Name Solution

Computer Architecture
EE 4720
Final Examination

