why. [6 pts]

☑ An `add` instruction raises an exception on an overflow.

ISA. Because this determines what a program would do and so it should be part of the ISA.

☑ The unused field in an instruction must be set to zero.

ISA. Specifies what instructions should look like and specifies the behavior of an illegal instruction (an otherwise legal instruction with an unused field that is not set to zero).

☑ Procedure call saves return address in a particular register.

ABI or ISA. Many ISAs do not have special call instructions, instead they have jump-and-link instructions that can save a return address in any register. The ABI defines which of those registers should be used for a return address. Some ISAs do have special call instructions that save to a particular register.

☑ Multiply instruction takes four cycles.

Implementation.

Name Solution

Computer Architecture
EE 4720
Final Examination

9 December 2003, 10:00–12:00 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias 17 December 1903

Good Luck!

Problem 1: (20 pts) The execution of a MIPS code fragment on a dynamically scheduled machine is shown on the next page. The diagram shows the values on certain wires at certain cycles. For example, 4:65 means that at cycle 4 the labeled wire holds value 65. The physical register file table is completed, ID- and commit-map tables are blank.

- The FP add unit has 3 stages, the FP multiply unit has 5, and the EA and ME are used for loads and stores.
- All destination registers are floating point.
- WB and commit can be done in the same cycle (indicate with a WC).
- To keep things simple the result of every instruction is zero and there are no cache misses.

(*a*) The ID and commit register map tables are blank ...

☑ ... complete them (the ID and commit register map tables.)

☑ Show the correct architected register numbers, or for partial credit make them up. (Two are easy, the rest are interesting.)

☑ Show the initial values (just before cycle 0) in the ID and commit map tables.

Solution appears on next page.

(*b*) Complete the pipeline execution diagram.

☑ Be sure to show Q, RR, WB, C (or WC), and a possible functional unit.

(*c*) Write a program consistent with these tables and labels.

☑ Choose consistent instructions.

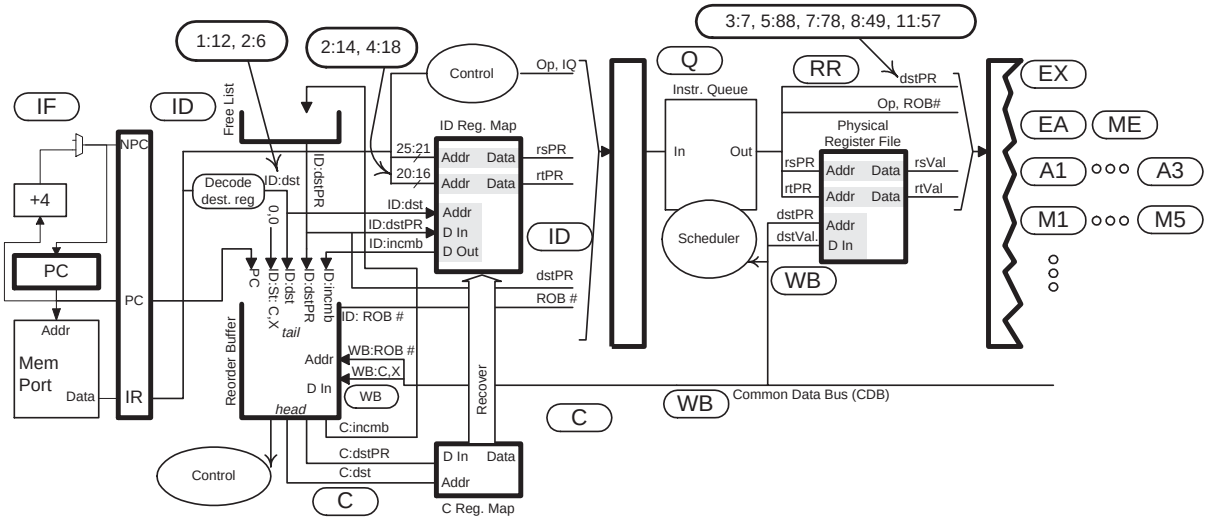
☑ Choose consistent registers. If a register number cannot be determined, use a question mark.

- *Hint 1: In the physical register file table, put a "1" next to the first (earliest) register removed from the free list, put a "2" next to the second register removed from the free list, and so on. Similarly, put a "1" next to the first register put back in the free list, etc. To figure out which physical register belongs to which instruction destination (easy) use the fact that certain events occur in program order.*
- *Hint 2: To figure out which architected register an instruction is writing (interesting) remember what causes a register to be put back in the free list.*

The solution appears on the next page. The key to completely solving the problem is understanding the physical register file. Physical registers are assigned in ID and that occurs in program order, and so the first physical register to be assigned (7) goes to the first instruction, the second physical register to be assigned (49) goes to the second instruction, etc. Physical registers are returned to the free list when instructions commit, so the first instruction commits at cycle 9, the second instruction commits at cycle 12, etc. When an instruction commits **it does not free its own physical register**, instead it frees the incumbent, the physical register used by the last instruction to write the same architected register. For example, when `ldc1` commits it frees physical register 7 which was assigned to `mul.s` because `mul.s` is the most recent instruction that writes `f12`, the same architected register that `ldc1` writes.

Every instruction goes to Q in the cycle after ID. The time an instruction uses RR is determined from the diagram (see the big bubble pointing to `dstPR`), the time it uses WB is determined by the physical register file (when it writes a value, 0. in each case here). The type of instruction is determined by the amount of time between RR and WB. For example, for the first instruction there are five cycles between RR and WB so it can only be using the multiply functional unit, and so it is most likely a multiply. (It's possible some other instruction uses the FP multiply unit, but in class it was always a multiply.)

Problem 1, continued: See previous page for instructions.




```
# Solution
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

mul.s f12, f14, f16   IF ID Q  RR M1 M2 M3 M4 M5 WC
add.s  f6, f12, f14      IF ID Q              RR A1 A2 A3 WC
ldc1.s f12, 0(r1)           IF ID Q  RR EA ME WB             C
add.s f12, f12, f18            IF ID Q     RR A1 A2 A3 WB       C
add.s f6, f12, f6                 IF ID Q              RR A1 A2 A3 WC
```

```
# ID Map       Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f6       95                 49       57
f12      33             7     88 78
# Commit Map   Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f6       95                                                49       57
f12      33                                        7           88 78
```

```
# Phys. Reg. File  Cy 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
 7                      [                                   0.              ]
33                                                          ]
49                          [                                       0.          ]
57                                  [                                           0.
78                              [                       0.
88                          [                   0.                      ]
95                                                                  ]
# Phys. Reg. File  Cy 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

Problem 2: (20 pts) In all of the problems below please check the code samples carefully for dependencies. All implementations below are fully bypassed. Please check the code samples carefully for dependencies.

Grading Note: A latency of 3 means a four-stage adder. Most people did not get this right, but points were not deducted for using a four-stage adder. (The initiation interval still has to be correct.) I think this will be the last test where the term latency in this sense is used.

(*a*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 2.

☑ Show a pipeline execution diagram for the code.

```
# Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
add.s f2, f4, f6    IF ID A1 A1 A2 A2 WF
add.s f8, f2, f12      IF ID -------> A1 A1 A2 A2 WF
add.s f14, f10, f16       IF -------> ID -> A1 A1 A2 A2 WF
```

Solution above. Note that the stage names are A1 and A2.

(*b*) The code below executes on a 1-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☑ Show a pipeline execution diagram for the code.

```
# Solution
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
add.s f8, f2, f12      IF ID -------> A1 A2 A3 A4 WF
add.s f14, f10, f16       IF -------> ID A1 A2 A3 A4 WF
```

Problem 2, continued:

(*c*) The code below executes on a 2-way statically scheduled machine in which the add FP functional unit has a latency of 3 and an initiation interval of 1.

☑ Show a pipeline execution diagram for the code.

```
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 # Solution
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 add.s f8, f2, f12   IF ID ----------> A1 A2 A3 A4 WF
 add.s f14, f10, f16    IF ----------> ID A1 A2 A3 A4 WF
 and r1, r2, r3         IF ----------> ID EX ME WB
```

Note that the third instruction does not enter ID even though there is space. This ensures that instructions in ID are in program order. Though its possible to design control hardware to handle out-of-order instruction in ID its probably not worth the trouble.

Grading Note: Many solutions used a second FP adder. It's not wrong, but its not necessary.

(*d*) In a correct solution to the problem above there should be at least one instruction for which a precise exception is impossible. If that describes your solution, show a pipeline execution diagram below in which all instructions could raise precise exceptions (even though they don't). It's also possible that in a correct solution to the problem above all instructions can raise precise exceptions. If so, show a pipeline execution diagram below in which some instructions cannot raise a precise exceptions. In the absence of exceptions all pipeline execution diagrams must show correct execution.

☑ Show the appropriate pipeline execution diagram, or show how the one above would be different.

☑ Identify those instructions for which precise exceptions are impossible (above or below) and explain why.

```
 # Solution
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID A1 A2 A3 A4 WF
 add.s f8, f2, f12   IF ID ----------> A1 A2 A3 A4 WF
 add.s f14, f10, f16    IF ----------> ID A1 A2 A3 A4 WF
 and r1, r2, r3         IF ----------> ID ----> EX ME WB
```

An instruction cannot raise a precise exception if an instruction after it writes back before the exception is detected. In the previous part suppose the third `add.s` raised an exception in cycle 10. It could not be precise because of the `and` instruction's writeback in cycle 9. One solution is to stall `and` so that it does not write back before the third `add.s`, that is what is shown above.

Problem 2, continued:

(*e*) The code below executes on a 2-way superscalar dynamically scheduled machine using method 3 (the only one covered this semester), the same one used in Problem 1. The FP add unit has a latency of 3 and an initiation interval of 1.

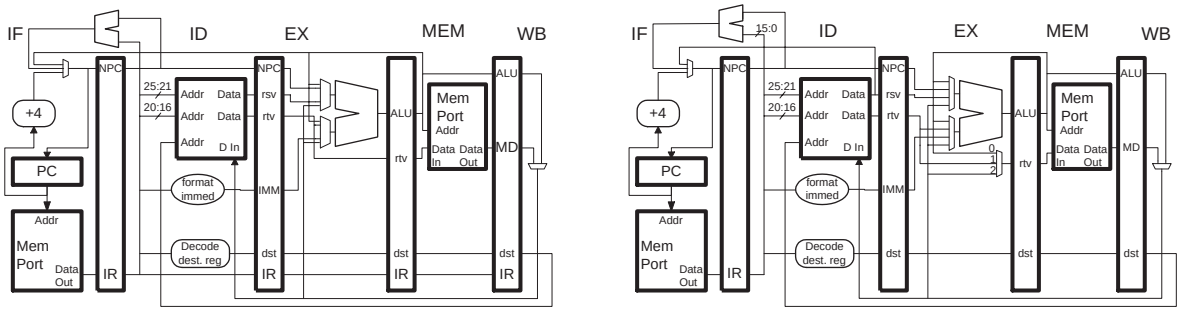
☑ Show a pipeline execution diagram. Don't forget the commit stage.

Assume an unlimited number of reorder buffer entries and physical registers.

```
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 # Solution
LINE: # LINE = 0x1000
 add.s f2, f4, f6    IF ID Q  RR A1 A2 A3 A4 WC
 add.s f8, f2, f12   IF ID Q              RR A1 A2 A3 A4 WC
 add.s f14, f10, f16    IF ID Q  RR A1 A2 A3 A4 WF       C
 and r1, r2, r3         IF ID Q  RR EX WB                   C
```

Grading Note: Many solutions had only one instruction committing in cycle 12. A 2-way superscalar processor must be able to commit two instructions per cycle. Also, many solutions used a second adder. A second FP adder is not needed in any of the problems.

Problem 3: (20 pts) Two MIPS implementations are illustrated below, the right one has a multiplexor at the input to the `EX/MEM.rtv` pipeline latch, the left one does not.



(*a*) Provide two code samples, one in which the multiplexor is useful and one in which it is not. Briefly explain.

```
# Mux Useful
add r1, r2, r3
sw r1, 0(r4)   # Mux used to bypass r1

# Mux Not Useful
add r1, r2, r3
add r4, r5, r1
sw r5, 0(r4)   # Not useful, r5 read from register file.
```

(*b*) Suppose version 5.11 of a compiler was written for the implementation on the left and is in the hands of customers. Version 5.99 of the compiler also includes the right implementation and is being released soon. Which two compilation options would you have to use to take advantage of the changes made for the right implementation? (The exact names of the compiler options is unimportant, but it should be obvious what they do.) Briefly explain why each option is necessary and how it would affect the code.

A compiler option is something put on the compiler command line or selected from a dialog box which tells the compiler how or what to compile.

The compiler for the left implementation would schedule instructions before a store so that the instruction producing the store value was more than two instructions before the store (the Mux Not Useful case). To take advantage of the new mux we would need to tell the compiler it is compiling for the right implementation and that it should optimize code. The options might be `-O` for optimization and `-impl=right` to select the right implementation. (Since many customers have the left implementation it would not be a good idea to compile for the right implementation by default.)

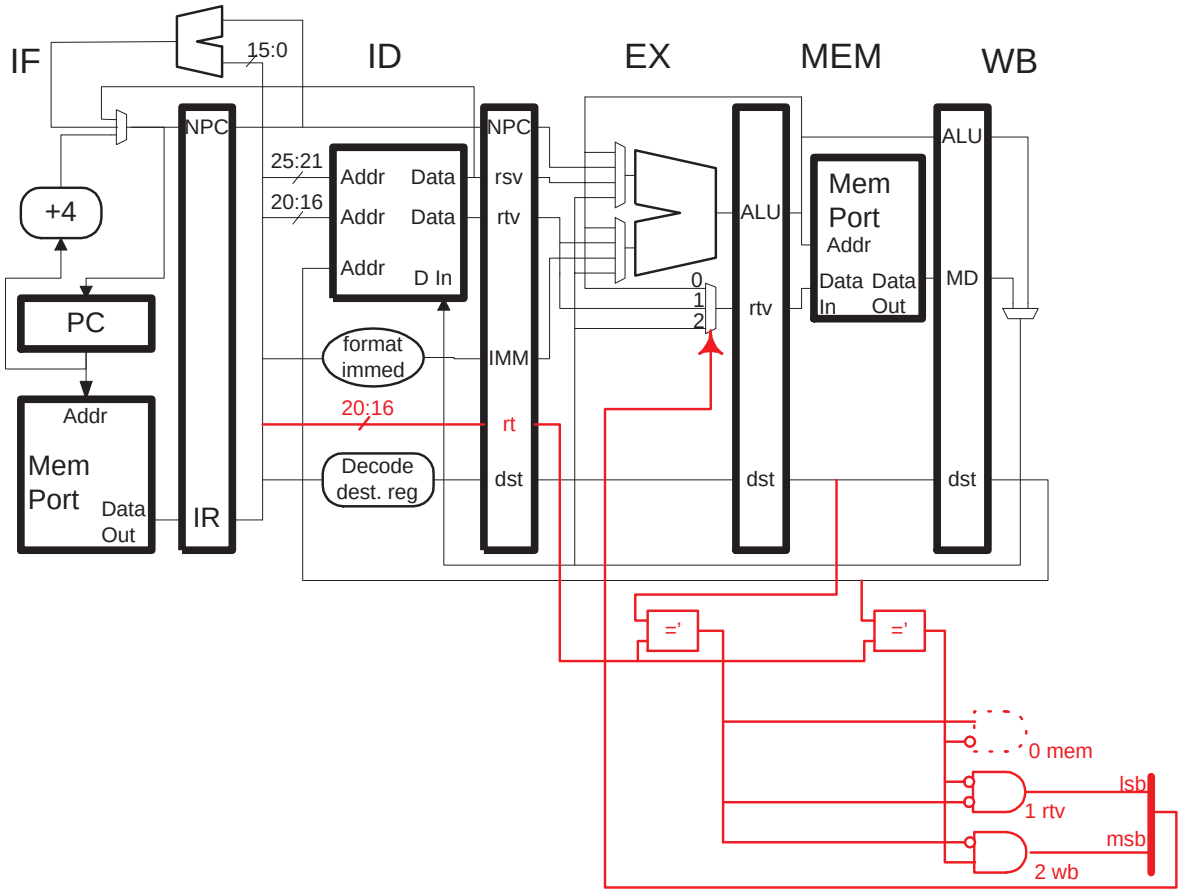
Grading Note: Many solutions correctly described what the compiler should do but did not specify anything like a compiler option. The question is asking about how compilers are used and so such answers did not receive full credit.

Problem 3, continued:

(*c*) Design the control logic for the `rtv` multiplexor. Unlike Homework 4, **the logic must be in the EX stage**. Where appropriate, show which bits are being used, e.g., `12:5`.

Changes shown in red below.

Grading Note: Too many solutions used `rtv` instead of `rt` in the `=?`. If it's not obvious why this is wrong then please review the basics of how the pipelined MIPS implementation works.



(*d*) There is a good reason why the control logic for the ALU input multiplexors should be in the ID stage that does not apply to the `rtv` multiplexor control logic. What is the reason, and why does it not apply to the `rtv` logic?

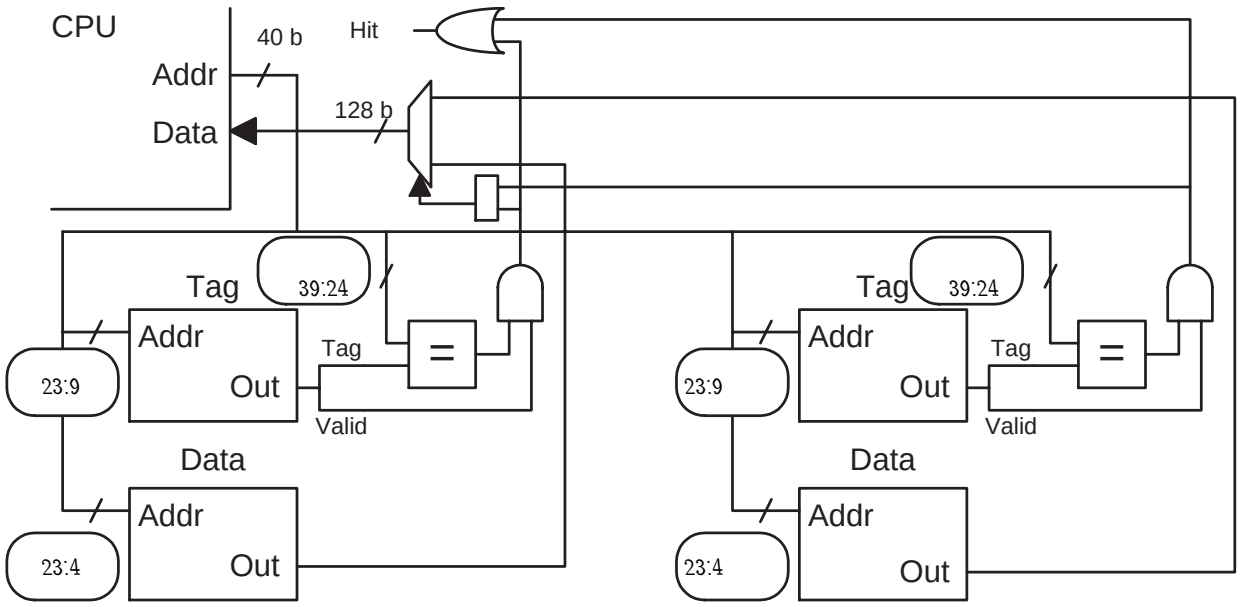
The ALU probably needs most of a clock cycle to compute. If the ALU input multiplexor logic were in EX then the ALU would not start useful computation until after the control logic generated the correct mux inputs and the muxen switched their inputs. Thus the control logic would be part of the critical path. Moving the logic to ID shortens the critical path. On the other hand, the `rtv` multiplexor is not part of a critical path, its output goes straight to the pipeline latch.

Grading Note: Some answered that the control logic for the ALU muxen would need information only available in ID, for example, the opcode. However, the `rtv` mux also needs information from ID (though not the opcode) so that reason applies to both. The ALU muxen would need the `rs` field value (and also `rt`) and one bit specifying whether the instruction uses the immediate, which could easily be passed through the pipeline latches.

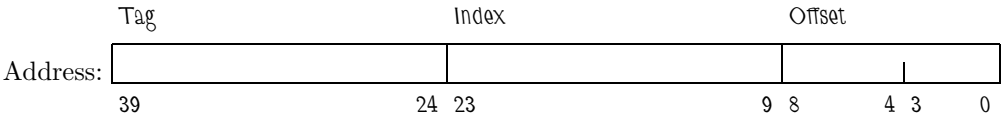
Problem 4: (20 pts) The diagram below is for a 32-MiB (2^{25} bytes) cache with 512-byte (2^9-byte) lines on a system with 8-bit characters.

(a) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

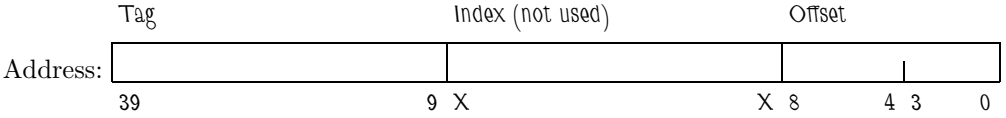


☑ Associativity: 2

☑ Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 \times 2^{24-9}(40 - 24 + 1)$ bits.

☑ Show the bit categorization for a **fully associative** cache with the same capacity and line size. *Note: Emphasis not included in original exam.*



Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array and assume the cache is cold (empty) when the code starts.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

The element size is one character. The line size is $2^9 = 512$ characters and so 512 consecutive `i` iterations will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $511/512$. The total amount of memory accessed is 1024 characters, which is much smaller than the 32 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{511}{512}+1\right)=\frac{1023}{1024}.}$

(*c*) Find the hit ratio for the code below running on the cache from the first part. Consider only accesses to the arrays and assume the cache starts out cold. State any assumptions made.

```
char *a = 0x1000000;  // sizeof(char) = 1 character
char *b = 0x2000000;  // sizeof(char) = 1 character
char *c = 0x3000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i ] + b[ i ] + c[ i ];
```

The cache is two-way set associative with a tag field that starts at bit 24, which would be the sixth hexadecimal digit in the arrays' starting address. Therefore the three arrays will have the same index (for the same value of `i`). Since at most cached two lines can have the same index the first access to `c` will result in the eviction of either `a` or `b`, if an LRU replacement policy is used then `a` will be the unlucky line. The next access to `a` will therefore be a miss instead of a hit, with LRU replacement `b` will be evicted. This pattern will continue resulting in a hit ratio of 0%.

(*d*) Modify the addresses of `a`, `b`, and `c` to maximize hit ratio. Explain how the modified addresses improve hit ratio.

The misses will be avoided if the three arrays use disjoint indices. There are many ways of doing this. One way is to keep the arrays close together, in that case: `a = 0x1000000, b = a + 1024 = a + 0x400 = 0x1000400`, and `c = b + 0x400 = 0x1000800`.

Problem 5: (20 pts) Answer each question below.

(*a*) The diagram below shows the branch outcome patterns for two branches.

```
# Loop contains only the branches shown.
BIGLOOP:
 Solution: Counter Value:      0  0  0  0  0  1  2  1  0  0  0  1  2  1  0  0  0  1  2
B1: 0x1000 beq $t1, $t2, SKIP1   N  N  N  N  T  T  N  N  N  N  T  T  N  N  N  N  T  T
 Solution: Misprediction        N- N- N- N- NX NX TX N- N- N- NX NX TX N- N- N- NX NX
...
B2: 0x1200 beq $v0, $v1, SKIP2    T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
...
    0x2010 j BIGLOOP
```

☑ How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table (BHT)?

The prediction accuracy is 50% (for a large number of iterations).

Grading Note: There were two common errors.

Error 1: Many predicted a branch *after* updating the counter using the branch outcome. (It should be obvious why this is wrong, or if not wrong, cheating.)

Error 2: Many computed the prediction ratio using the wrong number of branches. The repeating pattern contains six branches and that's what the prediction should be based on. The prediction outcomes have to repeat too which is why one should not base the prediction ratio on the first few branches.

☑ What is the minimum size of the BHT for which the accuracy in the previous part is possible? Explain.

The minimum size is $2^8 = 256$ entries.

If the first and second branch use the same entry in the BHT then they will interfere with each other. The first nine bits of the addresses of the two branches are identical. If those bits were used to index (as an address for) the BHT then the two branches would share an entry. The two low bits, being zero, are not used to index the BHT, and so in a table with a 7-bit address (2^7 entries) the two branches share an entry. If the table had an 8-bit address then the two branches would use separate entries and so the prediction accuracy would be preserved.

☑ Why might it be pointless to perform branch prediction in the ID stage of the 1-way statically scheduled pipeline used in class?

Because the branch outcome is determined in the ID stage and so there is no need to predict it there.

(*b*) Answer the following questions about exception codes as defined for SPARC V8 and using the class terminology.

☑ What is an exception code number (or trap type)? *Note: In the original exam the question was shorter: "What is an exception code?"*

It is a number that identifies the type of trap, hardware interrupt, or exception.

☑ How is it obtained for traps?

The trap instruction adds its `rs1` operand to an immediate or `rs2` operand, the low seven bits of the sum is the exception code.

☑ How is it obtained for hardware interrupts?

The exception code corresponds to the interrupt request port that was used to raise the interrupt.

☑ How is it obtained for exceptions?

It originates with the hardware that detects the exception, for example, an illegal opcode by the instruction decode hardware or segmentation fault by the data memory port.

☑ How is it used to start the handler?

The exception code is combined with the contents of trap base register to form the address of the handler. The high bits of the address are the contents of the TBR, the middle eight bits are the exception code, and the low four bits are zero (each table entry is 16 characters [4 instructions]).

(*c*) Consider two processors, one is a 6-way superscalar implementation of an ordinary ISA, say MIPS, the other is a 6-way implementation of a VLIW ISA, say Itanium (IA-64).

☑ Describe two features of Itanium (or some other VLIW ISA) that would allow it to execute faster than the superscalar implementation. Explain how the features allow faster execution.

Feature 1: Dependencies are specified in the bundle. This allows faster implementations by reducing the complexity of—and critical path length through—the logic checking for dependencies. With a shorter critical path the clock frequency can be higher.

Feature 2: Instructions arranged into bundles and CTI targets limited to bundle starts. In a non-VLIW superscalar processor in which instruction fetch is aligned on fetch groups unneeded (and ultimately unused) instructions may be fetched that precede the branch target because they are part of the same fetch group. With targets limited to bundle starts VLIW implementations do not suffer this inefficiency. A superscalar processor that was not restricted to fetch-group aligned fetches would have more complex logic in the fetch stage (or the cache port) to rearrange instructions, that would impact clock frequency or add stages, increasing branch penalty.

Feature 3: Instruction placement in VLIW is limited, for example, a VLIW ISA could forbid memory instructions in the first slot of a bundle. With such restrictions an implementation would require fewer multiplexors to send instructions to the proper functional unit. This certainly simplifies the logic (providing more room for cache or for other performance improving features) and might reduce critical path length.

Grading Note: Many people described the features but did not describe how they would improve performance.

☑ If the implementation for the VLIW ISA were faster why might the superscalar implementation still be better from a business perspective?

Suppose a company has many customers using an existing non-VLIW ISA. If the company were to offer a VLIW processor to customers using implementations of the non-VLIW ISA as the only way of getting higher performance from the company, those customers would have to re-compile their applications or buy new applications. Not every customer would be willing to do so. Some customers may be choosing the company's processor over competitors' to avoid the hassle of porting their applications to the competitors ISA. If such customers have to re-compile anyway they may switch to the competitors product.

70 Spring 2003 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 21 March 2003, 13:40–14:30 CST

Problem 1		(30 pts)
Problem 2		(30 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

Alias Freedom Quarter?

Good Luck!

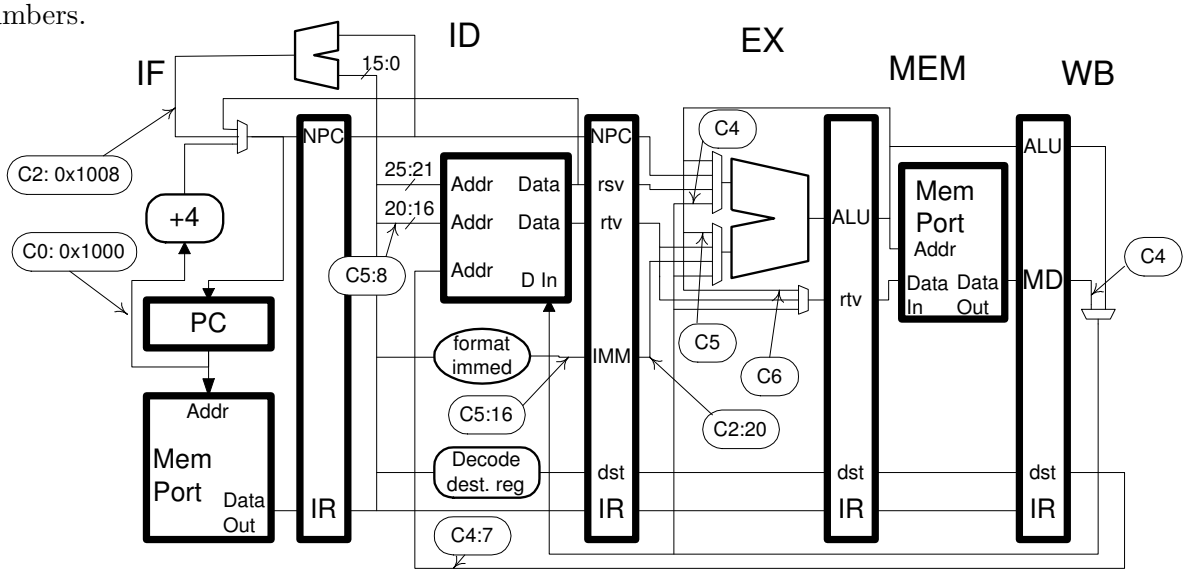
Problem 1: In the diagram below certain wires are labeled with cycle numbers and values that will then be present. For example, C5:8 indicates that at cycle 5 the pointed-to wire will hold an 8. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. There are no stalls during the execution of the code.[30 pts]

☑ Write a program consistent with these labels.

☑ The branch is taken. Use labels for branch targets and label the target line.

☑ Show the address of every instruction.

☑ Fill in every register number that can be determined and use `r10`, `r11`, etc. for other register numbers.



The solution appears below. One unusual feature is the branch. That the instruction in ID at cycle 2 is a branch can be determined by signal on the PC mux, C2:0x1008. The signal indicates that the branch target is `0x1008` which happens to be the delay slot instruction, and so the `add` executes twice, once as the delay slot instruction and once as the target. (The `C0:0x1000` indicates that the first instruction's address is `0x1000`.)

Grading Notes: Most people got this substantially correct. One relatively common error was omitting the branch.

The problem as given did not specify that there was a branch. Also, as given there was nothing to indicate that the branch was taken and so a solution with a not-taken branch was correct.

```
Solution:  Note that the add is executed twice.
       Cycle              0  1  2  3  4  5  6  7  8
0x1000 lw $7, 20($10)     IF ID EX ME WB
0x1004 bgt $11, TARG         IF ID EX ME WB
TARG:
0x1008 add $8, $7, $8           IF ID EX ME WB
0x1008 add $8, $7, $8              IF ID EX ME WB      [sic]
0x100c sw $8, 16($12)                 IF ID EX ME WB
       Cycle              0  1  2  3  4  5  6  7  8
```

Problem 2: The CISC-A ISA is making its world premier in this exam!

- It has 32 integer registers (the MIPS names can be used) and the address size is 32 bits.
- Each operand in each instruction can use any appropriate addressing mode, including register, immediate, and the full range of memory addressing modes described in class.
- Instructions are variable size, the entire first byte is the opcode.

(*a*) Re-write the code below in CISC-A, making up appropriate instructions as needed. [10 pts]

☑ Take advantage of CISC-A's characteristics to reduce code size within the loop (primary goal) and outside the loop (secondary goal). (See the next problem.)

☑ Take advantage of: the available *addressing modes* (memory, register, and immediate), the *variable instruction size*, and *operand flexibility*.

☑ A correct solution includes at least three big-improvement-over-MIPS instructions.

☑ Identify non-MIPS addressing modes used.

```
 lui $t0, 0x1234
 ori $t0, $t0, 0x5678
 lw $t2, 0($s0)
 lw $t2, 0($t2)
 addi $t3, $t0, 0x1000
LOOP:
 lw $t1, 0($t0)
 addi $t0, $t0, 4
 bne $t0, $t3  LOOP
 add $t2, $t2, $t1
```

```
 mov $t0, 0x12344578   # Uses 32-bit immediate.
 mov $t2, @($s0)       # Uses memory-indirect addressing.
 add $t3, $t0, 0x1000
LOOP:
 add $t2, $t2, ($t0)+  # Uses autoincrement addressing in an arithmetic insn.
 bneq $t0,$t3 LOOP     # No delayed branches.
```

(*b*) Describe a possible coding for three of the CISC-A instruction used above. Don't choose three similar instructions. [10 pts]

☑ It should be obvious (to a computer engineer) that the full range of operands is available.

☑ The coding must follow the • bulleted • points above.

☑ Instruction size should be small, though not the absolute minimum size.

Coding:

Every instruction consists of an opcode byte followed by zero or more sets of operand bytes. Each set of operand bytes specifies information about an operand (destination or one of the sources).

`opcode OPINFO1 [OPEXT1] [IMMED1] OPINFO2 [OPEXT2] [IMMED2] ...`

The first operand byte is OPINFO which is split into a 3-bit MODE field and a 5-bit NUM field. The MODE field specifies the addressing mode (see table) and the NUM might specify a register number, an immediate size, or something else. The OPEXT field is only used by some addressing modes as an extension of the MODE field (since the MODE field is only 3 bits it cannot specify all the addressing modes needed). The IMMED field, if present, holds an immediate, with the size specified in the size specified in OPINFO and possibly OPEXT.

```
MODE field values.
  0:  Register, NUM field holds register number.
  1:  Immediate, NUM field holds size and padding of immediate.
      The immediate itself is in the next NUM bytes.
  2:  Register indirect, NUM holds register number.
  3:  Register indirect autoincrement, NUM holds register number.
  4:  Register indirect autodecrement, NUM holds register number.
  5:  Memory indirect, NUM holds register number.
  6:  Direct, address in next four bytes.
  7:  Additional modes, NUM has mode details.
```

Problem 2, continued: Consider a second ISA, CISC-B, which differs from CISC-A in the following RISCy ways:

- Only load and store instructions can access memory.
- Each instruction has a fixed set of operand types, for example, an `addi` might be limited to one register destination, one register source and one immediate source.
- CISC-B still has variable-size instructions.

(*c*) Show how one such instruction might be coded. Try to choose a good example from the problem above. [5 pts]

☑ The coding should reflect and exploit CISC-B's operand restrictions.

Since the operand types are fixed there is no need for a MODE field. With the field omitted it is possible to specify three register operands in two bytes, rather than three bytes if each register number had to be accompanied by a MODE field.

(*d*) Show two program fragments: [5 pts]

☑ Show a program fragment that would be smaller in CISC-B than CISC-A. (No more than four instructions.)

```
add $t0, $t1, $t2
# CISC-A Coding: opcode opinfo1 opinfo2 opinfo3 (Four bytes)
# CISC-B Coding: opcode rd-rs1-rs2 (Three bytes, rd-rs1-rs2 is two bytes.)
```

☑ Show a program fragment that would be larger in CISC-B than CISC-A. (No more than four instructions.)

```
CISC-A:
 add $t0, $t1, ($t2)   # Four bytes.
CISC-B:
 load $t3, ($t2)       # Three bytes.
 add $t0, $t1, $t3     # Three bytes.  (Total six bytes.)
```

Problem 3: Answer each question below.

(*a*) Answer the following optimization questions. [10 pts]

☑ Explain the difference between front-end and back-end optimizations.

Back-end optimizations are performed at the machine-language level or close to it. Front-end optimizations are performed closer to the high-level language level.

☑ Can typical front-end optimizations be performed by the back end? Explain using an example.

Yes. For example, dead-code elimination. The optimizer would remove instructions that write registers that are never read (before being written again).

☑ Can typical back-end optimizations be performed by the front end? Explain using an example.

No. Many of these optimizations are based on machine instructions, or something close to it. For example, instruction scheduling. Since machine instructions are chosen after front-end optimization is done, there is no way for the front end to schedule (re-arrange) instructions.

(*b*) It is important to computer manufacturers to have high SPEC benchmark ratings. For each technique of improving SPEC ratings, describe whether it will work, and if it won't describe why not. [10 pts]

☑ Try to have a favorable benchmark added to the suite by sending the name of the benchmark and a little something for their trouble (a bribe) to morally weak SPEC members. *In the exam as given the sentence started "Have a favorable . . .".*

Some SPEC members work for the competition so they would not be easy to bribe. Could Compaq bribe Dell into making poor computers? If bribery did work the system would be corrupt and of little value because SPEC results could not be trusted. Without an accepted and trusted way of measuring performance, manufacturers would have little incentive to improve it. Societies that cannot control bribery will be doomed to non-competitive industries.

☑ Modify the source code to the benchmarks, honestly improving performance on your system.

It will not work because SPEC rules prohibit modifying source code.

☑ Modify your compiler so that it honestly produces faster benchmark executables.

This can be done, as long as the modified compiler is made available to the public (for free or as an ordinary product). If a company honestly improves its compiler it has every reason to make it available to their customers.

Grading Note: A common mistake was confusing compiler optimization switches with modifying the compiler itself.

☑ Report results with certain benchmarks omitted.

Not allowed. All benchmarks must be used.

(*c*) The typical ISA has one-, two-, four-, and eight-byte integers. [10 pts]

☑ Explain a possible benefit of having five-byte integers.

Smaller data size.

Grading Note: Many people gave the answer as smaller program size. This would be true to a small extent because of constant data, which could be five bytes, but the much larger benefit for most programs using the data type would be the reduction in space taken up by the data written to memory while the program is running.

☑ Explain a difficulty with adding five-byte integers, taking in to account a certain feature (some would call it a restriction) of many RISC ISAs. *Hint: The name of the feature begins with an "a".*

The a-word is alignment. If the memory system can only provide power-of-two aligned accesses then there would be no reasonable way of storing arrays of five-byte integers. If the first element of an array of five-byte items was stored in an aligned address the second one would not be.

(*d*) Provide the requested code examples and answer the questions. Two instructions suffice. [10 pts]

☑ Show code having a data dependency; circle the registers carrying the dependency.

```
add $t0, $t1, $t2  # Register t0 carries the dependency.
sub $t4, $t0, $t5
```

☑ What is the name of the corresponding hazard?

RAW hazard.

☑ Show code having an anti dependency; circle the registers carrying the dependency.

```
sub $t4, $t0, $t5  # Register t0 carries the dependency.
add $t0, $t1, $t2
```

☑ What is the name of the corresponding hazard?

WAR hazard.

Name Solution

Computer Architecture
EE 4720
Final Examination

14 May 2003, 15:00–17:00 CDT

Problem 1		(20 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

Alias The Next Spam

Good Luck!

Problem 1: The execution of a MIPS code fragment on a dynamically scheduled machine is shown in the tables and in the labels on the diagram, both on the next page. The tables show the contents of the ID Register Map, Commit Register Map, and the Physical Register File at each cycle. The diagram shows the values on certain wires at certain cycles. For example, 4:65 means that at cycle 4 the labeled wire holds value 65.

The following are functional unit segment labels: Load/store, L1 L2; floating-point add, A1 A2 A3 A4; floating-point multiply, M1 M2 M3 M4 M5 M6; integer, EX. The register maps handle both integer and floating-point registers.

(*a*) Write a program consistent with these tables and labels.(12 pts)

☑ Show a pipeline execution diagram, be sure to show where each instruction commits.

☑ Choose consistent instructions.

☑ Choose consistent registers. If a register number cannot be determined, use a question mark.

(*b*) Complete the tables on the next page as follows:(8 pts)

☑ Show where registers are added to, "]", and removed from, "[", the free list.

☑ Show the values on the line marked `X` in the illustration.

Solution is on the next page. Here is how the problem is solved:

Entries in the ID map are used to determine the destination registers (of the instruction in ID at that cycle) and when a physical register is removed from the free list.

Entries in the commit map are used to determine when an instruction commits. When an instruction commits the incumbent register (the same architected destination register used by an earlier instruction) is put back in the free list. The incumbent register is to the left of the committing register in the commit map table. (For example, at cycle 11 the instruction writing physical register 93 commits, the incumbent is 50 [and both are `f12`]). The incumbents are shown in the "X" row of the table, these physical registers are put back in the free list.

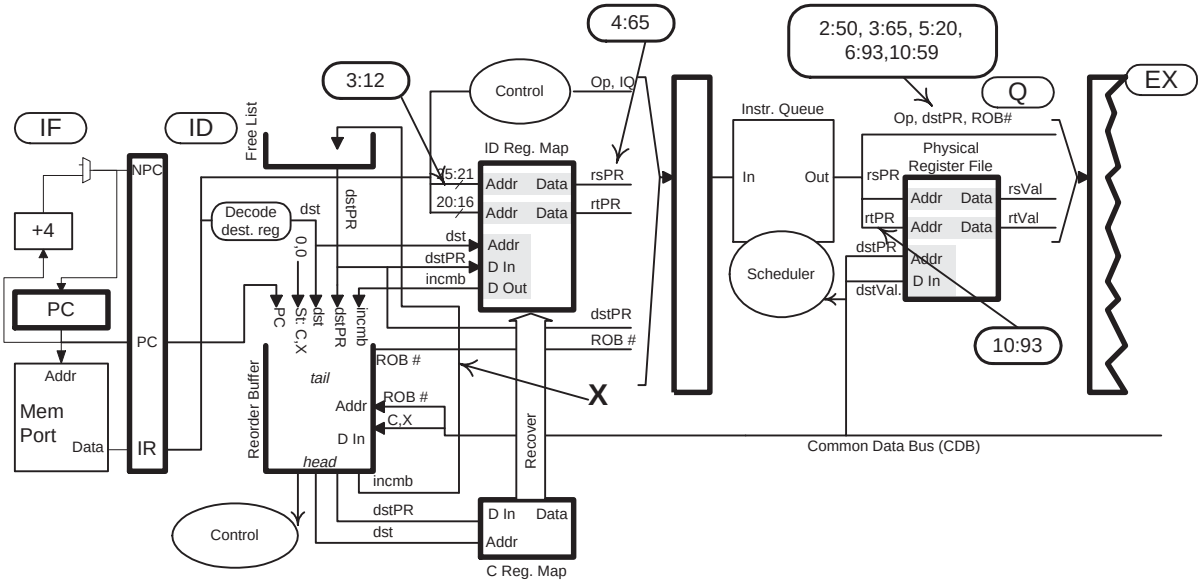
The big label in the diagram ("2:50, 3:65, etc.") shows when instructions are removed from the instruction queue to start execution. Qs are put in the pipeline execution diagram for this.

Entries in the physical register file table show when instructions write back. The time between Q and WB is spent in an execution unit, the amount of time (say four cycles) determines the type of unit and the type of instruction.

The three small labels in the diagram provide information about source registers, two are in ID, one is in Q.

The completed tables are on the next page.

Problem 1, continued: See previous page for instructions.




```
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
X (Solution)                                  7  3        50 10       93
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
# Solution
add.s f12, f??, f??   IF ID Q  A1 A2 A3 A4 WC
or   r5, r?, r?          IF ID Q  EX WB       C
add.s f12, f12, f??         IF ID       Q  A1 A2 A3 A4 WC
LwC1 f10, 0(r5)                IF ID Q  L1 L2 WB          C
add.s f12, f??, f12               IF ID             Q  A1 A2 A3 A4 WC

# ID Map       Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                 50    93    59
r5     3                    65
f10   10                          20
# Commit Map   Cycle  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
f12    7                                  50          93          59
r5     3                                     65
f10   10                                                    20

# Phys. Reg. File  Cy 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
# Solution
3             0                                 ]
7             0.                             ]
10            0.                                          ]
20                              [           0.
50                    [                     0.          ]
59                                  [                           0.
65                       [          100
93                          [                        0.          ]
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

Problem 2: The diagram below shows the branch outcome patterns for a branch. (15 pts)

```
BIGLOOP:

B1: 0x1000 beq $t1, $t2, SKIP1    N  N  N  T  T  N  N  N  T  T  N  N  N  T  T
...
B2: 0x1020 beq $v0, $v1, SKIP2
...
    0x2010 j BIGLOOP
```

☑ How accurately would branch B1 be predicted by a bimodal (one-level) branch predictor with a 2^{14}-entry branch history table?

For the version using a 2-bit counter the accuracy is $\frac{2}{5} = 40\%$. See the diagram below.

```
Counter: 0  0  0  0  1  2  1  0  0  1  2  1  0  0  1
B1:        N  N  N  T  T  N  N  N  T  T  N  N  N  T  T
Predict:  N  N  N  N  N  T  N  N  N  N  T  N  N  N  N
Correct:   Y  Y  Y  N  N  N  Y  Y  N  N  N  Y  Y  N
```

Repeating pattern has two correct, three wrong predictions.

☑ How accurately would branch B1 be predicted by a local history predictor with a 10-bit local history and a 2^{14}-entry branch history table?

The repeating pattern can easily be handled by a 10-bit local history so the accuracy is 100%.

☑ What is the minimum local history size needed to predict B1 with 100% accuracy (after warmup). No partial credit without an explanation.

Three outcomes. Just 3. See the table of patterns (NNN, etc.) and predictions.

```
Pat Pred
NNN T
NNT T
NTT N
TTN N
TNN N
```

☑ Find a pattern for branch B2 that will reduce the accuracy of the local predictor on branch B1. The branch history table (not to be confused with the pattern history table) remains 2^{14} entries and the history length remains 10 bits.

The pattern below has and extra "T" at the end. So the pattern history table entry for NNNTTNNNTT would be used by both branches, B1 would decrement the entry and B2 would increment it.

```
B2:  N  N  N  T  T  N  N  N  T  T  T   N  N  N  T  T  N  N  N  T  T  T
```

Problem 3: Answer the following load/store unit questions.

(*a*) Why don't store instructions write to the cache until they commit? (5 pts)

There's a chance they will be squashed, if they wrote to the cache there would be no way (ordinarily) to recover the data that was overwritten.

For the two problems below consider the four instructions (repeated) which are in the reorder buffer of a dynamically scheduled 1-way system. On a cache miss data will arrive in four cycles or more. The effective address for the second load is `0x1000`. (10 pts)

(*b*) Describe a scenario in which the data for address `0x1000` is not cached but the second load does not wait for its data more than a few cycles.

☑ Show a pipeline execution diagram.

☑ Explain the involvement of the load/store queue and why the second load does not wait more than a cycle or two.

The effective address of the second load (`0x1000`) is the same as that of the store and so the second load gets its data from the store value (there is no need to check the cache and it doesn't matter that the data is not there). For this to happen the store address must not be delayed (see the next part) which means the first load must hit the cache.

Involvement of the load/store queue: At cycle 7 the LSQ processes the second load. It scans "upward" (toward the head where the older instructions are) looking at each store instruction. The first entry it finds is the store instruction, since the addresses match and the store data is ready the load data will be taken from the store data.

```
# Cycle                     0  1  2 3  4  5  6  7  8
First:  lw $t1, 0($t2)      IF ID Q L1 L2 WB

        addi $t0, $0, 4720     IF ID Q EX WB

        sw 0($t1), $t0            IF ID Q L1 L2 WB

Second: lw $t3, 0($t4)               IF ID Q L1 L2 WB
```

(*c*) Describe a scenario in which the data for address `0x1000` is cached but the second load waits for its data at least four cycles.

☑ Show a pipeline execution diagram.

☑ Explain the involvement of the load/store queue and why the second load waits.

In this scenario the first load misses the cache and so the store cannot determine its effective address until cycle 10. Since the store address is unknown the second load cannot safely check the cache (because the store address might match the second load address). At cycle 11 the load can finally be processed. The timing below is correct for a store address of `0x1000` or some other address, either way the load waits until cycle 11 at which time it gets its data.

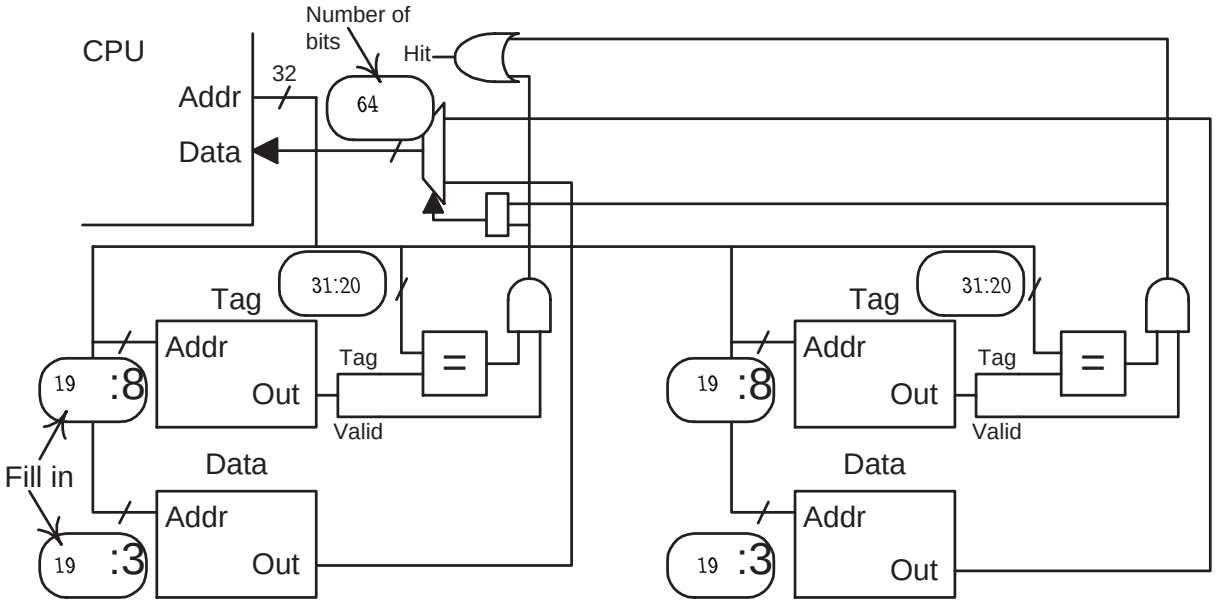
Involvement of load/store queue: The load store queue processes the second load at each cycle from 7 to 11. At cycles 7 to 10 it finds the store with an unresolved address and so the load cannot be completed. Finally at cycle 11 the store address is available. If it matches the second load address the store data is bypassed to the load, otherwise the cache is checked.

```
# Cycle                     0  1  2 3  4  5  6  7  8  9  10 11 12
First:  lw $t1, 0($t2)      IF ID Q L1 L2           L2 WB

        addi $t0, $0, 4720     IF ID Q EX WB

        sw 0($t1), $t0            IF ID Q              L1 L2 WB

Second: lw $t3, 0($t4)               IF ID Q L1           L2 WB
```

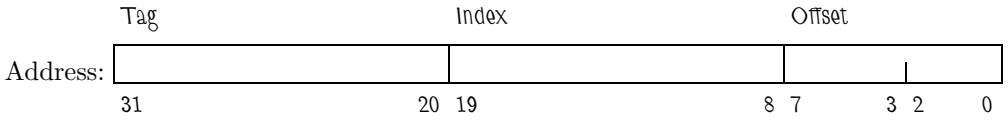
Problem 4: The diagram below is for a 2-MiB (2^{21} bytes) cache on a system with 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Associativity: 2

☑ Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 \times 2^{20-8}(32 - 20 + 1)$ bits.

☑ Show the bit categorization for a four-way set-associative cache with the same capacity and line size.

Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array. (7 pts)

```
char *a = 0x1000000;  // sizeof(char) = 1 character
int sum, i, j;
int ILIMIT = 1024;

for(j=0; j<2; j++)
  for(i=0; i<ILIMIT; i++)
    sum += a[ i * 2 ];
```

☑ What is the hit ratio for the program above?

The element size is one character but the code accesses every other element. The line size is $2^8 = 256$ characters and so $\frac{256}{2} = 128$ will access the same line. The first access will miss, the rest will hit. So for the first j iteration the hit ratio is $127/128$. The total amount of memory accessed is 1024 characters, but that covers 2048 characters of memory because of the skipping. That is much smaller than the 2 MiB capacity so on the second j iteration every access will hit.

The overall hit ratio is $\boxed{\frac{1}{2}\left(\frac{127}{128}+1\right) = \frac{255}{256}.}$

☑ What is the minimum value of ILIMIT needed to fill the cache?

The cache capacity is 2^{21} characters each iteration covers two characters (one read, one skipped) so to fill the cache $\boxed{\texttt{ILIMIT}=2^{20}}$.

Problem 4, continued:

(*c*) The code below runs on the same cache as parts above. Initially the cache is empty; consider only accesses to the arrays. (6 pts)

☑ Choose values for KLIMIT, MLIMIT, KSHIFT, and MSHIFT so that array `matrix` is completely removed from the cache with the **minimum** number of accesses (to `b`). That is, each `j` iteration must begin with `matrix` reloaded. Don't forget that the cache is set associative.

The first step is to determine how much space `matrix` takes. Since each element is a 4-character integer and 1024 of them are being accessed it covers 4096 characters or $4096/256 = 2^{12-8} = 16$ lines. The addresses range from `0x1000000` to `0x1000ffc`. Those addresses have indices from `0` to `0xf`. (The index bits start at the third hexadecimal digit.)

To remove `matrix` from the cache accesses to `b` must use each of those indices twice (because the cache is two-way set associative) and with a different tag each time. To generate each index with the minimum number of accesses set `MLIMIT = 0xf;` and `MSHIFT = 8;`, this shifts the `m` into the index part of the address. To generate each tag twice set `KLIMIT = 2;` and `KSHIFT = 20;`, this shifts `k` into the tag part of the address. The solution would also work with `k` and `m` swapped and with a larger `KSHIFT`.

Array `matrix` can be evicted with a small number of accesses because its index bits are the same as `b`, zeros. If index bits 12 or higher were different, say `b == 0x1001000`, it would take many more accesses (using the code below) to evict `matrix`.

Note: The exam was given on the opening day for *The Matrix Reloaded*.

```
int  *matrix = 0x1000000;    // sizeof(int) = 4 characters
char *b      = 0x2000000;
int sum, dummy, i, j, k, m;


// Solution
int KLIMIT =  2;                                    int MLIMIT = 16;

int KSHIFT =  20;                                   int MSHIFT = 8;


for(j=0; j<3; j++)
 {
    for(i=0; i<1024; i++)
      sum += matrix[ i ];

    for(k=0; k<KLIMIT; k++)
      for(m=0; m<MLIMIT; m++)
        dummy += b[ ( k << KSHIFT ) + ( m << MSHIFT ) ];
 }
```

Problem 5: Answer each question below.

(*a*) Suppose the instructions below are being considered for the latest extension of the MIPS ISA. For each new instruction explain why it should be added or why it should not be added. Consider a *variety* of factors related to the ISA and implementation. (10 pts)

☑ A mask instruction. Based on analysis of benchmarks.

```
New Instruction
 mask $t1, $t2, 5

Equivalent Code Using Existing Instructions
 srl $t1, $t2, 5
 sll $t1, $t1, 5
```

Yes, it saves an instruction and looks easy implement. Saving one instruction may not sound like much but if that one instruction is 5% of the dynamic instruction count the minor addition would be very cost effective.

Grading Note: Some students pointed out that an **andi** instruction can perform the same function and so the **mask** instruction is not needed. That's only partly correct since the immediate value is limited to 16 bits whereas **mask** could mask any number of bits.

☑ An integer negate instruction. Based on analysis of existing code. With this extension **sub** does not have to be used for negation!!!

```
New Instruction
 neg $t1, $t2

Equivalent Code Using Existing Instructions
 sub $t1, $0, $t2
```

Since **sub** does the exact same thing there is no reason to add **neg**. Adding **neg** would waste an opcode and complicate decoding. A better alternative is to have the assembler recognize **neg** as a synthetic instruction and have it substitute **sub**.

☑ An indirect load. Useful based on most existing benchmarks.

```
New Instruction
 lwi $t1, 0($t2)

Equivalent Code Using Existing Instructions
 lw $t1, 0($t2)
 lw $t1, 0($t1)
```

This would require using the MEM stage twice, something no other instruction does and which does not follow RISC principles (which are intended to simplify implementation). It should not be added.

Continued from previous page.

☑ Added functionality for the `sllv` instruction. The existing `sllv` looks at the low 5 bits of the `rt` register, ignoring the other bits. It only shifts left. The improved instruction also shifts right if the `rt` register holds a negative value and left if it's positive.

```
 Improved Instruction
   sllv $t1, $t2, $t3   # Shifts right if $t3 negative.

 Equivalent Code Using Existing Instructions
   bltz $t3, SHIFTRIGHT
   nop
   sllv $t1, $t2, $t3
   j DONE
SHIFTRIGHT:
   sub $at, $0, $t3
   srlv $t1, $t2, $at
DONE:
```

If bidirectional shifts were used a bidirectional shift instruction should be added, but **NOT** by changing the behavior of an existing instruction. There might be programs in which `sllv` instructions execute with a negative value in `rt`. The new behavior would break those programs and the programmers could rightly point out that "ignored" is not the same thing as "assumed to be zero." (And even if they didn't have a good argument it doesn't matter because the customer is always right!) The behavior of `sllv` should not be changed, instead a new instruction could be added.

Grading Note: No one got this one right. Maybe next semester I'll bring in an actual pair of golden handcuffs when talking about ISA compatibility and IA-32. Hmmm, would the student tech fee cover the cost?

(*b*) Answer the following questions about exceptions. (5 pts)

☑ Why are precise exceptions necessary for instructions like `lw` but optional for instructions like `div`?

If an instruction raises a precise exception then it can be re-executed after the handler does whatever it is it has to do. If an instruction raises an exception that is not precise then the best the handler can do is restart the program several instructions after the faulting instruction, there is no way to re-execute it.

In normal use `lw` (and other memory access instructions) will raise exceptions. The exception handler tends to the memory system (updating the TLB or swapping pages) and then restarts `lw` and the program proceeds as if nothing happened. Without precise exceptions there would be no easy way to implement modern memory systems.

A divide would raise an exception because of a division by zero or some other problem. In general, there is nothing the handler can do to fix the problem since the divide instruction gave the best answer it could. With no reason to restart there is no need for precise exceptions. There still may be situations where they are useful, say substituting a large value on a division by zero, but that's not as critical as handling TLB misses.

☑ What can handlers for instructions that raise precise exceptions do that handlers for other instructions cannot? Explain how that capability is used for `lw`.

They can see the state of the program just before the faulting instruction executed. That enables the handler to restart the instruction, if appropriate. It is used to restart `lw` after performing a routine memory system task.

(*c*) Delayed branches are common in RISC ISAs. (5 pts)

☑ Explain why delayed branches were useful in early RISC processors, such as the 1-way statically scheduled MIPS implementation covered in class.

Early RISC processors (and some modern ones too) used short pipelines, say five stages (as used in class). The delay slot instruction is fetched in IF while the branch computes its direction and target in ID; in the next cycle the target or fall-through is fetched. Since the delay slot instruction is executed either way (normally) there is no branch penalty on a taken branch (bubble).

☑ Why are delayed branches of less benefit with more recent implementations?

In a two-way or higher superscalar processor at least one instruction past the delay slot is fetched by the time the branch is resolved. If the branch is mispredicted those instructions will have to be squashed. The number of squashed instructions is one less than without delayed branches, in deeply pipelined systems that might mean squashing 15 instead of 16, the difference is greater with dynamic scheduling. This small benefit is only realized on a misprediction (say 5% of the time). It is not worth the complexity of the control logic.

Comment: Once an instruction is in an ISA it cannot be removed. If a new ISA were designed for compact implementations (say, embedded on a chip with other logic and memory and used to control something like a cell phone) it could use a delayed branch. If an ISA were designed for high-speed implementations (for general-purpose use) then a delayed branch would be omitted.

(*d*) Why might a gshare branch predictor with a longer global history register (GHR) have a significantly longer warm up time? (5 pts)

With a longer GHR there are more possible GHR values for a particular branch. (In an extreme case the branch is in a loop with another branch having random outcomes.) There is a pattern history table (PHT) entry for each GHR value, each of these entries must be warmed up for the branch. The more possible GHR values there are for the branch the longer it takes the GHR to warm up for the branch.

Grading Note: Many answered that it takes longer for the GHR to fill up with branch outcomes. Suppose the GHR were increased from 10 to 16 bits. That change would affect six branches, out of say, 100 million.

A correct answer could start with "It takes longer for the GHR to full up..." but it would have to explain that this was important when entering a loop and that the problem was the irrelevant outcomes from the part of the GHR written before the loop was entered.

(*e*) Why might a functional unit with an initiation interval of 4 and latency of 3 be less costly (as in dollars) than a functional unit with an initiation interval of 1 and a latency of 3?(5 pts)

☐ Include an illustration with your answer.

Sorry, no illustration. If someone asks I'll put one in.

If a functional unit has an initiation interval of 1 then new instructions can enter every cycle and so there must be enough hardware to simultaneously execute 4 instructions. If the initiation interval were 4 then the same small piece of hardware could be used repeatedly for the same instruction, as in a multiplier or divider. Multipliers usually are pipelined (initiation interval 1) but dividers are not.

71 Fall 2002 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 25 October 2002, 10:40–11:30 CDT

Problem 1		(30 pts)
Problem 2		(13 pts)
Problem 3		(13 pts)
Problem 4		(44 pts)
Exam Total		(100 pts)

Alias Who's in the Noose!

Good Luck!

Problem 1: A new MIPS branch instruction, `bieq rt,(rs) disp` (branch indirect equal) compares a register value to a memory location, if they are equal the branch is taken. The target is computed in the same way as other branches. In the code below, the contents of register `$s1` is compared to the contents of the memory location at address `$s2`. Like all MIPS control transfers, `bieq` has one delay slot. [30 pts]

(*a*) Modify the pipeline so that it can execute this new instruction. Show comparison units, multiplexors, and wires. **Do not** show control logic. For **partial credt** replace `bieq ...` with `bneq $s1,$s2, LOOP`.

See next page for solution discussion.

☑ Use as much existing hardware as possible. **Do not** add a new memory port.

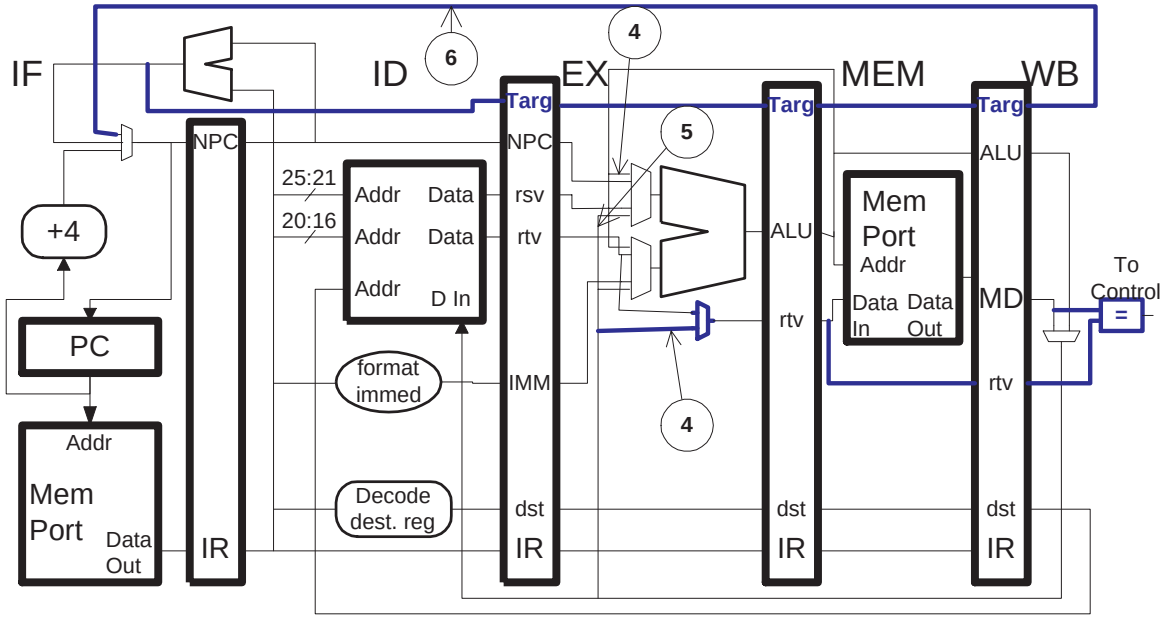
☑ The change should not reduce the clock frequency.

☑ Include the comparison unit for the branch condition.

☑ Add bypass paths so that the code below executes as shown.

☑ Label bypass paths with the cycle in which they are used. Include existing and any added bypass paths.

☑ Label the path carrying the branch target address with the cycle in which it is used.




```
LOOP:  # Assume bieq always taken.   # Solution
 # Cycle                0    1    2    3    4    5    6    7    8    9   10
 srl  $s1, $s1, 1       IF   ID   EX   ME   WB             IF
 addi $s2, $s2, 1            IF   ID   EX   ME   WB
 bieq $s1, ($s2), LOOP       IF   ID   EX   ME   WB
 sw   $s1, 100($s2)               IF   ID   EX   ME   WB
 add  $0, $0, $0                  IF   ID   EXx
 sub  $0, $0, $0                       IF   IDx
 or   $0, $0, $0                            IFx
 xor  $0, $0, $0
```

Problem 1, continued:

(*b*) The pipeline execution diagram on the previous page only shows the first iteration.

☑ Continue the pipeline execution diagram to the beginning of the second iteration (when `srl` is fetched), *consistent with your solution to the first part.*

See diagram on previous page.

☑ Show which instruction is in each stage of the pipeline a cycle eight. Include squashed instructions, if any.

```
Solution:
 IF: addi    ID: srl   EX: or(sq)   MEM: sub(sq)   WB: add (sq)
 sq indicates the squashed remains of an instruction.
```

☑ Determine the CPI for a large number of iterations.

Iterations start at cycles 0 and 7. There are no stalls or other reasons why later iterations would be any different, therefore the number of cycles per iteration is 7. The loop has 4 instructions, for a CPI of $\frac{7}{4}$.

Part (a) discussion.

Changes shown in **blue bold**. The branch condition is resolved in WB when one of the data items is retrieved from memory. (A MEM-stage comparison would stretch the critical path, lowering clock frequency.) In addition to the memory value, the value of the `rt` register is needed, a new path from MEM to WB is added for that, as well as an EX-stage bypass path. The existing ID-stage adder is used to compute the target address, which is sent through the pipeline and used in the WB stage. (An alternative design would use the ALU to compute the branch target, but that would require a mux at the memory address input to select the address, which might be held in a new `rsv` pipeline latch.)

The circles show the cycle numbers. Note that the arrows point unambiguously to the path that's being used. (In some solutions the arrows would point to a bypass path, say, coming from memory before it reached the upper ALU mux, so one could not tell if it was a bypass into the upper or lower ALU mux.

Note: the problem as originally given did not explicitly mention clock frequency and the diagram had a path from MEM to the IF-stage mux.

Problem 2: Convert the MIPS program below to SPARC, making use of condition codes. [13 pts]

☑ Make reasonable guesses about instruction names. Reasonable guesses will receive full credit.

☑ For the branch instructions, explain what the name stands for.

☑ Explain how each line of code uses the condition codes.

☑ Use the minimum number of registers.

☑ Do not rearrange instructions.

```
sub $s6, $s1, $s2
blt $s6, TARGET1
and $s3, $s6, $s4
beq $s3, $0, TARGET2
nop
beq $s6, $0 TARGET3
add $s5, $s6, $s3
```

Discussion: In SPARC, condition-code versions of arithmetic and logical instructions (such as `addcc`, `orcc`) set the condition codes based on the operation result, this is in addition to writing the result in the specified register. So the first instruction in the solution writes the difference in register `l6` and also writes the condition code register. For this first instruction the result of the subtraction is needed, by the `add` instruction.

The condition code register consists of four bits, Zero, Negative, Carry, and Overflow. Each is set based on the result of the operation.

```
subcc %l1, %l2, %l6  ! l6 = l1 - l2
bl TARGET1           ! Branch if < 0. (Negative, mnemonic assumes subtract.)
andcc %l6, %l4, %l3  ! l3 = l6 & l4
be TARGET2           ! Branch if equal to zero.
nop
subcc %l1, %l2, %g0  ! Set condition codes again. MIPS doesn't have to.
be TARGET3           ! Branch if equal to zero.
add %l6, %l3, %l5    ! Ordinary add.
```

Problem 3: Sometimes it's necessary to jump to a location specified in the program. [13 pts]

(*a*) SPARC V8 includes an instruction that can jump anywhere in the address space using an address specified in the instruction, for example, `call 0xabcd1234`.

☑ How does the `call` instruction, which is 32 bits, manage to specify a 32-bit jump target?

Because instruction addresses are aligned the least significant two bits of a jump target are always zero, so the instruction does not need to store them. The opcode in a SPARC instruction is just two bits, so the instruction can store a 30-bit immediate. (That immediate is a displacement, not the high 30-bits of the target address, but it can still jump anywhere in a 32-bit address space.)

☑ Switching now to MIPS, show the minimum number of instructions needed to jump to address `0xabcd1234`. (If `jr` is used the instructions must put the address in a register. The address cannot be loaded from memory.)

The solution appears below, followed by common mistakes.

```
# Correct
lui $10, 0xabcd
ori $10, $10, 0x1234
jr $10

## Common Mistakes Below

# Partial credit, could have used lui to reduce number of instructions.
addi $10, $0, 0xabcd
sll  $10, $10, 16
ori $10, $10, 0x1234
jr $10

# Partial credit, could have used lui to reduce number of instructions.
# Wrong shift amount
addi $10, $0, 0xabcd
sll  $10, $10, 4      # WRONG, a hexadecimal digit is four bits.
ori $10, $10, 0x1234
jr $10

## Common Mistakes Above
```

(*b*) Without introducing a new format, add a new instruction to MIPS that would allow a jump to an arbitrary address using two instructions. The address must be specified in the instructions themselves.

The new instruction is `jri rs, IMMED16` (jump register immediate). It jumps to an address constructed by oring the contents of register `rs` with `IMMED16`. It is a Type I instruction.

☑ Use the instruction in an example.

```
lui $s0, 0x1234
jri $s0, 0x5678  # Jumps to address 0x12345678.
```

☑ Show the coding for the new instruction.

Type I with low 16 bits of target address in the immediate field and the register number in the `rs` field.

Problem 4: Answer each question below.

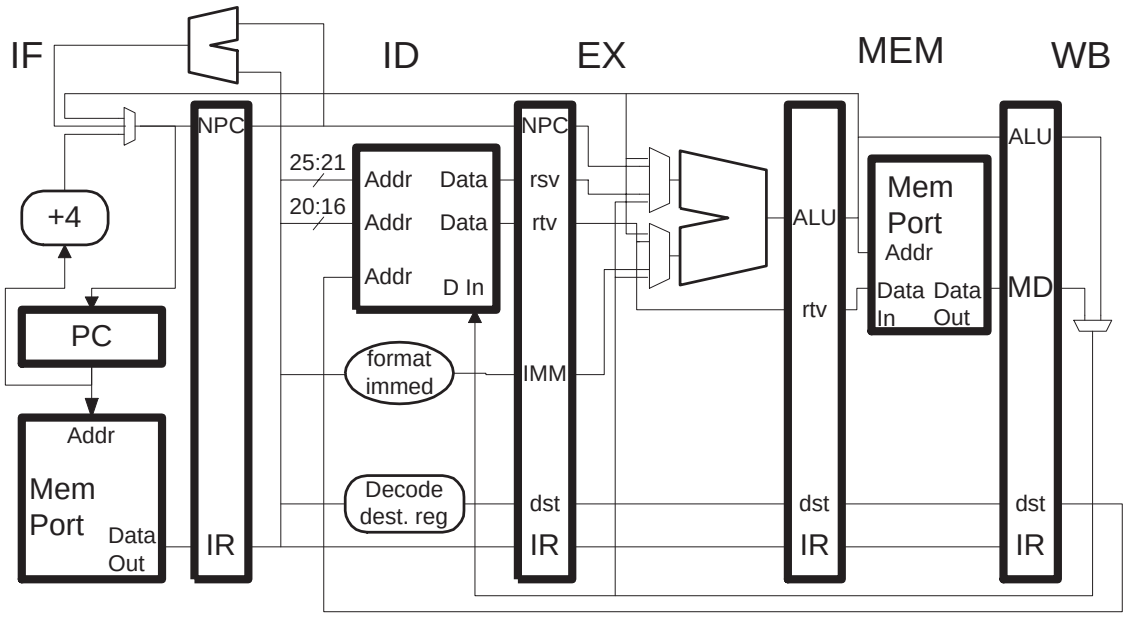
(*a*) Certain RISC ISA features are intended to facilitate pipelined implementations. [12 pts]

☑ Using the diagram below, briefly explain how RISC implementations benefit from fixed-length instructions and how things would be different with variable-length instructions.

Fixed length instructions allow implementations to easily compute the fetch address and to easily decode the instructions once fetched. With variable-length instructions the decode hardware might have to consider the instruction type before fetching registers. For example, different size formats might have the register numbers in different places, and so multiplexors would be needed at the input to the register file address ports. The IF stage could not just add 4 to get the next fetch address, the amount to add would have to be supplied by ID (if not slowing the fetch rate, requiring a shift-and-mask circuit to properly position the instruction).

☑ Using the diagram below, briefly explain how RISC implementations benefit from load/store instructions (restricting memory access to load instructions) and how relaxing the restriction would complicate things.

If source operands for ALU instructions could come from memory then either another memory stage would have to be added (adding to cost) or instructions would have to use the MEM stage before the EX stage, adding expensive new paths and greatly complicating the control logic (think about the logic for setting bypass paths).



(*b*) Before there was MMX there was BCD. [12 pts]

☑ Why are binary coded decimal (BCD) and packed-integer data types superficially similar? What is an important difference between them?

Discussion: A misconception (at least on the exam solutions) is that there are four integers packed into a register. The number of integers packed varies with the size of the integer (and the register). For example, a 64-bit register can hold 8 8-bit integers, 16 4-bit integers, etc. BCD digits are always four bits.

Answer: They are superficially similar because a decimal digit in a BCD representation might resemble a 4-bit number in a packed integer representation. An important difference is that the digits in a BCD representation are for one number whereas a packed integer holds a collection of numbers.

☑ Describe a computation in which packed integer data types yield a performance advantage over ordinary integers. Show a brief advantage of the computation.

```
// Part of solution.
int *a, *b, *c;  // Elements range from 0 to 15.
for(i=0; i<1024; i++) c[i] = a[i] + b[i];
```

The loop above compiled for an ordinary ISA would perform one addition per `add` instruction. If the compiler targeted a packed-operand ISA it might use an add instruction that adds two packed operands, each, say, holding sixteen numbers. (The compiler would have to know that numbers stored in the array are limited to 0-15.) The resulting code would require one sixteenth iterations. (Other code which accessed the arrays would also have to be written to handle the packed format.)

☑ Why were BCD data types added to some ISAs?

To avoid certain rounding errors.

(*c*) Describe two compiler optimizations that reduce instruction count. [8 pts]

Dead code elimination. If, say, the value of a variable is never used no instructions will be emitted for the code that computes a value for the variable.

Common subexpression elimination. If an expression appears multiple times then instructions for it might be emitted in one place, the instructions would store a value that would be used for each subsequent appearance of the expression.

(*d*) The BAPCO benchmarks used by PC magazine and others evaluates a computer by timing the execution of a suite of benchmarks. The suite of benchmarks is drawn from common applications, such as Adobe Photoshop. The applications are in executable form, source code is not used.

Like BAPCO, SPEC CPU2000 consists of a suite of common applications, unlike BAPCO source code is used. Assume that the BAPCO and SPEC benchmarks are both well chosen and aimed at roughly the same user community. [12 pts]

☑ How is source code used in preparing the SPEC CPU 2000 benchmark results?

The source code is carefully compiled for the system it will run on. Since manufacturers prepare the benchmarks for their own systems and knowledgeable customers respect the SPEC benchmarks every effort is made to select compiler options for the best possible compilation.

☑ Consider two processors, A and B. Suppose BAPCO ranks A faster but SPEC ranks B faster. How might the use of source code account for this difference?

Processor B had special instructions or scheduling characteristics that a compiler could take advantage of. Since SPEC benchmarks are compiled for the target system the results reflect their benefit. The BAPCo benchmarks are compiled by the software vendors and they target an average system, probably not taking advantage of B's special features. Without the special features A is faster.

☑ Taking in to account the differences due to the use of source code, who should make buying decisions based on BAPCO results? Who should make buying decisions on SPEC results?

Users buying "shrink wrapped" (or at least already compiled) software should use BAPCO results, especially if their applications are similar to BAPCO's. Those compiling their own applications or those buying code compiled for their particular system might use SPEC results.

Name Solution

Computer Architecture
EE 4720
Final Examination

12 December 2002, 10:00–12:00 CST

Problem 1		(10 pts)
Problem 2		(17 pts)
Problem 3		(11 pts)
Problem 4		(17 pts)
Problem 5		(45 pts)
Exam Total		(100 pts)

Alias Currently Accurate, Full, and Complete

Good Luck!

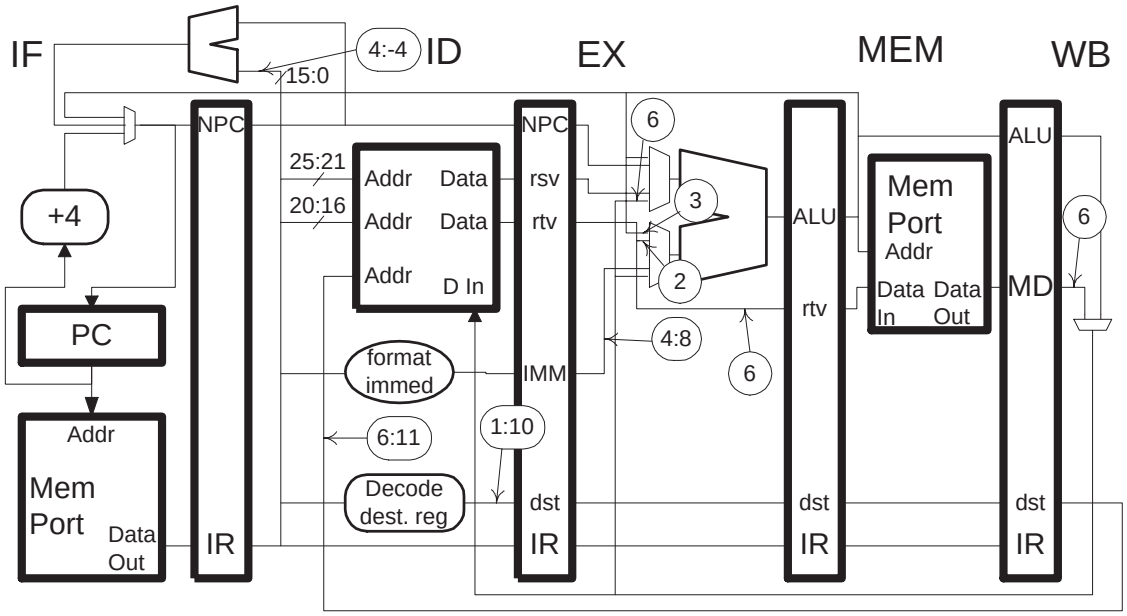
Problem 1: In the diagram below certain wires are labeled with cycle numbers and, in some cases, values that will then be present, for example, 2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions, use the PED to help solve the problem. (10 pts)

☑ Write a program consistent with these labels.

☑ Use labels for branch targets (if any) and label the target line.

Solution shown below.

Grading Note: Many people got this 100% correct, but I was hoping everyone would. For those currently taking EE 4720, study this well, it's not that difficult to get full credit.




```
 Cycle                  0  1  2  3  4  5  6  7  8
LOOP:
 add  $10, $2, $1       IF ID EX ME WB
 add  $9, $4, $10          IF ID EX ME WB
 lw   $11,  8($6)             IF ID EX ME WB
 bneq $1, LOOP                   IF ID EX ME WB
 sw   0($11), $7                    IF ID EX ME WB
```

Problem 2: The PED below shows execution on a defective 1-way dynamically scheduled machine using Method 3. A diagram appears on the next page. The circuitry that's supposed to restore the ID map after a branch misprediction is not working, the ID map is left unchanged. Everything else works correctly, in particular the free list is correctly restored to the exact state it was right after the `lw`. (17 pts)

(*a*) For this incorrect execution:

☑ Show where each instruction commits.

☑ Complete the ID map. (See the phys. reg. file for the free list.)

☑ Complete the commit map, include the initial state.

☑ Complete the physical register file.

☑ **In addition to other information** the physical register file should include a `[` in the cycle a register is removed from the free list and a `]` in the cycle it is returned to the free list.

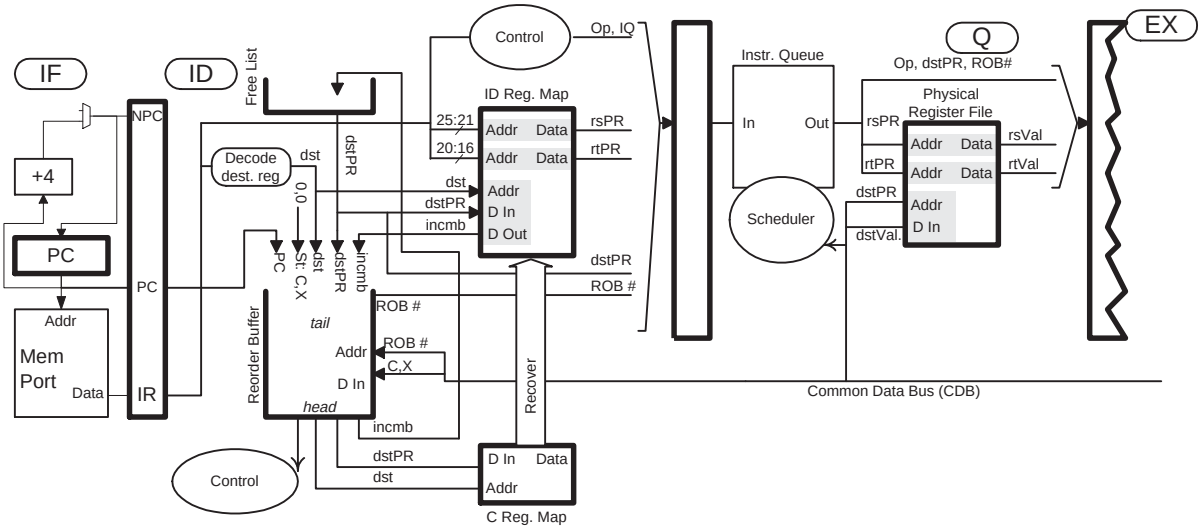
☑ Circle incorrect entries (due to the defect) in the tables below.

```
 # SOLUTION.  (See next page for notation and discussion.)
 # lw loads a 0x200, lb loads a 0x202
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
 bne $s1, $s2  SKIP  IF ID             Q  B  WC
 lw  $t1, 16($t1)       IF ID       Q  L1       L2 WC (18)
 addi $t4, $t4, 3          IF ID Q  EX WB    x (squash) (20)
 ori $t5, $t4, 0x30           IF ID Q  EX WB x (squash) (23)
 nop ... (lots of nops)
SKIP:
 lb   $t1, 16($t1)                                IF ID Q  L1       L2 WC   (20)
 addi $t4, $t4, 3                                    IF ID Q  EX WB (23)   C
 ori  $t5, $t4, 0xc0                                    IF ID Q  EX WB (30)   C
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
ID Map
 t1   3                   18                         20
 t4   7                      20                         23
 t5  10                         23                         30
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Commit Map
 t1   3                                           18                   20
 t4   7                                                                   23
 t5  10                                                                      30
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
Physical Reg File.  All values in hex.  Free List: 18, 20, 23, 30, 37
 3  100                                           ]
 7  101
 10 102
 18 103                   [                         200                  ]
 20 104                      [        104   ]         [                202{]}
 23 105                         [        134]           [       {107}       {]}
 30 106                                                    [       {1c7}
 37 107
 # Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

Problem 2, continued: The illustration below is similar to one used in class; it is provided for reference.

In the solution on the previous page incorrect entries are surrounded by braces, for example `107`. The numbers in parenthesis on the PED lines are the physical register numbers assigned to the instructions. They are there as an aid in solving the problem.

Because of the defect the ID map is not recovered after the branch misprediction. This causes two execution problems. First (most people got this) the `addi` on the correct path reads the wrong `t4`: it should read the one in physical register 7 but it reads the one in 20. As a result the values written in the physical register file are wrong. The second problem (few got this) is that when instructions pass through ID they read the wrong incumbent. For example, the `addi` on the correct path should read 7 as the incumbent, but instead it reads 20. As a result the wrong register will be placed in the free list when `addi` commits, which is what part (b) is about.



(*b*) Suppose 1000 instructions later an instruction finds `0x4720` in register `t4` despite the fact that the code above wrote something else and no instruction wrote `t4` after that. (If the free list is used improperly the question might apply to `t5`, not `t4`. See the diagram for hints on proper use of the free list.)

Note: On the original exam register `t5` *was used instead of* `t4`. *Register* `t5` *can not change unexpectedly, though it still has an incorrect value.*

☑ How could that have happened? Be specific in how it's due to the defect.

As explained above, the correct path `addi` reads the wrong incumbent and so frees the wrong physical register, in particular the physical register holding `t4`. At some point that physical register will be assigned to a new instruction which will write a value in it, perhaps `0x4720`.

(*c*) How could the defect have gone undetected?!? (That's not the question.)

☑ Suppose the `lw` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Remember, the only defect is with recovering the ID map on a branch misprediction. *Note: Original exam omitted "on a branch misprediction."*

The mechanism used to recover the ID map on a branch misprediction is different than the one used for exceptions. The problem stated that other than ID recovery on a branch misprediction, everything was working correctly. The `lw` and `lb` instructions execute before the `addi` and `ori` instructions. When the load reaches the ROB head exception recovery will start, part of that is copying the Commit Map to the ID Map, replacing the incorrect ID Map with a correct one. For that reason the code executes correctly.

☑ Now suppose the `lb` raised an exception in `L2` on this defective hardware. Would the code above execute incorrectly? Explain. Once again, the only defect is with recovering the ID map on a branch misprediction.

See the solution to the part above.

Problem 3: The code below runs on three systems: one using a bimodal (one-level) predictor, I; one using a two-level local history predictor, L; and one using a two-level gshare predictor, G. The global history register is 16 bits and the local history is 16 bits. The branch history tables have 256 entries. Assume that each branch below has its own BHT entry. The branch outcomes for B2 and B3 are provided for your solving convenience. Note that B3 has the same pattern as B2 but at a different phase, the phase difference is important. (11 pts)

```
BIGLOOP: LOOP:
 B1: bne $t1, $0, SKIP  # Random, independent, taken 25% of time.
 ...
 SKIP:
 B2: bne $t2, $t5, SKIP2 # Pattern: N N N N N N T T N N N N N N T T N N N N N N T T ....
 ...
 SKIP2:
 B3: bne $t7, $t8, SKIP3 # Pattern:  N N N N N T T N N N N N N T T N N N N N N T T N ....
 ...
 SKIP3:
 B4: bne $t9, $t10, LOOP # Iterates 50 times.
...
 j BIGLOOP
 nop
```

(*a*) Determine the prediction accuracy for each branch and each predictor. The accuracies should be after warmup. **Do not** compute the number of entries. Approximate the accuracy of B1 predictions.

☑ B1 on I: Accuracy: 70%

☑ B1 on L: Accuracy: 70%

☑ B1 on G: Accuracy: 70%

☑ B2 on I: Accuracy: 62.5%

☑ B2 on L: Accuracy: 100%

☑ B2 on G: Accuracy: 100%

☑ B3 on I: Accuracy: 62.5%

☑ B3 on L: Accuracy: 100%

☑ B3 on G: Accuracy: 85%

☑ B4 on I: Accuracy: 98%

☑ B4 on L: Accuracy: 98%

☑ B4 on G: Accuracy: 98%

Since branch B1 is taken 25% of the time the two-bit counter used to predict it will likely be 0 or 1 (with probability $\frac{36}{40}$), therefore the prediction accuracy of B1 is about 75% (it is exactly 70%, explained below). The accuracy is the same on all predictors because correlating the prediction with past occurrences of the same branch (local) or other branches (gshare) does not help predict the random branch.

The bimodal prediction accuracy for B2, B3, and B4 is straightforward.

Branch B2 and B3 follow a regular pattern that repeats every eight occurrences. This is small enough for the local predictor to achieve a 100% prediction accuracy. Each iteration of the loop has four branches, so the 16-bit GHR can "remember" four iterations. It so happens that B2 does whatever B3 did on the previous iteration. Gshare easily remembers one iteration back, so the prediction accuracy of gshare for B2 is 100%.

Determining the gshare prediction accuracy for B3 is more involved. The prediction accuracy depends upon how much of the repeating pattern the GHR holds. The pattern consists of four branches (B1-B4):

```
 B1: x   x   x   x   x   x   x   x    ...
 B2:  n   n   n   n   n   n   t   t   ...
 B3:   n   n   n   n   n   t   t   n  ...
 B4:    T   T   T   T   T   T   T   T ...
GHR: xnnTxnnTxnnTxnnTxnnTxntTxttTxtnT (repeats)
Iter: 0   1   2   3   4   5   6   7   0   1 ...
```

The **x** outcome for B1 can be either N or T, the outcomes for B2 and B3 are shown in lower case so they can be distinguished from B4 when looking at the GHR. The predictability of B3 depends on the iteration (see the diagram above). First consider iteration 0. The outcome is N and this outcome only occurs when the outcomes in the previous iteration (7) are **xtnT**. For iteration 0 the GHR needs to be five bits to see enough: **tnTxn** (the last **x** isn't needed). Similarly, iterations 6 and 7 can be predicted using only the previous iterations. Since the GHR is sixteen bits it is large enough to predict these iterations with 100% accuracy. Iterations 1 through 5 require more than five bits. Iteration 1 needs to see iteration 7, requiring 9 bits (which we have). For similar reasons iteration 2 needs 13 bits, iteration 3 needs 17 bits (bzzzt! [that's the GHR-is-too-small buzzer]), and iteration 4 needs 21 bits (bzzzt!). Iteration 5 also needs 21 bits (bzzzt!) since the GHR value at B3 for iterations 4 and 5 would be different. Since the GHR is only 16 bits the GHR contents will be identical when predicting B3 in iterations 3, 4, and 5. This would not be a problem if the outcome of B3 was the same in all these iterations (all taken or all not taken), but it's not taken in iterations 3 and 4 but taken in iteration 5. As a result the two-bit counter will mostly predict not taken but the branch will sometimes be taken. Because of the random branch there are really sixteen (four x's) two-bit counters for iterations 3-5 and they are accessed in random order. They will mostly be at zero or one. As a rough estimate, (which is enough for full credit) assume the counter is always zero or one, then the prediction accuracy of iterations 3 and 4 will be 100% and iteration 5 will be 0%. Taking into account that the counter is sometimes 2 and 3, (see below) we get a prediction accuracy of $\frac{12}{15} = 80\%$ for iterations 3 and 4 and a prediction accuracy of 20% for iteration 5.

The overall prediction accuracy for branch B3 is the average of each iteration: $\frac{1}{8}(5 * 100\% + 2 * 80\% + 20\%) = 85\%$. So gshare's accuracy on B2 is 85%.

To precisely determine the prediction accuracy on B1 (and also B2, and to a very minor extent, B4) one needs to compute the probability that the counter value is less than 2. In the other cases the counter value is not effected by the random branch so their exact value can be computed, usually 0 or 3. Consider B1 using the bimodal predictor. Because it's taken only 25% of the time the counter will mostly be less than two, but because outcomes are independent there is a chance the branch will be taken twice in a row, leaving the counter at 2 (or even 3). Computing these probabilities is straightforward, and uses a standard technique in queuing theory: Markov chains. A Markov chain is a mathematical model that can be used to represent the counter and other systems which can be represented by state transition diagrams with probabilities associated with arcs. There are four states, denote them s_0 (count is 0), s_1 (count is 1), etc. There is an arc from s_0 to s_1 and it is associated with the transition probability (branch taken probability), 0.25, which we will denote p. There are similar arcs from s_1 to s_2 and s_2 to s_3. There are also arcs from s_3 to s_2, etc. representing the not-taken cases, they are associated with the not taken probability, 0.75 or $1 - p$.

It turns out it is surprisingly easy to solve for the state probabilities. The key observation is that the number of times the counter is incremented from 0 to 1 (or 1 to 2, etc) must be equal to the number of times it is decremented from 1 to 0 (or 2 to 1, etc) (plus or minus 1, which is insignificant). Overloading the notation, let s_0 denote the probability of the counter being 0. Then $s_0 p = s_1(1-p)$ (what goes up must come down, or more formally, a balanced flow equation). Generalizing, $s_i p = s_{i+1}(1-p)$ for $i \in \{0, 1, 2\}$. With a few substitutions and some manipulation we find $s_i = s_0 \left(\frac{p}{1-p}\right)^i$. Obviously $\sum_{i=0}^{3} s_i = 1$ and so $\sum_{i=0}^{3} s_0 \left(\frac{p}{1-p}\right)^i = 1$. Solving for s_0 we get $s_0 = 1 \Big/ \left[\sum_{i=0}^{3} \left(\frac{p}{1-p}\right)^i\right]$. Since the summation is only over four terms one could compute s_0 at this point, but we don't have to since there is a closed-form expression for that summation: $\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$

and so

$$s_0 = \left[\frac{p}{1-p} - 1\right] \Big/ \left[\left(\frac{p}{1-p}\right)^4 - 1\right].$$

For branch B1 $p \to 0.25$ and so $s_0 = \frac{27}{40}$, $s_1 = \frac{9}{40}$, $s_2 = \frac{3}{40}$, and $s_3 = \frac{1}{40}$. The branch prediction accuracy for all predictors on B1 is exactly $\frac{3}{4}\frac{27+9}{40} + \frac{1}{4}\frac{3+1}{40} = 70\%$, reasonably close to the 75% estimate based on the assumption that the counter never reaches 2.

Grading Notes: Way too many people got B1 wrong, the most common wrong answer was 25%. That would only be correct if a taken prediction were made every time. (Yes, I understand that with more time ...) No one realized that 75% would be an upper bound on B1's prediction accuracy. Many had trouble with gshare, or decided not to spend time on it.

(*b*) A gshare predictor using a 15-bit GHR (instead of the 16-bit GHR used above) should give the same prediction accuracy for branch B2. How small can the GHR be made without changing the prediction accuracy of B2? Assume there are no collisions. *Note: The original question asked about B3.*

Branch B2 does whatever B3 did in the previous iteration, so the GHR can predict it at 100% accuracy with as few as three bits.

The original question asked about B3. Prediction accuracy is determined by how many previous branches the predictor can see. At four branches per iteration a 16 bit GHR holds four previous B3 outcomes. Reducing to 15 bits looses the oldest B3 outcome, however that information is in B2 because its outcome is the same as B3 in the previous iteration. Therefore the GHR can be reduced to 13 bits without changing predictor accuracy. (At 13 bits B2 is the oldest outcome.)

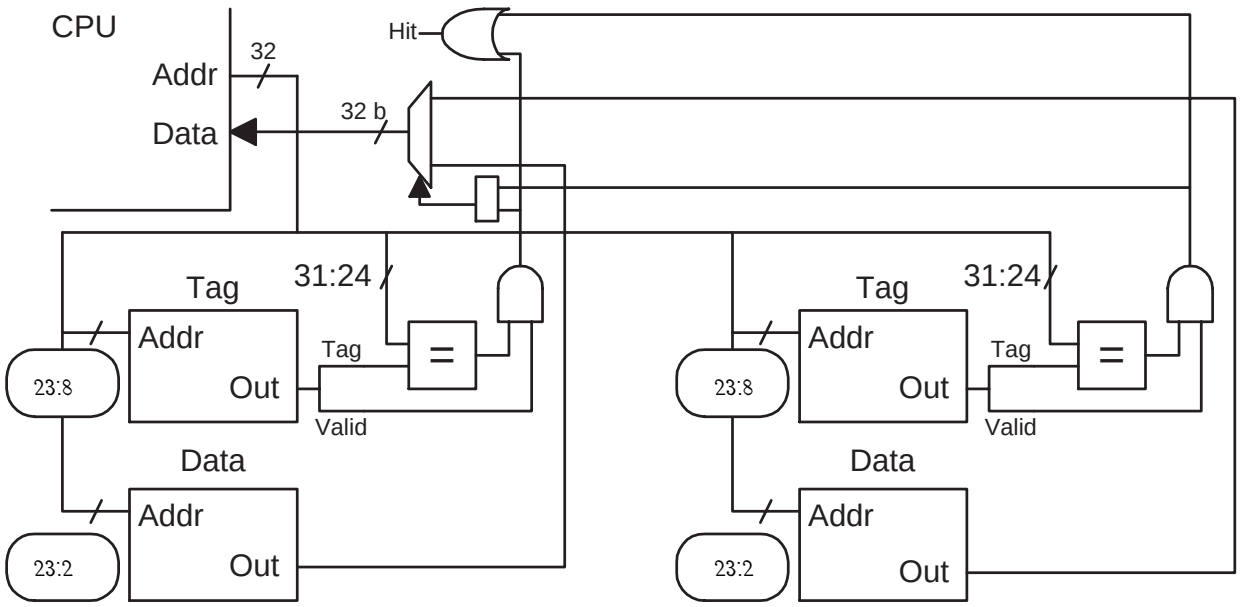
(*c*) For the local predictor, how small can the local history be made without affecting the prediction accuracy of B2 and B3?

To distinguish the first of the two taken branches from the last not-taken branch the local history must be at least six bits.

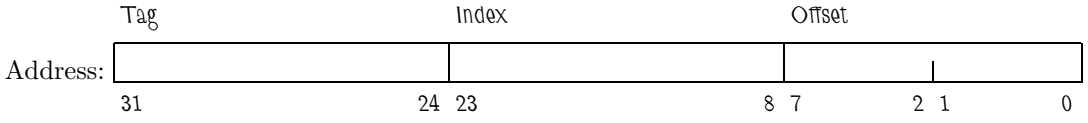
Problem 4: The diagram below is for a cache with 256-character lines on a system with 8-bit characters. (17 pts)

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Associativity: 2

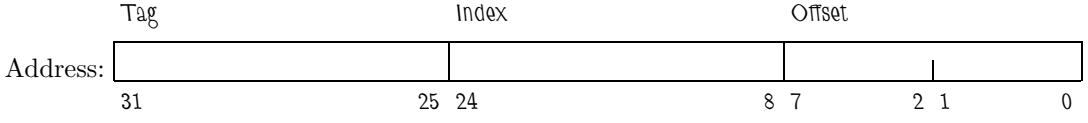
☑ Cache Capacity (Indicate Unit!):

It's $2 * 2^{24}$ characters (bytes in this case) or 32 MiB.

☑ Memory Needed to Implement (Indicate Unit!):

It's the cache capacity plus $2 * 2^{16}(32 - 24 + 1)$ bits.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 4, continued:

(*b*) The code below runs on the cache from the previous part. When the code below starts running the cache is empty. Consider only accesses to the array.

```
int *a = 0x100000;  // sizeof(int) = 4 characters
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ];
```

☑ What is the hit ratio for the program above?

There are $\frac{256}{4} = 64$ elements per line. During the first iteration the hit ratio is $\frac{63}{64}$, during the second it is 1 since the entire array can be cached. The total hit ratio is $\boxed{\frac{127}{128}}$.

(*c*) In the problem below, only consider accesses to the arrays.

```
int *a = 0x10000;    // sizeof(int) = 4 characters
int *b = 0x20000;
int sum, i, j;

for(j=0; j<2; j++)
  for(i=0; i<1024; i++)
    sum += a[ i ] + b[ i ];
```

☑ What is the minimum size of a direct mapped cache for which the program above will have a 100% hit ratio on the second j iteration? Explain.

The total size of the two arrays is 8192 characters but if the cache were 8192 characters the index bits for the two arrays would be the same and they could not be simultaneously cached. The only way for the two arrays to have different index bits is to make sure the index part of the address includes bit 16 (bit 16 of `0x10000` is 1, bit 16 of `0x20000` is zero). Such a cache would be $\boxed{2^{17} = 131,072 \text{ characters.}}$ What a difference!

☑ What is the minimum size of a two-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

In this case the cache can be made the combined size of the arrays, $\boxed{\text{8192 characters}}$. Corresponding elements (say, when `i=3`) have the same index but the cache can handle the two different tags.

☑ What is the minimum size of a three-way, set-associative cache for which the program above will have a 100% hit ratio on the second j iteration?

Again, waste. Why couldn't it be a four-way? One third of the cache goes to waste, the size is $\boxed{3 * 4096}$. Note that if it were a four way the size could be 8192 characters.

Problem 5: Answer each question below.

(*a*) In the PED below for a statically scheduled 2-way superscalar machine the `xor` instruction stalls in IF even though there is an empty space in ID. (5 pts)

```
add s1, s2, s3    IF ID EX ME WB
or  s4, s1, s5    IF ID -> EX ME WB
xor s6, s7, s8       IF -> ID EX ME WB
and s9, s10, s11     IF -> ID EX ME WB
```

☑ Why?

If `xor` did move in to ID then the instructions in ID would be out of order and ordinarily the decode logic is designed for in-order instructions, so it couldn't handle them.

☑ Would it be difficult or would it be impossible to modify the implementation (but keeping it statically scheduled) so that such an instruction could move to ID? Explain.

Tedious but not impossible. Logic would have to consider case where instruction in slot zero follows instruction in slot one.

(*b*) In a dynamically scheduled system why should store instructions wait until they are ready to commit before storing the data? (6 pts)

Because it might be squashed, and if it wrote before being squashed there would be no way easy to recovery the previous memory contents.

(*c*) The same code fragment executes on 1-way, statically scheduled systems with the specified FP add functional unit(s). For each show a pipeline execution diagram. Don't forget to consider *all* structural hazards and check carefully for dependencies. All adders take a total of four cycles to compute a result (latency is 3). (5 pts)

☑ One adder, `A`, initiation interval: 1.

```
# Solution
add.d f0, f2, f4    IF ID A1 A2 A3 A4 WB
add.d f6, f0, f8       IF ID -------> A1 A2 A3 A4 WB
add.d f10, f0, f12        IF -------> ID A1 A2 A3 A4 WB
```

☑ One adder, `A`, initiation interval: 2.

```
# Solution
add.d f0, f2, f4    IF ID A1 A1 A2 A2 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A2 A2 WB
add.d f10, f0, f12        IF -------> ID -> A1 A1 A2 A2 WB
```

☑ One adder, `A`, initiation interval: 4.

```
add.d f0, f2, f4    IF ID A1 A1 A1 A1 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A1 A1 WB
add.d f10, f0, f12        IF -------> ID -------> A1 A1 A1 A1 WB
```

☑ Two adders, `A` and `B`, each has initiation interval: 4.

```
add.d f0, f2, f4    IF ID A1 A1 A1 A1 WB
add.d f6, f0, f8       IF ID -------> A1 A1 A1 A1 WB
add.d f10, f0, f12        IF -------> ID B1 B1 B1 B1 WB
```

(*d*) One problem in Homework 5 was to analyze the execution of an unoptimized program to compute π. Many people got the question below wrong, here's your chance to get it right! (5 pts)

☑ Should computer engineers analyze the performance of processors running unoptimized code? Explain.

They should analyze the kinds of programs that people run, which is optimized. Software is compiled without optimization only for special circumstances such as debugging.

(*e*) Consider a new data type, binary coded trianary. This 32-bit data type consists of 16 radix-3 digits, each coded with two bits, in the same way a BCD data type would consist of 8 radix-10 digits. (8 pts)

☑ Why might 3-fingered, three-armed aliens (one hand per arm) find this data type useful?

For the same reason we've adopted a radix-10 number system they might adopt a radix-9 or radix-3 number system. Binary-coded trianary would have the same advantage for them.

☑ Explain how the data type would be useful in accessing arrays in which the element size was a power of three.

Suppose the size were $3^3 = 81$ and we wanted to access element 21. Instead of multiplying by $21 * 81$ we would shift the trianary 21 by six bits (three trianary digits). Either we would use a new machine instruction to convert it to binary or the array would be 25% padding.

Grading Note: A common incorrect answer was that it would be easy to increment an index. That's easy anyway since it would take one cycle to add 81 (incrementing the address by one element).

In a meeting next week you plan to argue that this data type should be included in the next revision of the ISA, one of several proposed new data types. Only one will be chosen. *Note: The following sentence was not in the original exam.* Three-fingered aliens are **not** expected to make up a large portion of your customer base.

☑ How will you argue that the data type should be included?

First, show that multiplying by a power of three and computing modulo 3 (or a power) would be only a single cycle with the new data type, versus many cycles (especially for modulo now). Then argue that existing software performs enough modulo-3 arithmetic to benefit from the extension (if recompiled).

Grading Node: Many answers involved aliens. They were fun to read.

☑ What will you do in the next week to prepare your argument?

Analyze existing code to find how often they perform power-of-three multiplication and modulo operations.

The following incorrect answer was common: write example programs to demonstrate the data types usefulness. It's wrong because the example programs don't tell us how often the data type would be used in real programs, which is very important. A sample program should not be necessary to demonstrate that a modulo 3 operation can be done in 1 cycle rather than 15 (it would be argued by the design of the arithmetic unit, in this case comparing an existing division/modulo unit to a shifter or logic unit [for masking to do the modulo]).

(*f*) In Homework 6 the performance of a linear search of an array and linked list were compared. On the dynamically scheduled systems the search was much faster on the array. Now consider the array program on a statically scheduled system. (8 pts)

```
\scoreable Would the array program perform as well compared
it the linked list program?  Consider
the effect of memory latency and line size.\par

\solution{The array program would still perform better than the linked
list program, but with longer memory latency or shorter line size the
array program on the dynamically scheduled system would run
substantially faster.  }
```

☑ Explain. *Hint: the solution above was intentionally included.*

The array program does better because a single miss brings in multiple lines. Both statically and dynamically scheduled systems enjoy this benefit. If the line size is short or memory latency is large the dynamically scheduled system could have misses to more than one line at once, overlapping two time consuming memory accesses. This is something the statically scheduled processor could not do.

(*g*) Predicated instructions can be used to avoid branches. (8 pts)

☑ Why do predicated instructions have maximum benefit over branches that skip over one instruction (compared to predicated instructions used to avoid branches that skip over more than one instruction)?

Because of the overhead needed for branches, especially in superscalar machines. Even on a one-way machine the branch overhead is worse than sometimes wasting time by executing a predicated instruction with a false predicate. As more instructions are skipped over by the branch, the benefit of predication drops since the branch overhead remains fixed while the waste of executing predicated instructions increases.

☑ Suppose the compiler is considering whether to replace a branch that skips n instructions with predicated instructions on a dynamically scheduled machine. How could the compiler use an estimate of prediction accuracy, f_{correct}, and resolution time, t_{resolve}, as well as implementation details to make its decision?

The performance potential lost due to mispredicted branches is proportional to the resolution time (the time needed to determine the branch condition). If a branch has a low prediction accuracy and a high resolution time then it will be responsible for squashing many instructions, and so the waste of predication is a better alternative. The higher $t_{\text{resolve}}(1 - f_{\text{correct}})$ the more instructions (higher n) one could predicate.

72 Spring 2002 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 22 March 2002, 13:40–14:30 CST

Problem 1		(35 pts)
Problem 2		(25 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

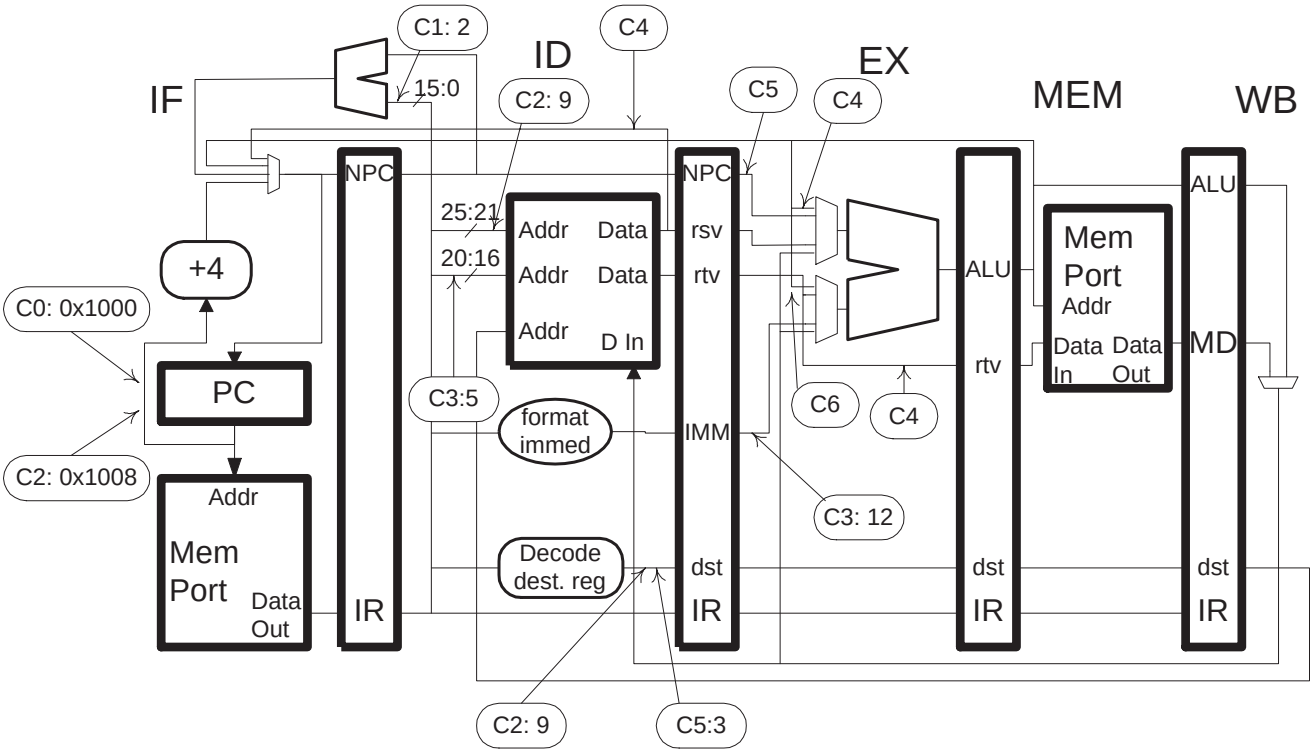
Alias Vaxinated

Good Luck!

Problem 1: In the diagram below certain wires are labeled with cycle numbers and values that will then be present, for example, C2:9 indicates that at cycle 2 the wire will hold a 9. Other wires are labeled just with cycle numbers, indicating that the wire is used at that cycle. Write a program consistent with these labels. There are no stalls during the execution of the code. The code should use five instructions. *Note: The diagram in the original exam had an error that resulted in contradictory information about the third instruction (fetched in cycle 2). The error was a* C3:5 *pointing to the input to the* `ID/EX.dst` *latch; that now points to the rt address input to the register file.* [35 pts]

☑ Write a program consistent with these labels.

☑ Use labels for branch targets (if any) and label the target line.



Solution shown below. Upper-case letters in instruction mnemonics indicate parts that are fixed. For example, **Sw** indicates that the instruction must be some kind of a store instruction, but instead of **sw** it could be **sh** or **sb**. Register numbers below $10 are based on the labels above, registers $10 and above are arbitrary. For example, $3 in the **and** must be $3, but $14 could be $15.

```
 # Cycle            0    1    2    3    4    5    6    7    8    9
 Bne $10,$10 TARG   IF   ID   EX   ME   WB
 addI $9, $9, 12         IF   ID   EX   ME   WB
 Sw $11, 0($9)                IF   ID   EX   ME   WB
TARG:
 JALR $31, $12                     IF   ID   EX   ME   WB
 and $3, $14, $31                       IF   ID   EX   ME   WB
 # Cycle            0    1    2    3    4    5    6    7    8    9
```

Problem 2: Consider the CISCy instruction below: [25 pts]

(*a*)

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

☑ What makes it CISCy? (CISCy means characteristic of CISC ISAs.)

It's an ALU instruction *and* it accesses memory, that's CISCy. Also, because of the large immediates that can be present, a reasonable coding would have variable-size instructions, another CISCyness.

☑ Ignoring instruction size, why would it be difficult to implement such an instruction on the kind of five-stage pipeline used in class for MIPS?

Lots of reasons. For one, because execution of the instruction includes memory access before the addition (to get the first operand) but in the five-stage pipeline memory is accessed after the execute stage.

Problem 2, continued:

```
add 0x123456(r1), 8(r2), r3   # 0x123456(r1) is the destination.
```

(*b*) How could the single-issue (not superscalar) pipeline used in class be modified so that instructions like the one above could be executed with a potential for a CPI of 1 without impacting clock frequency? (Instructions like the one above have one destination and two sources, the destination and first source can either use a register value or memory value specified with displacement addressing, the second source can be either a register value or an immediate.) Feel free to throw hardware at the problem, but do not assume that the hardware is any faster than what we've been using.

☑ Show a sketch of the pipeline, briefly explaining what should be done in each stage.

Here are the stages that would be needed. A correct solution might include a quick sketch.

IF: Same (pretend, because instructions are variable width).

ID: Same (pretend, because instructions are variable width).

EA: Compute effective addresses for destination and source operands (if needed).

M1: Load first source operand from memory (if needed).

EX: Same

M2: Store result in memory (if necessary).

WB: Store result in register file (if necessary). WB and M2 might be combined.

☑ Show a pipeline execution diagram for the following code on your new pipeline, assuming no dependencies.

Solution appears below.

```
# Solution

add 0x1234(r1), 8(r2), r3  IF ID EA M1 EX M2 WB

or r4, 7(r5), r6              IF ID EA M1 EX M2 WB

and 0x1234(r7), r8, r9           IF ID EA M1 EX M2 WB
```

☑ In the part above, why was it necessary to *assume* no dependencies?

First, about that word "assume." I might assume that you had breakfast this morning, but I never assume that I had breakfast because I know for sure whether I did or didn't. So the question above is asking why the code might still have dependencies alluding to the fact that there are no registers in common. The question does NOT ask "why was an example with no dependencies chosen for the problem."

Back to the question. The assumption was made because dependencies are still possible, through memory. The effective address of the destination of the `add` instruction might be the same as the first source operand of the `or` instruction. That is, the `add` writes the memory location that the `or` reads, a true dependency. (Such dependencies do not affect statically scheduled RISC processors because load and store instructions pass through the same memory stage in order.)

Problem 3: Answer each question below.

(*a*) What'll it be? One FP adder with an initiation interval of 2 and a latency of 3 (four cycles of computation) or two FP adders each with an initiation interval of 4 and a latency of 3? [10 pts]

☑ What is the maximum number of FP adds per second with each alternative on a 1 GHz system?

In either alternative a result can be produced every two cycles, at 1 GHz thats 500 million FP adds per second.

For some reason few answered this question correctly.

☑ Show a code fragment in which one of the alternatives is slightly faster. For the alternative with two adders, use label "A" for one adder and "B" for the other.

```
# Solution.

# On first alternative
add.d f0, f2, f4   IF ID A1 A1 A2 A2 WB
add.d f6, f8, f10     IF ID -> A1 A1 A2 A2 WB

# On second alternative
add.d f0, f2, f4   IF ID A1 A1 A1 A1 WB
add.d f6, f8, f10     IF ID B1 B1 B1 B1 WB
```

☑ Ignoring the cost of the adders themselves, which alternative would be more costly and why?

The second alternative, since the outputs of the two adders have to be multiplexed together.

(*b*) SPEC benchmark numbers are provided in "base" and "peak" forms. [10 pts]

☑ How are they different?

Programs compiled for the base numbers are prepared with normal (actually, a bit on the aggressive side) optimization. Programs compiled for the peak numbers are prepared with your-life-depends-on-it extreme optimizations.

☑ When should an intelligent (passed EE 4720) computer buyer use the base numbers, and when should the buyer use the peak numbers?

Use base numbers when you plan to use purchased software or when you can't put alot of effort in to tuning code. Use the peak numbers when compiling your own software and have the expertise and time to super-optimize, or if purchasing super-optimized software.

(*c*) Stack ISAs had burrowed their way in to our past and percolated to the fore in the past decade. [10 pts]

☑ Why are programs compiled for stack ISAs smaller than the same programs compiled for other ISAs? (Assume all compilers are of high quality.)

Many instructions in stack ISAs refer to the stack so they don't need register operands, and so they can be small, say one byte.

☑ Why would superscalar implementations of stack ISAs be less efficient (further from the ideal CPI) than superscalar implementations of RISC and some other "conventional" ISAs? Ignore instruction fetch problems. Consider the simple stack programs presented in class.

If the instructions in a fetch group in a statically scheduled superscalar processor (the only kind covered so far) are dependent some will stall. The stack organization forces dependent instructions to be near each other, guaranteeing lots of stalls in a superscalar implementation.

(*d*) At last superscalar implementations and VLIW ISAs will wage their epic [tm] battle now at the dawn of the twenty-first century.[10 pts]

☑ What distinguishes a superscalar implementation from a nonsuperscalar implementation? (VLIW isn't part of this question).

Superscalar processors can sustain execution of more than one instruction per cycle by duplicating fetch, decode, register read, and other parts of the system.

☑ Explain two ways in which VLIW machines overcome problems encountered in superscalar implementations of conventional (say RISC) ISAs?

The dependency information in VLIW bundles reduces the need for the dependency-checking hardware that is present in superscalar processors. Limiting branch targets to bundles reduces the inefficiencies due to un-aligned fetch groups.

Name Solution

Computer Architecture
EE 4720
Final Examination

18 May 2002, 12:30–14:30 CDT

Problem 1		(20 pts)
Problem 2		(37 pts)
Problem 3		(20 pts)
Problem 4		(23 pts)
Exam Total		(100 pts)

Alias Map or RAT?

Good Luck!

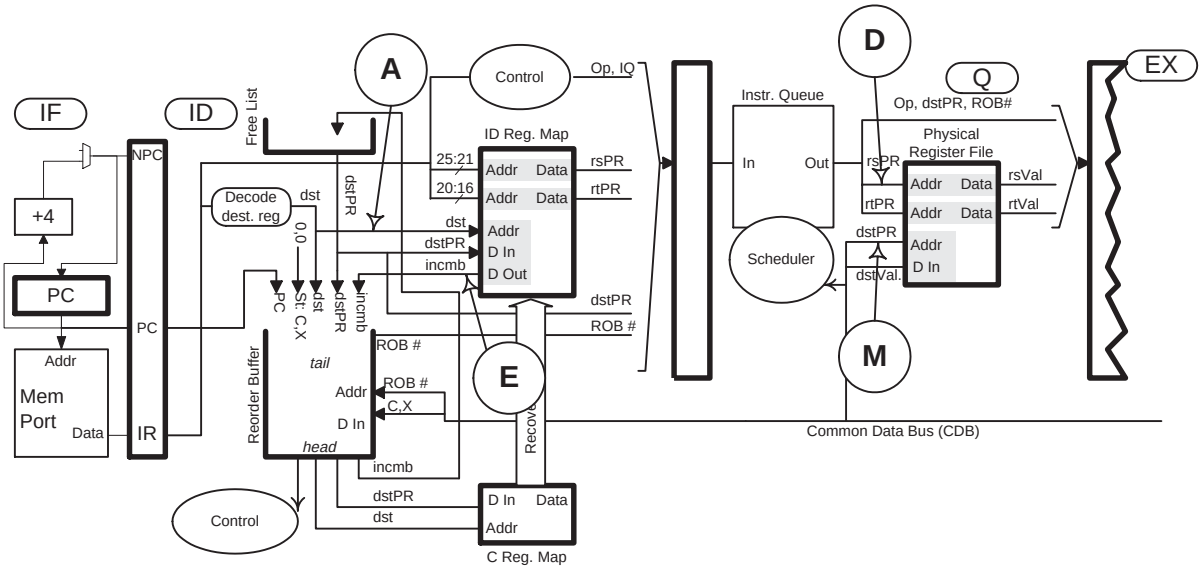
Problem 1: The (hopefully familiar) code fragment on the next page executes as shown on the dynamically scheduled system illustrated below.

(*a*) For each entry in the Physical Register File table on the next page place a "[" at the cycle(s) at which a physical register is allocated (removed from the free list) and place a "]" at the cycle(s) at which it is placed back in the free list. (This was part of the solution to Homework 5.) (10 pts)

(*b*) The diagram below includes circled letters, these letters appear in a Signals Values Table on the next page.

☑ (10 pts) Fill in the table showing the signals that will appear on the labeled wires.

- Use register names (`f0`, `$1`, etc.) for architected registers (but not for physical registers!).
- Use a question mark for a physical register number that cannot be determined from the tables.
- Since the machine is four-way superscalar each wire holds four values. Do not show values for the instructions that are not in the table, such as the two instructions following `sub`.
- Assume that values have been correctly computed, even if they depend upon preceding instructions in the same group. (This affects row E in the table.)



The filled-in tables appear on the next page.

The D row shows the physical register used for the rs operand being read for instructions starting execution. Instructions start execution when their operands are ready, with one exception that is the cycle after the instruction is decoded. The exception is multiply which waits two cycles.

The M row shows the physical register number for instructions being written back. The easy way to fill this row is to check when entries are written in the physical register file table.

```
LOOP: # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 ldc1 f0, 0($1)   IF ID Q  L1 L2 WC
                           IF ID Q  L1 L2 WB             C
                                    IF ID Q  L1 L2 WB            C
                                             IF ID Q  L1 L2 WB
 mul.d f0, f0, f2 IF ID Q        M1 M2 M3 M4 M5 M6 WC
                           IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                    IF ID Q        M1 M2 M3 M4 M5 M6 WC
                                             IF ID Q        M1 M2 M3 M4 M5
 sdc1 0($1), f0   IF ID Q  L1                      L2 WC
                           IF ID Q  L1                      L2 WC
                                    IF ID Q  L1                      L2 WC
 addi $1, $1, 8   IF ID Q  EX WB                   C
                           IF ID Q  EX WB                   C
                                    IF ID Q  EX WB                   C
 bne $2, $0 LOOP     IF ID Q  B  WB                C
                             IF ID Q  B  WB                C
                                      IF ID Q     B  WB              C
 sub $2, $1, $3      IF ID Q  EX WB                C
                             IF ID Q  EX WB                C
                                      IF ID Q  EX WB                C
 # ID Map         0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99              97,96    94,93    91,90
 $1 98              95       92       89
 # In cycle one first 97 is assigned to f0, then 96 (replacing 97).  The
 # same sort of replacement occurs in cycles 4 and 7.
 # Commit Map     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 f0 99                          97               96    94 93    91 90
 $1 98                                              95       92       89

 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 Physical Register File
 99               1.0           ]
 98               0x1000                        ]
 97                 [           10              ]
 96                 [                           11    ]
 95                 [         0x1008                        ]
 94                           [          20               ]
 93                           [                           2.2   ]
 92                           [       0x1010                        ]
 # Cycle          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17

 # Signal Values  0    1    2    3    4    5    6    7    8    9    10   11   12
 # Solution
 A                f0   $2             f0   $2        f0   $2        f0   $2
                  f0                  f0             f0             f0
                  $1                  $1             $1             $1

 D                           98   ?    97   95   ?    94   92 ...
                             98   95        95   92        92 ...
                             98             95             92 ...


 E                      99   ?         96   ?         93
                        97             94             91
                        98             95             92

 M                                     95   97        92   94             96

 # Cycle          0    1    2    3    4    5    6    7    8    9    10   11   12
```

Problem 2: The `add` instruction below is a predicated version of a MIPS instruction (for this exam). If the contents of register `$s5` (for the example) is non-zero the `add` instruction executes normally. If it's zero then it's as though the `add` never executed. Any GPR can be used to hold the predicate.

```
 # Cycle                          0  1  2  3  4  5  6
          xor $v1, $a0, $a1       IF ID EX ME WB
          lw $s5, 0($t2)             IF ID EX ME WB
     ($s5) add $s1, $s2, $s3            IF ID EX ME WB
```

(*a*) Suppose there are predicated versions of other two-source, one-destination instructions. How might they be coded in MIPS? The coding should be chosen to ease implementation. (5 pts)

Use the Type-R format and use the SA field to hold the predicate register number.

(*b*) Modify the pipeline below to implement predication; it should run the code above correctly and without stalls (as illustrated). The added hardware must detect whether the predicate is true or false and take appropriate action. It should include bypassing, for example, to handle the dependency carried by `$s5` or from `xor` if the predicate were `$v1`.

Hint: A correct solution uses multiplexors, an is P *(box that recognizes a predicated instruction), assorted gates, and some modifications to existing components.*

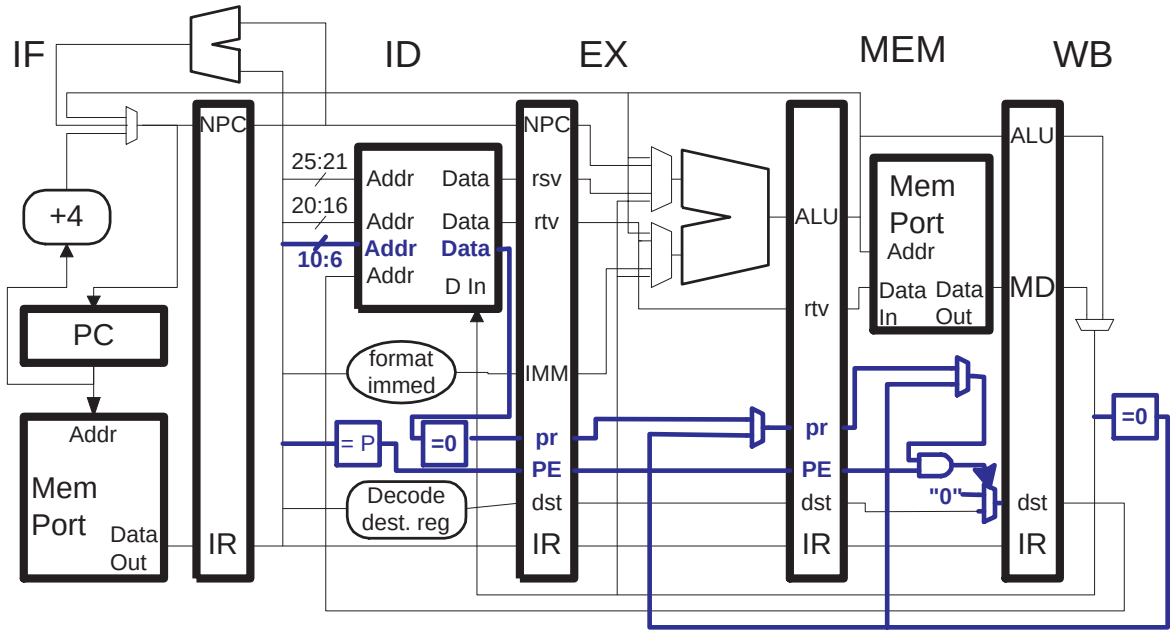
(9 pts)

Changes shown in **blue bold**.

In the `ID` stage a port has been added to the register file to read the predicate value, it feeds a NOR gate (to test if it's equal to zero) and is written to a new `pr` pipeline latch. The predicate is examined in the MEM stage, it its true (`pr` is `1`) and the instruction is predicated (`PE` is `1`) the "0" is select from the `dst` multiplexor, changing the destination register to zero. It's necessary to put this hardware in the MEM stage so that predicated instructions with dependencies like `add` can execute without stalling.

Bypass multiplexors are included in the EX and MEM stages, the EX mux would be used if the predicate were `v1`, the MEM mux is used, to bypass `s5`.

Note that the `pr` pipeline latches are just one bit. An alternative would be to store a full 32-bit value and check if it's zero in the MEM stage, but that would be more costly.



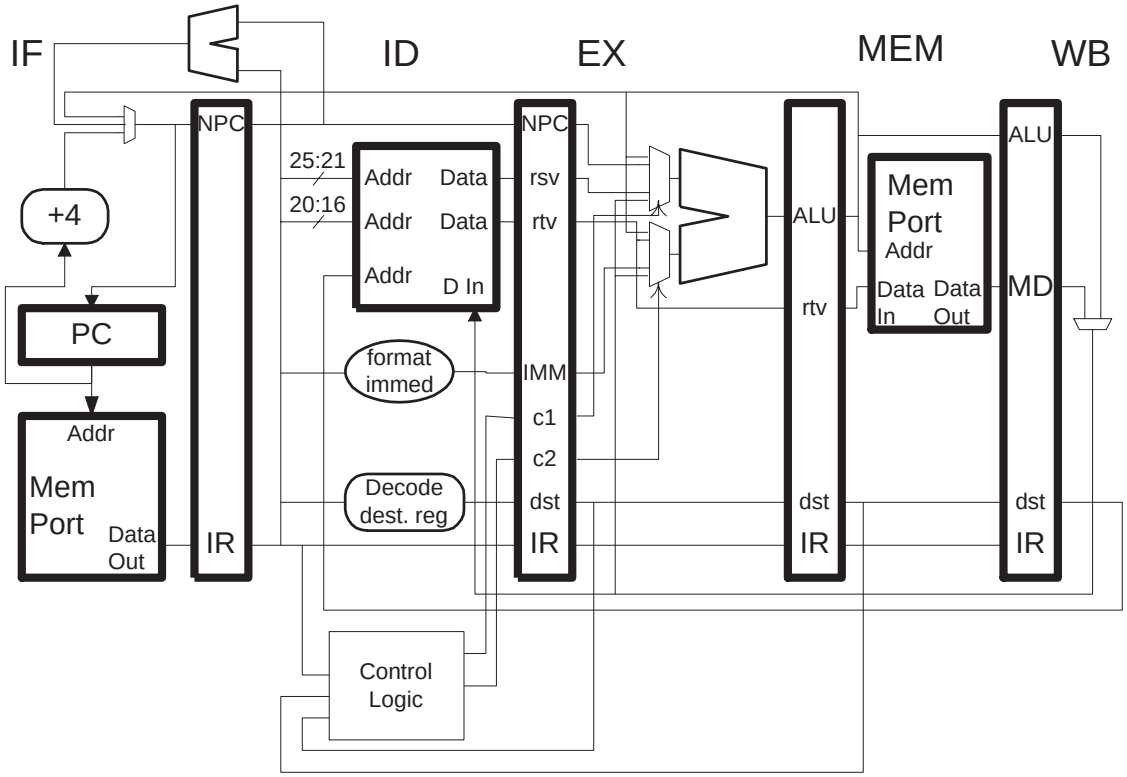
Problem 2, continued:

(*c*) Assume predicates were implemented as requested in the previous part. Explain why bypassing of the dependency carried by `$s1` in the program below won't work for the hardware below. Explain how the problem might be fixed. (8 pts)

It won't work because the bypass control logic is in the ID stage. The `or` will need to bypass a value from `add` only if its predicate is true, otherwise it will read `s1` from the register file. When the `or` is in ID the predicate on `add` is not yet resolved and so there is no way the bypass logic, which is in ID, can set the ALU multiplexors correctly.

A solution would be to move at least some of the control logic to the EX stage, and have that logic compute bypass connections based on how predicates are resolved. A less expensive and lower performance solution would be to stall instructions in ID if they depend on a value produced by an unresolved predicated instruction in EX or ME.

```
 # Cycle                        0  1  2  3  4  5  6
           xor $s5, $a0, $a1    IF ID EX ME WB
           lw $s5, 0($t2)          IF ID EX ME WB
      ($s5) add $s1, $s2, $s3         IF ID EX ME WB
           or  $s4, $s1, $v0             IF ID EX ME WB
```



Problem 2, continued: Consider such predicated instructions in a dynamically scheduled implementation using method 3. The implementation without predicate prediction (the next two parts) should realize the benefit of predicated instructions: avoiding the need for branches.

(*d*) How might the ID Register Map be updated for predicated instructions? (For partial credit: Why is this a good question?)(5 pts)

Update the ID map for a predicated instruction in the same way as an ordinary instruction: write the physical register number assigned to the destination. If the predicate turns out to be false just copy the value from the incumbent physical register to the one assigned to the instruction.

(*e*) Based on your answer above, how is execution affected if the predicate turns out to be false? (5 pts)

The only impact on execution is the need to copy a physical register and the ultimately unnecessary waiting of instructions dependent on the destination register.

(*f*) Consider a dynamically scheduled system using predicate prediction. What should be done when a predicate is mispredicted? Under what circumstances (properties of program being run) would it be faster than a system without predicate prediction? Under what circumstances would it be slower? (5 pts)

First things first. There is a big difference between a false predicate and a mispredicted one. When a predicate is mispredicted following instructions may read the wrong data. For example, if a predicate is mispredicted true then some instructions may read the value written by the predicated instruction or if its mispredicted false they read an "outdated" value rather then the one the predicated instruction would have written. In either case they read the wrong value and so will have to be squashed or re-executed. The problem is that following instructions started executing based on a predicted predicate value, not the real one. If there is no predicate prediction following instructions will wait for the predicate to resolve so they will always execute with the correct data, whether or not the predicate is true or false.

Now back to the solution: What one does on a predicate misprediction depends on how the ID map was updated. If the ID map was updated as described above then on a misprediction instructions dependent on the result can be re-executed (not something covered in class), a simpler solution would be to squash all instructions following the predicated instruction, restore the ID map and re-start fetching after the predicated instruction. If instructions are re-executed the ID map does not have to be restored because it is correct, only the values in the physical registers are wrong.

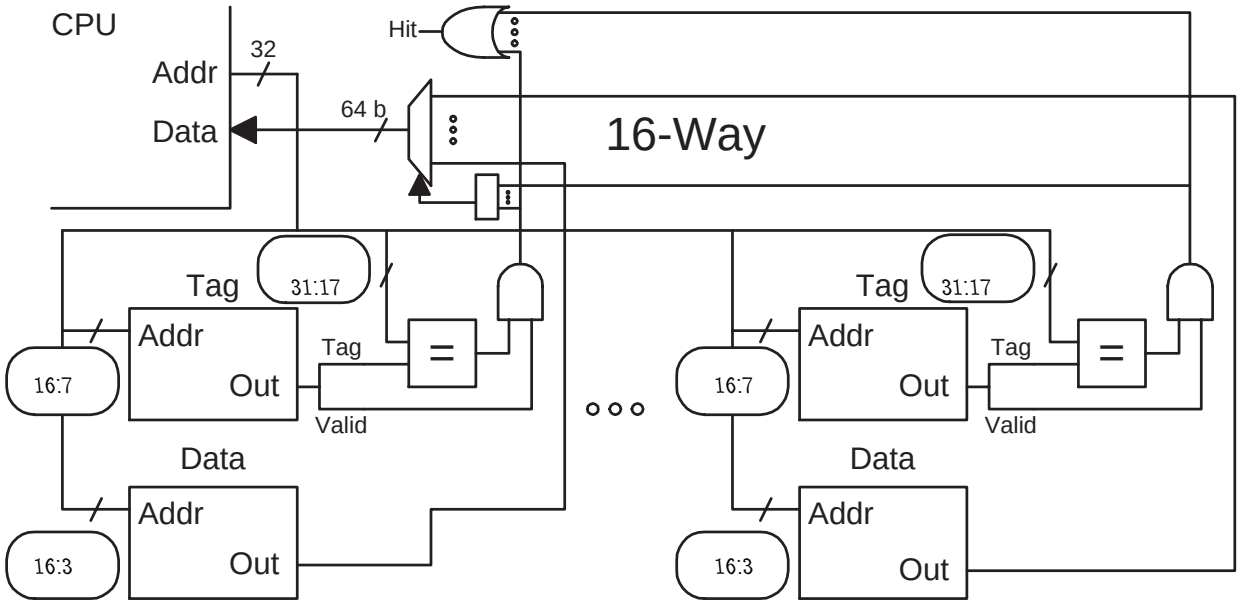
Another way to handle the ID map is to not update it for predicates that are predicted false. In that case squashing would be necessary (re-execution would not be possible) since the ID map would be invalid.

Programs benefit from predicate prediction if predicates become available later than other operands to predicated instructions and there are close dependencies with predicated instructions. It would be slower if prediction accuracy were low since valid instructions would be squashed and re-executed. There would be little benefit if predicates were never available later than other operands to the instruction. (The little benefit would be from scheduling latency, something not covered in class very much.)

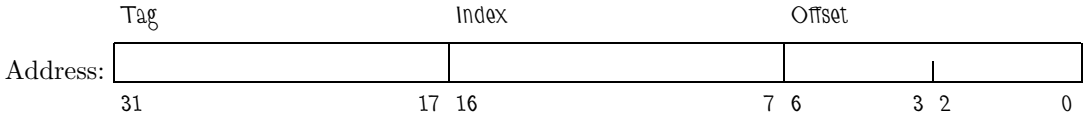
Problem 3: The diagram below is for an 2 MiB (2^{21} byte) 16-way set-associative cache, with a line size of 128 characters, for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)

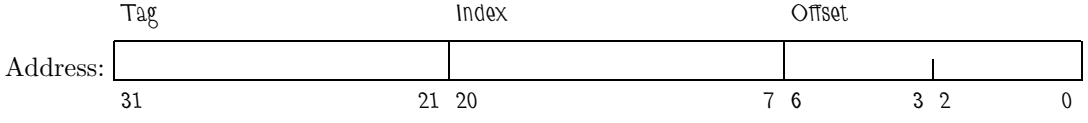


☑ Memory Needed to Implement (Indicate Unit!):

It's 2^{21} characters (data) plus $(32 - 17 + 1)$ 16 $2^{17-7} = 2^{18}$ bits for the tag store.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.

Since it's 16-way set associative the index will need to be made $\log_2 16$ bits larger. Since the line size can't change the offset bits can't be expanded (or shrunk) and so the tag size is reduced.



Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
short *h = 0x100000;  // sizeof(short) = 2 characters
int sum;
int i,j;
int ISTRIDE = 1;
int ILIMIT = 1024;

for(j=0; j<4; j++)
  for(i=0; i<ILIMIT; i++)
    sum += h[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

☑ What is the hit ratio for the program above?

Each line holds $\frac{2^7}{2} = 64$ short integers. Since data is read sequentially on the first `j` iteration there will be a miss followed by 63 hits for a hit ratio of $\frac{63}{64}$. The total data read in a single `j` iteration is $1024 \quad 2$ characters, which is much less than 2 MiB and so on subsequent `j` iterations all accesses will hit.

The total hit ratio is thus $\boxed{\frac{\frac{63+3}{4}}{\frac{64}{64}} = \frac{255}{256} = 0.996094}$.

☑ Find the *minimum* value of `ISTRIDE` needed to maximize the miss ratio. (That is, *minimize* the hit ratio, make things really slow.) Don't forget that the cache is set associative. Do not modify `ILIMIT`.

On a *single* `j` iteration the hit ratio can be maximized by accessing a different line every `i` iteration, that is done by setting `ISTRIDE` to one half the line size, (2^6), (since each element is two characters). *But* every access would be a hit on the next `j` iteration. *So* we need to make sure that a line cached in one `j` iteration is evicted by the next `j` iteration. Since its a 16-way cache a line is evicted after 16 different lines having the same index are accessed (assuming LRU replacement). Making `ISTRIDE` really big and a power of two would do this since then every access would have the same index. *But* the problem asked for the minimum `ISTRIDE` so we need to think a little more.

Think of `i` as a 10-bit number (since it iterates from 0 to $2^{10} - 1$). In order to get exactly 32 distinct tags for each index we need to "shift" `i` so that its five most significant bits are in the tag region, and the rest are in the index region. This way, for each index there will be 32 tags, forcing evictions. (If there were just four bits in the tag region then the cache would be able to hold the first j iteration.) To do that, set `ISTRIDE` to 2^{11}, and so the address will be computed by shifting `i` 12 places to the left (one extra place since a short is two characters).

So the answer is $\boxed{\texttt{ISTRIDE}=2^{11}}$.

Problem 4: Answer each question below.

(*a*) In the pipeline execution diagram below the multiply is squashed to avoid the WAW hazard. How does this make a precise exception impossible? (5 pts)

```
mul.d  f0, f2, f4  IF ID M1x
add.d  f0, f6, f8     IF ID A1 A2 A3 A4 WB
```

Suppose the `add.d` raises an exception. The exception handler will not see the value of `f0` written by `mul.d` (because it was squashed). For an exception to be precise the handler must see the effect of all instructions that precede the faulting instruction.

(*b*) How do processes share memory in a virtual memory system? Provide an example in which two processes share address `0x12000` but address `0x34000`, used by each process, is not shared. The addresses must be used in the example. (5 pts)

They share memory by having virtual addresses map to the same physical address. Call the two processes A and B. The page tables for process A and B might both map `0x12000` to physical page `0x1000`, and so reads and writes would be writing the same memory. Process A's page table might map `0x34000` to physical address `0x2000` while process B's page table might map it to `0x3000` and so virtual address `0x34000` would be separate for each process.

(*c*) Show how the branch history table and pattern history tables are connected in a local history branch predictor. Show how the table is indexed and how its contents are used to make a prediction. (5 pts)(For partial credit show a one-level [bimodal] predictor.)

The address input of the BHT is connected to the low bits of the program counter (or maybe NPC). The output of the BHT contains the local history for the branch, it is connected to the address input of the PHT. The PHT output is a two-bit counter, if the upper bit is 1 predict taken. A correct answer should include a sketch.

(*d*) One way to improve performance is to divide the pipeline into more stages, as in the Pentium 4 compared to the Pentium III. This does not reduce the amount of time it takes to compute things, such as sums, though. Given that: (8 pts)

☑ How are dependencies potential performance limiters when dividing pipeline stages?

The more pipeline stages are divided the higher the latency for functional units, such as the integer unit. In the classical case the integer functional unit has zero latency so dependent instructions can start in the next cycle. If it's divided in two the latency becomes one and so dependent instructions that are decoded in the next cycle will have to stall. At some point dividing pipeline stage will yield no additional performance because every instruction would be stalling.

☑ How does the Pentium 4 Fast ALU get around that?

By dividing the ALU in to low and high halves and allowing data from the output of the low half to be bypassed to the input of the low half. The same is done for the high half. Therefore even though it takes two cycles to compute a complete result (three counting conditions) a bypassable value is available in one cycle.

☑ Are dependencies still a problem or can we now use zillion-stage pipelines (at least as far as dependencies are concerned)? Explain.

Yes. This trick only works for certain integer operations, so other instructions would dominate performance. Even without those other operations, the number of bits in an integer operation is a limit.

☑ Why is branch prediction accuracy more important when there are more stages?

Because the number of instructions that are fetched between the time a prediction is made and the time it resolves is higher. For a given branch prediction accuracy more pipeline stages mean more instructions squashed. This doesn't mean lower performance, it means that performance gained by dividing pipeline stages is limited. If each pipeline stage were divided in two clock frequency and so performance, could be nearly doubled. (Nearly because of greater logic complexity, register setup times, etc.) But because roughly the same amount of time is needed to resolve a mispredicted branch (regardless the number of pipeline stages) at best the part of execution which does not include resolving mispredicted branches will nearly double.

73 Fall 2001 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

Friday, 26 October 2001, 13:40–14:30 CDT

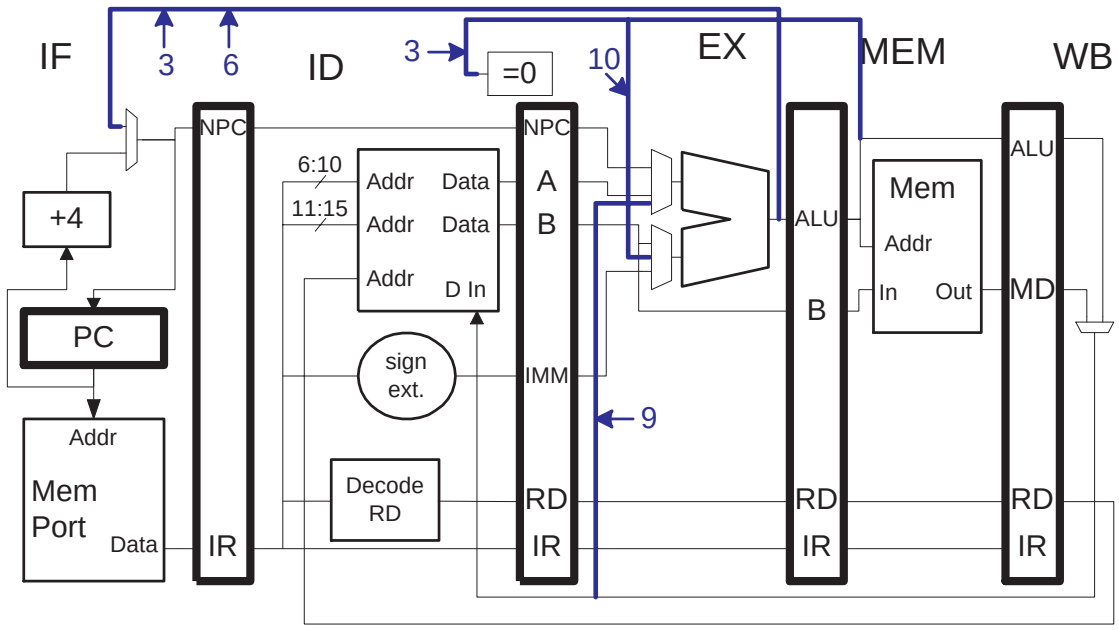
Problem 1		(15 pts)
Problem 2		(15 pts)
Problem 3		(10 pts)
Problem 4		(60 pts)
Exam Total		(100 pts)

Alias ΨΨΨ

Good Luck!

Problem 1: The DLX implementation below lacks bypass paths and, worse than that, lacks the hardware needed for control-transfer instructions.

Changes and cycles shown in blue in the diagram below.



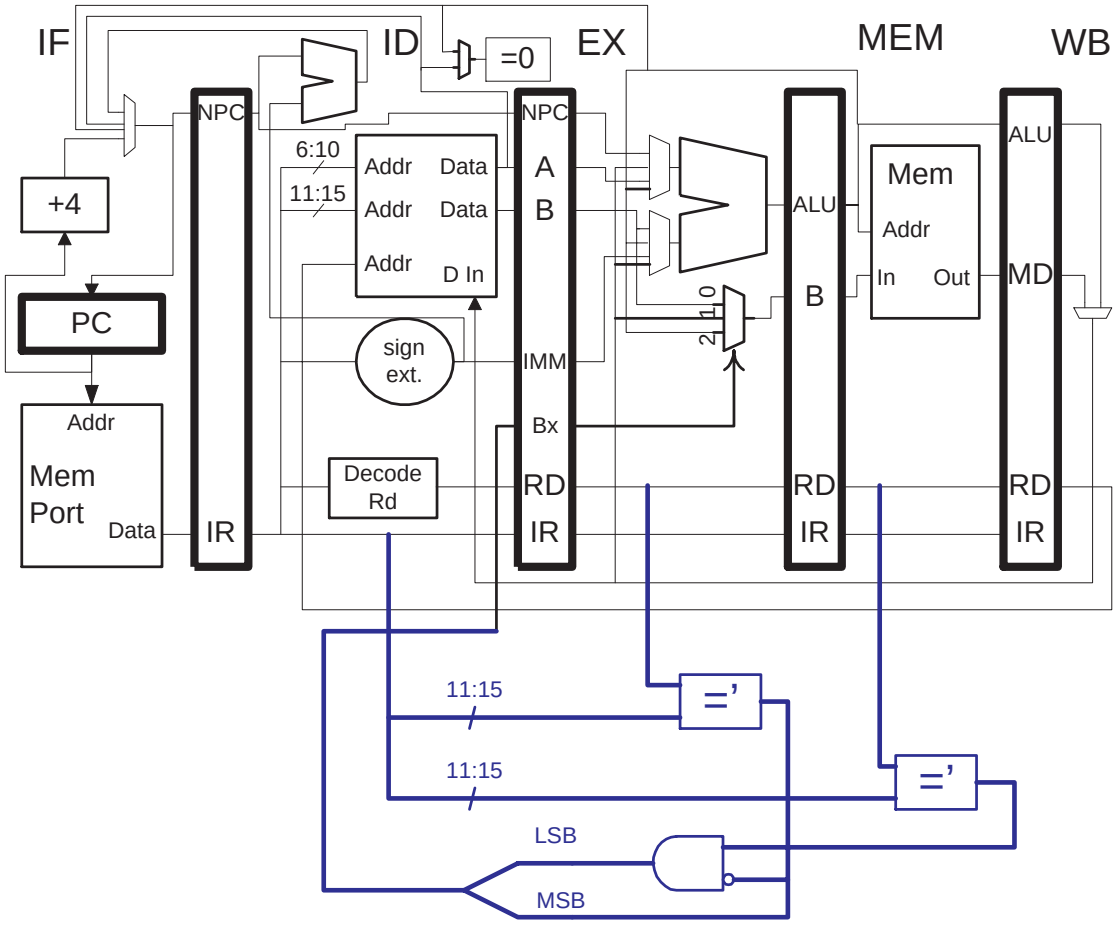
☑ [5 pts] Add exactly the hardware needed so that the control-transfer instructions execute *as shown* below. Include a connection to the =0 box used in determining whether a branch is taken.

☑ [5 pts] Add exactly those bypass paths necessary so that the code below executes *as shown*. Check the code carefully for dependencies, including all those related to the `jalr` instruction.

☑ [5 pts] Show the cycles in which each added wire will be used.

```
! Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
 ori r1, r1, #15    IF ID EX ME WB
 bnez r1, SKIP         IF ID EX ME WB
 add r0, r0, r0           IF IDx
 xor r0, r0, r0              IFx
SKIP:
 sub r20, r20, r21              IF ID EX ME WB
 jalr r20                          IF ID EX ME WB
 xor r0, r0, r0                       IFx
 ...
 add r15, r31, r0                        IF ID EX ME WB
 or  r16, r16, r15                          IF ID EX ME WB
! Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12
```

Problem 2: The DLX implementation below includes bypass paths into the `EX/MEM.B` register.



☑ [5 pts] Write a program that uses all three paths into the `EX/MEM.B` register. *Hint: It's easy.*

```
addi r2, r2, #1  IF ID EX ME WB
sw 0(r1), r2        IF ID EX ME WB
sw 4(r1), r2           IF ID EX ME WB
sw 8(r1), r2              IF ID EX ME WB
```

☑ [10 pts] Design the control logic for the multiplexor feeding the `EX/MEM.B` register. The control logic should be in the ID stage and feed into the `ID/EX.Bx` pipeline latch provided in the diagram above.

Changes shown in blue.

Problem 3: Registers `r1` and `r2` each contain a signed integer, call them i and j. Register `r10` contains a double-word-aligned address, call the address A. Let $p = i \quad j$.

☑ [10 pts] Starting at address A write p in three formats: integer, double-precision floating-point, and single-precision floating-point. Maintain as much precision as possible.

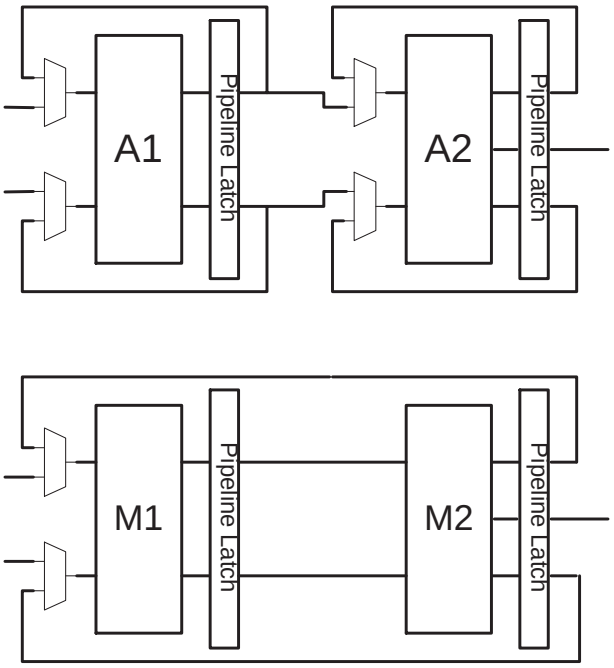
Hint: A reasonable solution would use `movitof` *and* `cvtXtoY` *instructions.*

```
! Initially: r1 and r2 each contain an integer, i and j.
!            r10 contains an address.
!
!            At Mem[r10]    write i * j (integer format).
!            At Mem[r10+?]  write i * j (double-precision FP)
!            At Mem[r10+??] write i * j (single-precision FP)
```

```
# DLX Solution
movitofp f0, r1
movitofp f2, f2
cvtitod  f0, f0
cvtitod  f2, f2
multd    f4, f0, f2
cvtdtoi  f6, f4
sf 0(r10), f6
sd 8(r10), f4
cvtdtos f4, f4
sf 16(r10), f4

# MIPS Solution
mtc1    $1, $f1           # Move to FP (co-processor 1) register.
mtc1    $2, $f2
cvt.d.w $f10, $f1         # Convert from integer to double FP.
cvt.d.w $f20, $f2
mul.d   $f4, $f10, $f20
sdc1    $f4, 8($10)       # Store double-sized item (from $f4 and $f5).
cvt.s.d $f6, $f4
swc1    $f6, 16($10)      # Store word-sized item
cvt.w.d $f6, $f4
swc1    $f6, 0($10)
```

Problem 4: Answer each question below.

(*a*) In the two pipelined functional units below an instruction must pass through each segment twice. The `A`'s are for FP addition and `M`'s are for FP multiplication.



☑ [10 pts] Complete the pipeline execution diagram below for a system using these functional units. Don't overlook the dependency through `f6`.

```
! Solution

addd f0, f2, f4     IF ID A1 A1 A2 A2 WB

addd f6, f8, f10       IF ID -> A1 A1 A2 A2 WB

muld f12, f6, f14         IF -> ID -------> M1 M2 M1 M2 WB

muld f16, f18, f20              IF -------> ID M1 M2 M1 M2 WB
```

(*b*) The code below, which of course is not DLX, uses memory-indirect and autoincrement addressing.

```
lw r1, @(r2)  ! Memory-indirect load.
lw r4, (r5)+  ! Autoincrement
lh r6, (r7)+  ! Autoincrement
```

☑ [10 pts] Rewrite the code in DLX.

```
lw r1, 0(r2)
lw r1, 0(r1)
lw r4, 0(r5)
addi r5, r5, #4
lh r6, 0(r7)
addi r7, r7, #2
```

(*c*) The three types of interrupts discussed in class are traps, hardware interrupts, and exceptions.

☑ [6 pts] For each one explain how the exception code (number) is determined.

Traps: the code is specified in the trap instruction itself. Hardware Interrupts: the code is based on the interrupt request line that was asserted. Exceptions: the code is based on what went wrong with the faulting instruction.

☑ [6 pts] For each one explain where control returns after the handler completes.

Trap: The instruction following the trap instruction. Hardware Interrupt: The instruction just after the last one to complete. Exception: the faulting instruction.

(*d*) An ISA has two implementations, A and B; each implementation has a well-written compiler.

☑ [5 pts] Would the code compiled by A's compiler run on implementation B? Briefly explain.

Yes, since they are compiled for the same ISA.

☑ [5 pts] A program is compiled using A's compiler and B's compiler. How might the compiled code differ? Provide a reason for the difference.

Scheduling of instructions might be different because of differences in functional unit latencies. For example, in A loads might have a latency of 1 (as in Chapter 3 DLX) while in B they might have a latency of 2, and so for B the compiler try to move two instructions, rather than one, between a load and a use of the loaded value.

(*e*) You have become the owner of a large American computer company, congratulations.

☑ [6 pts] How can you (legally) influence the decision-making process so that SPECs next benchmark suite does not unfairly put your company's products at a disadvantage? (Note: Bribery is illegal in the U.S.)

Become a member of SPEC and help create, and vote for, benchmarks fair benchmark suites.

(*f*) DLX does not have delayed branches, but many other RISC ISAs do.

☑ [6 pts] What is a delayed branch and how does it help?

A delayed branch is one in which the instruction following the branch (or the d instructions following the branch, though d is almost always 1) in program order is executed regardless of whether the branch is taken, followed by the branch target if the branch is taken. Many implementations will be forced to squash the instruction following a taken non-delayed branch, with a delayed branch the instruction following the branch is not squashed and so can do useful work.

(*g*)

☑ [6 pts] How and why is the CPI affected in an implementation re-designed for a higher clock frequency?

With a higher clock frequency less work can be done per clock cycle. This might increase the latency of certain instructions (for example, from six to twelve multiply segments), increasing the number of stall cycles and thus increasing CPI.

Name Solution

Computer Architecture
EE 4720
Final Examination

11 December 2001, 7:30–9:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

Alias Solution!!!

Good Luck!

Problem 1: The code fragment below executes as shown on a single-issue dynamically scheduled system using a physical register file (Method 3). Initially register `f0` contains a 0, `f2` contains a 0.2, and `f4` contains a 0.4. The value computed by each instruction is shown near the right margin. (20 pts)

☑ Show where each instruction commits.

☑ Complete the ID- and commit-stage register map tables.

☑ Complete the physical register file table.

☑ Be sure to show the initial values for `f0`, `f2`, and `f4` in the register maps and the register file.

☑ In the physical register file use a [to indicate when a register is removed from the free list and use a] to indicate when it's put back.

```
!Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
addd f0, f0, f2   IF ID Q                          A1 A2 WC                        (.2)
addd f4, f2, f4      IF ID Q  A1 A2 WB                    C                        (.6)
addd f2, f0, f8         IF ID Q                       A1 A2 WC                    (1.)
addd f0, f4, f8            IF ID Q  A1 A2 WB                   C                 (1.4)
addd f4, f2, f8               IF ID Q                             A1 A2 WC       (1.8)
!Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
```

```
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
ID:
f0   0                3        6
f2   1                   5
f4   2                4        7
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
Commit:
f0   0                                                 3        6
f2   1                                                       5
f4   2                                                    4        7
```

```
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
PRF:
0    0                                                 ]
1    .2                                                      ]
2    .4                                                   ]
3                     [                                 .2       ]
4                        [           .6                           ]
5                           [                                 1.
6                              [           1.4
7                                 [                                 1.8
8
9
!Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22
```

Problem 2: The code fragment below executes on three different systems, I, L, and G, each using a different branch predictor:

I: One-level predictor, 2^{16}-entry BHT. L: Sixteen-bit (two-level) local history. G: Sixteen-bit (two-level) gshare.

In the spaces provided below show the accuracy of, and the number of entries used by, the indicated branch on the indicated system. The accuracy and number of entries should be after warmup. The number of entries used for the gshare scheme can be approximate. If there is more than one table, show the number of entries in each table separately.(20 pts)

☑ B1 on I: Accuracy: 50% Entries: One entry.

☑ B2 on I: Accuracy: 99% Entries: One entry.

☑ B3 on I: Accuracy: $66\frac{2}{3}$% Entries: One entry.

☑ B1 on L: Accuracy: 50% Entries: $1 + 2^{16}$

☑ B2 on L: Accuracy: 99% Entries: $1 + 17$

☑ B3 on L: Accuracy: 100% Entries: $1 + 3$

☑ B1 on G: Accuracy: 50% Entries: $\approx 2^8$

☑ B2 on G: Accuracy: 99% Entries: $\approx 2^8$

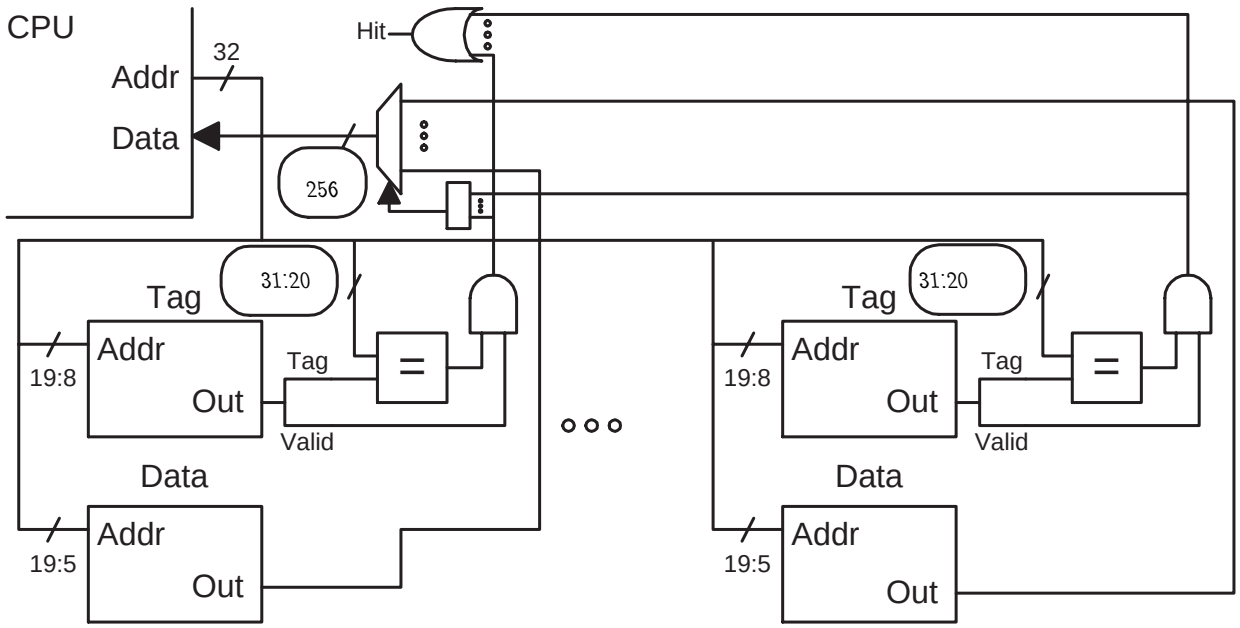
☑ B3 on G: Accuracy: 100% Entries: $\approx 1 + 2^9$

```
BIGLOOP:

LOOP2: ! Iterates 100 times
 ! Random, 0 or 1
    lw r1, 0(r2)
    addi r2, r2, #4
B1: bnez r1, SKIP1
    add r10, r10, r11
SKIP1:
    sub r3, r2, r12
B2: bnez r3, LOOP2

    addi r1, r0, #3
LOOP3:
    subi r1, r1, #1
B3: bnez r1, LOOP3

j BIGLOOP
```

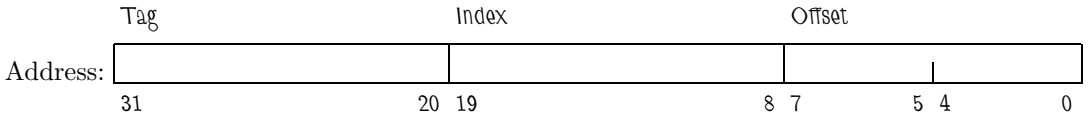
Problem 3: The diagram below is for an 8 MiB (2^{23} byte) set-associative cache for a system with 8-bit (how ordinary) characters.

(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Associativity:

Eight-way.

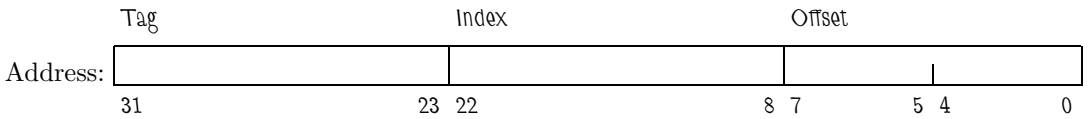
☑ Line Size (Indicate Unit!):

It's 2^8 characters.

☑ Memory Needed to Implement (Indicate Unit!):

It's 2^{23} characters (data) plus $8 \times 2^{12}(12+1)$ bits for the tag store.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 3, continued: Continue to use the set-associative cache from the previous part.

When the code below starts running the cache is empty. Consider only accesses to the array.

```
double *d = 0x100000;  // sizeof(double) = 8 characters
double sum;
int i;
int ISTRIDE = 1;
int ILIMIT = 64;

for(i=0; i<ILIMIT; i++)
  sum += d[ ISTRIDE * i ];
```

(*b*) Answer the following. (10 pts)

☑ What is the hit ratio for the program above?

It's $\frac{31}{32}$. (Note that the size of a double is 8 characters.)

☑ Modify `ISTRIDE` and `ILIMIT` so that the cache, which remember is set-associative, is completely filled *with the minimum number of accesses*.

Set `ISTRIDE` so that successive values of `i` access successive lines. That would be the line size divided by the array element size, `ISTRIDE` $= \frac{2^8}{2^3} = 2^5$. Set ILIMIT to the number of lines, which is the associativity times the number of sets (two to the power of the number of index bits): `ILIMIT` $= 8 \times 2^{20-8} = 2^{3+20-8} = 2^{15}$.

Problem 4: Answer each question below.

(*a*) The contents of the load/store queue in a dynamically scheduled system is shown below for $t = 4720$. At this particular time the cache is empty, but with a load/store queue full of instructions it won't be empty for long. The instruction in entry `L0` is the oldest.

The EA field indicates the effective address, a `#1` in this field indicates that the effective address cannot be computed until the instruction in ROB entry #1 (not shown) completes. It completes at $t = 7700$.

The Data field indicates the data that will be written by the store instruction in the entry.

The cache is nonblocking and the load/store queue can process an unlimited number of instructions per cycle.

☑ (5 pts) For each load instruction at $t = 4721$ if it's complete show what data it has loaded otherwise indicate the first thing it's waiting for.

Solution under the "Data loaded or reason for waiting" column.

```
     Type  EA      Data   Data loaded or reason for waiting.
 L8: load  #1             Need value from instruction in ROB entry #1 for EA
 L7: load  0x1000         Loaded 55
 L6: store 0x2000  66
 L5: store 0x1000  55
 L4: load  0x1000         Waiting for L3 to compute its address (could be 0x1000)
 L3: store #1      33
 L2: store 0x1000  22
 L1: load  0x2000         Waiting for the cache.
 L0: load  0x1000         Waiting for the cache.
```

(*b*) The exception recovery mechanism in the R10000 does not need a commit map, the one in Method 3 does.

☑ (5 pts) But the R10000 takes longer to recover. Why does it take longer and how much longer does it take?

It takes longer because rather than replacing the ID map with a commit map it re-creates the older ID map by reversing the changes made by the soon-to-be-squashed instructions four at a time. If there are sixteen instructions in the ROB that would take 4 steps.

(*c*) In some schemes for recovering from branch mispredictions on dynamically scheduled systems recovery can start at completion rather than waiting for commitment.

☑ (5 pts) What additional hardware is needed for this? How is it used?

Storage for backup copies of the ID map. When a branch is predicted a backup copy of the ID map is stored. In the writeback stage of a mispredicted branch the ID map is replaced by the backup copy corresponding to the mispredicted branch.

(*d*)

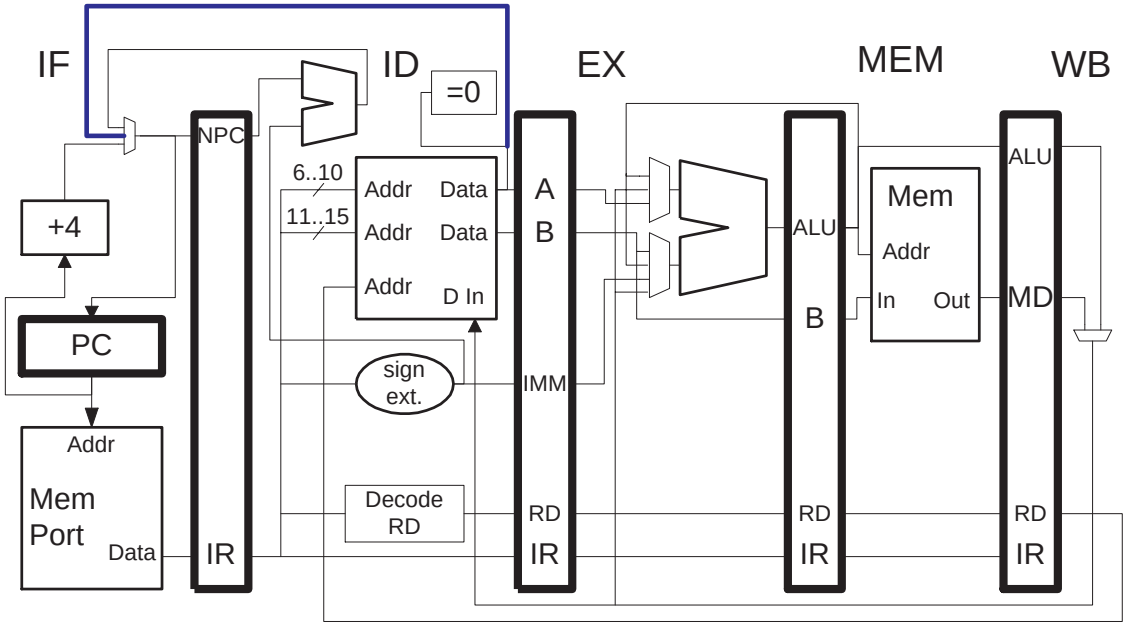
☑ (5 pts) Show how bits are categorized in a real address and a physical address in a system with a 64-bit virtual address space, a 40-bit real address space and 2^{14}-character pages.

Bits 0 to 13 in real and virtual addresses are the offset. In a virtual address bits 14 to 63 are the virtual page number and in a real address bits 14 to 39 are the real page number.

(*e*) Connections for the `jr` instruction are omitted from the pipeline below.

☑ (5 pts) Add them.

Change shown in blue below.



☑ (5 pts) Show a pipeline execution diagram for the code below consistent with the modified hardware above.

```
 slli r10, r1, 2    IF ID EX ME WB
 addi r10, r10, r2     IF ID EX ME WB
 lw   r10, 0(r10)         IF ID EX ME WB
 jr   r10                    IF ID ----> EX ME WB
 ...                            IF ---->x

TARG: !Target of jr.
 add r20, r21, r22                      IF ID EX ...
```

(*f*) Here are the IA-64 instructions used in the solution to Homework 3: `ld1` (load byte), `cmp.eq` (compare equal), `cmp.le` (compare less than or equal), `cmp.gt` (compare greater than), `add` (add of course), `st1` (store byte), and `br` (branch).

☑ (5 pts) Convert the DLX code below to IA-64, be sure to use predicated instructions.

☑ Show the stops.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```

```
      cmp.le  p1,p2 = r2, r3;;
 (p1) add r10 = r10, r11
 (p2) sub r12 = r13, r13;;
      add r14 = r12, r10
```

☑ (5 pts) Besides needing fewer instructions, how does predication speed execution? (Not necessarily in IA-64 implementations.) Modify the DLX program (without changing the branch and jump) so that predication would be less useful.

With predication branches can be eliminated, along with their inefficiencies. The inefficiencies include the time needed to fetch the target and the un-executed instructions in the fetch group following a taken branch.

```
 sle r1, r2, r3
 beqz r1, SKIP
 add  r10, r10, r11
 j CONT
SKIP:
 sub r12, r12, r13
CONT:
 add r14, r12, r10
```

74 Spring 2001 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

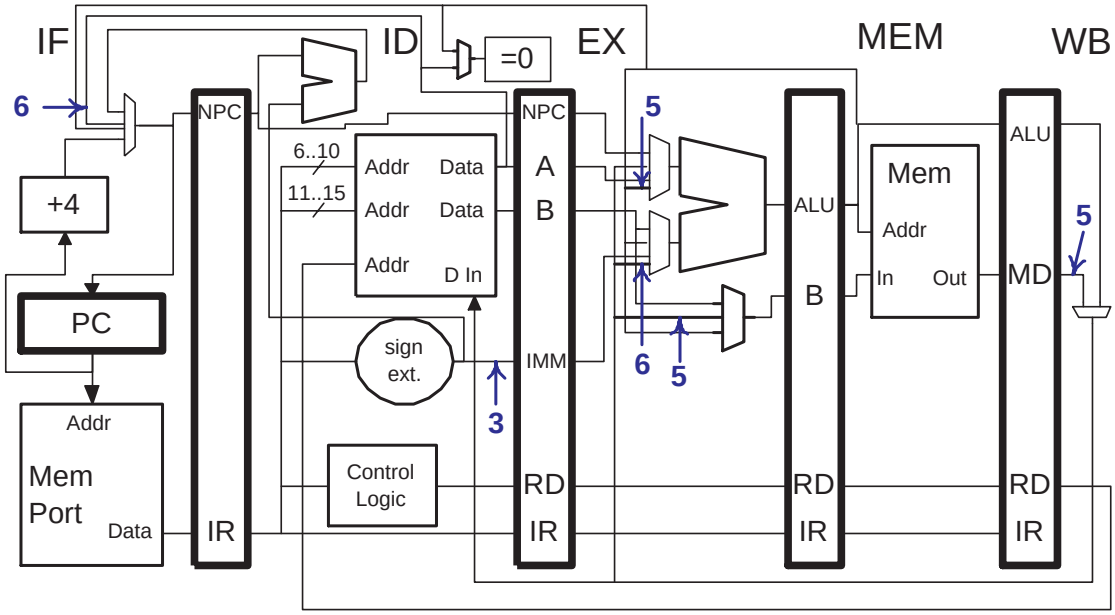
Wednesday, 21 March 2001, 13:40–14:30 CST

Problem 1		(35 pts)
Problem 2		(25 pts)
Problem 3		(40 pts)
Exam Total		(100 pts)

Alias Farewell, Mir!

Good Luck!

Problem 1: In the DLX implementation below six paths are marked with a number, a cycle in which the path will be used. Write a DLX program that uses those six paths at the indicated cycles. Some marked paths are *bypass* paths and some are not; the bypass paths can be used **only** at the cycles indicated. A pipeline execution diagram is provided for your convenience.



☑ [15 pts] Choose the register operands so the bypass paths are used **only** at the cycles indicated.

☑ [15 pts] Choose the instructions so that the marked paths will be used as indicated. There need be only one control transfer instruction.

☑ [5 pts] One instruction will be squashed. Show where by crossing out the segment labels (`ID`, etc.) in the diagram.

Solution:

```
    ! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
nop             IF  ID  EX  MEM WB

lw   r1,0(r20)      IF  ID  EX  MEM WB

addi r2, r21, #1        IF  ID  EX  MEM WB

sw   0(r2), r1              IF  ID  EX  MEM WB

add  r22, r23, r2               IF  ID  EX  MEM WB

jr   r24                            IF  ID  EX  MEM WB

nop                                     IFx

nop                                             IF  ID  EX  MEM WB
    ! Cycle     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Problem 2: The DLX code below runs on a dynamically scheduled system that uses reorder buffer entries to name destination registers (Method 1). The floating-point adder has **five stages**, `A1` through `A5`, and is fully pipelined. The common data bus (CDB) can handle an unlimited number of writebacks per cycle, but other parts of the implementation are ordinary. The cache is non-blocking. The `cvtftoi` instruction uses the FP adder; the `movfptoi` instruction uses the integer (`EX`) functional unit. The pipeline is fully bypassed.

(*a*) For this part no instructions raise exceptions.

☑ [7 pts] Show a pipeline execution diagram for the code. Reorder buffer and reservation station numbers **do not** have to be shown.

☑ [5 pts] Indicate where each instruction commits.

Check the code carefully for dependencies! Register `r1` *is part of a long chain of dependencies. Pay attention to the register equality and inequality comment.*

```
! Solution
! r4 = r2, r6 != r2, r6 != r1
!Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
addf f0, f1, f2  IF ID A1 A2 A3 A4 A5 WC
cvtftoi f0, f0      IF ID RS          A1 A2 A3 A4 A5 WC
movdtoi r1, f0         IF ID RS                      EX WC
subd f0, f4, f6           IF ID A1 A2 A3 A4 A5 WB          C
sd 0(r1), f10                IF ID RS                   L1 L2 WC
sd 0(r2), f0                    IF ID L1       L2 WB             C
ld f12, 0(r4)                      IF ID L1       L2 WB             C
ld f14, 0(r6)                         IF ID L1                L2 WB   C
!Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

Notes on the solution: Load and store instructions compute their effective addresses (use the `L1` stage) as soon as possible. The first `sd` does not move in to the `L1` stage until the rs1 operand, `r1`, is available in cycle 13. The second `sd` moves in to `L1` in cycle 7 but waits for the store value before moving to `L2`. The first load uses the `L2` stage in cycle 11 to bypass a result from the second `sd` (since their effective addresses are the same). The second `ld` must wait for the address of the first store to be computed. Without that address there is no way to tell whether the first store writes to the same location that the second load reads from.

(*b*) Suppose an arithmetic exception is discovered for the `subd` instruction when it is in the first FP adder stage, `A1`, in an execution of the program above.

☑ [6 pts] At what cycle is the reorder buffer flushed and the handler fetched?

When `subd` reaches the head of the reorder buffer, at cycle 14.

☑ [7 pts] Show the contents of the reorder buffer when the status of the `subd` instruction is set to exception. (For partial credit show the contents of the reorder buffer halfway through execution, be sure to indicate the cycle.)

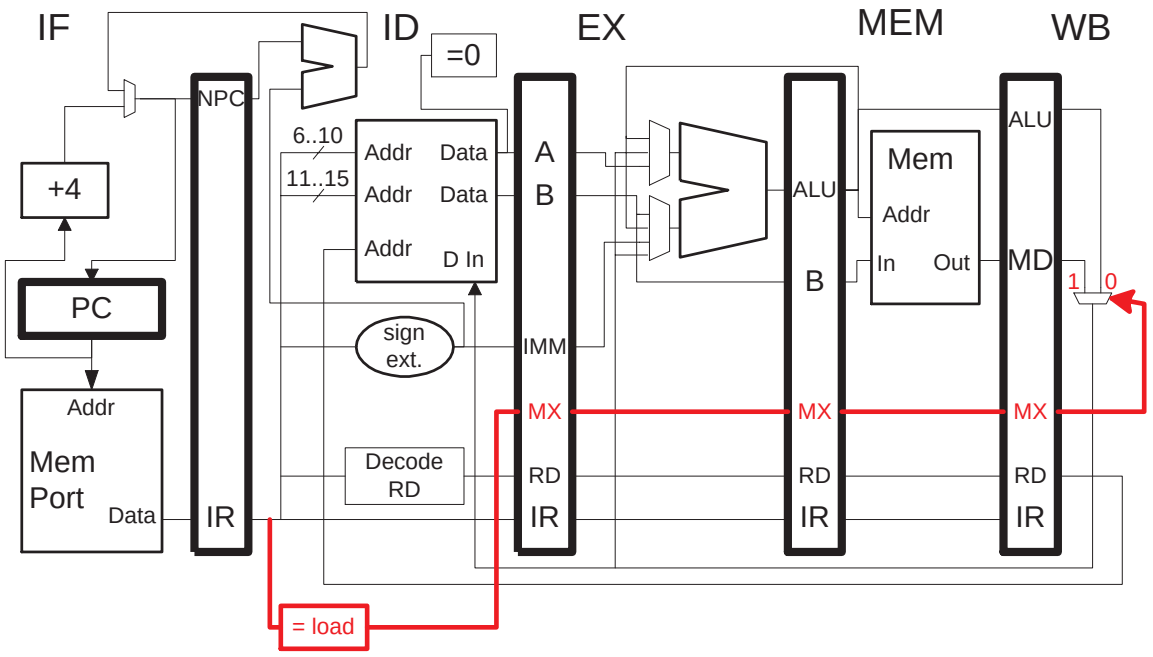
The status is set to exception when `subd` is in writeback, at cycle 10. The contents of the reorder buffer from head (next to commit) to tail (last to enter) is: `cvtftoi`, `movdtoi`, `subd`, `sd`, `sd`, `ld`, `ld`.

Problem 3: Answer each question below.

(*a*)

☑ [8 pts] Design the control logic for the WB-stage multiplexor. The control logic itself must be in the ID stage. Show the connection to the mux and number the mux inputs. *Hint: This is an easy problem.*

Changes shown in red.



(*b*) In the pipeline execution diagram below `multd` is delayed because of a true dependency with the `addd` instruction.

☑ [8 pts] Why can't any realistic implementation do things this way?

```
addd  f0, f2, f4   IF ID A1 A2 A3 A4 WB
multd f6, f0, f8               IF ID M1 M2 M3 M4 M5 M6 WB
subd  f10, f12, f14               IF ID A1 A2 A3 A4 WB
```

Because the true dependency could not be detected before the instruction is fetched and decoded.

(*c*) Unlike DLX, many ISAs, such as SPARC, use a condition code register for integer branch conditions.

☑ [4 pts] In what way is DLX's method more flexible?

Since any register can be used for a condition the code can test one condition, another condition, and then the first one without recomputing it.

☑ [4 pts] In what way is a condition code register more flexible?

Condition code registers typically have flags indicating that a value is negative, zero, overflowed, etc, and these can be set by an instruction doing normal computation. A branch instruction can test various combinations of these conditions. After executing a single set condition-code instruction, several different tests can be made on the conditions, such as ≥ 0, overflow, etc.

(*d*) An ISA may have variable-width instructions, fixed-width instructions, and bundled instructions.

☑ [4 pts] How do the different alternatives affect displacement branches?

If the instructions are variable width then the displacement in displacement branches must be the number of characters away the branch target is. If the instructions are fixed width the displacement is the number of instructions to skip, which spans a larger area of memory (assuming instructions are aligned). With bundled instructions the displacement is the number of bundles to skip, and so an even larger area of memory is covered.

[4 pts] Name an ISA category (type) that uses each instruction format:

☑ Variable-Width Instructions:

CISC

☑ Fixed-Width Instructions:

RISC

☑ Bundled Instructions:

VLIW

(*e*) Packed-operand data types and instructions are *de rigueur* for any *au courant fin-de-siècle* ISA. (Are a must-have for any up-to-date end-of-the-century ISA.)

☑ [8 pts] What are packed-operand data types and instructions? Show a short program that would benefit from these. The program can be in a high-level language or even pseudo code. Do not show the packed-operand instructions, just explain what they would do.

A data type in which several values are held in a single register, for example four eight-bit values in a 32-bit register. A packed-operand instruction operates on all of the values in parallel. For example, a single `add` would, using the last example, sum four pairs of eight-bit numbers.

```
extern unsigned char *a, *b, *c;
for(i=0; i<1024; i++) a[i] = b[i] + c[i];
```

Name Solution

Computer Architecture
EE 4720
Final Examination

11 May 2001, 15:00–17:00 CDT

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(25 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias Solution!!!

Good Luck!

Problem 1: The *reference count* of a value stored in a register is the number of times the value is referenced (read) by instructions before being overwritten. In the example below the reference count of the value written by the `add` is zero because it is overwritten before being read. The reference count of the value written by the `lw` is two.

```
add r1, r2, r3   ! A first value for r1 is defined (written).
lw  r1, 0(r4)    ! A second value for r1 is written, the first one is never used.
sub r5, r5, r1   ! The second value is referenced (read).
xor r6, r1, r7   ! The second one is referenced again.
or  r1, r0, r0   ! A third value for r1 is defined.
```

Three new instructions are to be added to DLX to obtain reference count information. The instructions refer to two registers, `rc.z` and `rc.nz`. Register `rc.z` holds the number of values written to `r1` with a zero reference count and `rc.nz` holds the number of values written to `r1` with a non-zero reference count. The counts are updated on either the first reference or when a new value is defined, whichever is earlier.

Instruction `rc.reset` sets both of these registers to zero. Instruction `movstoi RD, rc.z` moves the contents of `rc.z` to a general-purpose register (shown as `RD`), `movstoi RD, rc.nz` moves the contents of `rc.nz` to a general-purpose register.

The instructions are used in the code below.

```
rc.reset           ! rc.z -> 0,  rc.nz -> 0
add r1, r2, r3
lw  r1, 0(r4)      ! rc.z -> 1
sub r5, r5, r1     ! rc.nz -> 1
xor r6, r1, r7
and r1, r0, r0
or  r1, r0, r0     ! rc.z -> 2
slli r1, r1, #1    ! rc.nz -> 2
movstoi r8, rc.z   ! r8 -> 2
movstoi r9, rc.nz  ! r9 -> 2
add r10, r8, r9    ! Total number of writes to r1.
```

The new instructions should work for `r1` and no other register. (25 pts)

(*a*) For each new instruction below cross out the instruction type that could not reasonably be used to code it. Circle the type that would best be used to code it.

☑ `rc.reset` Possible types: [Type R], Type I, Type J.

☑ `movstoi RD,rc.z` Possible types: [Type R], Type I, ~~Type J~~.

(*b*) As the alert test taker has noticed, there is already a `movstoi` instruction, used to move a special-purpose register (such as the processor status word, not covered much in class) to a general-purpose register.

☑ The new `movstoi RD,rc.z` instruction can be coded as a new instruction (with its own opcode) or as a new use of `movstoi`. That would depend on how the existing `movstoi` is defined. Explain that. *Hint: Suppose there was only one special register in DLX.*

If the `rs1` field in `movstoi` in the existing instruction is always set to some specified value, say `1`, then the existing instruction can be used by indicating that `rs1` specifies which special register to move. A might `1` indicate the special purpose register, a `2` might indicate the new `rc.z` register, and a `3` might indicate the new `rc.nz` register.

Problem continued on next page.

(*c*) Modify the pipeline below to implement the new instructions. The code on the previous page should execute properly.

Assume that all Type-R instructions have two source operands and all Type-I instructions have one source operand. Use `is Type R` and `is Type I` to recognize these instruction types.

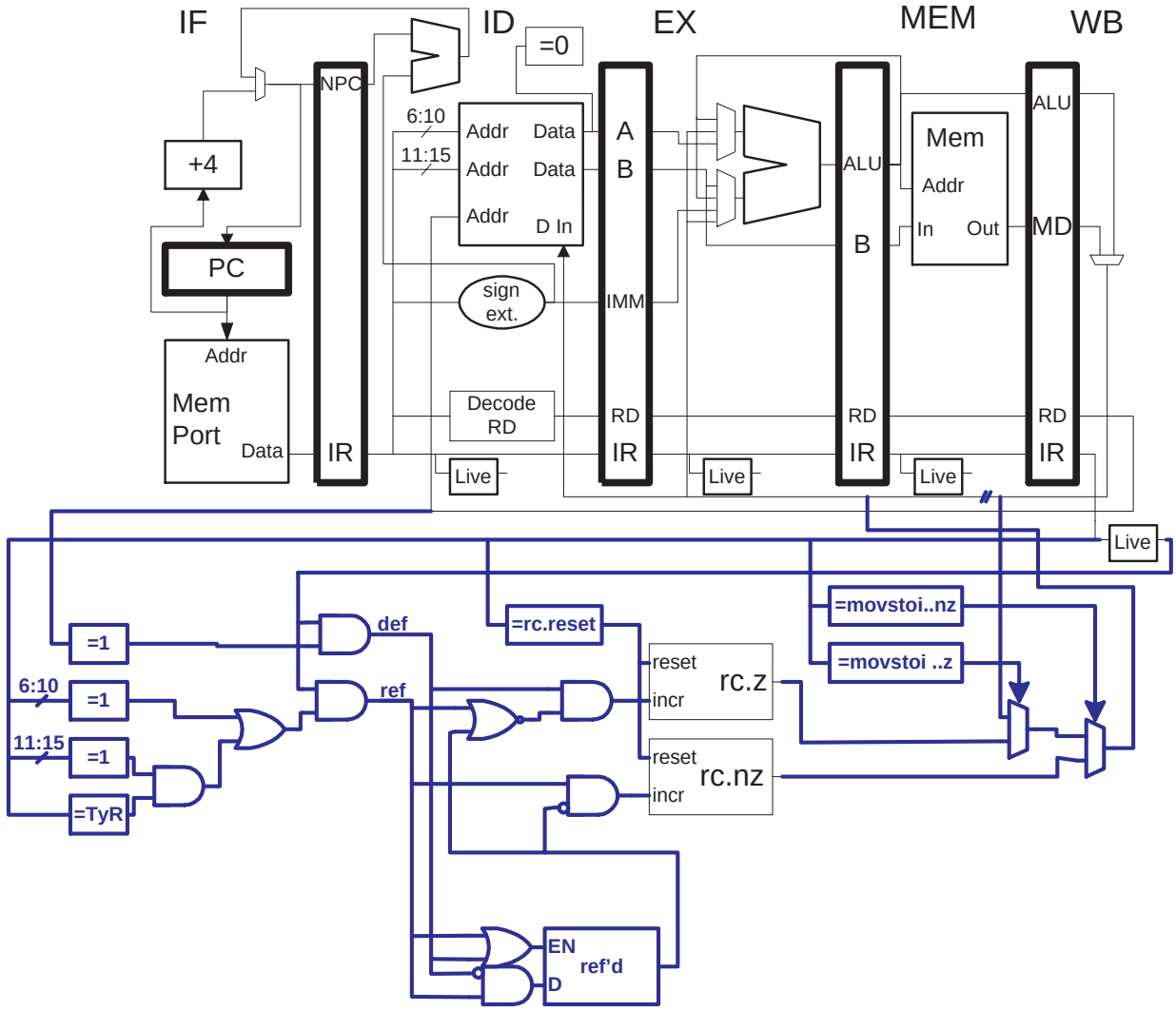
Assume that Decode RD in the diagram can recognize the new instructions

☑ The counts should not include squashed instructions.

☑ Show control logic for all multiplexor and registers that you add. Use symbols like `=rc.reset` to recognize instructions, **show the inputs to these boxes**.

☑ Don't forget the modifications for `movstoi RD, rc.z` and `movstoi RD, rc.nz`.

Changes shown in the diagram below in blue. The "ref'd" box is a D flip flop that indicates whether `r1` has been references. Input "EN" is an enable and "D" is the data input.



Problem 2:

(*a*) The code below executes on a dynamically scheduled single-issue (not superscalar) machine using Method 1, register names are reorder buffer entry numbers. The multiply unit is six stages and is fully pipelined (latency 5, initiation interval 1). The add unit has a latency of 3 and an initiation interval of 2. The CDB can handle an unlimited number of writebacks per cycle. Floating point exceptions are **not** precise. (15 pts)

☑ Show a pipeline execution diagram.

☑ Show where instructions commit.

☑ Show changes to the reorder buffer, register map and register file at each cycle. Register `f0` initially contains zero, `f2` initially contains 20.0; `f4`, 40.0; etc. Use line numbers for reorder buffer entry numbers.

```
L1:multd  f0, f2, f4

L2:addd   f0, f0, f6

L3:addd   f2, f2, f8

L4:addd   f10, f12, f14
```

```
! Solution
 multd  f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WC
 addd   f0, f0, f6     IF ID RS             A1 A1 A2 A2 WC
 addd   f2, f2, f8        IF ID A1 A1 A2 A2 WB             C
 addd   f10, f12, f14        IF ID RS A1 A1 A2 A2 WB          C

 ID Map
 f0     0              L1 L2                              860
 f2     20.0                 L3             100.
 f10    40.0                    L4                 260

 Commit File
 f0     0                                     100.        860
 f2     20.0                                                 100.
 f10    40.0                                                    260.

 ROB                           L4 L4 L4 L4 L4
                             L3 L3 L3 L3 L3 L3 L4 L4 L4 L4
                          L2 L2 L2 L2 L2 L2 L2 L3 L3 L3 L3 L4
 ROB Head              L1 L1 L1 L1 L1 L1 L1 L1 L2 L2 L2 L2 L3 L4
```

☑ The pipeline execution diagram above is the same whether or not exceptions are precise. Why?

Because when a ROB is used for exception recovery there is no need to delay the write back or otherwise change the way long-latency instructions execute.

(*b*) The code below executes on a two-way superscalar statically scheduled DLX implementation with perfect branch and jump target prediction. (10 pts)

☑ Show a pipeline execution diagram for the worst-case execution of the code below for two iterations. (The worst case is achieved by proper choice of the labels, `LOOP` and `THERE`.)

☑ Explain why it's worst case.

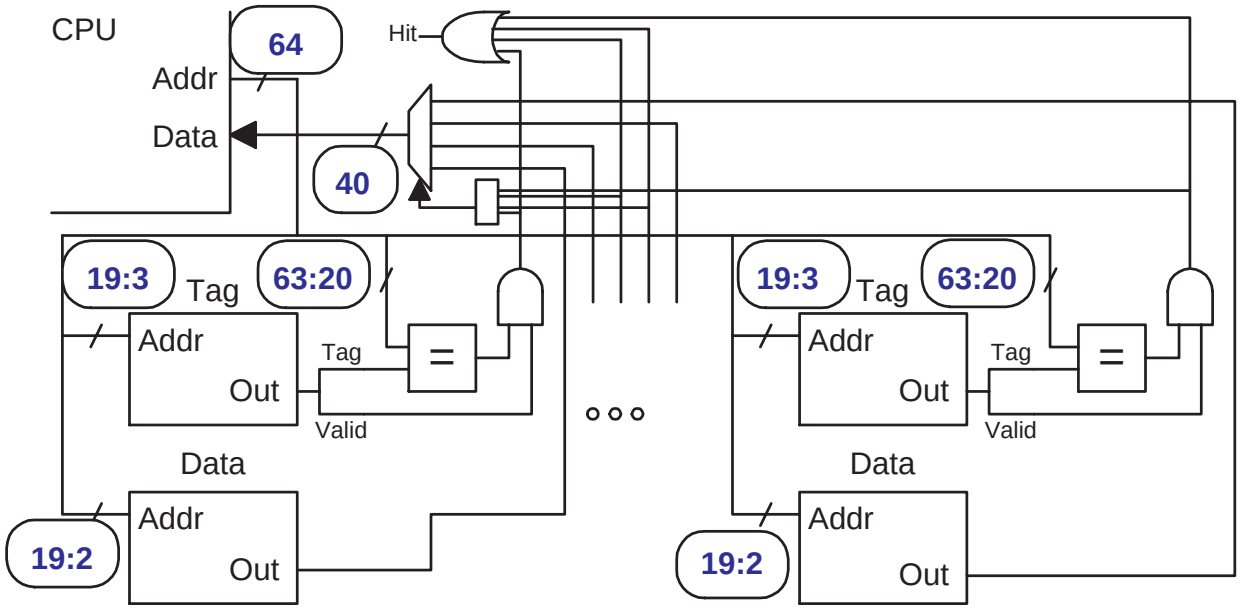
For `LOOP = 0x1004`, the `subi` is the second instruction in an aligned group, so after the first iteration the `addi` and `bnez` will be fetched uselessly. Similar reasoning for `THERE = 0x2004`.

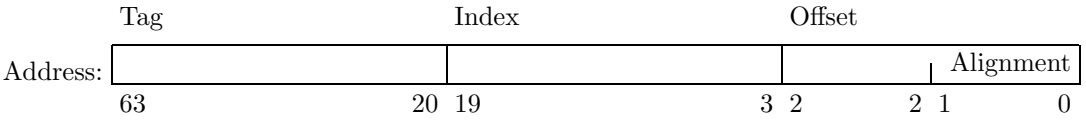
☑ What is the CPI of the for a large number of iterations?

Solution not yet available.

```
 ! Solution not yet available.

 addi r1, r0, #2000
LOOP:
 subi r1, r1, #1
 j THERE
HERE:
 bnez r1, LOOP

THERE:
 add r2, r2, r3
 j HERE
```

Problem 3: The cache for a system implementing an ISA with 10-bit characters is described by the following incomplete schematic diagram and address bit categorization.





(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☑ Fill in the blanks in the diagram.

Changes shown in blue.

☑ In the diagram above the processor's data out port is not shown. Given what is shown is the cache probably write back or write through? Argue your answer in terms of what is missing or what is present.

Write through because there's no dirty bit. In a write through cache whenever data is written to the cache it is also written to memory. In a write back cache data written to a line in the cache is not written to memory until the line is replaced. A line that has not been modified does not have to be written to memory, the dirty bit is used to distinguish them. It is set to zero when a line is loaded into the cache and it is set to one when the line is written.

☑ What is the associativity of this cache? (Look carefully.)

Four, there are four inputs to the hit or gate.

☑ What is the capacity of the cache? Specify units!

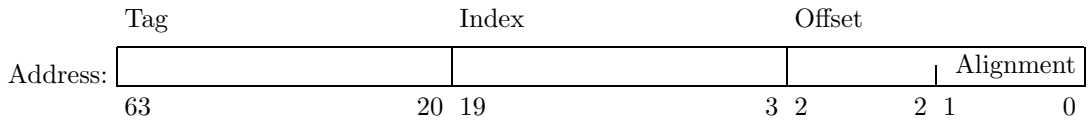
The capacity is $4 \times 10 \times 2^{20}$ bits. The formula above would get full credit, no need to write it in another form such as forty-one million, nine hundred forty-three thousand, forty.

☑ How much memory does it take to implement the cache? Specify units!

The storage used for the tag store is $4 \times 44 + 1) \times 2^{17}$ bits, the total amount of memory is $4 \times 44 + 1) \times 2^{17} + 4 \times 10 \times 2^{20}$ bits.

Problem 3, continued: Continue to use the cache from the previous part.

(*b*) When the program below starts execution no data is cached. Consider only memory accesses needed for the array access. The address bits are repeated for convenience. (15 pts)




```
extern char *a;    // & a[ 0 ] = 0x1000000;
int i, j, k, t;
int ISTRIDE = 1024;
for(k=0; k<2; k++) for(i=0; i<256; i++) for(j=0; j<32; j++)
  t += a[ ISTRIDE * i + j ];
```

☑ What is the hit ratio?

The hit ratio is $\frac{15}{16}$.

Each line holds eight characters. In the inner loop the array is accessed sequentially. When `k=0` it is being accessed for the first time, and so the hit ratio there will be $\frac{7}{8}$. When `k=1` it is being accessed the second time. Since the entire array fits in the cache the hit ratio is $\frac{1}{2}\left(\frac{7}{8}+1\right)=\frac{15}{16}$.

☑ What is the smallest line size that would maximize the hit ratio?

The smallest line size would be 2^{18} characters, which is unreasonably large.

The hit ratio is maximized when the entire array fits on one line. It fits on one line if the tag and index parts of every address are the same. Loop variable `j` "modifies" bits 4:0 of the address. (The address is `a + ISTRIDE * i + j`, where `a` is the address of the first element of the array `a`.) Loop variable `i` modifies bits 17:10. The other bits of the address are the same for every array element. Neither `i` nor `j` modify tag bits, and so the entire array can be cached. The highest bit number modified is bit 17, and so bits 19:18 of the index are the same for every element. If the line size were increased to 2^{18} characters without changing the cache size the index bits would be 19:18, and so the entire array would fit on one line.

☑ What is the smallest *reasonable* line size that would maximize the hit ratio?

2^5 characters.

Increasing it to 2^6 would not improve hit ratios; improvements do not occur until it's increased to 2^{11} which is moving in to unreasonable territory.

☑ What is the smallest power-of-two value of ISTRIDE for which a two-way set-associative cache is necessary to avoid conflict misses?

`ISTRIDE` $= 2^{13}$.

A two-way set-associative cache is necessary when two elements have the same index but different tags. That will happen when exactly one of the bits affected by `i` is part of the tag. For that to happen, `ISTRIDE` $= 2^{13}$. With that value of ISTRIDE 256 values of `i` yields two different tag values and 128 different indices (ignoring `j`). The elements for `i`=0 have a tag of zero and those for `i`=128 have a tag of 1. For `j`=0 both have an index of 0. As long as it's a power of two `ISTRIDE` shifts `i` to the left.

☑ If ISTRIDE were larger than the answer to the previous question but not a power of two, would there be lots of conflict misses? Explain.

There would be few conflict misses because potentially all parts of the address would be affected by `i`, so for 256 different values of `i` there would be 256 distinct index values.

For example, suppose ISTRIDE $= 2^{13} + 2^2$. Now `i` affects bits 20:13 and bits 9:2. There will be a distinct index or offset for each value of `i`.

Problem 4: Answer each question below.

(*a*) One-level predictors are usually outperformed by two-level predictors. (5 pts)

☑ Write a code fragment containing a branch that would be predicted well by a two-level predictor such as gshare but poorly by a one-level predictor.

```
! Solution
 lw r1, 0(r2)  ! r1 hard to predict
 bnez r1, SKIP ! Not predicted well by any predictor.

SKIP:
 beqz r1, SKIP2 ! Since it's correlated with the last branch
                ! a two-level predictor that uses global history will do well.
```

(*b*) Virtual memory allows two processes on the same processor to use the same address as though they were on different systems, that is, one process never loads what the other stores. (5 pts)

☑ Why doesn't one process loading from, say, address `0x1000` get data previously stored by another process at address `0x1000`?

Because the real page numbers in their addresses are different, and so different locations are being accessed.

(*c*) A processor implementing a 64-bit ISA has the following integer add latencies: 8- and 16-bit integers, 0 cycles; 32-bit integers, 1 cycle; 64-bit integers, 2 cycles; 128-bit integers, 3 cycles. (5 pts)

☑ Why are these latencies not appropriate for a 64-bit ISA? In what important way would the processor suffer?

Because 64-bit integer operations are needed to compute memory addresses. Since these are done frequently the processor will suffer.

(*d*) Precise exceptions are optional for floating-point instructions but required for integer instructions. (5 pts)

☑ Why is this so?

Because load and store instructions encounter exceptions in normal use (in virtual memory systems) and they must be re-started after exceptions as though nothing happened.

☑ Illustrate your answer with an example showing an integer instruction that raises an exception.

```
! Illustration for Answer:
! Cycle:            0  1  2  3  4  5  6  7
lw   r1,0(r2)       IF ID EX *M WB          !  <-  Faulting instruction.
add  r30, r29, r28     IF ID EX ME WB
addi r2, r1, #4           IF ID EX ME WB     !  <-  Last before handler.
and  r27, r26, r25           IF ID EX ME WB  !  <-  Resume here.
```

☑ Explain what goes wrong if the exception is not precise.

In the example above `lw` raises an exception but `addi` is the last instruction, it executes with a wrong value in `r1`. Execution resumes at the `and` and so the correct value is never put in `r1`.

(*e*) What are stops and lookaheads? (5 pts)

☑ How are they used?

Stops are placed between instructions (in assembler source) to indicate that there is a true dependency between an instruction before the stop and an instruction after the stop. The assembler places this information in an instruction bundle's template.

The lookahead of a bundle, call it x, is the number of following bundles that can be executed before finding a bundle that has a true dependency with bundle x.

☑ What benefit to they have to implementations?

They lower the cost and increase the speed of implementations by eliminating the need for some dependency checking hardware.

☑ Explain the similarities and differences between stops and lookaheads.

They both provide dependency information. Lookaheads only indicate dependencies between bundles, while stops can specify intra-bundle dependencies.

75 Fall 2000 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination, Part I

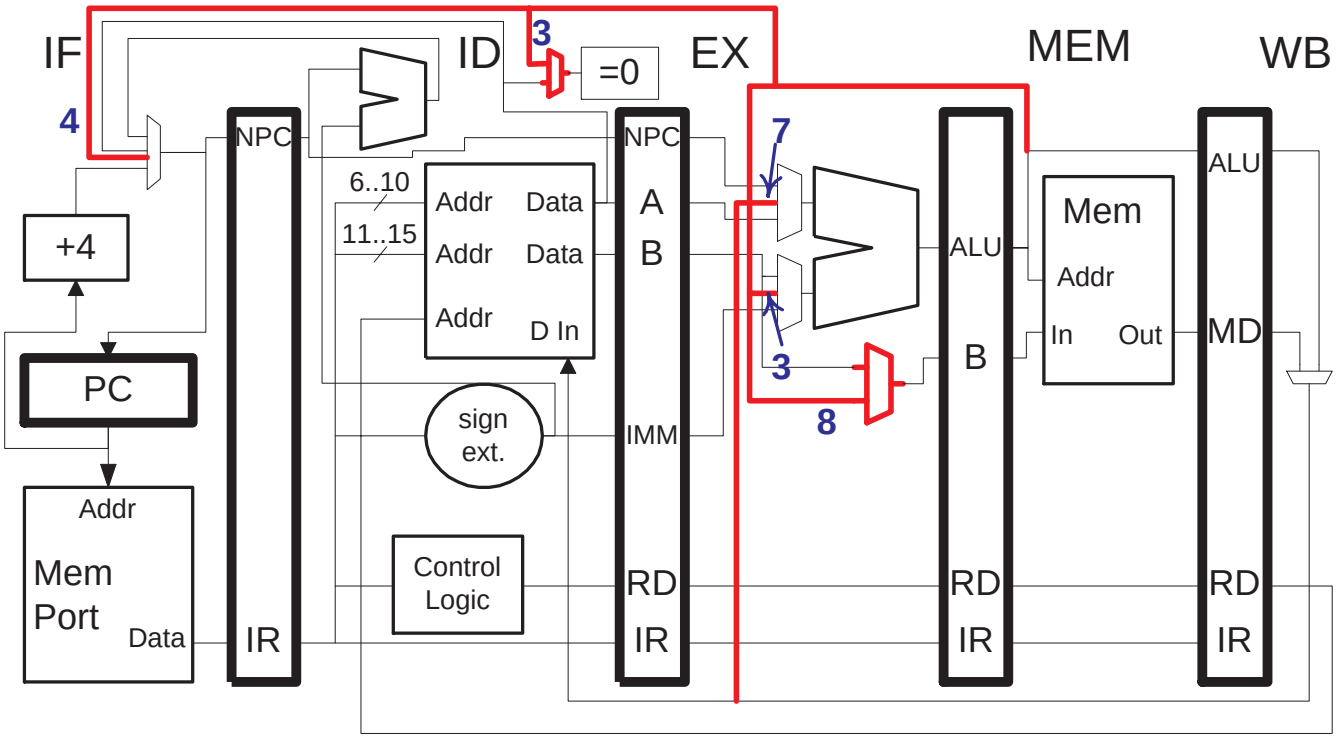
Monday, 16 October 2000, 12:40–13:30 CDT

Problem 1		(17 pts) Mon.
Problem 2		(17 pts) Mon.
Problem 3		(16 pts) Mon.
Problem 4		(13 pts) Wed.
Problem 5		(17 pts) Wed.
Problem 6		(20 pts) Wed.
Exam Total		(100 pts)

Alias Lets go Mets!!

Good Luck!

Problem 1: The program below executes on the pipeline below as illustrated in the pipeline execution diagram below. Bypass paths do not appear in the illustration (below).




```
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11
andi r8, r8, #31  IF ID EX ME WB
add  r10, r9, r8     IF ID EX ME WB
bnez r8, LINEX          IF ID EX ME WB
jalr r10                   IF ID EX ME WB
xor                           IFx
...
subi r31, r31, #8                IF ID EX ME WB
sw   0(r10), r31                    IF ID EX ME WB
! Cycle           0  1  2  3  4  5  6  7  8  9  10 11
```

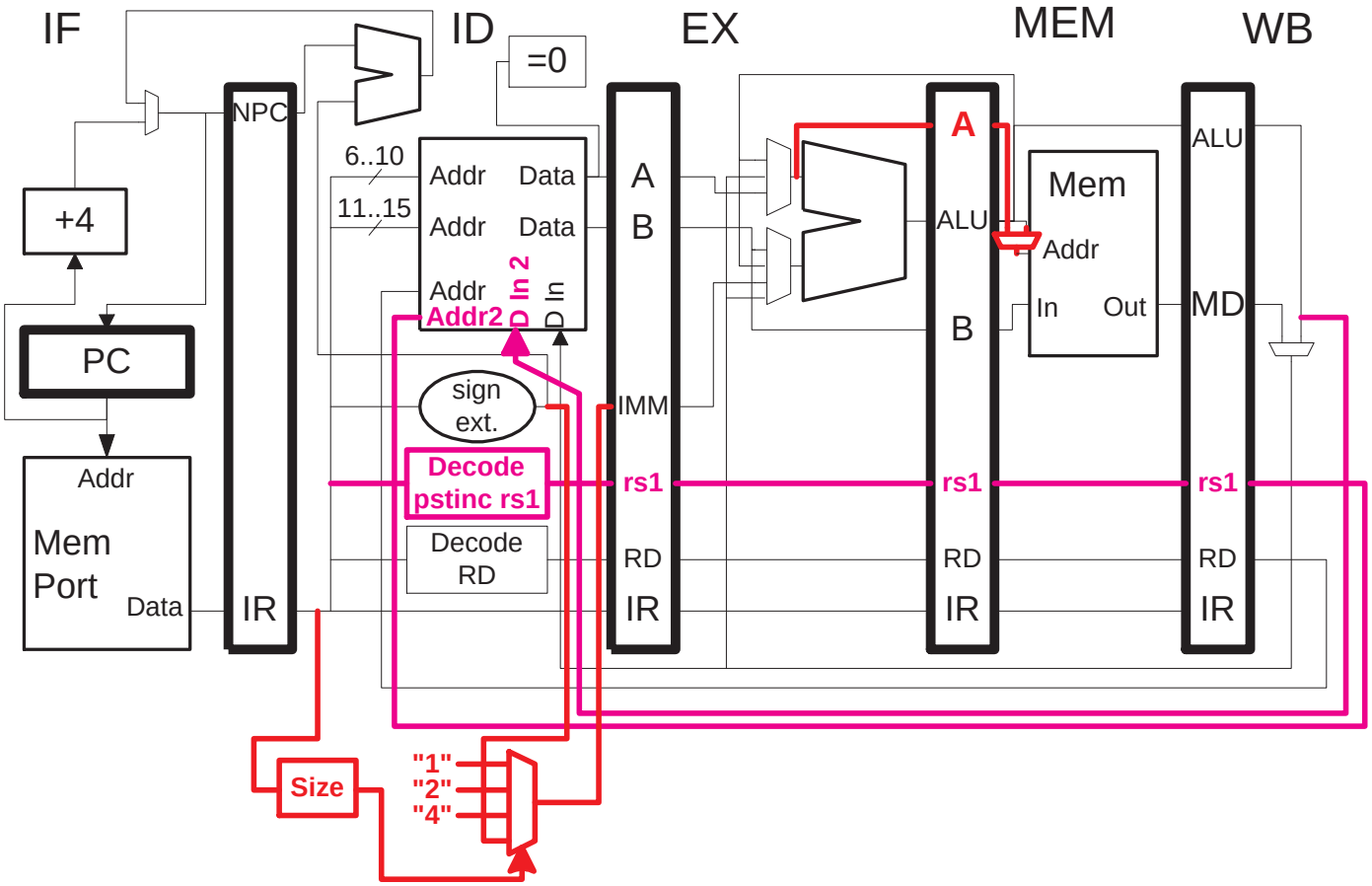
☑ [10 pts] Add *exactly* those bypass paths that are needed so that the code (above) executes as shown. Credit will be deducted for unneeded bypasses. *Please, please, please check the code carefully for dependencies.*

The bypass paths are shown in **red bold**.

☑ [7 pts] Next to each bypass path indicate the cycle(s) in which it will be used.

The cycle numbers are shown in blue.

Problem 2: As described in class, postincrement instruction `lw r1,(r2+)` loads the value at memory address `r2` into register `r1` and stores `r2+4` in `r2`. Postincrement stores are similar. The pipeline below is to be modified so that it can execute postincrement loads and stores for bytes, half words, and words. A logic block size can be used; its input is the `opcode` and `func` fields; the output is 0 for a postincrement with a byte-size load or store value, 1 for a postincrement with a half-word value, 2 for a postincrement with a word-size value, and 3 if the instruction is not a postincrement load or store.



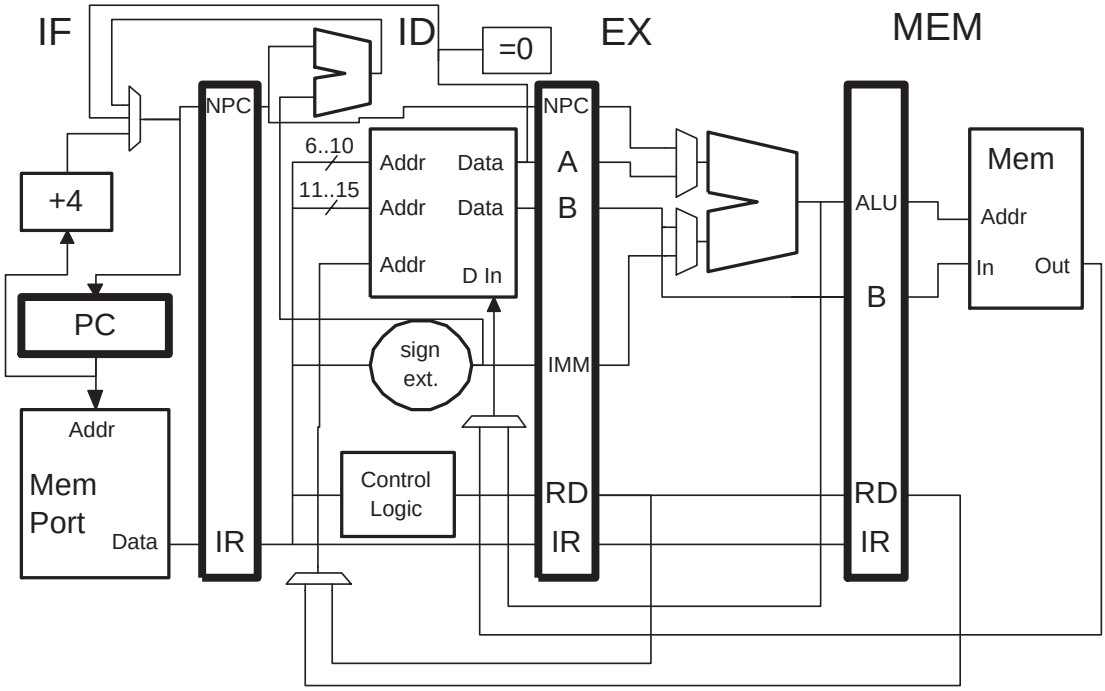
☑ [10 pts] Show the datapath changes needed so that the pipeline can execute postincrement store instructions. Include the control logic for obtaining the amount to increment the address by but do not include any other control logic.

These changes are shown in **red bold**.

☑ [7 pts] Show the additional datapath changes needed so that the pipeline can execute postincrement loads. These changes must not add structural hazards and the load must execute without stalling the pipeline (assuming favorable dependencies).

These changes are shown in **purple bold**. The changes include a second register file write port so the loaded value and incremented address can be stored at the same time. Also shown is a new control field, `rs1`, which is the `rs1` register number used in the postincrement instruction, or zero if any other instruction is used. The `rs1` was not required for the solution, but is included anyway.

Problem 3: The four-stage (no WB) pipeline below includes an *Express Writeback* feature, eliminating the need for bypass connections. Instructions proceed through the pipeline slightly differently than the DLX pipeline presented in class. **Do not** add or assume the presence of bypass connections.



☑ [6 pts] Show a pipeline execution diagram for the code below on this pipeline. (Don't forget to look for dependencies.) State any assumptions about how the register file operates.

```
add r1, r2, r3  IF ID EX ME
lw  r4, 8(r1)      IF ID EX ME
sw  12(r5), r4        IF ID -> EX ME
```

One disadvantage of *Express Writeback* is that it introduces a structural hazard.

☑ [6 pts] Show a program that encounters the hazard and a pipeline execution diagram showing how the hazard can be avoided by stalling. (*Hint: the program can be just two instructions.*)

```
lw  r1, 0(r2)   IF ID EX ME
add r3, r4, 45     IF ID -> EX
```

☑ [4 pts] Explain how *Express Writeback* affects the critical path.

The register write occurs in the same cycle as the ALU operation or memory operation, and so the critical path is longer.

Part II on Wednesday at 12:40 precisely.

Name Solution

Computer Architecture
EE 4720
Midterm Examination, Part II

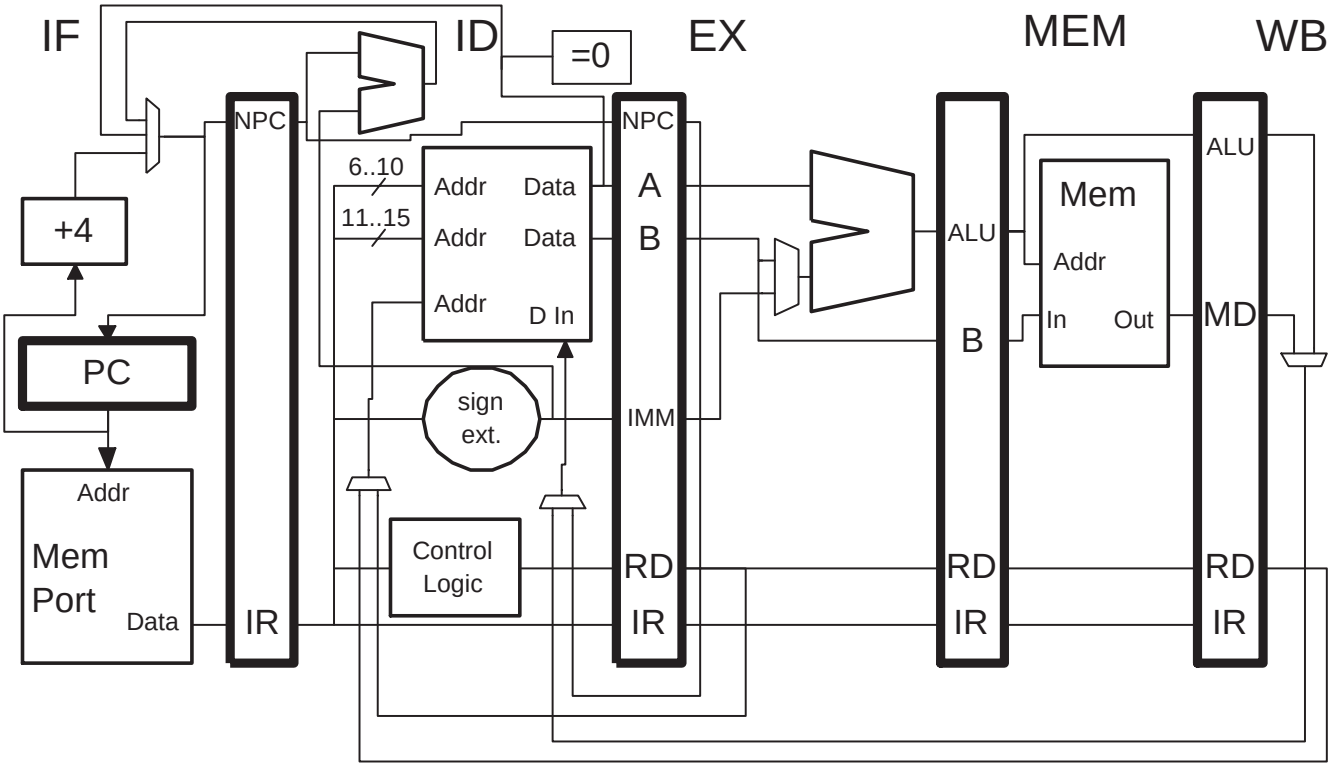
Wednesday, 18 October 2000, 12:40–13:30 CDT

Problem 1	(17 pts) Mon.
Problem 2	(17 pts) Mon.
Problem 3	(16 pts) Mon.
Problem 4	(13 pts) Wed.
Problem 5	(17 pts) Wed.
Problem 6	(20 pts) Wed.
Exam Total	(100 pts)

Alias _____

Good Luck!

Problem 4: The diagram below includes naïve hardware for implementing the `jal` and `jalr` instructions. With this hardware precise exceptions are impossible.



☑ [6 pts] Explain why precise exceptions are impossible here.

Because the return address is written before it is certain that a preceding instruction will not raise an exception. If a preceding instruction does raise an exception there is no way to restore the registers to the state they were in before the faulting instruction was executed.

☑ [7 pts] Show a program including a `jalr` that encounters an exception and which does not run correctly (after the exception handler returns) because of the way `jalr` is implemented. Briefly explain why it does not run correctly. (For partial credit answer the question using an instruction other than `jalr`.)

```
sw 0(r1), r31   IF ID EX*ME*WB
jalr r4           IF ID EXx
```

The `sw` encounters a page fault. When it is re-executed `r31` will have the value stored by `jalr` rather than its previous value.

Problem 5: For some, 16 bits is just not enough. Consider a new DLX instruction, `mbi` (move big immediate), which moves a large immediate into register `r1`. (Register `r1` is always used, a different register cannot be specified.) The following code uses the new instruction:

```
mbi  0x12345     ! Move 0x12345 into register r1.
add  r2, r1, r2  ! Use 0x12345 in an add.
```

☑ [5 pts] Describe how the instruction could be coded using an *existing* DLX instruction type to get the biggest immediate possible. Specify the size of the immediate.

The instruction type with the largest immediate is type J, so code it as a type-J instruction. The immediate would be 26 bits.

Ignoring the previous part, suppose one wanted the immediate to be 30 bits.

☑ [4 pts] Why are 30-bit immediates impossible using an existing instruction type?

Because no existing type has a 30-bit immediate field, or any 30-bit field for that matter.

☑ [4 pts] Describe how a 30-bit immediate `mbi` could be coded using a *new* DLX instruction type. (See the next subpart before answering.)

Have a new type, call it type B, with a two-bit opcode and a 30-bit immediate.

Whether it is possible to add a 30-bit immediate instruction and maintain compatibility depends on certain details of DLX which during lectures were usually made up on the spot.

☑ [4 pts] What kind of details were those? Make them up so that the new instruction is compatible.

Opcodes. The `mbi` opcode will be 11_2, and so no other instruction can have an opcode starting with two 1's.

Problem 6: Answer each question below.

(*a*) Loads and stores in DLX are aligned.

☑ [6 pts] What does that mean? Provide examples of aligned and unaligned accesses.

That means the address of the item being loaded or stored must be a multiple of its size. For words, the address must be a multiple of 4 and for shorts a multiple of 2. Aligned: `lw r1,0x4(r0)`. Unaligned: `lw r1,0x2(r0)`.

(*b*) As described in class, an ISA implementing a time data type might have instructions to determine the number of days between two times. (Say between 18 October 2000 and 15 December 2000.)

☑ [7 pts] Give two reasons why an ISA should *not* include such a time data type.

It wouldn't be used very often and it would'nt be much faster.

(*c*) RISC and CISC are sometimes seen as two competing architectural styles.

☑ [7 pts] Name two features that distinguish RISC processors from CISC processors.

Fixed size instructions (RISC). ALU instructions that can access memory (CISC).

Name Solution

Computer Architecture
EE 4720
Final Examination

8 December 2000, 12:30–14:30 CST

Modified

Problem 1		(20 pts)
Problem 2		(14 pts)
Problem 3		(20 pts)
Problem 4		(17 pts)
Problem 5		(29 pts)
Exam Total		(100 pts)

Alias Solution!!!

Good Luck!

Problem 1: New DLX instruction `jalr.safe` is like a `jalr` instruction except that it automatically returns after a certain number of instructions in the called routine have been executed. Suppose `r1` contains `0x1000` and `r2` contains 23 when `jalr.safe r1, r2` is executed. Execution would jump to address `0x1000` and `PC+4` would be saved in `r31`. Nothing special happens if a `jr r31` (a return) is executed within 23 instructions. If after 23 instructions a return is not executed control will automatically return to the instruction following `jalr.safe`. When `jalr.safe` is used the called procedure cannot call another procedure.

Note: The `jalr.safe` instruction might be used in time-critical systems to prevent a procedure from taking too long (though it would probably be better to base the automatic return on time rather than an instruction count). A procedure called using `jalr.safe` might compute a rough estimate of a return value first, then start working on an accurate return value. If it returned automatically the rough value would be used.

The instruction would be "safe" if used properly but if used improperly would be very dangerous. Any procedure called with `jalr.safe` must be written so that it could return any time. Otherwise, data structures might be left half-updated, code accessing them later might execute incorrectly.

Real systems using *watchdog timers* rather than instructions like `jalr.safe`. A calling procedure would set a timer (sort of like an alarm clock) to expire after the called procedure was supposed to return. If the called procedure return on time the timer is cancelled. Otherwise the timer expires and a timer-expiration interrupt handler is called. That handler might terminate the overdue procedure.

(*a*)

☑ (5 pts) Show how `jalr.safe` might best be coded.

A good coding is one that is similar to the coding of existing instructions. Since `jalr.safe` is similar to `jalr` it would be best to use the same instruction type as `jalr`, if possible. The `jalr` instruction uses one source register, `jalr.safe` uses two. This can be accommodated in the type I coding used by `jalr` by using the `rd` field as the second source.

(*b*) Modify the pipeline on the next page so that it executes the `jalr.safe` instruction.

- The output of the Live box is 1 if the corresponding stage contains a non-squashed instruction that will advance to the next stage in the next cycle. (The instruction is neither squashed nor stalled.)
- =Ret can be used for detecting return instructions, =jalr.safe for `jalr.safe` instructions, etc.

Continued on next page.

Problem 1, continued:

☑ (7 pts) Show the hardware needed to properly save the automatic return address and count. (For the automatic return address do not use the contents of `r31`, instead add a register for this address.)

☑ (2 pts) Base the return on the number of instructions that will complete, squashed instructions should not be counted.

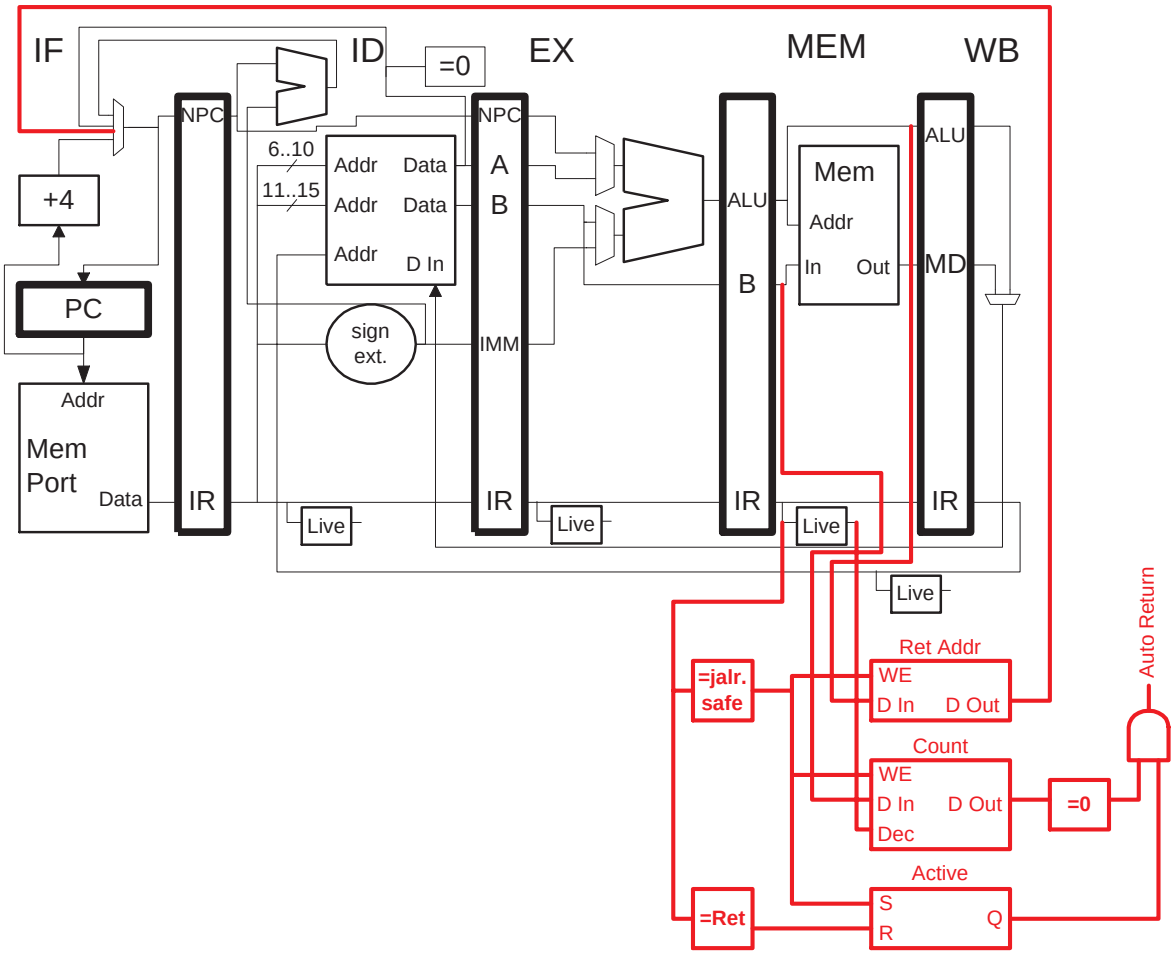
☑ (2 pts) Make sure `jalr.safe` can be squashed before it sets a return address.

☑ (2 pts) Generate a signal named `Auto Return`, it shall be true when there is to be an automatic return and false other times.

☑ (2 pts) Explain what the controller must do when `Auto Return` is asserted.

The modifications appear in red below. When `Auto Return` is asserted instructions in IF, ID, and EX are squashed and new input to the PC mux is selected.

Since the added hardware is in the MEM stage `jarl.safe` can be squashed. (If it were in the ID stage, for example, it might set a return address before being squashed in the EX stage by an instruction in the MEM stage.)



Problem 2: The programs below run on statically and dynamically scheduled systems. All systems are single issue (not superscalar), have perfect branch prediction, have an unlimited number of functional units, and use non-blocking caches. The programs run for a large number of iterations, and the first `lw` in **every** iteration will miss the cache. On a cache miss data arrives 10 cycles after `MEM` or `L2` is entered. The line size is 1024 characters.

```
! Program 1                    ! Program 2
LOOP:                          LOOP:
 lw   r1, 0(r2)                 lw   r1, 0(r2)
 addi r2, r2, #1024             lw   r2, 4(r2)
 add  r3, r3, r1                add  r3, r3, r1
 bneq r1, LOOP                  bneq r1, LOOP
```

(*a*) Suppose the programs were are run on a statically scheduled machine and loop unrolling was being considered. *Note: The following important point was not included in the 2000 final exam.* The statically scheduled machine treats load misses like floating-point operations: it allows them to complete out of order if there are no name or data dependencies with following instructions.

☑ (1 pts) For a statically scheduled system applying loop unrolling to Program 1 would improve performance: (*a*) ☑ by a large amount; (*b*) ☐ by a small amount; (*c*) ☐ not at all; (*d*) ☐ none of the above.

☑ (1 pts) For a statically scheduled system applying loop unrolling to Program 2 would improve performance: (*a*) ☐ by a large amount; (*b*) ☐ by a small amount; (*c*) ☑ not at all; (*d*) ☐ I do not wish to reveal my intent.

☑ (5 pts) Explain the two answers above. In particular explain, if appropriate, why loop unrolling is more effective on one program than the other.

Program 1 can easily be unrolled. Without loop unrolling the processor would have to suffer one miss at a time, so the execution time is at least the number of iterations times the miss delay. In the unrolled loop there can be several misses being serviced in parallel (since the cache is nonblocking), so the execution time is roughly the number of iterations in the unrolled loop times the miss delay. The number of iterations in the unrolled loop is the number of iterations in the original loop divided by the degree of unrolling, so the execution time is a lot lower.

In program 2 the data to fetch on one iteration depends on the data fetched in the previous iteration so it cannot be unrolled.

(*b*) Suppose the programs were to be run as is (not unrolled).

☑ (1 pts) Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 1: (*a*) ☑ much faster; (*b*) ☐ slightly faster; (*c*) ☐ about the same speed; (*d*) ☐ dimple.

☑ (1 pts) Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 2: (*a*) ☐ much faster; (*b*) ☑ slightly faster; (*c*) ☐ about the same speed; (*d*) ☐ I assume that if my answer below is correct points will not be deducted for this choice.

☑ (5 pts) Explain the two answers above. In particular explain, if appropriate, why the dynamically scheduled machine is more effective on one program than the other.

With dynamic scheduling several loads in program 1 can be active at once, and so execution time will be greatly reduced.

There is little dynamic scheduling can do for program 2 for the same reason it could not be unrolled.

Problem 3: The code below executes on a dynamically scheduled system with branch prediction but without branch target prediction. The branch is predicted taken but it ultimately is not taken. The processor is single-issue (not superscalar) but, conveniently, has an unlimited number of functional units and can handle any number of write-backs per cycle. At most one instruction per cycle can be committed.

(*a*) The code below executes on such a machine in which the register map is **not** backed up when branches are decoded. Registers are intentionally omitted from the last three instructions, assume those instructions are not data-dependent on the loads.

☑ (6 pts) Complete the pipeline execution diagram for this system until all instructions complete.

☑ (2 pts) Show instruction commitment and squashing.

- Don't forget to check for dependencies!

```
! Solution:

! Branch predicted taken, but branch is NOT taken.

! Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r3, 0(r4)     IF ID L1 L2 RS RS RS RS RS L2 WC
 lw r1, 0(r2)        IF ID L1 L2 RS L2 WB          C
 bneq r1, TARGET        IF ID RS       B  WB          C
 xor r5, r6, r7            IFx                        IF ID EX WC
 sgt r8, r9, r10                                         IF ID EX WC
 ...

TARGET:
 add r11, r12, r13            IF ID EX WB             x
 sub r14, r15, r16               IF ID EX WB          x
 and r17, r18, r19                  IF ID EX WB       x
```

Problem 3, continued: (*b*) The code below executes on a version of the machine in which the register map is backed up (checkpointed) when branches are decoded.

☑ (6 pts) Complete the pipeline execution diagram for this system until all instructions complete.

- Show instruction commitment and squashing.

```
! Solution:


! Branch predicted taken, but branch is NOT taken.

! Cycle           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r3, 0(r4)     IF ID L1 L2 RS RS RS RS RS L2 WC
 lw r1, 0(r2)        IF ID L1 L2 RS L2 WB          C
 bneq r1, TARGET        IF ID RS       B  WB          C
 xor  r5, r6, r7           IFx            IF ID EX WB    C
 sgt  r8, r9, r10                            IF ID EX WB    C
 ...

TARGET:
 add  r24, r25, r26           IF ID EX WB
 sub  r21, r22, r23              IF ID EX WBx
 and  r9, r20, r30                  IF ID EXx
```

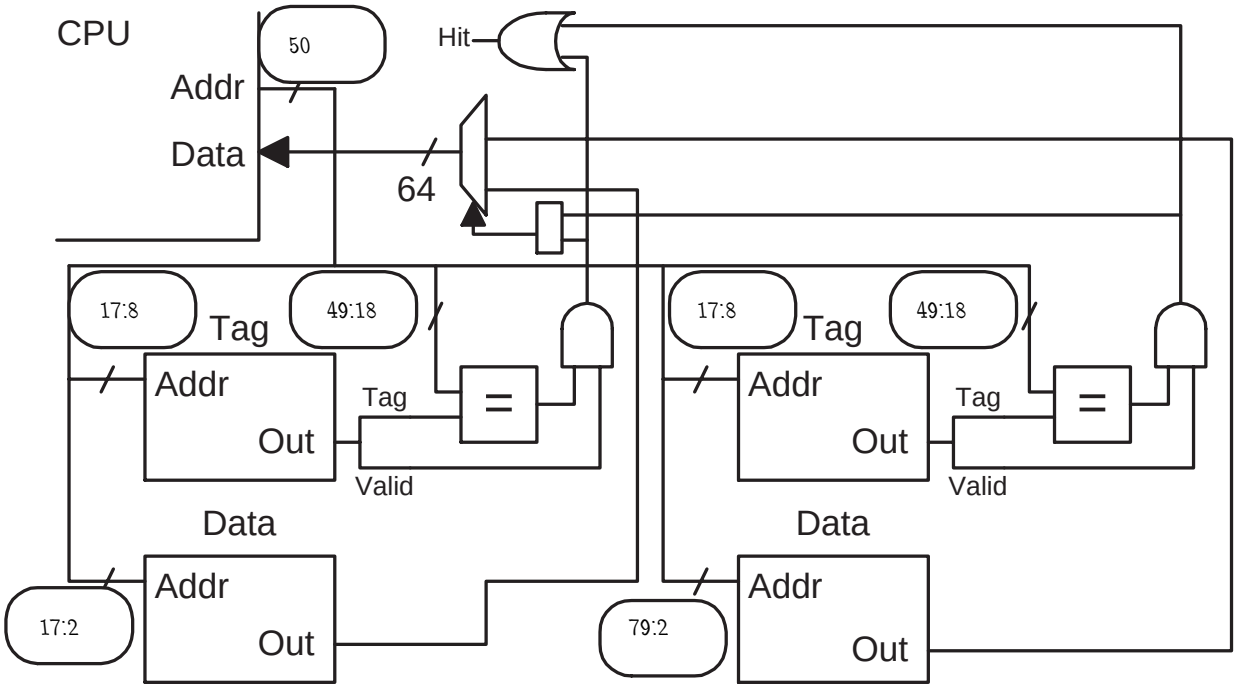
(*c*) Suppose the processor from the previous part has the following defect: it does not back up or restore the register map, but it executes instructions as though it did. Suppose execution up until the code fragment above is okay.

☑ (6 pts) Add registers to the last three instructions in the previous part so that the `xor` instruction executes correctly but the `sgt` (set greater than) executes incorrectly.

The registers have been added. For `sgt` to execute incorrectly the register map entry for one of its source registers must be incorrect. For that to happen an instruction past `TARGET` with destination `r9` or `r10` must be decoded. (It does not have to go any farther than ID, since the new register specified in the register map for `r9` or `r10` will have the wrong value whether or not the instruction completes write back.)

Problem 4: A system has a 1-megabyte (2^{20} byte) two-way set-associative cache with 256-character blocks and a 50-bit address space addressing **16**-bit characters.

☑ (7 pts) Fill in the rounded boxes in the diagram below so that it describes this cache.



☑ (5 pts) Show the smallest set of addresses that cannot all be in this cache at the same time.

Because the associativity is 2 the smallest set of addresses would have three elements. They would have the same index and different tags. For example, {0x543210, 0x643210, 0x743210}.

☑ (5 pts) What would be the associativity of a fully associative cache with the same capacity and block size as this one?

The associativity of a fully associative cache is equal to the number of blocks in the cache. The cache in part (a) has 2^{11} blocks, so a fully associative cache with the same capacity and block size would have an associativity of 2^{11}.

Problem 5: Answer each question below.

(*a*) The DLX program below runs on a system using a one-level branch predictor with a 2^{16}-entry BHT, each BHT entry is a two-bit saturating counter. The loop iterates many times.

Please do not confuse `andi` with `addi`.

```
LOOP:
 addi r1, r1, #1
 andi r2, r1, #1
 bneq r2, SKIP
 nop
 nop
 nop
 nop
SKIP:
 sub  r3, r1, r4
 bneq r3, LOOP
```

☑ (4 pts) What are the best-case and worst-case prediction accuracies for the first branch. Briefly explain.

The worst case occurs when the counter is 2 when a not-taken branch is being predicted. The prediction will always be wrong. (Worst case accuracy of 0%.) Otherwise the prediction accuracy will be 50%.

☑ (3 pts) What is the smallest BHT size for which there will not be a collision between the two branches?

The two branches are six (110_2) instructions away from each other, and so their addresses will be the same at bit position 2 and different at bit position 3. Bit position 2 is the LSB of the BHT address. If the BHT had two entries the two branches would share an element, if it had four or more, they would be in different entries.

(*b*) Consider a dynamically scheduled four-way superscalar processor with a common data bus (CDB) that can handle two write-backs per cycle.

☑ (3 pts) Compare its speed to that of an ordinary dynamically scheduled two-way superscalar processor. Justify your answer.

Ordinary n-way superscalar processors can write back n instructions per cycle.

The four-way will be able to fetch and decode faster than the two-way. Since write back is not always a bottleneck, it will be faster.

☑ (3 pts) Compare its speed to that of an ordinary dynamically scheduled four-way superscalar processor. Justify your answer.

Since it can't do as many write backs it will be slower.

(*c*)

☑ (4 pts) Why is branch target prediction potentially more useful for DLX `jalr` instructions than it is for `bneq` and `beqz` instructions?

Because the target address from the branch instruction can be determined from the NPC and the instruction itself, whereas for the jump a register value is needed, which might not be available for several cycles.

(*d*)

☑ (4 pts) What is a predicated instruction? Show how predicated instructions can be used in the code below. (Exact syntax is not important.)

```
 beqz r1, SKIP
 add  r2, r2, r3
SKIP:
 or   r4, r4, r5
```

A predicated instruction is one that writes its result only if a condition (the predicate) is true.

```
(r1) add r2, r2, r3  ! r2 only written if r1 nonzero.
     or  r4, r4, r5  ! Always executed.
```

(*e*) Consider a statically scheduled DLX implementation in which the floating-point add functional unit is two stages and the floating-point multiply functional unit is four stages, and both are fully pipelined. An exception can occur in any stage of the FP functional units.

☑ (4 pts) Show how the code below would execute to ensure precise exceptions.

```
muld  f0, f2, f4   IF ID M1 M2 M3 M4 WB
addd  f2, f8, f10     IF ID ----> A1 A2 WB
or    r1, r2, r3         IF ----> ID EX ME WB
```

☑ (4 pts) Suppose that floating-point instructions did not have precise exceptions. Show how a test instruction could be used to ensure that an exception in `muld` was precise. Illustrate with a pipeline execution diagram.

```
muld  f0, f2, f4   IF ID M1 M2 M3 M4 WB
testexc               IF ID ----> EX ME WB
addd  f2, f8, f10        IF ----> ID A1 A2 WB
or    r1, r2, r3                 IF ID EX ME WB
```

76 Spring 2000 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

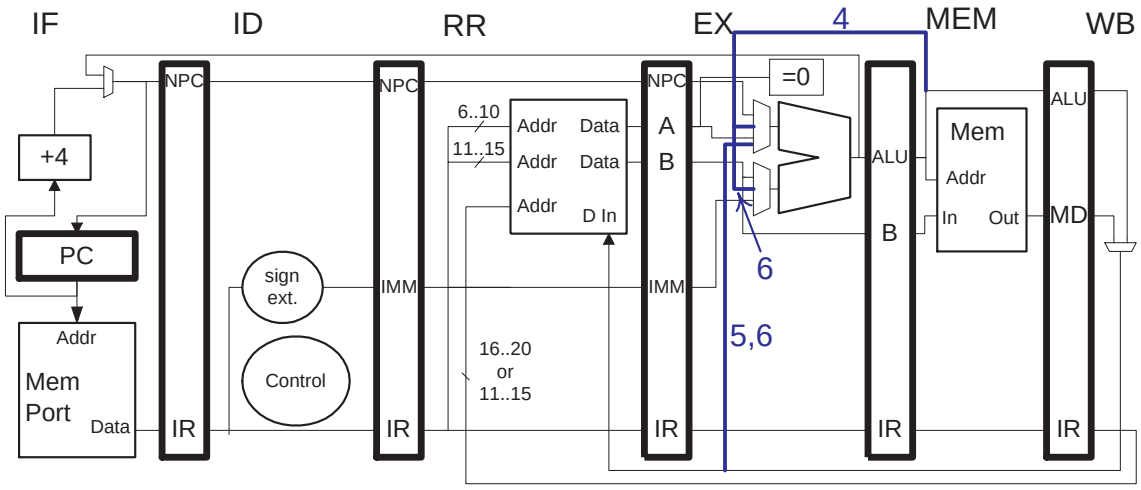
22 March 2000, 13:40–14:30 CST

Problem 1		(35 pts)
Problem 2		(20 pts)
Problem 3		(45 pts)
Exam Total		(100 pts)

Alias

Good Luck!

Problem 1: The DLX implementation below has six stages. (The work done by ID is now done by ID and RR.)



(*a*) The execution of some code on this pipeline is shown below. Add *exactly* the bypass paths needed so that the code executes as illustrated. Next to each bypass path indicate the cycle(s) in which it will be used. (Do not add bypass paths that won't be used in the execution of the code below.) (10 pts)

```
! Cycle          0  1  2  3  4  5  6  7  8
add  r1, r2, r3  IF ID RR EX ME WB
lw   r5, 10(r1)     IF ID RR EX ME WB
xor  r4, r1, r2        IF ID RR EX ME WB
and  r6, r5, r4           IF ID RR EX ME WB
```

The changes are shown in the diagram above in **blue bold.**

(*b*) Show the execution of the code below on this pipeline until `bneq` reaches IF a second time. The branch is taken. Be sure to base the CTI behavior on the hardware shown above. Show where instructions are squashed. (10 pts)

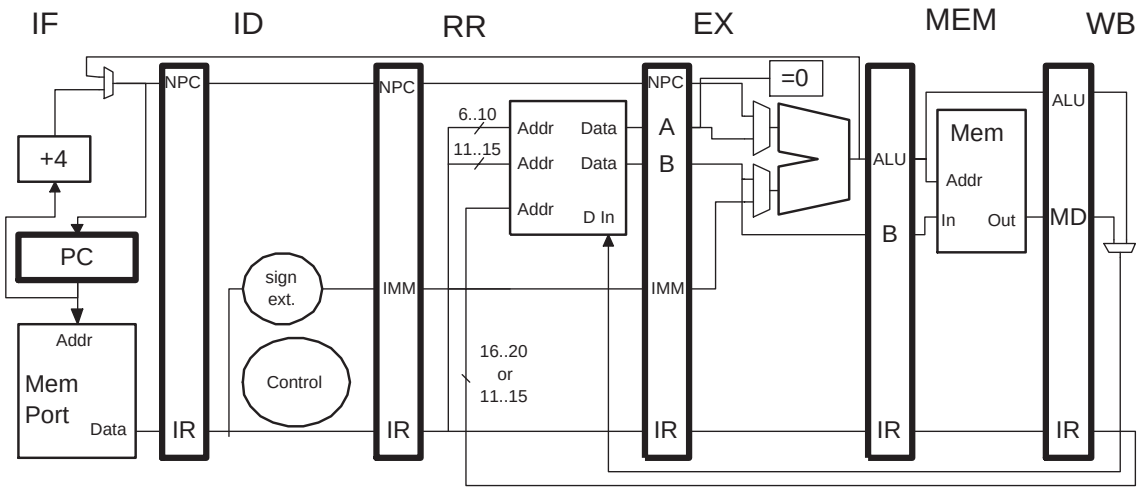
The RR stage adds a cycle of branch penalty, but one cycle is saved, compared to the original pipelined DLX implementation, because the ALU output, rather than the `EX/MEM.ALU` latch, is fed back to the PC multiplexor.

```
LOOP:
! Solution.
! Cycle          0  1  2  3  4  5  6  7  8
 bneq r1, SKIP   IF ID RR EX ME WB       IF
 add  r2, r3, r4    IF ID RRx
 sub  r5, r6, r7       IF IDx
 and  r8, r9, r10         IFx
 or   r11, r12, r13
SKIP:
 j LOOP                      IF ID RR EX ME
 add  r2, r3, r4                IF ID RRx
 sub  r5, r6, r7                   IF IDx
 and  r8, r9, r10                     IFx
 or   r11, r12, r13
```

Problem 1, continued: The figure and code below are identical to the ones on the previous page.

(*c*) Add branch and jump hardware so that the code executes as quickly as possible. Additional adders can be used, the fewer the better. Branches and jumps can be handled separately. The register file cannot be moved or duplicated and cannot be read before the RR stage. (Jumps and branches both use displacement addressing. In branches the displacement is in bits 16 to 31 and in jumps the displacement is in bits 6 to 31.) Do not show control hardware. (10 pts)

Solution to be added by May 2000.



(*d*) Show how the code below executes on the modified pipeline. As before, show execution until `bneq` enters IF a second time. (5 pts)

```
LOOP:
! Solution.
! Cycle           0  1  2  3  4  5  6  7  8
 bneq r1, SKIP    IF ID RR EX ME WB
                                 IF ID RR EX
 add  r2, r3, r4     IF IDx
 sub  r5, r6, r7        IFx
 and  r8, r9, r10
 or   r11, r12, r13
SKIP:
 j LOOP                    IF ID RR EX ME WB
 add  r2, r3, r4              IFx
 sub  r5, r6, r7
 and  r8, r9, r10
 or   r11, r12, r13
```

Problem 2: The code below executes on a dynamically scheduled processor that uses reorder buffer entry numbers to name registers. There are 128 reorder buffer entries, the next free entry is #1.

(*a*) Show when each instruction commits and show the contents of the register map, register file and reorder buffers using the space provided. Show only the instruction line numbers (`A`, `B`, `C`, `D`) in the reorder buffer. Indicate cycle numbers above the reorder buffer. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`A:addd f0, f2, f0`	IF	ID	A0	A1	A2	A3	WC										
`B:addd f6, f0, f6`		IF	ID	RS	RS	RS	A0	A1	A2	A3	WC						
`C:addd f0, f0, f6`			IF	ID	RS	RS	RS	RS	RS	RS	A0	A1	A2	A3	WC		
`D:addd f6, f2, f8`				IF	ID	A0	A1	A2	A3	WB						C	

Register Map

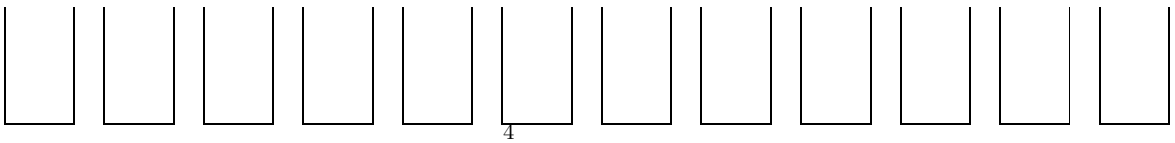
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
`f0`	10	#1		#3											120		
`f2`	20																
`f6`	60		#2		#4					100							
`f8`	80																

Register File

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
`f0`	10						30								120		
`f2`	20																
`f6`	60										90					100	
`f8`	80																

Cycle:

Reorder buffer:



Problem 2, continued: (*b*) The code below is identical to the code above, however the second add instruction raises an exception in cycle 9 (indicated by a star). Complete the pipeline execution diagram and other tables for this situation for the instructions shown. (Do not show the trap handler.) Show the contents of the reorder buffers after each change. (10 pts)

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A:addd f0, f2, f0	IF	ID	A0	A1	A2	A3	WC										
B:addd f6, f0, f6		IF	ID	RS	RS	RS	A0	A1	A2	*A3*	WF						
C:addd f0, f0, f6			IF	ID	RS	RS	RS	RS	RS	RS	A0	X					
D:addd f6, f2, f8				IF	ID	A0	A1	A2	A3	WB	X						

Register Map

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
f0	10	#1		#3							30						
f2	20																
f6	60		#2		#4					100	60						
f8	80																

Register File

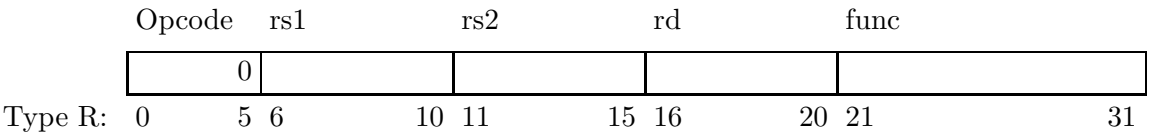
Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
f0	10						30										
f2	20																
f6	60																
f8	80																

Cycle:

Reorder buffer:

Problem 2: Answer each question below.

(*a*) Why do DLX Type-R instructions have a `func` field? (7 pts)



Because there is not enough room in the opcode field to code all DLX instructions. If the `func` field were eliminated and the size of the opcode field were increased Type I and Type J instructions would have to have smaller immediates.

(*b*) Would there be any advantage in pipelining the integer ALU used in the statically scheduled DLX implementation? Explain. (7 pts)

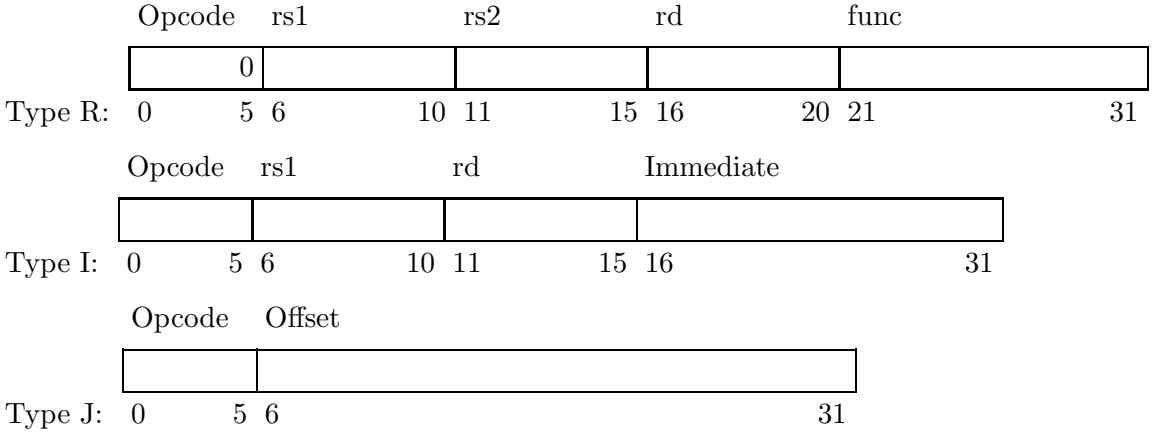
No, since it produces an answer in one cycle. Well, maybe. If functional units in the other stages could be pipelined then the clock frequency could be doubled.

(*c*) Why do ISAs include one-byte integers? (What are the advantages of using them when running on typical implementations.) (7 pts)

They take less space and so are useful for storing large amounts of small numbers. The are **not** faster than full-size integers, arithmetic operations on both can be completed in one cycle.

(*d*) The DLX implementations covered in class start reading registers before the instruction is decoded. Why is this possible? Modify the instruction codings so that it is no longer possible. That is, with the modified codings decoding would have to be performed *before* register read (as in problem 1). (8 pts)

The DLX codings are given below for reference and can be used to explain your answer.



Swap the rd and rs1 fields in the Type R instructions.

(*e*) Ignoring floating-point instructions, how can precise exceptions be implemented on the statically scheduled (Chapter 3) DLX implementation? Illustrate your answer with an example in which exceptions occur out of order but are handled in order. Show the code and a pipeline execution diagram, indicate where the exceptions are occurring. (9 pts)

Pass an exception bit and other information down the pipeline, test for an exception at the end of the MEM stage. In the example below the illegal instruction exception occurs before the page fault, but the page fault is seen at MEM before the illegal instruction exception.

```
lw r4, 0(r5)     IF  ID  EX *ME* WB
ant r1, r2, r3       IF *ID* EX  ME  WB
```

(*f*) What is an advantage of a stack ISA over a RISC ISA? What is a disadvantage of a stack ISA over a RISC ISA? (7 pts)

Advantage of stack: smaller code. Disadvantage: slower implementations.

Name Solution

Computer Architecture
EE 4720
Final Examination

8 May 2000, 10:00–12:00 CDT

Problem 1		(20 pts)
Problem 2		(10 pts)
Problem 3		(10 pts)
Problem 4		(21 pts)
Problem 5		(39 pts)
Exam Total		(100 pts)

Alias MPI phone home!!!

Good Luck!

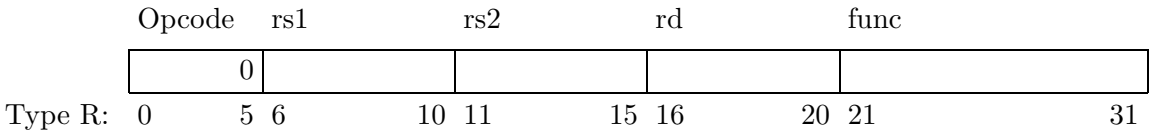
Problem 1: An extended DLX ISA, Triple-DLX [tm], includes new three-operand integer ALU instructions. (Assume that the integer ALU can perform integer multiply.) Some sample instructions appear below:

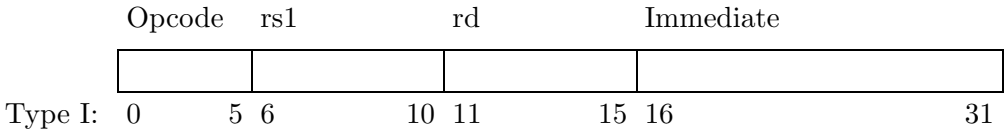
```
add_add r1, r2, r3, r4 ! r1 = r2 + r3 + r4
add_mul r5, r6, r7, r8 ! r5 = ( r6 + r7 ) * r8
mul_add r5, r6, r7, r8 ! r5 = ( r6 * r7 ) + r8
```

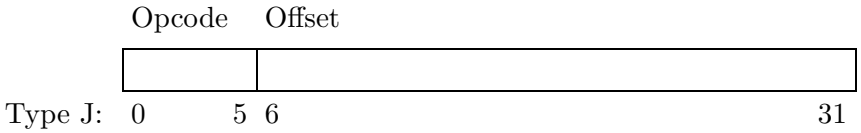
(*a*) (8 pts)A new instruction type, Type-T, will be used for these instructions. Show how the new Type-T instructions can be coded. *The coding should be chosen to ease implementation and should allow for at least 64 three-operand instructions.* Assume that there are six free opcode field values and seven free func-field values available for your use.

- Explain how to distinguish Type-T instructions from Type-R, Type-I, and Type-J instructions.
- How many Type-T instructions can be provided using your coding?

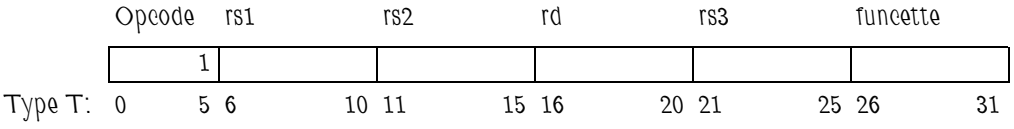
The DLX codings are given below for reference and can be used to explain your answer.





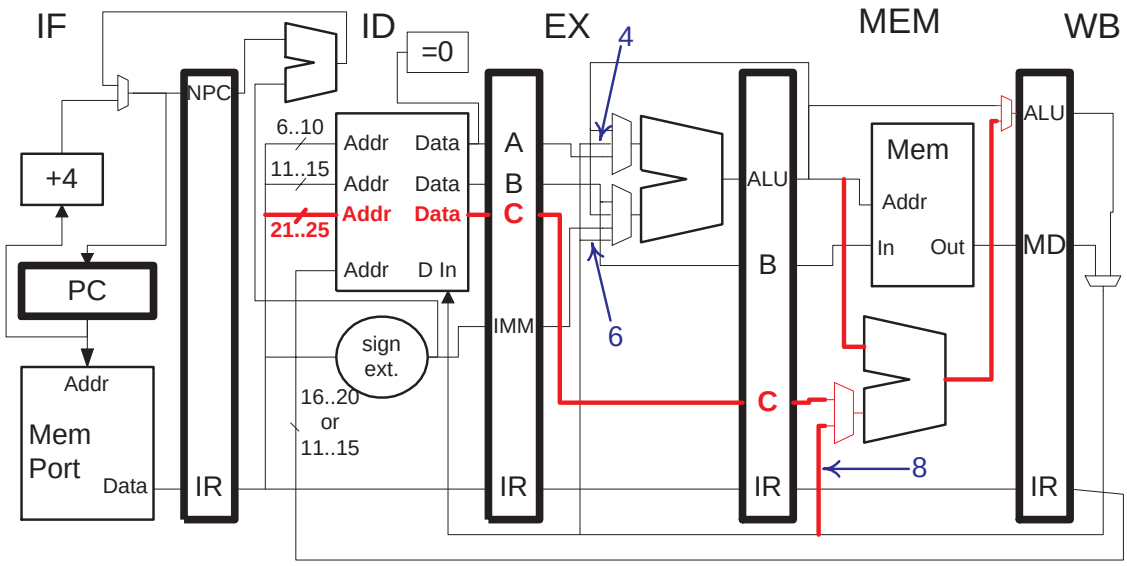


The instruction format is similar to Type R, except an rs3 field is added. It's added after rd, rather than before, so that decoding logic would not have to look for rd in a third place. Type T instruction use one of the 6 opcodes, opcode 1 is used in the example. The funcette field, at 6 bits, specifies which of 64 instructions to perform.



(*b*) Modify the pipeline below so that it can execute the three-operand instructions. A second ALU has been placed in the MEM stage; it should be used to help implement the instructions. The register file is among the parts that need to be modified. (6 pts)

Changes to the pipeline are shown in **red bold.**



(*c*) Show a pipeline execution diagram for the code below assuming that all needed bypass paths are available. The code should execute as fast as possible. **Add** any needed bypass paths to the diagram above. Do not add any bypass paths that are not needed by the code below. Label each bypass path (added or already present) with the cycle in which it is used. Please be sure not to miss any true dependencies. (6 pts)

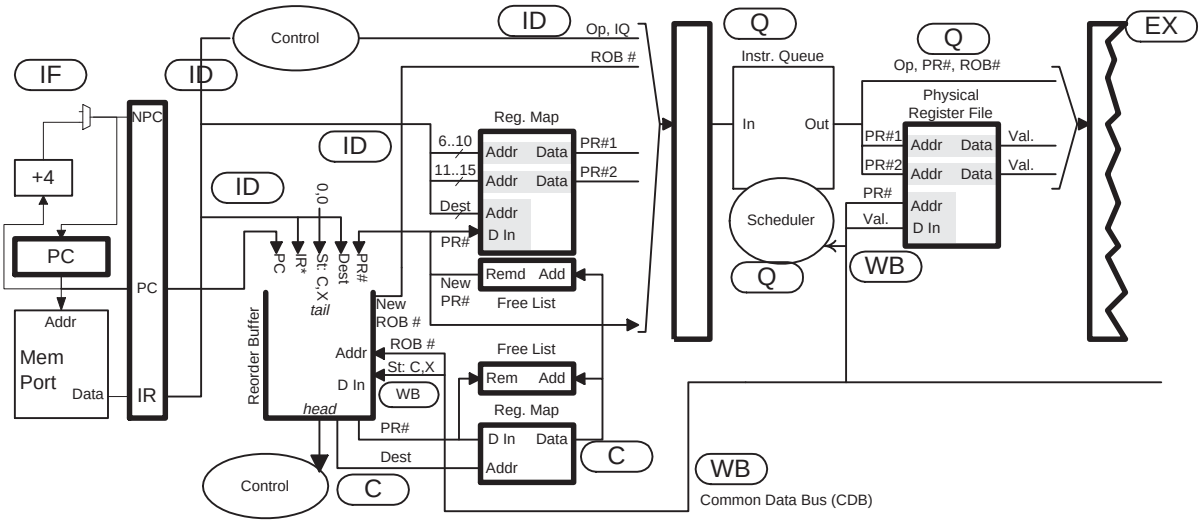
The pipeline execution diagram is shown below. Note that the MEM stage has been labeled E2 to emphasize its role. Also note that in cycle 8 r6 is bypassed to the second ALU in the E2 (MEM) stage. The bypass paths and the cycles in which they are used are shown in blue in the illustration above.

```
! Solution
! Cycle                  0  1  2  3  4  5  6  7  8  9
 add_add r1, r2, r3, r4  IF ID EX E2 WB
 add_sub r5, r1, r2, r3     IF ID -> EX E2 WB
 sub_sub r6, r1, r5, r6        IF -> ID -> EX E2 WB
 sub_add r7, r1, r5, r6              IF -> ID EX E2 WB
```

Problem 2: The code appearing on the next page executes on a dynamically scheduled machine using physical registers to name destination registers.

Complete the tables, showing only changes. Show where instructions commit. Show only entries associated with registers `f0` and `f6` in the physical register file.

At cycle zero, register `f0` contains a 5, `f2` contains a 20, `f4` contains a 40, and `f6` contains a 60. None of the instructions raise exceptions. The diagram below is provided for convenience; it is the same one used in class. (10 pts)



Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
add f0,f2,f4	IF	ID	Q								A0	A1	WC					
add f6,f0,f4		IF	ID	Q									A0	A1	WC			
add f0,f4,f4			IF	ID	Q					A0	A1	WB				C		
add f6,f0,f4				IF	ID	Q								A0	A1	WB	C	
sub f0,f2,f4					IF	ID	Q	A0	A1	WB								C

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Arch. Reg.	ID Register Map																	
f0	17	12		8		3												
f6	6		7		4													

	ID Free List																	
	12																	17
	7	7															17	6
	8	8	8													17	6	12
	4	4	4	4											17	6	12	7
	3	3	3	3	3								17		6	12	7	8

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

	Commit Register Map																	
f0	17												12			8		3
f6	6														7		4	

	Commit Free List																	
	12												7		8	4	3	17
	7												8		4	3	17	6
	8												4		3	17	6	12
	4												3		17	6	12	7
	3												17		6	12	7	8

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Phys Reg.	Physical Register File																	
3										-20								
4																120		
6	60																	
7															100			
8												80						
12													60					
17	5																	

Problem 3: Provide a pipeline execution diagram for the code below running on a dynamically scheduled *two-way superscalar* machine using reorder buffer entry numbers to name destination operands. Be sure to show when instructions complete.

There are more than enough reservation stations and reorder buffer entries. Do not show reservation station numbers in the diagram. There are two load-store units and the *cache is nonblocking*.

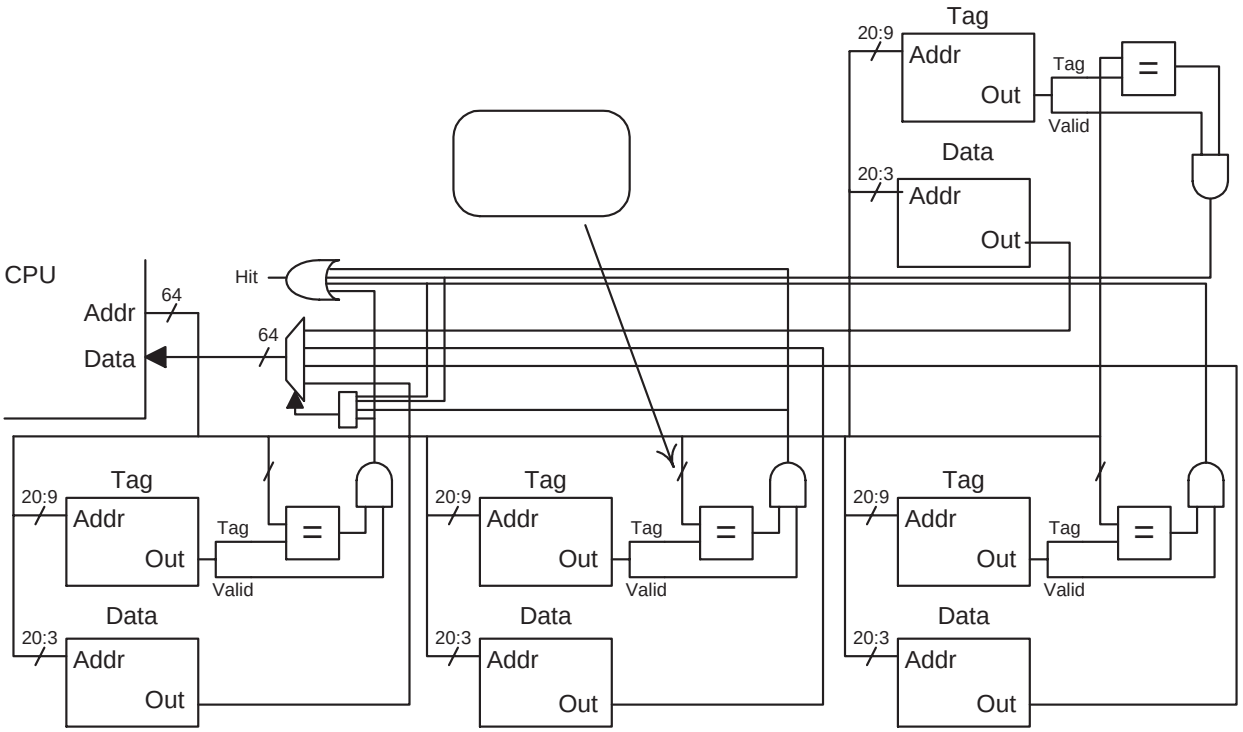
The first `lw` misses the cache, the data arrives 6 cycles after it first enters the L1 stage; the second `lw` also misses the cache, its data arrives 4 cycles after it first enters the L1 stage. (10 pts)

Carefully check the code for dependencies before working on a solution.

The pipeline execution diagram appears below. Note: Because the effective address of the second store is not known in cycle 4, the last load cannot proceed (because of a possible dependency).

```
! Solution
! Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12
lw r3, 0(r1)  IF ID L1 L2 RS          L2 WC
lw r5, 0(r2)  IF ID L1 L2 RS    L2 WB    C
sw 0(r4), r5     IF ID L1 RS       L2 WB    C
lw r7, 0(r9)     IF ID L1 L2 WB             C
sw 0(r3), r1        IF ID RS             L1 L2 WC
lw r8, 0(r10)       IF ID L1                L2 WC
```

Problem 4: A system uses the following cache:



(*a*) Determine the value of the following parameters for the cache illustrated above. Be sure to specify units (bits, bytes, etc.). Answers can be in the form of mathematical expressions. (8 pts)

Associativity: 4

Number of Sets: Solution: $2^{20-9+1} = 2^{12}$.

Address Space Size: 64 bits

Block (Line) Size: Solution: $2^9 = 512$ characters

Cache Capacity: Solution: 4×2^{21} characters.

Amount of Memory to Implement Cache: Solution: 4×2^{21} characters plus 4×2^{12} $((63 - 21 + 1) + 1)$ bits.

Character Size: 8

Tag Bits: (Can show on diagram.) Solution: 63:21.

(*b*) Write a program that will fill the cache using the minimum number of accesses. (Ignore instruction accesses.) (8 pts)

```
void fill()
{
  extern char *a;

  int line_size_lg = 9;
  int line_size = 1 << line_size_lg;
  int associativity = 4;
  int set_count_lg = 12;
  int line_count = associativity * ( 1 << set_count_lg );
  int i, dummy;

  for(i=0; i<line_count; i++) dummy += a[ i * line_size ];

}
```

(*c*) Determine the parameters for a direct-mapped cache with the same block size and the same capacity designed for the same system. (5 pts)

Associativity: One, because it's direct mapped.

Number of sets: Solution: 2^{12+2}

Address Space Size: Solution: 64, no change.

Block (Line) Size: (Same, don't answer.)

Cache Capacity: (Same, don't answer.)

Character Size: Same, 8 bits

Tag Bits: Two less: 63:23.

Problem 5: Answer each question below.

(*a*) The program below runs on a system using a gselect branch predictor. What is the minimum global history size needed so that there is a good chance that the last branch will be correctly predicted at the last iteration (after warmup)? Assume that there are no collisions. Explain your answer. (7 pts)

```
 addi r1, r0, #10
 add  r5, r0, r0
LOOP:
 lw r2, 0(r3)
 add r5, r5, r2
 subi r1, r1, #1
 addi r3, r3, #4
 bneq r1, LOOP
```

If execution is at the last branch and at least one of the previous 9 branches was not taken, then execution surely has not reached the 10th iteration. It would take a global history length of 9 to hold this information. Prediction would not be perfect, it would depend on whether the last branch encountered before reaching the loop is taken.

(*b*) What is the advantage of backing up (checkpointing) the register map when a branch is encountered in a system using branch prediction and dynamic scheduling? Explain how execution would be different if the register map were not backed up. (7 pts)

If the register map is backed up and a branch misprediction is discovered the register map can be restored to the state it was in when the branch was encountered without having to wait for preceding instructions to commit. With a register map backup recovery can start in the WB stage of the branch, without a backup recovery must wait for the commit stage.

(*c*) Why are three-way superscalar machines difficult to build but three-instruction-bundle VLIW ISAs are common? (5 pts)

Superscalar machines are built on existing ISAs, many of which use 32-bit instructions. If 32-bit instructions were fetched in groups of three the fetches would be unaligned and so would take more expensive hardware. In many VLIW ISAs, three instructions are placed in 128-bit bundles, so their addresses are aligned.

While its true that VLIW ISAs allow for less expensive hardware, a three-way superscalar machine is still small and so the hardware needed for dependency checking, register renaming, and scheduling would be less than difficult.

(*d*) An ISA is almost like DLX except, oops, the `lbu` (load byte unsigned) instruction was omitted, and so the program below won't run on an implementation of this ISA. Modify the program so that it will run, and run as though an `lbu` instruction was used. (In other words, replace `lbu r1, 1(r2)` by instructions that do the same thing.) (5 pts)

```
lbu r1, 1(r2)

! A correct answer:
lb r1, 1(r2)
andi r1, r1, #0xff

! A WRONG answer:
lw r1, 1(r2)       ! Error, since address may not be aligned.
andi r1, r1, #0xff
```

(*e*) The table below shows virtual and physical addresses in use in a virtual memory system having a 32-bit address space. What is the largest possible page size this system can have? Using this page size (or some other one, but show what page size is being used) show the possible contents of a two-level page table storing this virtual-to-physical mapping. State any assumptions made. (For partial credit solve the problem for a one-level page table.) (10 pts)

Virtual	Physical
0xfea34b62	0x74b4b62
0xfeb92b90	0x14b2b90
0xeaa31f16	0x77b1f16

The number of offset bits is the log-base-2 of the page size. The offset bits of physical and virtual addresses are the same, so the largest possible page size here is 2^{17} characters (because the low 17 bits of 0xfea34b62 and 0x74b4b62 are the same, the low 17 bits of 0xfeb92b90 and 0x14b2b90 are the same, and the low 17 bits of 0xeaa31f16 and 0x77b1f16 are the same). Note that it's possible the page size is smaller.

The solution below is for a 2^{16} character page. The level-one table is index using the upper 8 bits of the address and the second-level table is index using the next 8 bits.

Level One Page Table:

Address	Level Two Base
⋮	⋮
0xea	3000
⋮	⋮
0xfe	7000
⋮	⋮

Level Two Page Table:

Address	Physical Page Number
⋮	⋮
3000	⋮
⋮	⋮
30a3	77b
⋮	⋮
7000	⋮
⋮	⋮
70a3	74b
⋮	⋮
70b9	14b
⋮	⋮

(*f*) Describe two ways that loop unrolling improves performance. (5 pts)

Fewer branches are needed, eliminating the instruction themselves plus the branch penalty. Certain operations, such as index variable increments can be eliminated. Greater scheduling freedom.

77 Fall 1999 Solutions

Name Solution

Computer Architecture
EE 4720
Midterm Examination

20 October 1999, 10:40–11:30 CDT

Problem 1		(33 pts)
Problem 2		(33 pts)
Problem 3		(34 pts)
Exam Total		(100 pts)

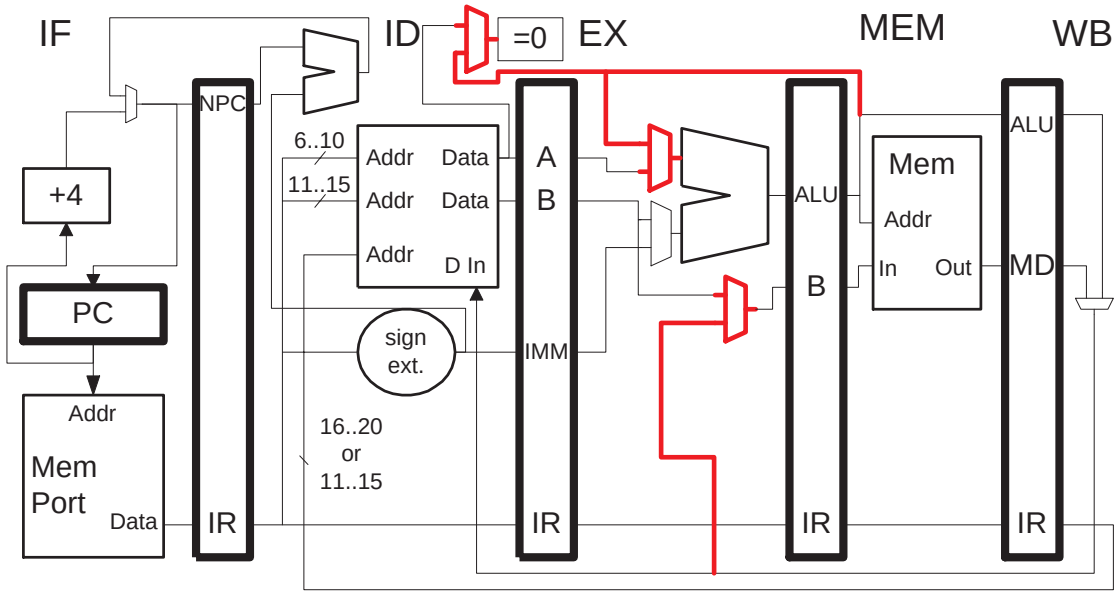
Alias

Good Luck!

Problem 1:

(*a*) Add *exactly* the bypass paths that are needed so the code below executes as shown in the pipeline execution diagram. (**Don't** add any bypass paths that are not needed by the code.) (15 pts)

Added bypass paths appear in **red bold** in the diagram below.



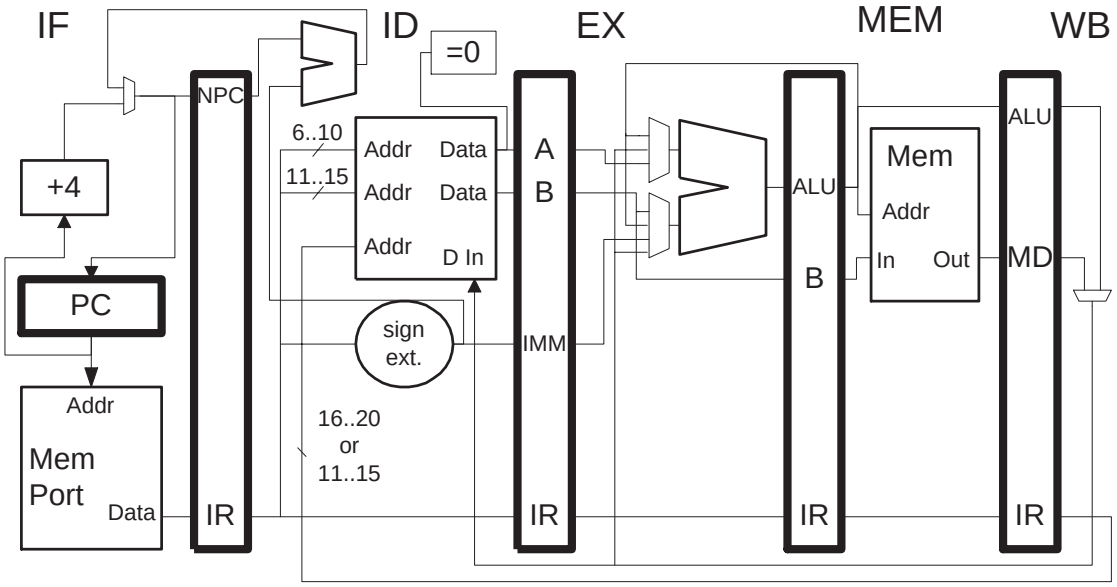

```
! r1 = 0x1010  r2 = 0x2020, r3 = 0x3030, r4 = 0x4040, r5 = 0x5050
! Cycle            0   1   2   3   4   5   6   7
LINE1:  LINE1 = 0x100
 lw   r1, 16(r2)   IF  ID  EX  MEM WB  IF  ID  EX ...
 addi r2, r2, #4       IF  ID  EX  MEM WB  IF  ID ...
 sw   20(r2), r1           IF  ID  EX  MEM WB  IF ...
 beqz r2, LINE1                IF  ID  EX  MEM WB ...
 and  r3, r4, r5               IFx
```

(*b*) Show the values of the registers listed below in **cycle 4** of the execution above using the added bypass paths. Assume that the load instruction retrieves a `0x1234` from memory. Instructions are squashed by replacing them with an `or r0, r0, r0`. (18 pts)

IF.PC 0X110	ID/EX.A 0X2020	EX/MEM.ALU 0X2024	MEM/WB.ALU 0X2030
IF/ID.NPC 0X110	ID/EX.B 0X1010	EX/MEM.B 0X2020	MEM/WB.MD 0X1234
IF/ID.IR beqz	ID/EX.IMM 0X20	EX/MEM.IR addi	MEM/WB.IR lw
	ID/EX.IR sw		

Problem 2:

(*a*) Show a pipeline execution diagram for the code below on the implementation shown until `lw` is fetched a second time. The first branch is not taken but the last one is. The only bypass paths available are the ones shown. (18 pts)




```
LOOP:
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)  IF  ID  EX  MEM WB                                          IF
 andi r3, r1, #4     IF  ID  --> EX  MEM WB
 beqz r3, LINE1          IF  --> ID  -----> EX  MEM WB
 add  r4, r4, r1             IF  -----> ID  EX  MEM WB
 j LINE2                                IF  ID  EX  MEM WB
LINE1:
 add  r5, r5, r1                            IFx
LINE2:
 sw   4(r2), r1                                 IF  ID  EX  MEM WB
 addi r2, r2, #8                                    IF  ID  EX  MEM WB
 bneq r2, LOOP                                          IF  ID  -----> EX
 xor  r5, r6, r7                                            IF  ----->x
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

(*b*) Rewrite the code below (which is the same as the code on the previous page) using one-cycle delayed branches and predicated instructions and schedule the code so that it executes as quickly as possible. (Do not unroll the loop.) Assume that bypass paths are provided for the predicated instructions. It should be possible to remove all stalls, but if any remain point them out (for partial credit). (15 pts)

```
LOOP:
 lw   r1, 0(r2)
 andi r3, r1, #4
 beqz r3, LINE1
 add  r4, r4, r1
 j LINE2
LINE1:
 add  r5, r5, r1
LINE2:
 sw   4(r2), r1
 addi r2, r2, #8
 bneq r2, LOOP
 xor  r5, r6, r7
```

Predicated instructions are used in a straightforward manner.

To remove the stall between **lw** and **andi** the **addi** instruction is placed between them. Since **addi** was moved above **sw** eight is subtracted from the immediate in **sw**.

Now that branches have one-slot delays, **sw** can be placed after the branch.

Note that the code below encounters no stalls (assuming cache hits) and something useful is done in the branch delay slot.

```
LOOP:
       lw   r1, 0(r2)
       addi r2, r2, #8
       andi r3, r1, #4
 (~r3) add r4, r4, r1
 (r3)  add r5, r5, r1
       bneq r2, LOOP
       sw   -4(r2)
```

Problem 3: Answer each question below.

(*a*) Why do DLX branches (and branches in many other ISAs) use displacement addressing? Why don't branches use indirect addressing (destination address in a register) instead of displacement addressing? (8 pts)

The target of branches are usually nearby so the displacement (in the immediate field) in a type-J instruction would usually be large enough. If indirect addressing were used a register would have to be loaded with the target address, which would require two instructions before most branches.

(*b*) The code below executes on an implementation that uses a reservation register to detect WB structural hazards. At cycle zero the reservation register contains all zeros. Show the state of the reservation register at the end of each cycle below. Indicate which (if any) bit positions are tested in each cycle. (9 pts)

```
! Cycle              0  1  2  3  4  5  6  7  8  9  10
multf f0, f1, f2    IF ID M0 M1 M2 M3 M4 M5 WB
addf  f3, f4, f5       IF ID A0 A1 A2 A3 WB
subf  f6, f7, f8          IF ID -> A0 A1 A2 A3 WB
gtf   f9, f10, f11           IF -> ID A0 A1 A2 A3 WB
nop
nop
...
! Solution:
! Cycle             0  1  2  3  4  5  6  7  8  9  10
                Pos
                 7     1* 0  0  0  0  0  0  0  0  0
                 6     0  1  0  0  0  0  0  0  0  0
                 5     0  1* 1* 1* 1* 0  0  0  0  0
                 4     0  0  1  1  1  1  0  0  0  0
                 3     0  0  0  1  1  1  1  0  0  0
                 2     0  0  0  0  1  1  1  1  0  0

! Cycle             0  1  2  3  4  5  6  7  8  9  10
```

The contents of the reservation register is shown vertically each cycle except zero. An asterisk marks the bit position that is checked.

(*c*) Many packed operand instructions perform saturating arithmetic. What is saturating arithmetic? Provide an example. (8 pts)

In saturating arithmetic if the result of an operation is too large to represent (*e.g.*, 10 using 3 bits) it is replaced by the largest representable value (7 using 3 bit unsigned integers). Similarly if the result is too small, it is replaced by the smallest representable value.

Example: Eight-bit unsigned integers $100 + 200 = 255$.

(*d*) In homework 3, a special return address register (`ERA`) was used to hold exception return addresses. The jump and link instructions, `jal` and `jalr`, use `r31` for the return address; is this an option for exceptions? Explain. (9 pts)

It's not an option because `r31` must be left unchanged. After some exceptions (*e.g.*, page fault) execution is supposed to resume normally, so all register values must be left unchanged. The interrupted program might have been using `r31` for a jump and link return address, or for some other purpose.

Name Solution

Computer Architecture

EE 4720

Final Examination

Primary: 6 December 1999, 10:00–12:00 CST
Alternate: 7 December 1999, 15:00–17:00 CST

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(25 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias MPI phone home!!!

Good Luck!

Problem 1: The code in the table on the next page executes on a dynamically scheduled machine using reorder buffer entry numbers to rename registers. The implementation performs branch and branch target prediction.

There are an unlimited number of reorder buffer entries, reservation stations, and functional units. Integer instructions use the `EX` functional unit, branches use the `B` unit, and loads and stores use the load/store unit, consisting of segments `L1` and `L2`.

When the code in the table starts to execute all register values are available, as shown in the tables on the next page.

The `lw` instruction will suffer a miss; it will finish `L2` five cycles after entering `L2`, loading a `100`.

The `bneq` instruction is predicted not taken but is, in fact, taken.

(*a*) For this subproblem the register file is not backed up when branches are encountered. Using the tables provided on the next page show a pipeline execution diagram for the code and the changes to the register map and register file at the end of each cycle. Do not show reservation station numbers or reorder buffer entries in the pipeline execution diagram itself. The entry number for the next available reorder buffer entry is 1. Register values are in hexadecimal. Show when instructions commit or when they are squashed. (15 pts)

(*b*) Explain how execution would be different if the register file were backed up when branches are encountered. A second diagram is not necessary, just show where execution differs and how it differs. (5 pts)

If the register map were backed up the fetching on the correct path would start one cycle after the branch reached WB, instead of one cycle after it committed. (The one-cycle delay is needed to detect the misprediction and move the correct address into the PC.

There is one more part.

Problem 1, continued: Completed table appears below. Only changes to register map and file are shown.

Pipeline Execution Diagram

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw r4, 0(r5)`	IF	ID	L1	L2	RS				L2	WC							
`add r1, r2, r3`		IF	ID	EX	WB						C						
`or r2, r1, r3`			IF	ID	EX	WB						C					
`bneq r4, SKIP`				IF	ID	RS				B	WB		C				
`sub r1, r2, r5`					IF	ID	EX	WB					X				
`SKIP:`																	
`add r2, r1, r2`						IF	ID	EX	WB				X	IF	ID	EX	WC

Register Map

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val. or ROB#																
`r1`	10		#2		50	#5		20					50				
`r2`	20			#3		70	#6		90				70		#5		60
`r3`	30																
`r4`	40	#1								100							
`r5`	50																

Register File

Cycle:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Arch. Reg.	Val.																
`r1`	10										50						
`r2`	20											70					60
`r3`	30																
`r4`	40									100							
`r5`	50																

Problem 1, continued:

(*c*) The DLX code in the table below executes on the dynamically scheduled machine described above. The machine uses a load/store queue and a nonblocking (lockup free) cache as described in class. For this part the cache miss latency is lower: If an instruction in `L2` encounters a cache miss it returns to its reservation station for three cycles then returns to `L2`.

Show the execution of the code in the table below. Show when instructions commit but **do not** show reservation station or reorder buffer entry numbers.

The instructions at lines `Line1` and `Line3` miss the cache. The contents of each register is different. (5 pts)

Completed table appears below.

Pipeline Execution Diagram															
Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`Line1: sw 0(r1), r2`	IF	ID	L1	L2	RS			L2	WC						
`Line2: lw r3, 0(r2)`		IF	ID	L1	L2	WB				C					
`Line3: lw r1, 0(r4)`			IF	ID	L1	L2	RS			L2	WC				
`Line4: sw 0(r1), r2`				IF	ID	RS					L1	L2	WC		
`Line5: lw r5, 0(r2)`					IF	ID	L1	RS					L2	WC	

Don't forget part (b)!!!!

This problem consists of four parts. For full credit do parts (a), (b), and (c) only. For **reduced** *credit do parts (a) and (d). If you have time but lack confidence do all parts, the grade will be* $\max\{a+b+c, a+d\}$.

Problem 2: The code below is run on systems having a 32-bit address space, 1-byte characters, and using 256-kibibyte (2^{18}-byte) caches as described below. Before the code is run the cache is cold (empty). Only consider accesses to the array, `a`.

```
// sizeof(int) = 4 characters
int *a = 0x1000000;  // Storage allocated elsewhere.
for(x=0; x<4; x++)
  for(i=0; i<512; i++)
    for(j=0; j<8; j++)
      sum += a[ i * 1024 + j ];
```

(*a*) Find the hit ratio encountered executing the code above on a direct-mapped 2^{18}-byte cache with a line size of 8 characters. How much of the cache is filled? (9 pts)

Each 8-byte line holds two four-byte integers. The innermost loop accesses integers sequentially so initially the hit ratio will be $\frac{1}{2}$. Each iteration of the middle loop skips ahead 1024 integers. Each iteration of the outermost loop is identical (the same addresses are used) so maybe there will be more hits in the second iteration.

Because the cache is direct mapped no two lines can have the same index. (An access to a second line would evict the first.) In the cache the index bits are 3:17. The access to the array varies address bits 2:4 (the j loop) and 12:20 (the i loop). Three bits, 18, 19, and 20 are in the tag (outside the index). Therefore there will be eight different addresses that use the same index. When the second iteration of the outermost loop starts the lines needed (in the beginning) will have been evicted. Therefore the overall hit ratio is $\frac{1}{2}$.

To find the amount of data cached, find the number of bits in the intersection of the index and offset bits with the bits varied by the code. The intersection is bits 17:12 and 4:2 for a total of 9 bits. The amount of the cache filled is $2^9 = 512$ integers or 2048 bytes.

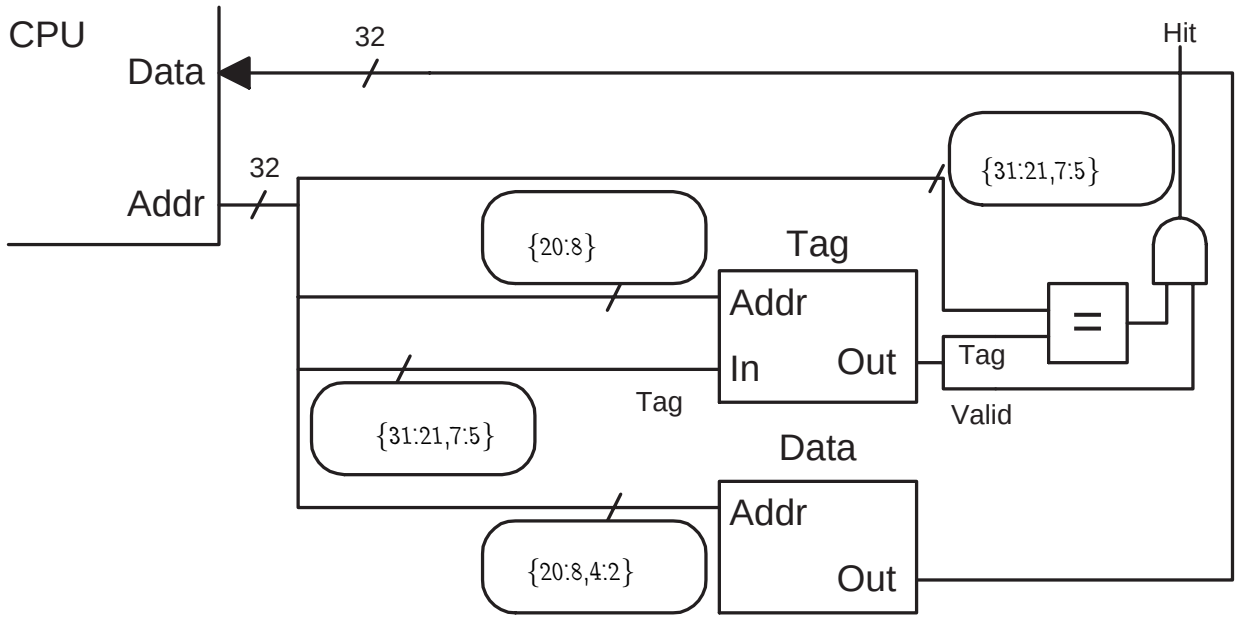
(*b*) Suppose the associativity of the cache could be increased while fixing the cache capacity at 2^{18}-byte and the line size at 8. What would the hit ratio be if the associativity were 2? What is the smallest associativity needed to achieve the maximum hit ratio on the code above? Is that associativity practical? Explain. (8 pts)

The hit ratio is still $\frac{1}{2}$ when the associativity is 2. Because doubling the associativity reduces the number of index bits by one, it would take an associativity of 512 to maximize the hit ratio. This associativity is impractically high.

Problem 2, continued:

(*c*) Suppose **any** address bits could be used to form the index (set number, address used in the tag store) and any line size could be used. (Lines must still be contiguous.) In the diagram below show which address bits should be used (using the rounded boxes) to maximize the hit ratio of the code above. The capacity of the cache must still be 2^{18}-byte. (8 pts)

Figure includes solution.

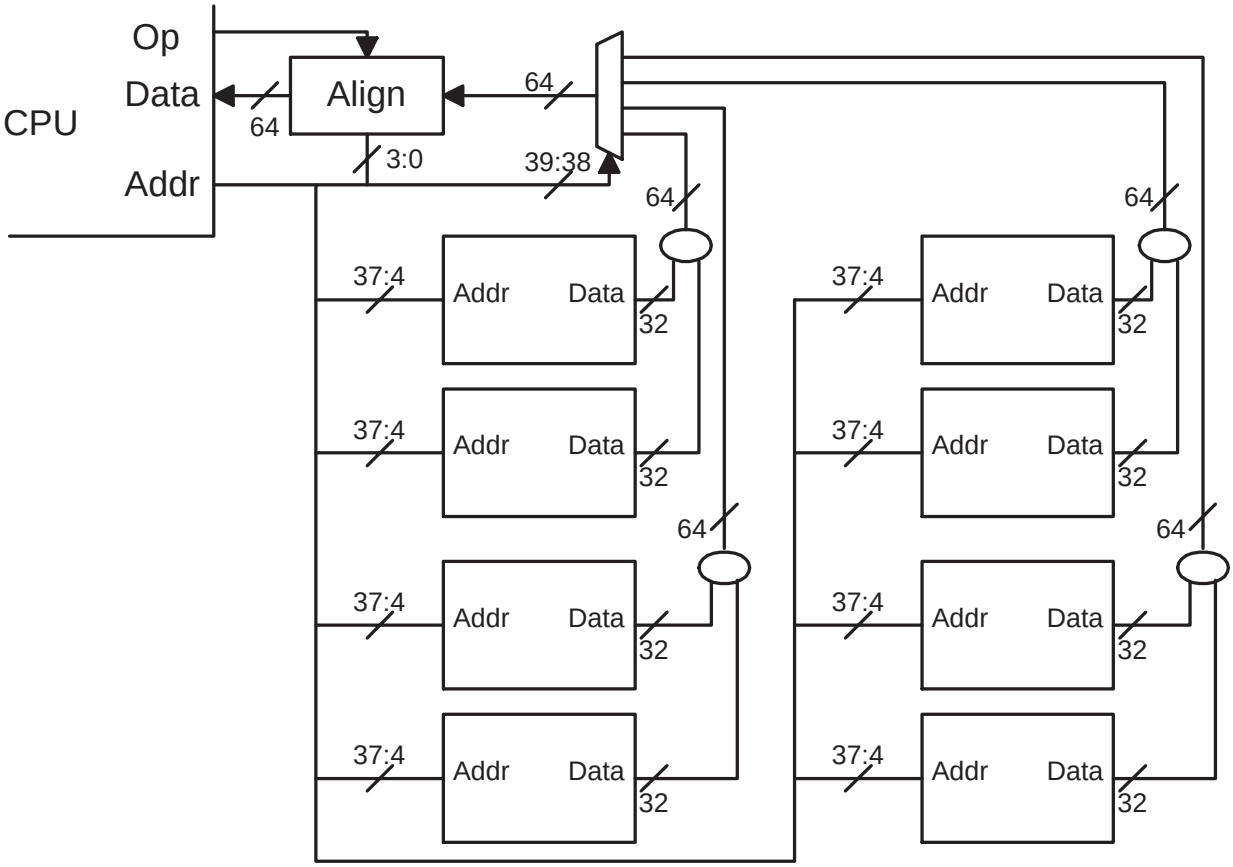


The line size is increased to 32 bytes to minimize cold misses. To eliminate conflict misses the index bits are moved to positions 20:8 (so that none of the bits varied by the code are in the tag).

(*d*) **Optional:** *Only solve this part if you cannot solve parts (b) and (c). Total credit will be lower.*

This part is unrelated to the previous parts. Show how $2^{34} \times 32\,\mathrm{b}$ memory devices should be connected to implement a 40-bit address space on a system with four-bit (one-nibble) characters and a bus width of 64 bits. Include the alignment network. (8 pts)

Solution:



Problem 3: The code below is compiled and run on two machines, one using a one-level branch predictor and the other using a two-level gselect branch predictor. Both predictors use a 4096-entry BHT, the gselect predictor uses a 3-branch global history. Each entry holds a 2-bit saturating counter.

```
for(i=0; i<100000; i++)
{
Line1:  if( a == 1 ) aa++;
Line2:  if( b == 1 ) bb++;
Line3:  if( c == 1 ) cc++;
Line4:  if( i & 0x2 ) { x++; }  /*  N N T T N N T T N N T T N N T T ... */
Line5:  if( i & 0x4 ) { y++; }  /*  N N N N T T T T N N N N T T T T ... */
Line6:  if( i & 0x2 ) { z++; }
}
```

(*a*) The ISA has a 32-bit address space, all instructions are 32 bits (four characters) and must be aligned. How is the address for the BHT in the gselect predictor obtained? Be sure to specify bit positions. (9 pts)

Since there are 4096 entries the BHT must have 12 address bits. In gshare the global branch history is concatenated with some instruction address bits. There are 3 bits of global history so 9 instruction address bits are needed. Bits are taken from the low end, skipping the alignment. In this case bits 10:2.

Note: `i & 0x2` is the bitwise and of the value of i and 2. (0x2 is the hexadecimal representation of 2). The if is taken when the second bit of i is 1. For example:

```
 i     i & 0x2
000     0  (False)
001     0  (False)
010     10 (True)
011     10 (True)
100     0  (False)
101     0  (False)
110     10 (True)
111     10 (True)
```

Note that a value of zero is false and any non-zero value is true.

(*b*) Assume that exactly one branch instruction is generated for each `if` statement and that the compiler does no optimizing. What is the prediction accuracy for each of the last three `if` statements using the one-level predictor after a large number of iterations? (8 pts)

The branches at `Line4` and `Line6` suffer a prediction "accuracy" of 25% and `Line5` has an accuracy of 50%.

(*c*) As above, assume that exactly one branch instruction is generated for each `if` statement and that the compiler does not do any optimizing. What is the prediction accuracy for each of the last three `if` statements using the gshare predictor after a large number of iterations? *Hint: The solution to this part does not require tedious computation or the construction of lengthy tables.* (8 pts)

The global history is not much help to the branch at `Line4` because the preceding three branches are always taken the same way, so the prediction accuracy remains 25%. Knowing whether the branch at `Line4` was taken is no use in predicting the branch at `Line5`, so its accuracy remains at 50%. The outcome of `Line4` is useful in predicting `Line5`, in fact it can be predicted with 100% accuracy.

Problem 4: Answer each question below.

(*a*) Synthetic instruction `clr 12(r2)` writes a zero to the memory location at address `12 + r2`. How could it be added to DLX? (5 pts)

This would correspond to DLX instruction `sw 12(r2),r0`.

(*b*) Besides the large amount of storage, what would be the disadvantage of a RISC ISA that had 1048576 (2^{20}) integer (general-purpose) registers? (5 pts)

It would take 20 bits to specify each operand and so the instructions would be too large.

(*c*) Why would it be inappropriate to add a memory indirect load instruction to DLX. (For example, `lw r1,@(r2)`.) Justify your answer for the statically scheduled DLX implementation. Weigh the complexity of changes needed for the implementation against expected benefit over a software-only solution. Be brief. (5 pts)

Such an instruction would require two accesses to memory. In the static DLX implementation this would require either a second MEM stage (too expensive and only used for a few instructions) or the instruction would have to make two passes through MEM, complicating the simple integer pipeline. Two load instructions would be just as fast.

(*d*) The code below runs on a dynamically scheduled 4-way superscalar machine with perfect branch target prediction and a cache that never misses. There are an unlimited number of reorder buffer entries, reservation stations, and functional units. This machine is not 100% perfect: its fetch mechanism is the type described in class. What is the minimum and maximum CPI executing the code below for a large number of iterations? Explain the conditions under which minimum and maximum CPI are encountered. (5 pts)

```
LOOP:
 lb   r1, 0(r2)
 addi r2, r2, #1
 bneq r1, LOOP
```

The minimum CPI is $\frac{1}{3}$, the maximum CPI is $\frac{2}{3}$. The minimum CPI is encountered when all three instructions lie on one (aligned) fetch block, as when LOOP is a multiple of 16. The maximum occurs when the instructions fall on two fetch blocks.

(*e*) What is the difference between a write-through cache and a write-back cache? Which one needs a dirty bit and why? (5 pts)

In a write-through cache stores are always immediately written through to the next level of the hierarchy, for example, to memory. In a write back cache data is not written to the next level of the hierarchy until a line is replaced. (Or under other circumstances covered in later courses.)

78 Spring 1999 Solutions

Name Solution_____

Computer Architecture
EE 4720
Midterm Examination

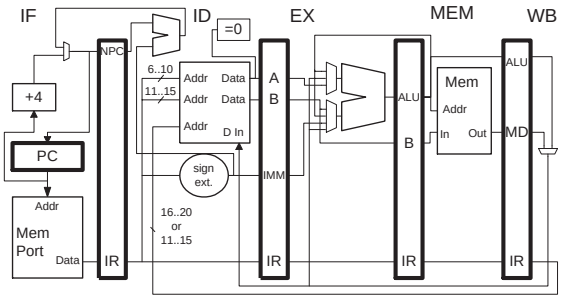
12 March 1999, 10:40–11:30 CST

Problem 1	_____	(25 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(45 pts)
Exam Total	_____	(100 pts)

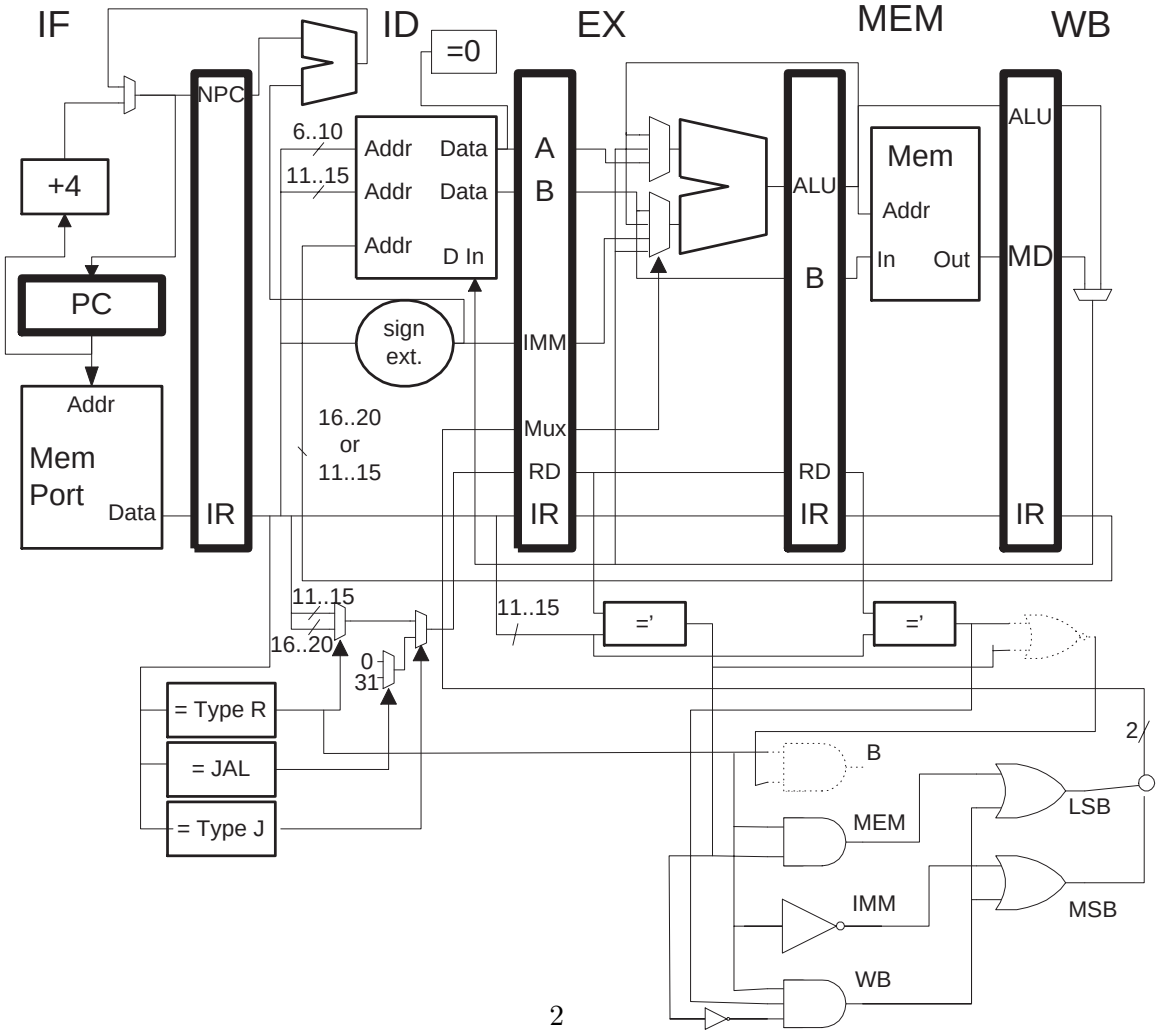
Alias _____

Good Luck!

Problem 1: Design control logic to generate the control signal for the multiplexor at the lower input to the ALU. The control logic should be located in the ID stage and should generate a two-bit integer for the multiplexor. The integer specifies which multiplexor input to use, they are numbered from zero starting at the top. (Input 0 connects to `ID/EX.B`, 1 connects to `EX/MEM.ALU`, etc.) The logic can use units that test for equality of their two inputs, `=`, and units that test for instruction formats, `= Type I`, `= Type R`, and `= Type J`, and can use the usual logic gates. Base the setting on instruction type, rather than the exact opcode. Show how the control signal is connected to the multiplexor. (25 pts)



Solution:



Problem 2: Consider the code below.

```
LOOP:
Cycle              0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)    IF  ID  EX  MEM WB                                  IF  ID  EX
 addi r3, r1, #12      IF  ID  --> EX  MEM WB                              IF  ID
 sw   4(r2), r3            IF  --> ID  -----> EX  MEM WB                       IF
 add  r5, r5, r1               IF  -----> ID  EX  MEM WB
 addi r2, r2, #8                          IF  ID  EX  MEM WB
 slt  r6, r2, r7                              IF  ID  EX  MEM WB
 bneq r6, LOOP                                    IF  ID  -----> EX  MEM WB
 xor  r8, r9, r10                                     IF  -----> x
```

(*a*) Show a pipeline execution diagram for execution up to the second time `lw` enters instruction fetch. Use the pipeline from problem 1. As with homework 3, a bypass path is unavailable if it's not shown. What is the CPI for a large number of iterations? (10 pts)

The pipeline execution diagram appears above. The CPI is $\frac{13}{7} = 1.857\,\text{CPI}$.

(*b*) Unroll the loop so that two iterations of the code above is performed by one iteration of the unrolled loop. (Assume the number of iterations in the original loop is a multiple of two.) Schedule the unrolled loop to minimize stalls. (10 pts)

```
LOOP:
!Cycle:           0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)   IF  ID  EX  MEM WB                              IF  ID  EX  MEM
 lw   r11, 8(r2)      IF  ID  EX  MEM WB                              IF  ID  EX
 addi r3, r1, #12         IF  ID  EX  MEM WB                              IF  ID
 addi r13, r11, #12           IF  ID  EX  MEM WB                              IF
 addi r2, r2, #16                 IF  ID  EX  MEM WB
 slt  r6, r2, r7                      IF  ID  EX  MEM WB
 add  r5, r5, r1                          IF  ID  EX  MEM WB
 add  r5, r5, r11                             IF  ID  EX  MEM WB
 sw   -12(r2), r3                                 IF  ID  EX  MEM WB
 sw   -4(r2), r13                                     IF  ID  EX  MEM WB
 bneq r6, LOOP                                            IF  ID  EX  MEM WB
 xor  r8, r9, r10                                             IF  x
```

(*c*) What is the CPI of the unrolled and scheduled loop found above? What conclusions about performance improvement can and cannot be made by comparing the CPI of the original and unrolled loop? What is the performance improvement? (Give a number for performance improvement, don't just say "it's good.") (10 pts)

The unrolled loop has 11 instructions and only suffers a 1-cycle branch delay, for a CPI of $\frac{12}{11} = 1.091\,\text{CPI}$.

Though the CPI is lower, this doesn't tell the whole story because fewer instructions do the same amount of work and so the performance improvement is more than CPI improvement would suggest.

Performance improvement will be expressed as speedup. The speedup of the unrolled loop will be found using the time needed to do two iterations of the original loop and dividing it by the iteration time of the unrolled and scheduled loop: $\frac{13\times 2}{12} = 2.167$.

Problem 3: Answer each question below.

(*a*) Show an example of DLX code that encounters a WAW hazard on the Chapter-3 implementation of DLX (in which the multiply floating-point functional unit has an initiation interval of 1 and a latency of 6 and the add floating-point functional unit has an initiation interval of 1 and a latency of 3) but which does not encounter a WAW hazard on a DLX implementation which is identical except the FP add latency is 1 and the FP multiply latency is 4. The code should not encounter a structural hazard on either implementation. (12 pts)

```
! Code execution on Chapter-3 DLX (unmodified).
! Cycle           0   1   2   3   4   5   6
addf  f0, f1, f2  IF  ID  A0  A1  A2  A3  WB
lf    f0, 0(r1)       IF  ID  EX  MEM WB

! Code execution on Chapter-3 DLX with fast FP functional units.
! Cycle           0   1   2   3   4   5   6
addf  f0, f1, f2  IF  ID  A0  A1  WB
lf    f0, 0(r1)       IF  ID  EX  MEM WB
```

(*b*) In DLX, why are there separate `lh` (load half) and `lhu` (load half unsigned) instructions, a `sh` (store half) instruction but **no** `shu` (store half unsigned) instruction? (11 pts)

Because the register contents is stored into a location of the right size and so there is no need for sign extension and therefore no need to distinguish a signed and unsigned value.

(*c*) The code below is for a stack architecture. Rewrite the code below in DLX using as few instructions as possible. Assume that `ADDRA` is in `r10`, `ADDRB` is in `r11`, `ADDRX` is in `r12`, and `ADDRY` is in `r13`. The data at the addresses are double-precision floating-point values. (11 pts)

```
push ADDRX
push ADDRX
mult
push ADDRA
push ADDRB
add
push ADDRX
mult
push ADDRA
push ADDRB
mult
add
add
pop ADDRY
```

```
ld    f0, 0(r10)  ! A
ld    f2, 0(r11)  ! B
ld    f4, 0(r12)  ! X
multd f6, f4, f4
addd  f8, f0, f2
multd f8, f8, f4
multd f10, f0, f2
addd  f10, f10, f8
addd  f10, f10, f6
sw    0(r13), f10
```

(*d*) Explain two ways in which precise exceptions can be made optional for floating-point instructions. That is, the programmer may choose to have precise exceptions where they are needed or may choose to not have precise exceptions were performance is most important. Explain what the programmer would have to do for each of the two ways. (11 pts)

Method 1: Provide precise and non-precise versions of floating-point instructions. The programmer uses the appropriate version.

Method 2: Provide a test instruction that can be used after floating-point instructions for which exceptions must be precise.

Name Solution_____

Computer Architecture
EE 4720
Final Examination

5 May 1999, 7:30–9:30 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: With the following extension to the DLX ISA a program can get a count of how many times a particular address has been read or written by load and store instructions. The count is kept in a *load/store counter (LSC)* which is incremented whenever a load or store instruction uses a particular effective address. Two new instructions are added, `setlsc` (type I) and `getlsc` (type R). Instruction `setlsc r1,r2+#3` sets the effective address to `r1` and initializes the counter to `r2+#3`. Instruction `getlsc r3` copies the LSC to `r3`.

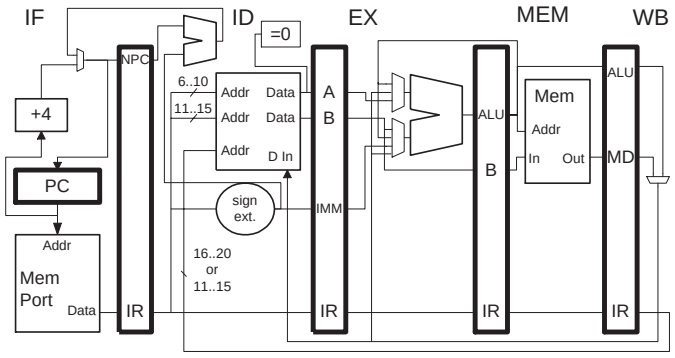
For example, consider the code below. The first instruction, `setlsc`, initializes the count to zero and sets the address to watch to the value of `r1`. When `lw` executes the count will be incremented because the addresses match. If `r1=r10-8` then the count will be incremented again when `sw` executes, otherwise the count will remain at one. The `getlsc` instruction will load `r2` with a two, if `r1=r10-8`, or a one, otherwise.

```
setlsc r1, r0+#0
lw r5, 0(r1)
sw 8(r10), r11
getlsc r2
add r3, r3, r2
add r4, r4, r2
```

(*a*) Show the changes necessary to add the two instructions to the pipeline below. (The illustration and program are repeated on the next page for convenience.)(20 pts)

- Include the hardware, including control, needed to set, increment, and read the LSC.
- The code above (and any other valid DLX program) must execute correctly and with as few stall cycles as possible.
- The solution can use basic gates, multiplexors, instruction recognizers (`=getlsc`, `=setlsc`, `=load/store`, etc.), equality testers (`=`), incrementers, and similar logic.
- Equality testers produce their output in $\frac{1}{2}$ cycle, instruction recognizers produce an output in much less than one cycle.

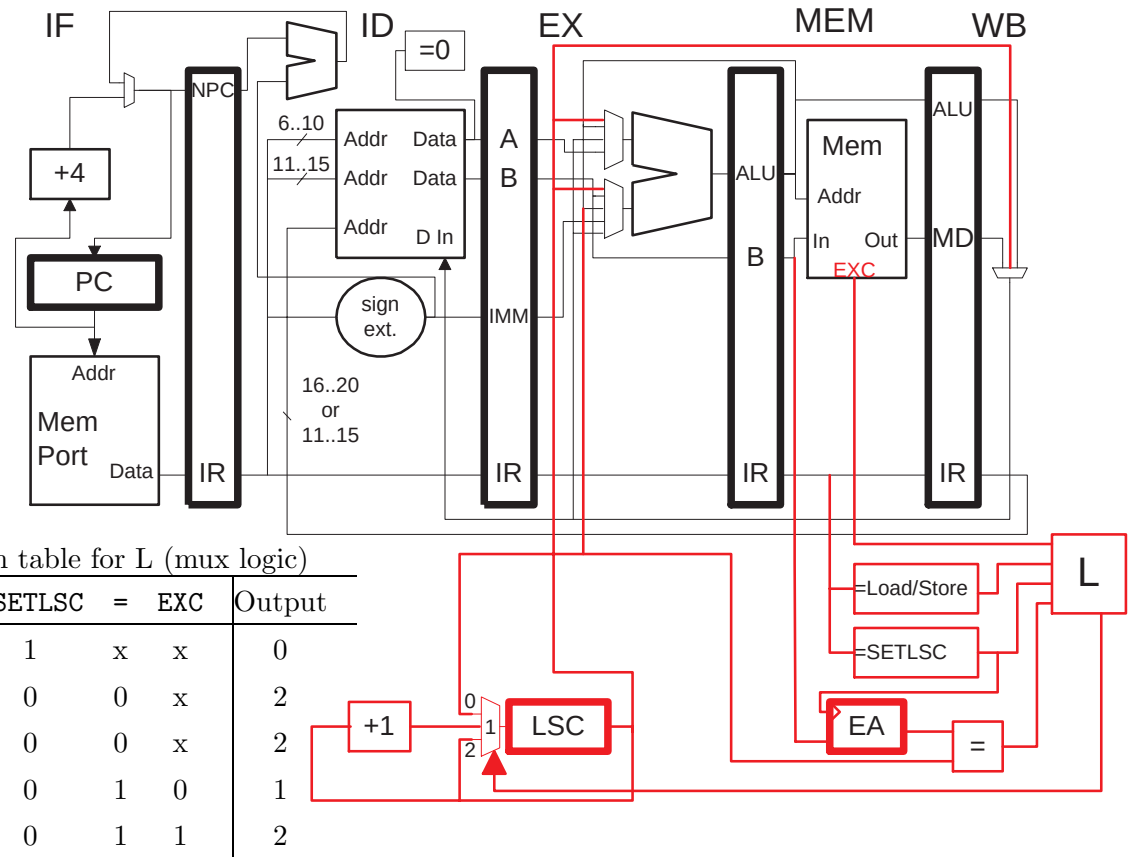
(*b*) If the solution above works correctly when instructions raise exceptions you've solved this part too. Otherwise, explain how an exception could result in incorrect execution and how it might be fixed. (Of course, the pipeline handled exceptions correctly before the changes for the first part.) (5 pts)



Use the next page for the solution.

Problem 1, continued:

```
!Cycle              0   1   2   3   4   5   6   7   8   9
 setlsc r1, r0+#0   IF  ID  EX  MEM WB
 lw r5, 0(r1)           IF  ID  EX  MEM WB
 sw 8(r10), r11             IF  ID  EX  MEM WB
 getlsc r2                      IF  ID  EX  MEM WB
 add r3, r3, r2                     IF  ID  EX  MEM WB
 add r4, r4, r2                         IF  ID  EX  MEM WB
```



Truth table for L (mux logic)

=L/S	=SETLSC	=	EXC	Output
x	1	x	x	0
0	0	0	x	2
1	0	0	x	2
1	0	1	0	1
1	0	1	1	2

Added portions shown above in red. The truth table describes the control logic for the LSC multiplexor. Logic is not shown for the ALU-input and WB-stage multiplexors. The EXC output on the memory is 1 when the memory operation raises an exception.

Operation: the added hardware includes two registers, LSC and EA, both are in the MEM stage.

When a `setlsc` instruction is in the MEM stage the effective address will be in the `EX/MEM.B` latch and the count value will be in `EX/MEM.ALU` latch. These values will be clocked into the LSC and EA registers at the end of the cycle.

When a load or store instruction is in the MEM stage the effective address is in the `EX/MEM.ALU` latch; it is compared to the current value in EA, and based on the comparison LSC is either loaded with the same or an incremented value. (LSC could also have been implemented using a loadable counter.)

When a `getlsc` instruction is in the WB stage the current value of LSC will be selected from the WB-stage multiplexor.

LSC values can be bypassed back to the ALU using the existing WB-to-ALU path and a new LSC-to-ALU path. The new bypass path would be used by the first instruction after `getlsc` in the code above (while `getlsc` is in MEM and the

`add` is in **EX**). The existing bypass path is used by the second `add`.

If a load or store instruction raises an exception the memory's **EXC** output is 1, the **LSC** multiplexor will select the unchanged value (input 2). If the incremented value were selected (that is, if the exception were ignored), then **LSC** would be incremented and incremented again when the instruction re-executes. The resulting value would be too high.

Correct operation can be verified by examining the pipeline execution diagram. For example, `setlsc` initializes **LSC** before it is incremented by the following `lw` and after it might have been incremented by any preceding instructions.

Problem 2: Provide pipeline execution diagrams for the code and machines indicated below. All machines have the following features in common:

- Dynamically scheduled processor using register renaming with a reorder buffer. Registers named using reorder buffer entries.
- Branch and branch target prediction (but no branch folding).
- Speculative execution past predicted branches with the reorder buffer used for misprediction recovery.
- Functional unit inputs connect to CDB (and reservation stations).
- Functional units and reservation stations are as listed below. The quantity of functional units is given for both an ordinary single-issue machine and a 4-way superscalar machine.

Quantity Single	Quantity 4-Way	Functional Unit	Abbr.	Latency	Initiation Interval	Reservation Station Nums
1	1	Load/Store	L	1	1	0-1
1	4	Integer	EX	0	1	2-3,13-15
1	2	F.P. Add	A	1	1	4-6
1	1	F.P. Mul.	M	7	2	7-8
1	1	Branch	BR	0	1	9-10
1	1	F.P. Divide	DIV	22	23	11-12

(*a*) Show the execution of the code below on a single-issue (not superscalar) machine as described above.

The branch is predicted taken and its target is correctly predicted. However, the branch is in fact not taken. Show execution until the last instruction completes. Show when each instruction commits; indicate canceled instructions with an X. (Note: `ltf f0,f3` sets the floating-point condition register to `f0<f3`; `bfpt SKIP` branches to `SKIP` if the floating-point condition is true. The `ltf` instruction uses the FP add unit. (9 pts)

```
!Cycle             0    1    2    3    4    5    6    7   8   9   10   11  12  13  14  15
 addf  f0, f1, f2  IF   ID   A1   A2   WC
 ltf   f0, f3           IF   ID   RS   A1   A2   WC
 bfpt  SKIP                IF   ID   RS   RS   BR   WC
 lf    f0, 0(r1)                                      IF  ID  L1   L2  WC
SKIP:
 addf  f4, f0, f5               IF   ID   A1   A2   WX      IF  ID   RS  A1  A2  WC
 addi  r1, r1, #4                    IF   ID   EX   X           IF   ID  EX  WB      C
```

(*b*) Show the execution of the code below on a single-issue (not superscalar) machine as described above. The branch is correctly predicted not taken. Show execution until the last instruction completes. Show when each instruction commits. Show the state of the reorder buffer when `addf` reaches writeback. (8 pts)

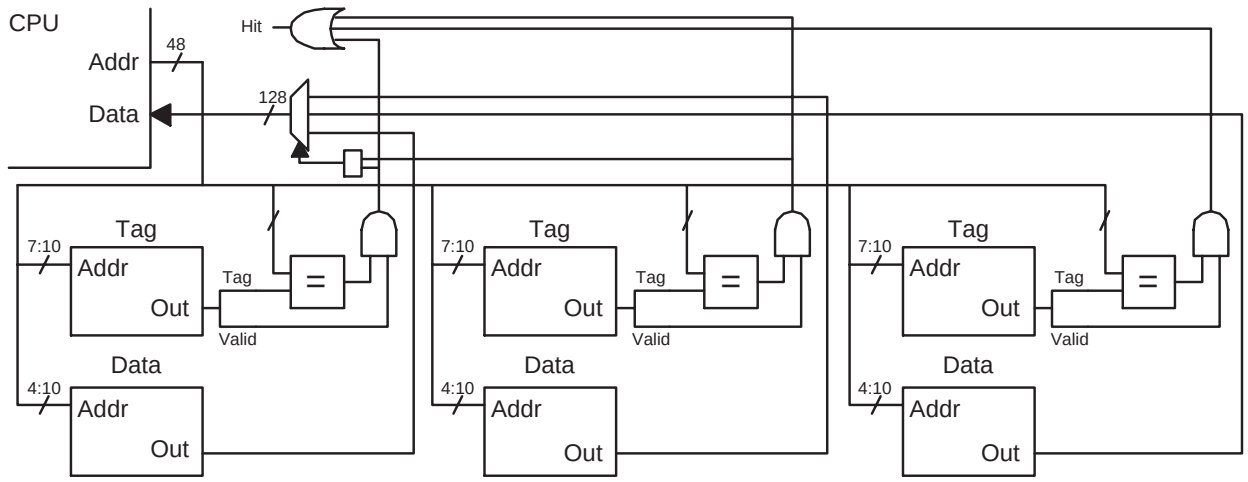
```
!Cycle             0  1   2    3    4    5    6    7    8    9    10   11  12  13 14 15 16
 multf f0, f1, f2  IF ID  M1   M1   M2   M2   M3   M3   M4   M4   WC
 ltf   f0, f3         IF  ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS   A1   A2  WC
 bfpt  SKIP               IF   ID 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS 9:RS  BR  WC
 lf    f0, 0(r1)               IF   ID   L1   L2   WB                            C
SKIP:
 addf  f4, f0, f5                   IF   ID 4:RS   A1   A2   WB                      C
 addi  r1, r1, #4                        IF   ID   EX   WB                           C
```

(*c*) Show the execution of the code below on a 4-way superscalar processor as described above. Instructions are fetched in aligned blocks of four. The branch is correctly predicted taken. All targets are correctly predicted. Show execution until the last instruction completes. Show when each instruction commits. (8 pts)

```
LINE0: ! LINE0 = 0x100c
 beqz r1, LINE1   IF  ID  BR  WC
 addf f0, f0, f1      IF  x
 j LINE2              IF  x
LINE1
 addf f0, f0, f2      IF  ID  A1   A2   WC
 add  r2, r2, r3      IF  ID  EX   WB   C
 and  r2, r2, r4          IF  ID   EX   WC
LINE2
 addf  f0, f0, f3         IF  ID 4:RS   A1    A2   WC
 addf  f5, f5, f0         IF  ID 5:RS 5:RS  5:RS   A1   A2   WC
 addf  f6, f6, f0         IF  ID 6:RS 6:RS  6:RS   A1   A2   WC
 addf  f7, f7, f0             IF   ID 4:RS  4:RS 4:RS   A1   A2  WC
 addf  f8, f8, f0             IF   ID ---------> 5:RS   A1   A2  WC
 and   r5, r6, r7             IF   ID   EX    WB                C
 or    r8, r9, r10            IF   ID   EX    WB                C
 xor   r11, r12, r13               IF ---------->   ID   EX    WB      C
 sub   r14, r15, r16               IF ---------->   ID   EX    WB      C
```

Problem 3:

(*a*) A system is equipped with the cache shown below.



Determine a value for each of the following. **Be sure to specify units (bits, bytes, etc.)** (10 pts)

Character Size (Size of item at a single address.):8 bits

Associativity:3

Block Size:128 bytes or 1024 bits

Number of Sets: 16

Cache Capacity (Amount of data that can be stored.):3× = 6144 bytes

Memory Needed to Implement Cache: $3 \times 16 \times (128 \times 8 + (37 + 1)) =$ 50976 bits = 6372 bytes

The problems on this page are for the cache described below, **not** *the cache from the previous page.*

Consider a system with a 64-bit address space ($a = 64$) which addresses the usual 8-bit (1-byte) characters ($c = 8$) and is equipped with a 2-way set-associative cache with 64-byte lines and 256 sets. The cache uses LRU replacement.

(*b*) Initially the cache is empty. What is the hit ratio for accesses to the array in the code below. Ignore all other memory accesses. (7 pts)

```
extern char *the_array;
// Note: sizeof(char) = 1 byte.
// &the_array[0] = 0x10000
for(i=0; i<16; i++) a += the_array[ i * 8 ];
```

Hit ratio will be $\frac{7}{8} = 0.875$.

(*c*) Choose values for `i_limit` and `stride` so that the program below completely fills the cache with the minimum number of accesses. Assume the cache is initially empty and only include accesses to the array. (8 pts)

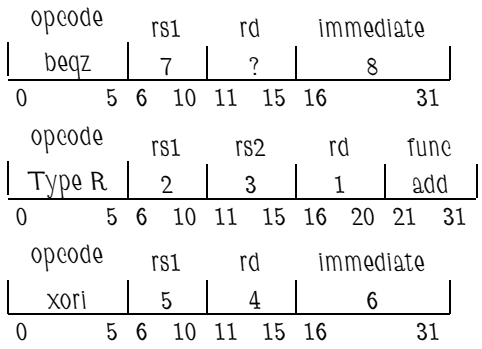
```
extern char *the_array;
// Note: sizeof(char) = 1 character. (What else?)
// &the_array[0] = 0x10000

int i_limit =

int stride  =

for(i=0; i<i_limit; i++) a += the_array[ i * stride ];
```

Solution: `i_limit` = 512 and `stride` = 64.

Problem 4: Answer each question below.

(*a*) Show how the DLX instructions below are encoded. That is, write the instructions as numbers. Make up numbers for opcode and func fields, but use actual values for other fields. *Hint: If you can't remember field sizes look at the illustration for problem 1.* (5 pts)

```
 beqz r7, SKIP
 add r1, r2, r3
 xori r4, r5, #6
SKIP:
```



(*b*) Describe an advantage of VLIW over superscalar processors. Describe a disadvantage of VLIW over superscalar processors. (Use the VLIW ISA described in class.) (5 pts)

Advantage of VLIW: dependencies specified, superscalar machine devotes alot of silicon to finding dependencies. Advantage of superscalar: compatible with existing code.

(*c*) What is a translation lookaside buffer (TLB) and what does it do? (5 pts)

It is a cache indexed by virtual page number that stores real page numbers.

(*d*) Show how an address can be constructed for the branch history table in an (m, n) two-level correlating branch predictor. (Two ways were presented in class, show either one.) (5 pts)

Maintain the outcome of the last m branches in a shift register. Exclusive or the contents of the shift register with the low-order m bits past the alignment bits of the address of the branch instruction being predicted. Voilá.

(*e*) Re-write the program below using DLX instructions. Additional registers may be used. (5 pts)

```
lw  r1,@(r2)          ! Memory indirect addressing.
add r3, r4, (r5)      ! Register indirect addressing.
ld  f0, (0xfedcba01) ! Direct addressing.
```

Solution:

```
lw  r21, 0(r2)
lw  r1, 0(r21)
lw  r25, 0(r5)
add r3, r4, r25
lh  r22, 0xfedc
ld  f0, 0xba01(r22)
```

79 Spring 1998 Solutions

Name _____

Computer Architecture
EE 4720
Midterm Examination

17 March 1998, 19:00–21:00 CST

Modified

Preliminary, Partial Solutions

Problem 1 _____ (42 pts)

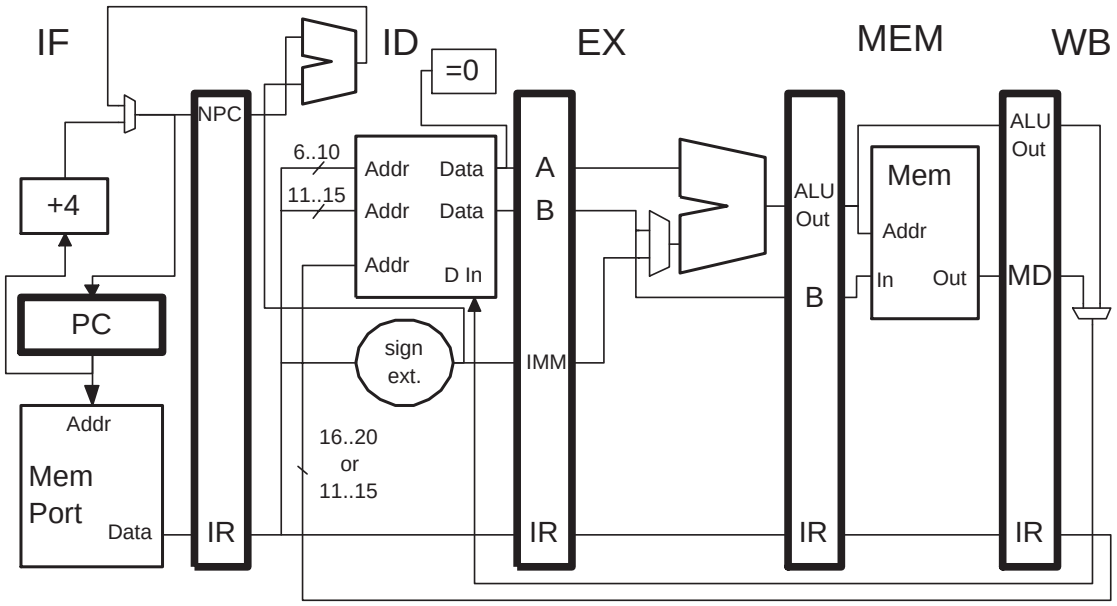
Problem 2 _____ (32 pts)

Problem 3 _____ (26 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Problem 1: The DLX implementation below uses ID-stage address calculation (as illustrated) and bypassing (which is not illustrated) including the branch condition bypassing developed in homework 3. The branches do not include delay slots.




```
! Initial values: r1=2028, r2=0xf11, r3=5, r4=5004, r5=-1000, LOOP=0x3000, MEM[0xf11]=0x97
LOOP:  ! Cycle      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
 lb r1, 0(r2)       IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 add r2, r2, r3         IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 lw r4, 3(r2)               IF  ID  EX  ME  WB          IF  ID  EX  ME  WB

 sub r5, r1, r4                 IF  ID  ->  EX  ME  WB      IF  ID  ->  EX  ME  WB

 bneq r5, LOOP                      IF  ->  ID  EX  ME  WB      IF  ->  ID  EX  ME  WB

 addi r2, r2, #1                            IF                      IF
! Cycle             0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

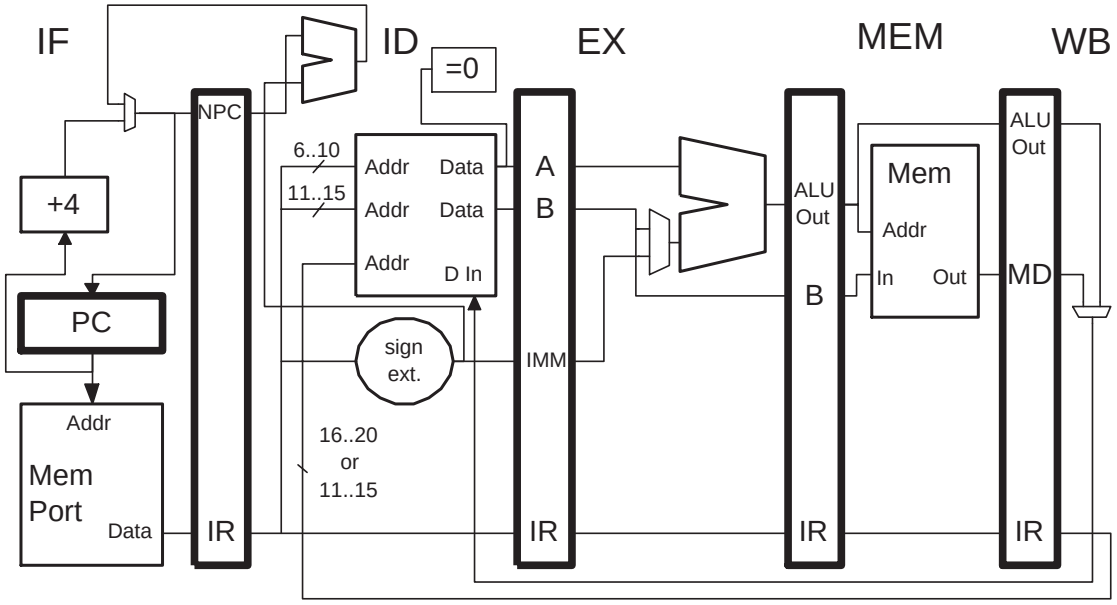
(*a*) Show a pipeline execution diagram for two iterations of the code on the DLX implementation. *Don't forget, bypass paths are present but not shown.* The space to the right of the program or a separate sheet may be used. A description of some mnemonics appears on the next page. (7 pts)

(*b*) On the figure above show exactly those bypass paths needed to execute the code. (7 pts)

(*c*) What is the CPI while executing the loop (assuming a large number of iterations)? (7 pts) $7/5 = 1.4$ CPI.

(*d*) Show the contents (values, not functions of register names) of the pipeline latches at the middle of the first cycle that `lb` is in WB. Include PC and NPC. For IR contents, just show the mnemonic; if an IR contains a nulled instruction show the original mnemonic and "(nulled)." Show the values on the diagram above, write the value within the stage to which it applies with an arrow pointing to the latch or register. (7 pts)

(*e*) Show a pipeline execution diagram of two iterations of the loop below on the DLX implementation illustrated (it's the same as the earlier one). As before, bypass paths are provided but not shown, including hw3 bypass paths. The target of the `beqz r2, SKIP2` instruction is really just the next instruction. Assume that no special hardware optimizations have been made. (7 pts)




```
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20

 addi r1,r0,#0x200  IF  ID  EX  ME  WB
SKIP1:
 andi r2, r1, #1        IF  ID  EX  ME  WB      IF  ID  EX  ME  WB  IF

 beqz r2,SKIP2              IF  ID  EX  ME  WB      IF  ID  EX  ME  WB
SKIP2:
 subi r1, r1, #1                IF  IF  ID  EX  ME  WB  IF  ID  EX  ME  WB

 bneq r1,SKIP1                          IF  ID  EX  ME  WB  IF  ID  EX  ME  WB

 xor  r1, r1, r1                            IF
! Cycle            0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

(*f*) Compute the CPI of the execution of the loop above for a large number of iterations. (7 pts)

Note that first branch executes every other iteration, so CPI should be based on two consecutive iterations. $\frac{11}{8} = 1.375$ CPI.

For Reference:

Mnemonic	Description
`addi`	Add immediate
`andi`	Logical "and" immediate
`beqz r1, TARGET`	Branch if `r1` equals zero.
`bneq r1, TARGET`	Branch if `r1` not zero.
`lbu`	Load byte unsigned.
`lb`	Load byte.
`lw`	Load word.

Problem 2: Consider an ISA, DLXPC, which is like DLX except it uses a condition code bit and predicated execution. The condition code bit is set by the execution of new integer arithmetic instructions, to zero if the result is zero and to one if the result is nonzero. The ISA also includes predicated arithmetic instructions which complete only if the condition code bit (set by the most recent condition-code setting instruction) is 1, otherwise they have no effect. The mnemonic for a predicated instruction has a "p" appended, *e.g.*, `add_p`, and the mnemonic for an instruction that sets the condition code has a "`c`" appended, *e.g.*, `add_c`. An instruction can be both predicated and set the condition code; for example, `add_pc` only executes if the condition code (set by a previous instruction) is 1, and sets the condition code itself.

(*a*) Using these instructions rewrite the code below so that it uses fewer instructions and registers. Assume that the value in register `r1` is not used after `beqz`.(6 pts)

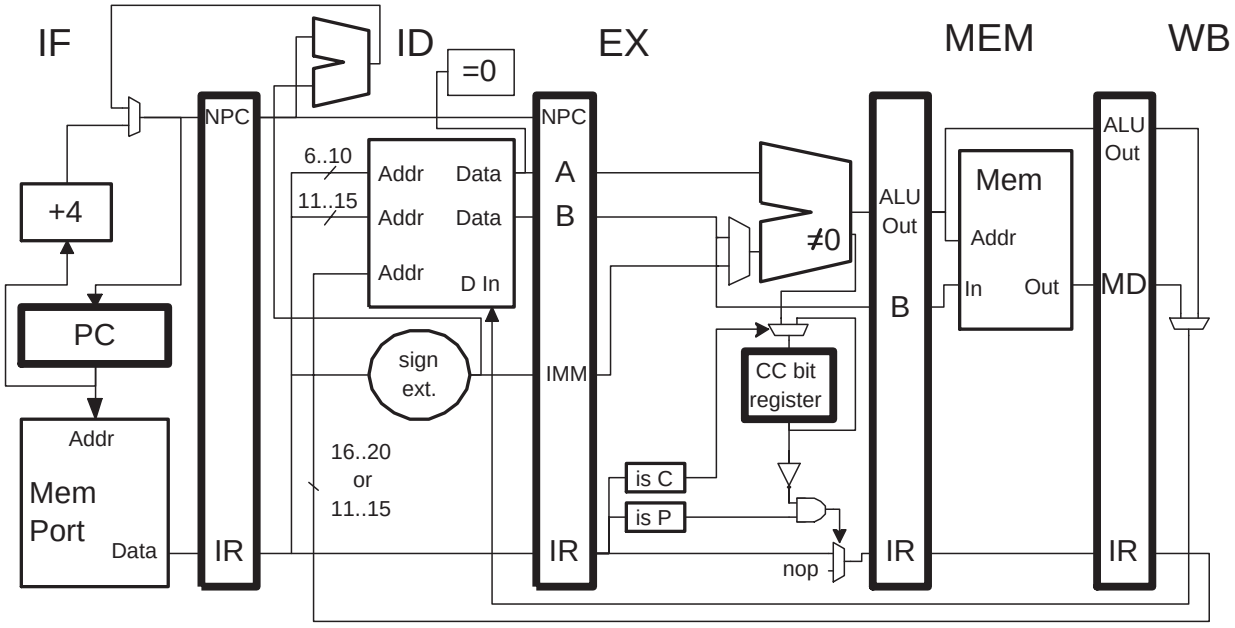
```
 sub r1, r2, r3
 beqz r1, SKIP
 add r4, r5, r6
SKIP:
 addi r4, r4, #1
```

Solution below:

```
sub_c r0, r2, r3
add_p r4, r5, r6
addi r4, r4, \#1
```

(*b*) The pipeline below implements DLXPC—almost: it will not execute predicated, condition-code setting instructions (such as `xor_pc`) correctly if an exception occurs at a certain time. Show code and a pipeline execution diagram that illustrates the problem. Be sure to point out where the exception occurs and what goes wrong. *Hint: the problem would not occur if the execution of some other instructions could be stopped in the cycle that an exception was detected.* (9 pts)

In the figure below the ALU has a second output, which is 1 if the main output is not equal to zero, 0, otherwise. The output of [is C] is 1 if the instruction is a type that sets the condition code (*e.g.*, `add_c`); the output of [is P] is 1 if the instruction is predicated, zero otherwise.



Condition-bit instructions change processor state (the condition bit register) one cycle earlier than other instructions.

```
Cycle   0   1   2   3     4   5
lw      IF  ID  EX  *ME*  WB
xor_pc      IF  ID   EX   ME  WB
```

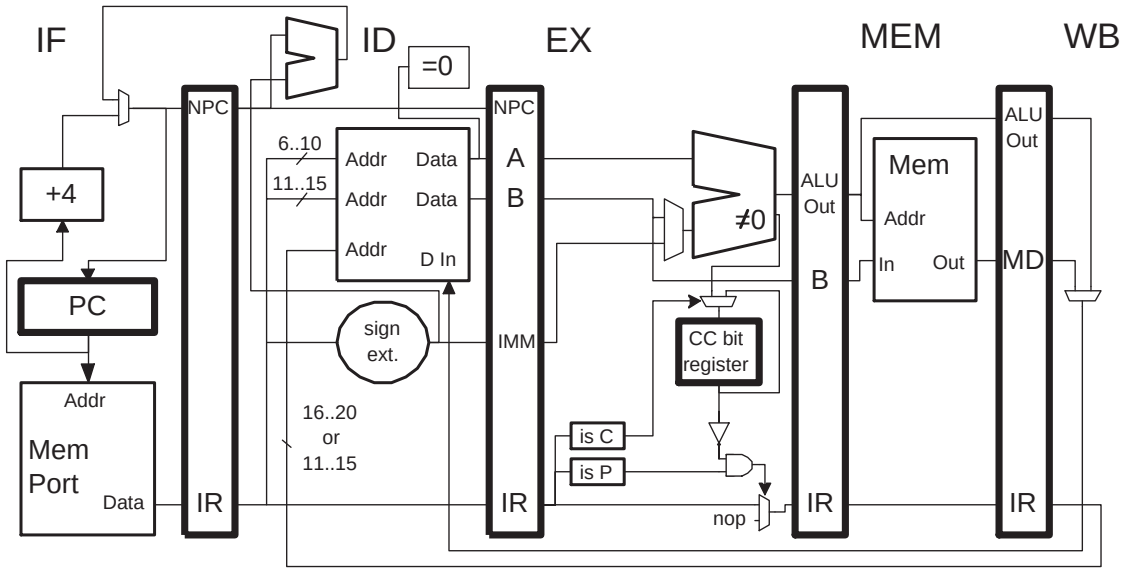
In cycle 3, the lw instruction raises a page-fault exception in the same cycle xor_pc is writing the condition bit. If the condition bit is overwritten then it will be impossible to get the previous value and so when the trap handler returns, xor_pc may not execute, even though the condition bit was 1 (before being overwritten).

(c) Modify the pipeline to fix the problem. (9 pts)

One possibility is to add a condition-bit-in-last cycle register. (This would have the condition bit value that was valid in the previous cycle.) It would be used to provide the old value, details would be needed for a complete solution.

(*d*) Suppose the ISA also includes a new branch instruction, mnemonic `b_p`, that tests the condition bit (instead of a register) and branches if it's true. To implement the branch instruction an ID-stage `take_branch` signal is needed. The signal should be 1 when a `b_p` instruction that will be taken is in the ID stage (zero otherwise). Show how the pipeline below would have to be modified to provide this signal. Be sure that the modified pipeline executes the code below correctly and without adding an unnecessary stalls. (8 pts)

```
sub_c r0, r1, r2  ! Set condition code.
b_p TARGET
add_c r3, r4, r5  ! Set condition code.
add r6, r7, r8    ! Doesn't affect condition code.
b_p TARGET2
```



Outline: In most cases the CC bit could be used to generate `take_branch`, the exception is when the bit is set by the instruction immediately preceding the branch. For that case the input to the condition bit register would be used (perhaps stretching the clock cycle). A multiplexor would select the proper signal, controlled by the [= C] box.

Problem 3: Answer each question below.

(*a*) In class execution time has been modeled using the equation $t = \frac{1}{\phi}\sum_i \mathrm{IC}_i\mathrm{CPI}_i$. Given what has been covered in class so far, to what degree is CPI_i a function of the implementation and to what degree is CPI_i a function of the program? Explain. (9 pts)

CPI is strongly determined by implementation, for example, bypass paths would reduce the CPI. CPI is effected to a small degree by the program, since different instruction orderings might effect their execution. For example, if a program has a load instruction immediately followed by an instruction that uses the loaded value than the CPI_l, where l is the class of `lw` would be larger than a program in which loads were followed by non-dependent instructions.

(*b*) Synthetic instruction `neg r1` assigns register `r1` to the negation of its previous value, that is, `r1 = -r1`. How would such a synthetic instruction be defined in DLX? (8 pts)

Solution: `sub r1, r0, r1`

(*c*) Packed-operand instructions (as found in MMX or VIS) and elaborate procedure call instructions (that perform extensive stack-frame preparation actions including register saves) are similar in that they can replace many individual instructions. Provide contrasting reasons on whether or not such instructions should be added to an ISA, including implementation factors. For example, _____instructions should be added because the implementation _____while _____should not be added because _____though certainly if _____it would _____, but that's not enough[1]. (9 pts)

The procedure call instruction should not be added because not much time would be saved over individual instructions that perform the same actions. A procedure call function would constrain how procedure could be called. Packed operand instructions would (usually) take no more time to execute than ordinary arithmetic instructions. The instructions that a packed-operand instruction replace would take many cycles to execute, so there is a good reason to add them to an ISA.

[1] This is just an example, don't try to fill in the blanks!

Name David M. Koppelman

Computer Architecture
EE 4720
Final Examination

11 May 1998, 10:00–12:00 CDT

Modified

Problem 1 _____ (20 pts)

Problem 2 _____ (30 pts)

Problem 3 _____ (20 pts)

Problem 4 _____ (30 pts)

Alias The Solution Exam Total _____ (100 pts)

Good Luck!

Problem 1: An extended version of DLX is to include a new *morph* instruction, mnemonic `mrph`. Morph takes two arguments, an instruction address and a new instruction. After executing `mrph IADDR INSTR`, when execution reaches IADDR instead of executing the instruction at IADDR, INSTR is executed. This substitution only happens once, if execution reaches IADDR a second time the instruction at IADDR executes normally (unless `mrph` was executed again). For example consider:

```
 mrph POINTA, [subd f0, f2, f6]
 ...
LOOP:
 subi r2, r2, #3
POINTA:
 muld f0, f0, f2  ! On the first iteration subd will execute.
 bneq r2, LOOP
 addd f0, f0, f4
```

In the first iteration of the loop the `subd` instruction specified by `mrph` will be executed instead of the `muld`. After that the loop will execute normally. Morph might be useful for debuggers (the substituted instruction would be a jump to a debug routine).

A system can have at most one morph active at any time. The morph instruction cannot modify memory[1].

(*a*) Determine a format for the morph instruction that fits naturally into the DLX ISA. (An instruction fits naturally if it uses an existing type and its implementation requires little new hardware.) The format should include the type and how the arguments are specified, that is how mrph's arguments relate to IADDR and INSTR. In other words, how is the address specified (not too difficult since many existing DLX instructions specify addresses) and how is the instruction specified (the interesting part). (The three DLX formats types are type-R, used by `add r1,r2,r3`; type-I, used by `addi r1,r2,#3`; and type-J, used by `j LOOP`. Don't confuse the assembly language instruction with the actual instruction format, the assembly language form used above may be misleading.) *Hint: If you're not sure what to do in this part, attempt the next part first.* (5 pts)

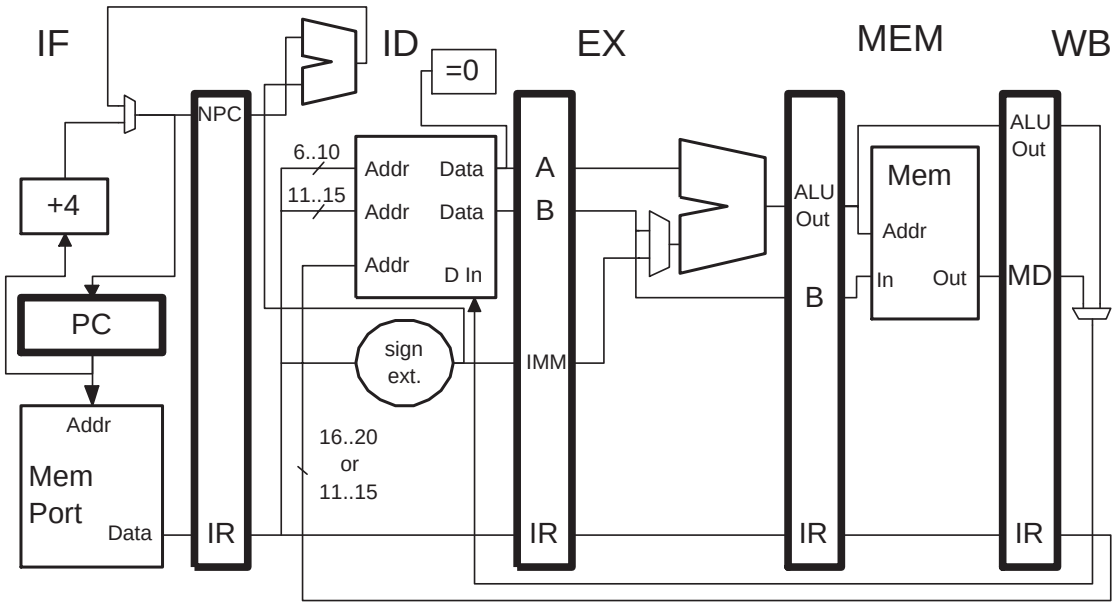
The morph instruction needs another instruction and an address. To fit "naturally" into the DLX ISA mrph would have to be 32 bits, so there would be no way to literally have the address and instruction present. Two options for the address are displacement addressing (as with branches) and register-indirect (as with some jumps). Register indirect is chosen because it does not require addition and because it leaves a source register operand free. The instruction (INSTR) is specified by a register, that is, the instruction itself is stored in a register. Since the type-R format has two source registers, it will be used.

In summary, mrph is a type-R format instruction in which register rs1 holds the address of the instruction to replaced (IADDR) and rs2 holds the instruction to substitute (INSTR).

[1] Morph could be implemented in software by replacing the instruction at IADDR by a jump to a morph routine. The morph routine would execute the substituted instruction and then return.

(*b*) Modify the pipeline below to implement `mrph` using the format chosen above. The pipeline must always execute the code below correctly. (The code itself is correct.) (For partial credit if your design will not execute the code below correctly explain why and how it might be fixed.) (15 pts)

```
 bneq r10,L1
 mrph A,[add r1,r2,r3]
 j L2
L1:
 mrph SKIP,[add r0,r0,r0]
L2:
 ...
 bneq r1, SKIP     ! Hint: May cause trouble.
A:
 sub r1, r2, r3    ! Maybe add r1, r2, r3 substituted.
 lw r4, 0(r1)      ! Hint: May cause trouble.
SKIP:
 addi r4, r4, #12  ! Maybe add r0, r0, r0 substituted.
```



Preliminary: The added hardware performs two actions: the execution of mrph, and substituting the instruction as specified by the last mrph. To execute mrph the presence of mrph is detected in the MEM or WB stage, if present IADDR and INSTR are clocked into special registers added for morph. (The values must be copied because the register contents can change.) A `morphactive` bit is also set. If the steps above were done in ID there would be no way to cancel the execution of mrph, for example if the preceding instruction encountered a page fault.

The hardware that performs the substitution is in the IF stage. The PC address which is being used for instruction fetch is also compared with the stored morph IADDR. If they match and morphing is active then the stored morph INSTR is clocked into the IF/ID.IR register rather than the value retrieved from memory and the `morphactive` bit is reset.

Solution continued on next page.

To implement morph the stage latches have an extra bit, called the M bit. This bit is set to 1 in IF if the substitute instruction is clocked into IF/ID.IR. The bit is passed down the pipeline unchanged. If a substituted instruction is cancelled and the M bit is 1 the **morphactive** bit is set back to one. Two examples of morphed instructions being cancelled appear in the code fragment above. Suppose the branch before **A** is taken after executing the first morph. While the branch is in ID the **sub** will be in IF and will be replaced with the **add**, the M bit will be set to 1, and the **morphactive** bit will be reset. Since the branch is taken the **add** (nee **sub**) will be cancelled in the next cycle. Since the M bit will be one, **morphactive** will be set back to one. Next consider the second morph above. Suppose the **lw** encounters a page fault. The morphed instruction **add** (nee **addi**) will be in EX when the fault occurs and (depending on implementation) in EX or MEM when cancelled. Again, because of the M bit the **morphactive** bit can be set back to 1.

In the original exam, the question asked "what might go wrong in the code above," in the modified exam that was changed to "if your design will not execute the code correctly explain why." Many had misinterpreted the original phrase to mean: "The programmer who wrote the code was careless and so the code has bugs. What did the programmer intend and what will go wrong?" There was nothing in the question to indicate that the program was other than correct, so the interpretation was not correct.

Problem 2: The program below executes on a 2-way, dynamically scheduled superscalar processor. The processor's features are summarized in the list below, the table gives details of the functional units. (The load/store unit computes the address in its first cycle and does the actual access in its second cycle. An operation does not enter the load/store unit until all its operands are ready.)

- ♢ Two-way superscalar instruction issue.
- ♢ Dynamically scheduled (see table), register renaming.
- ♢ CDB can accommodate results from any two functional units in any cycle.
- ♢ Reorder buffer for speculative execution and precise exceptions.
- ♢ Reorder buffer can retire as many as two instructions per cycle.
- ♢ Reservation stations, *not* reorder buffer, used for renaming.
- ♢ Zero-delay branch and branch target prediction. **No** branch folding.
- ♢ Branches do not have delay slots.

Name	Abbreviation	Latency	Initiation Interval	Reservation Station Nums
Load/Store	L	1	1	0-1
Integer	EX	0	1	2-3
F.P. Add	A	1	1	4-5
F.P. Mul.	M	5	2	6-7
Branch	BR	0	1	8-9
F.P. Divide	DIV	22	23	10-11

```
! When loop first entered r2-r1 large (loop iterates many times).
LOOP: ! LOOP = 0x1000
 lf   f0, 0(r1)     ! Don't overlook true dependencies on f0!
 mulf f0, f0, f2
 addf f0, f0, f3
 sf   4(r1), f0
 addi r1, r1, #8
 slt  r3, r1, r2
 bnez r3, LOOP
 div  f4, f5, f6
```

(*a*) Using the grid on the next page show the execution of the code above up to and including the last cycle shown on the grid. Assume perfect branch and branch target prediction, and no cache misses. Include instructions even if they have not yet finished executing at the end of the grid. (10 pts)

(*b*) Either determine and justify the CPI of an execution of a large number of iterations of the loop (ignoring cache misses and assuming perfect target prediction) or explain why it cannot easily be determined from the pipeline execution diagram. (A correct explanation of why it cannot be easily determined will get full credit, a correct CPI that left no time for problems 3 and 4 will get full credit for this subproblem and sympathy—but not credit—for omitting the others.) (4 pts)

In the execution shown on the next page (solution to first part) no two iterations are the same, thus the CPI is not easily determined given the severe time constraints imposed.

```
! First & Third Iteration
Cycle:          0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21

lf   f0, 0(r1)   IF  ID 0:L1 0:L2 0:WB
                                                          IF   ID ------------> 1:LS 1:LS 1:WB
mulf f0, f0, f2  IF  ID 6:RS 6:RS 6:M1 6:M1 6:M2 6:M2 6:M3 6:M3 6:WB
                                                          IF   ID ------------> 6:RS 6:RS 6:M1 6:M1 6:M2 6:M2 6:M3
addf f0, f0, f3      IF   ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:A1 4:A2 4:WB
                                                               IF  ------------>  ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS
sf   4(r1), f0       IF   ID 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:L1 1:L2 1:WB
                                                               IF   ------------> ID --------> 1:RS 1:RS 1:RS 1:RS
addi r1, r1, #8           IF   ID 2:EX 2:WB
                                                                                   IF -------->   ID 2:EX 2:WB
slt  r3, r1, r2           IF   ID 3:RS 3:EX 3:WB
                                                                                   IF -------->   ID 3:RS 3:EX 3:WB
bnez r3, LOOP                  IF   ID 8:BR 8:WB
                                                                                                  IF   ID 8:BR 8:WB
dif fr,f5,f6                   IF   ID x
                                                                                                  IF   ID X
! Second & Fourth Iteration
Cycle:          0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21

lf   f0, 0(r1)                 IF   ID 0:LS 0:LS 0:WB
                                                                                                       IF   ID 0:L1
mulf f0, f0, f2                IF   ID 7:RS 7:RS 7:M1 7:M1 7:M2 7:M2 7:M3 7:M3 7:WB
                                                                                                       IF   ID 5:RS
addf f0, f0, f3                     IF   ID 5:RS 5:RS 5:RS 5:RS 5:RS 5:RS 5:RS 5:A1 5:A2 5:WB
                                                                                                            IF   ID
sf   4(r1), f0                      IF   ID --------> 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:L1 0:L2 0:WB
                                                                                                            IF   ID
addi r1, r1, #8                          IF --------->   ID 2:EX 2:WB
                                                                                                                 IF
slt  r3, r1, r2                          IF --------->   ID 3:RS 3:EX 3:WB
                                                                                                                 IF
bnez r3, LOOP                                            IF   ID 8:BR -->  8:WB

dif fr,f5,f6                                             IF   ID x
Cycle:          0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21
```

(*c*) Suppose the second time the load (`lf`) executes it triggers a page fault exception, perhaps due to a bad load address.

In what cycle will the exception occur? Show the contents of the reorder buffer at that cycle and place a check mark next to instructions that have completed execution.

Explain how the reorder buffer will allow the exception to be precise. Specify what happens to the reorder buffer as a result of the exception, when it happens, and the contents of the buffer before and after it happens. At what cycle will the trap be inserted in the pipeline? (If solutions to the previous parts are not ready, make up a pipeline execution diagram just for this question.) (8 pts)

The exception occurs in cycle 7, when the second execution of `lf` is in L2 (MEM).

The reorder buffer at cycle 7 appears to the right, with oldest instructions at the bottom. The faulting instruction is marked with a ⋆.

addf
mulf
lf ⋆
bnez √
slt √
addi √
sf
addf
mult

The reorder buffer maintains precise exceptions by insuring that all instructions before the faulting instruction (`lf` in this case) complete while ensuring that no instructions following the faulting instruction can change anything. Permanent changes are made when instructions retire (write results and leave the reorder buffer). Retirement must proceed in program order and can only occur after the instructions complete. If an instruction at the bottom of the reorder buffer has faulted, all following instructions are deleted and a trap is inserted in the pipeline. It's important that the faulting instruction reach the bottom of the reorder buffer before inserting the trap so that the trap starts when all preceding instructions have committed. Based on the two-instruction-per-cycle retirement rate the `lf` reaches the bottom of the reorder buffer at cycle 16, the contents before all following instructions are deleted is shown to the right.

addf
mult
lf
bnez √
slt √
addi √
sf
addf √
mulf √
lf ⋆

(*d*) What are the minimum number of reservation stations of each type needed to attain a minimum CPI on the code above? What is that CPI? (There is a grid on the next page to work this out, other methods may be faster.)(8 pts)

The number of reservation stations is found by completing a pipeline execution diagram up to when every instruction in the first iteration finishes. The minimum number of reservation stations (without stalling execution) is used. The numbers are: L, 4; M, 3; A, 3; EX, 2; DIV 0, and BR 1. With at least these many reservation stations execution will proceed at 4/7 CPI.

First Iteration, Third Iteration, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`lf f0,0(r1)`																					
`mulf f0,f0,f2`																					
`addf f0,f0,f3`																					
`sf 4(r1),f0`																					
`addi r1,r1,#8`																					
`slt r3,r1,r2`																					
`bnez r3,LOOP`																					
`div f4,f5,f6`																					

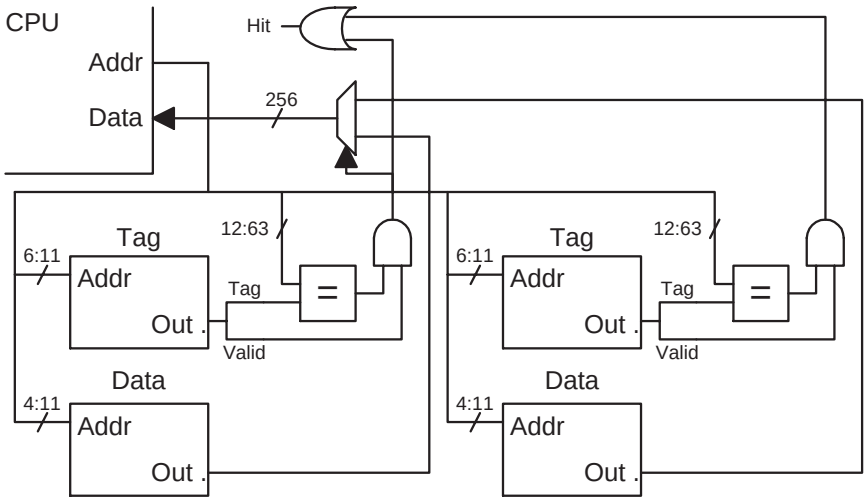
Second Iteration, Fourth Iteration?, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`lf f0,0(r1)`																					
`mulf f0,f0,f2`																					
`addf f0,f0,f3`																					
`sf 4(r1),f0`																					
`addi r1,r1,#8`																					
`slt r3,r1,r2`																					
`bnez r3,LOOP`																					
`div f4,f5,f6`																					

Problem 3: A system has a 64-bit address space ($a = 64$), addresses 16-bit characters ($c = 16$), and has a 256-bit data bus ($w = 256$).

(a) Show how memory devices could be connected to construct a 2-way set-associative cache with 1024-bit lines and 64 sets. Memory devices of any size can be used, be sure to specify their sizes (*e.g.*, $x\,\mathrm{b} \times 2^y$). Show only the connections needed to retrieve the data and tag information, determine if the access is a hit, and pass the data to the CPU. (10 pts)

Each of the two data memories are 256×2^8 bits. Each of the two tag memories are 53×2^6 bits.



(b) Suppose the cache is write-through. What is the capacity of the cache and how much memory is needed to implement it? Be sure to specify units. (5 pts)

The capacity of the cache is $1024 \times 2 \times 2^6 = 2^{17} = 131072$ bits or $2^{14} = 16,384$ bytes or $2^{13} = 8192$ characters. To implement it $(52 + 1) \times 2 \times 2^6$ additional bits are needed. (No storage is needed for a dirty bit.)

(c) Find the addresses specified below. (5 pts)

- *Three* different address that are part of the same line and require exactly two loads to access. (Assume there are 256-bit load instructions.)

All have identical index and tag, two have same offsets but different alignment: For example: 0x0, 0x1, 0x10.

- Two addresses that can be in different lines but the same set.

Same indices, different tags. For example: 0x0, 0x1000.

- Three addresses that are in different sets.

Different indices. For example, 0, 0x40, 0x80.

Problem 4: Answer each question below.

(*a*) What is branch folding? In the implementations of DLX covered in class, why must the predictions used for branch folding always be correct? (6 pts)

Branch folding is the substitution of a branch target for a branch so that the branch is never seen further down the pipeline. It is implemented using a branch target buffer which holds the predicted target address *and* the target instruction. The prediction must be correct because in the DLX implementations a folded branch instruction does not get a chance to execute so there is no way to detect an incorrect prediction. Because predictions must always be correct branch folding is only appropriate for unconditional branches, which are usually called jumps. Systems with special branch units and flexible issue hardware can fold conditional branches, the branches execute in the branch unit leaving a bubble (wasted issue slot).

(*b*) Why might the cost of a functional unit with an initiation interval of 2 be less than one performing the same operations but with an initiation interval of 1 but having the same latency? Given such a cost relationship, what should the minimum number of reservation stations be for a functional unit with an initiation interval of ι and a latency of λ? Explain. (6 pts)

With an initiation interval of 2 only half the number of functional-unit segments may be needed, data would make two passes through each segment. We want to keep all the functional-unit segments busy (we could have gotten a cheaper functional unit with fewer segments, but we didn't so we must have had a good reason) so we would need at least $\lceil(\lambda+1)/\iota\rceil+1$ reservation stations. The $+1$ is for write back.

(*c*) What are some difficulties that might be encountered in developing a superscalar implementation of a stack ISA? (For partial credit, list some distinguishing features of a stack ISA.) (6 pts)

Stack ISAs typically have a variety of instruction sizes, making it difficult to fetch and decode several of them simultaneously. Code for stack ISAs have many consecutive instructions with true dependencies, forcing an implementation to rely on dynamic scheduling.

(*d*) Explain how a reservation register can be used to detect MEM-stage structural hazards while an instruction is in the ID stage. (6 pts)

A reservation register is a shift register indicating the status of a resource, in this case the MEM stage. Each position in the register is for a different number of cycles in the future. At each cycle the shift register is clocked, maintaining the time relationship. An instruction in ID checks the reservation register to see if the MEM stage will be free when it needs it. If so, it writes a 1 corresponding to the time it will be using MEM and will move to its functional unit in the next cycle. If not, it waits for the next cycle to try again (stalling the pipeline).

(*e*) What is the difference between a RAW hazard and a true dependency? (6 pts)

Pipelines have hazards, instructions have dependencies.

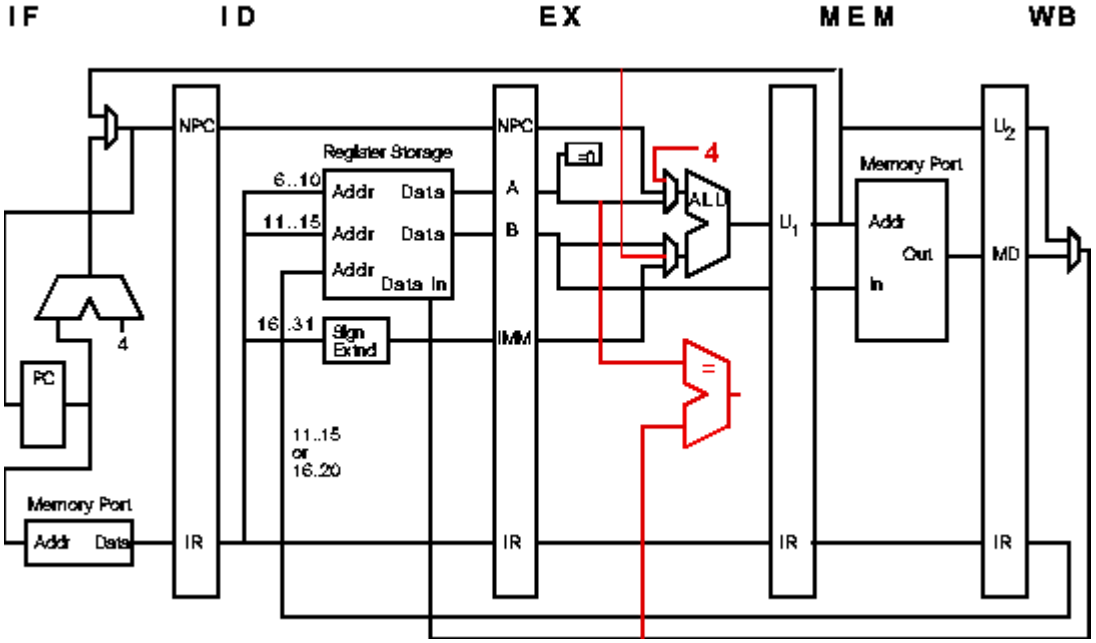
80 Spring 1997 Solutions

EE 4720 1997 Midterm Exam Solution

Problem 1

Part (a)

The modified pipe below implements a high-speed version of mems, the modified portions are shown in red.



mems executes as follows:

```
      0   1   2    3     4          19    20
mems  IF  ID  EX1  MEM1  MEM1  ...  MEM1  MEM1  MEM2  WB
                   EX2   EX2        EX2   EX3
```

As usual, the instruction is fetched in IF. In ID rs1 (e.g., r1) is loaded into latch buffer A and rs2 (e.g., r2) is loaded into B. The EX stage is used three different ways, EX_1, EX_2, and EX_3. In EX_1 the address, rs2, is passed through the ALU unchanged. The ID/EX latches are not clocked until the cycle after the element is found, so that rs1 is held in the B buffer. (rs2 is also held, but is not needed.) In MEM_1 the element is retrieved from memory while at the same time the address of the next element is computed in EX_2. Using an added comparison unit, EX_2 also compares the element retrieved in the previous cycle with rs1. The mems instruction will continue using the EX and MEM stages (inserting NOPs into WB and stalling instructions in IF and ID), until the comparison unit signals a match (cycle 19 in the example). In the cycle following a match (20) the execute stage performs EX_3, subtracting 4 to compute the address of the element found in the previous cycle. The address makes its way back to rs2 in MEM_2 and WB.

Part (b)

```
LOOP:
 lw   r3, 0(r2)
 addi r2, r2, #4        ! Words are four bytes long.
 seq  r3, r3, r1        ! Compare. Note that lw stall avoided.
 beqz r3, LOOP
 subi r2, r2, #4        ! Saves a branch inside or before loop.
```

Part (c)

The mems instruction executes for three cycles before reaching MEM_1. MEM_1 is used once for each element fetched, plus 1, for a total of 1001 cycles. Two more cycles complete execution for a total of 1006 cycles.

An iteration of the program above executes as follows (on a implementation that includes bypass paths)

```
      0   1   2   3   4   5   6   7
lw    IF  ID  EX  MEM WB          IF
addi      IF  ID  EX  MEM WB
seq           IF  ID  EX  MEM WB
beqz              IF  ID  EX  MEM WB
```

Each iteration takes seven cycles, one additional cycle is needed after the last iteration to complete the subtraction. The total number of cycles is $7 \times 1000 + 1 = 7001$.

Advanced techniques, covered in chapter 4, will reduce mems advantage.

Problem 2

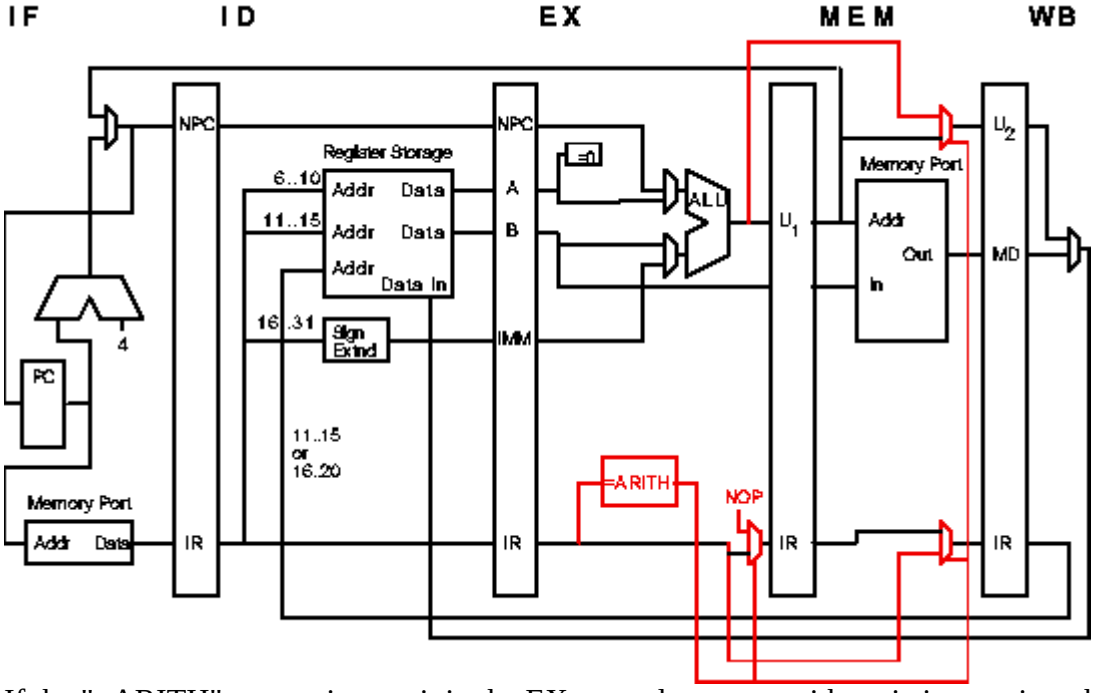
Part (a)

```
     0   1   2   3   4
lw   IF  ID  EX  MEM WB
add      IF  ID  EX  WB
```

The hazard occurs in cycle 4.

Part (b)

The modifications (additions) are shown in red.



If the "=ARITH" comparison unit in the EX stage detects an arithmetic instruction, the IR and ALU outputs are written to the MEM/WB latches, skipping a stage. Note that this replaces whatever was in MEM, resulting in incorrect execution if a WB structural hazard is present.

Part (c)

The structural hazard could be avoided by adding a second write port to the register file (additional changes would have to be made to the part (b) solution) or by having an arithmetic instruction execute normally if skipping the MEM stage would result in a structural hazard.

Part (d)

The benefit of skipping the MEM stage is that registers are written one cycle early. If register forwarding is not used then early writeback reduces the number of stall cycles due to RAW hazards, which is worthwhile. If register forwarding is used, then there is no benefit to early writeback since results from any arithmetic instruction are available to instructions immediately following.

Problem 3

Part (a)

An undefined instruction in an ISA might still do something useful in an implementation of that ISA. (This might happen by accident.) Suppose programmers discover such an instruction in a current implementation and use it in their code. These programs would run fine on the current implementation, but the computer engineers working on the new implementation of the ISA may make no attempt to see that undefined instructions execute the same way. (Why should they, the instructions were undefined and programmers were warned not to use them.) Therefore, computers using the new chip may not be able to run the programs using the undefined instructions.

The problem is avoided if undefined instructions raise an illegal instruction exception.

Part (b)

The target address of a trap instruction is found in a table using an index provided in the instruction. The target address of a subroutine call is given in the instruction, be it PC relative, register indirect, immediate, etc.

The target address of a trap instruction is typically in the system part of the address space while subroutines are in the user part.

As a result of executing a trap, the processor switches to privileged mode, while subroutine calls do not change the processor mode.

Part (c)

All instructions that read registers have the register numbers in the same place, bits 6..10 and 11..15, so they can be fetched without knowing the identity of the instruction. There is no harm in fetching registers for an instruction that does not need them, so registers can be fetched for any instruction and so no decoding is necessary.

Part (d)

Program weights in a geometric mean of normalized execution times (or rates) do not depend on their run time on the base machine, so a particular base machine will not exaggerate or diminish the impact of any program in the suite.

Part (e)

The clock rate might change because of faster technology (e.g., the new fabrication process has a smaller feature size) or because engineers chose to make pipeline segments shorter, allowing for a faster clock.

The instruction count might change in two implementations *of the same ISA* because relative execution times for individual instructions might change (although the instructions available in the old and new implementations are the same), leading programmers to re-write time-critical portions. For example, an integer add might be replaced by a speeded-up floating-point add, perhaps saving integer-to-floating-point and floating-point-to-integer conversion steps.

CPI might change because the number of stall cycles was reduced using advanced techniques, etc.

David M. Koppelman - *koppel@ee.lsu.edu* Modified 17 Mar 1998 17:31 (23:31 UTC)

EE 4720 1997 Final Exam Solution

Problem 1 (a), *EE 4720 1997 Final Exam Solution* Top Next

With a larger-immediate instruction:

```
lw rll, 0(r10)
addli #0x981234
```

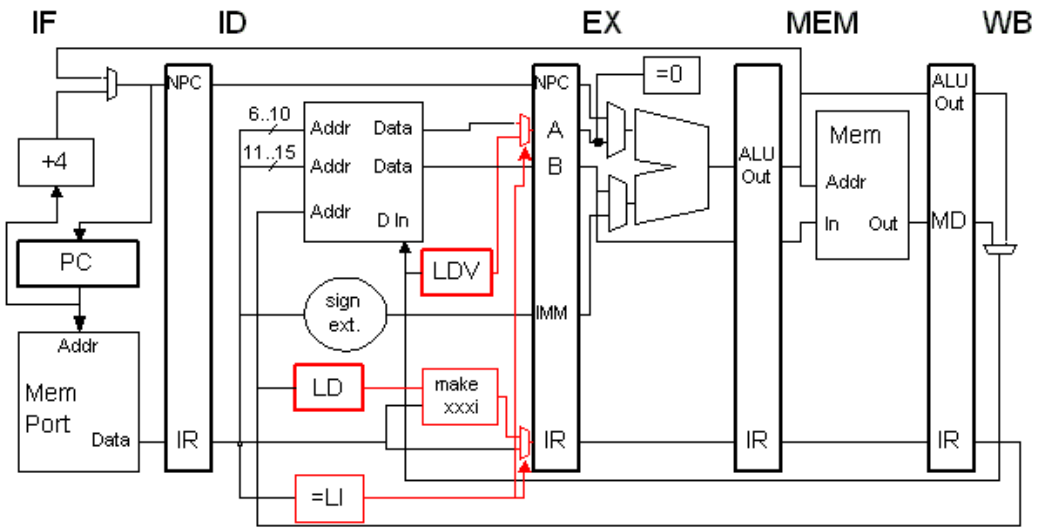
Without larger-immediate instructions:

```
lw r11, 0(r10)
lhi r9, 0x98
ori r9, r9, 0x1234
add r11, r11, r9
```

Problem 1 (b), *EE 4720 1997 Final Exam Solution* Top Previous Next

To implement larger-immediate instructions the processor needs some way of remembering which register was used by the last arithmetic or logical (ALU) instruction. A last-destination register in the ID stage would serve: whenever an ALU instruction is in ID it will write its rd field (the destination register number) into the last-destination register. When a larger-immediate instruction is in ID it will use the contents of last-destination as its destination and rs1 operand register number.

There is one problem with the solution so far: to retrieve operand 1 ID must first determine that a larger-immediate instruction is present (otherwise it will use IR bits 6-10 for the register number). This may lengthen the critical path, so the solution below uses a register to store the last destination.

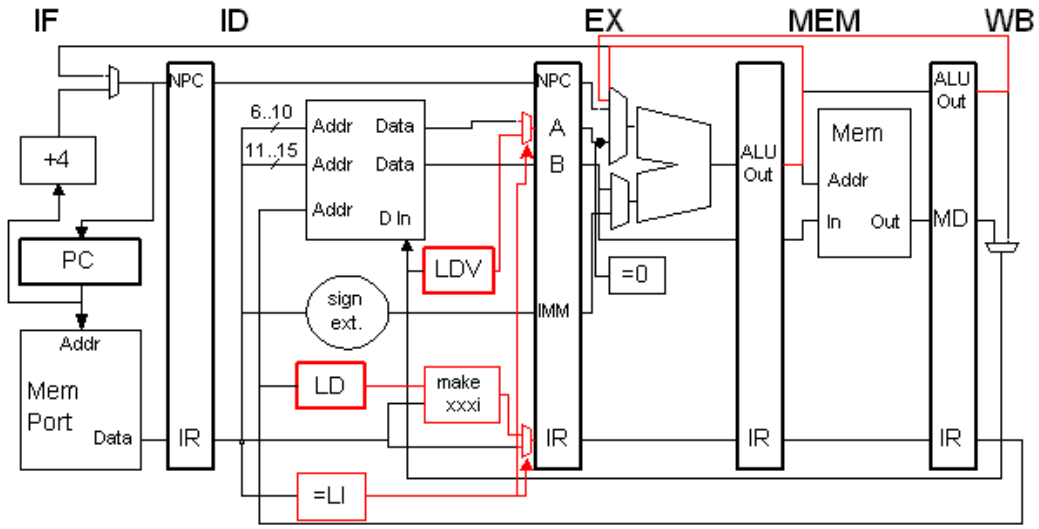


In the solution above, changes are shown in red, and registers have bold outlines. The "=LI" block examines the instruction opcode and returns 1 if it's a larger immediate, 0 otherwise. The "make xxxi" box combines a larger-immediate instruction (or control bits) and the last destination register into a new IR value or control bits; these are used if a larger-immediate instruction is present. The new IR value or control bits would be the same as an immediate type instruction, with the LD contents appearing as rs1 and rd.

LD is the last-destination register, LDV is the last destination value. The controller must only clock the LD and LDV registers during an ALU instruction writeback phase. The sign extension unit, since it is used for J-type instructions, can already sign-extend 26-bit quantities.

Problem 1 (c), *EE 4720 1997 Final Exam Solution* Top Previous Next

The bypassing changes are minor, connections are added from MEM and WB back to the ALU. Unlike general-purpose bypass connections, these only go to the top ALU input.



Consider the execution below:

```
Time              0  1  2  3  4  5
add   r1, r2, r3  IF ID EX ME WB
addli #123456        IF ID EX ME WB
```

In cycle 1 the identity of the destination register, r1, is written to LD. The actual data is written to LDV in cycle 4. The addli instruction arrives in ID during cycle 2. The makexxxi box constructs an "addi r1, r1, #123456" instruction, and the same interlock hardware that detects RAW hazards in regular instructions (not shown) detects the RAW hazard with the newly constructed addi, and sets the control bits needed to use the MEM-to-EX bypass paths (not shown). In cycle 3 the bypass path is used to connect the new r1 to the top ALU input.

A simpler solution would place the LDV and LD registers in the EX stage, avoiding the need for bypass paths.

Problem 2 (a), *EE 4720 1997 Final Exam Solution* Top Previous Next

```
div f0,f10,f11 IF   ID    DV1 DV1 DV1 DV1 DV1 DV1 DV1 DV1 DV1 MF  WBX
mul f0,f3,f2        IF    ID  MU1 MU1 MU2 MU2 MF  WB
add f8,f9,f0              IF  ID              A1  A2  MF  WB
add f0,f1,f2                  IF              ID  A1  A2  MF  WB
mul f3,f0,f4                                  IF  ID      MU1 MU1 MU2 MU2 MF  WB
mul f5,f0,f6                                      IF      ID      MU1 MU1 MU2 MU2 MF  WB
```

Because the initiation interval of the multiply unit is 2, the last multiply instruction must wait an extra cycle. The problem requires that "distinct pipeline segments have distinct names", so the multiply unit's segments are labeled MU1 and MU2. With an initiation interval of 2, an instruction spends 2 cycles in each segment.

Even though its result won't be needed, the divide instruction is allowed to continue but it will not write the register file (indicated by a WBX instead of WX.

Problem 2 (b), *EE 4720 1997 Final Exam Solution* Top Previous Next

```
div f0,f10,f11 IF   ID    7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:WB
mul f0,f3,f2        IF    ID    3:MU1 3:MU1 3:MU2 3:MU2 3:WB
add f8,f9,f0              IF    ID    1:RS  1:RS  1:RS  1:RS  1:A1  1:A2  1:WB
add f0,f1,f2                    IF    ID    2:A1  2:A2  2:WB  2:WB
mul f3,f0,f4                          IF    ID    4:RS  4:RS  4:RS  4:MU1 4:MU1 4:MU2 4:MU2 4:WB
mul f5,f0,f6                                IF    ID    ID    3:RS  3:RS  3:RS  3:MU1 3:MU1 3:MU2 3:MU2 3:WB
```

The last multiply spends an extra cycle in ID, waiting for a free multiply reservation station.

Problem 2 (c), *EE 4720 1997 Final Exam Solution* Top Previous Next

```
div f0,f10,f11 IF   ID    7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:DV1 7:WB
mul f0,f3,f2   IF   ID    3:MU1 3:MU1 3:MU2 3:MU2 3:WB
add f8,f9,f0        IF    ID    1:RS  1:RS  1:RS  1:RS  1:A1  1:A2  1:WB
add f0,f1,f2        IF    ID    2:A1  2:A2  2:WB
mul f3,f0,f4              IF    ID    4:RS  4:RS  4:MU1 4:MU1 4:MU2 4:MU2 4:WB
mul f5,f0,f6              IF    ID    ID                3:RS  3:MU1 3:MU1 3:MU2 3:MU2 3:WB
```

Problem 2 (d), *EE 4720 1997 Final Exam Solution* Top Previous Next

```
div f0,f10,f11 IF   ID      DV1   DV1   DV1   DV1   DV1   DV1   DV1   DV1   DV1   WBX
mul f0,f3,f2   IF   ID      MU1   MU1   MU2   MU2   WB
add f8,f9,f0        IF      ID                      A1    A2    WB
add f0,f1,f2        IF      ID                            A1    A2    WB
mul f3,f0,f4                IF                            ID          MU1   MU1   MU2   MU2   WB
mul f5,f0,f6                IF                            ID                      MU1   MU1   MU2   MU2   WB
```

Problem 3 (a), *EE 4720 1997 Final Exam Solution* Top Previous Next

Bit positions: Alignment 0:2, Offset, 3:7; index, 8:19; tag, 20:63. (In a previous version of this solution bits 0:7 were specified for the offset. That's okay, as long as all of those bits are not used to construct an address for the data part of the cache.)

Cache size: $3 \times 4096 \times 256 = 3$ & $2^{12} \times 2^{8} = 3{,}145{,}728$ bytes.

To implement include tag, valid, and dirty bit storage: 3×4096 (256+(44+1+1)/8)=3,216,384 bytes.

Problem 3 (b), *EE 4720 1997 Final Exam Solution* Top Previous Next

Rewritten program:

```
double *a = 0x10000000;  /* Assume the entire address space is ours. */

/* Block size is 256 bytes or 32 doubles.  Touch consecutive blocks. */
for( i=0; i<4096; i++ ) total += a[ i * 32 ];

/* To fill set, touch two blocks with different tags than above. */
for( i=0; i<2048; i++ ) total += a[ 0x20000 + i * 32 ] + a[ 0x40000 + i * 32 ];
```

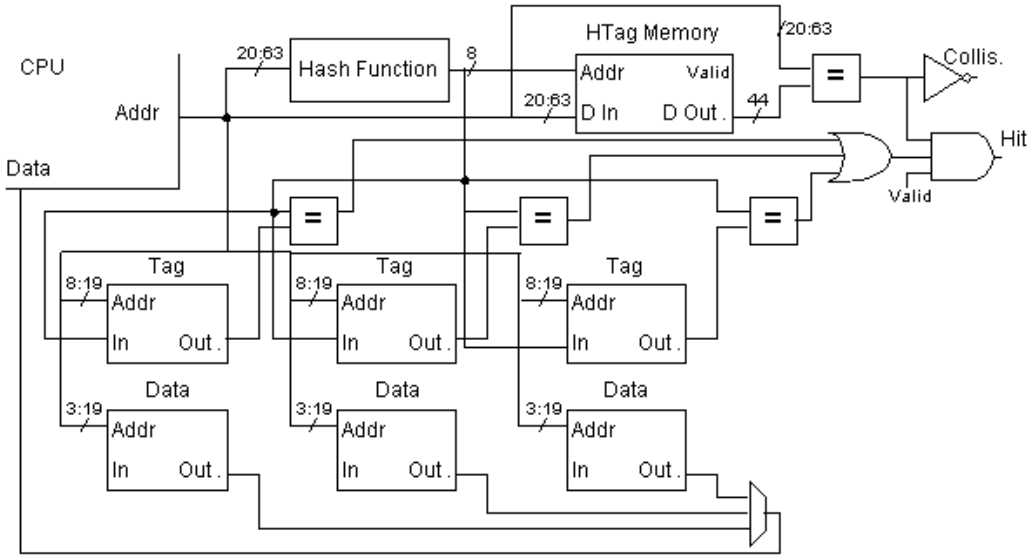
Problem 3 (c), *EE 4720 1997 Final Exam Solution* Top Previous Next

The design is shown in the figure below. The lower part of the illustration shows a conventional cache, the only difference is that the tag memory stores 8-bit HTags rather than the 44-bit tags an ordinary cache would use. The upper half shows the hardware for converting the tag of the accessing address to an HTag. An HTag memory is provided for storing the full 44-bit tags corresponding to each HTag, this is needed to detect collisions.

(1)When an address with a new tag is presented, the lookup in the HTag memory will yield an invalid (perhaps never written) HTag. The new tag will be written to the HTag memory. This could only be a cache miss; the remainder of the accesses proceeds as a normal miss.

(2) If an address is presented with an already cached tag the value retrieved from the HTag memory will be valid and will match the tag of the access address.

(3) When an address collides with a tag the value retrieved from the HTag memory will not match the access address. This means that two different addresses have the same HTag (a risk when squishing 44 bits to 8), one of these is for cached item(s) the other is the current access. This is trouble because those already cached items have to be evicted before the current access can be cached. (Otherwise there would be no way to distinguish them.) The lines to be evicted could be anywhere, so eviction will be time-consuming, perhaps performed by software (using a trap routine).



Not shown is the control input to the data multiplexor. That would be derived from the three HTag comparison boxes (which connect to the or gage). The valid input to the and gate also comes from the tag memories. Note that the collision signal is meaningless when the valid output of the HTag memory is false.

Problem 4 (a), *EE 4720 1997 Final Exam Solution* Top Previous Next

The data to write will not be ready until the end of a cycle and will be needed at the beginning of the next cycle. With many inputs and outputs, it can take longer for a signal to reach its destination on a CDB than on a bypass path, which has fewer inputs and outputs.

Problem 4 (b), *EE 4720 1997 Final Exam Solution* Top Previous Next

A benchmark suite is more appropriate for overall performance evaluation, the engineers might use it to assess the net effect of many changes. Synthetic benchmarks might be designed to test and tune specific hardware features. The engineers might use it to determine why a particular branch prediction scheme is or is not effective.

Problem 4 (c), *EE 4720 1997 Final Exam Solution* Top Previous Next

```
Time 0   1   2   3   4   5
lw   IF  ID  EX  MEM WB!
sub      IF  ID  EX  MEM EX
add          IF  ID! EX! MEM
mul              IF  ID  EX
sw

<trap>               IF  ID
```

The pipeline checks for exceptions in the WB stage. The `add` exception at cycle 2 results in add being nulled and an exception code being passed along with the nulled instruction. (The exclamation point indicates the instruction is in that stage and is nulled because of an exception.) Suppose memory access done by the `lw` instruction results in a page fault; this would occur at cycle 3, and would result in a trap being inserted--for the `lw` instruction--in cycle 4, replacing the `sw` that was being fetched.

Problem 4 (d), *EE 4720 1997 Final Exam Solution* Top Previous Next

It's similar since dependencies must span several instructions so that multiple instructions can be issued in parallel. With superscalar processors that scheduling is optional, unscheduled code would run but with more stalls. VLIW describes an ISA (although an implementation is often what's meant) so code in non-VLIW ISAs must be recompiled. Unlike superscalar processors, VLIW sub-instructions issued in parallel cannot have any dependencies (in correct code).

Problem 4 (e), *EE 4720 1997 Final Exam Solution* Top Previous Next

As *m* increases prediction accuracy improves as a branch's global-history-dependent behaviors can be individually maintained. As *m* increases further, two effects can reduce performance. First, a greater number of branches might share the same entry. Second, a greater amount of irrelevant global history might spread a branches counters thin. With 10 bits of correlation (*m*=10), each branch would have 1024 counters, if the least recent five branches were irrelevant and evenly distributed, then each useful behavior would have to be "learned" by 32 counters.

ECE HOME PAGE 4720 HOME PAGE UP

David M. Koppelman - *koppel@ee.lsu.edu* Modified 13 Apr 1999 11:43 (16:43 UTC)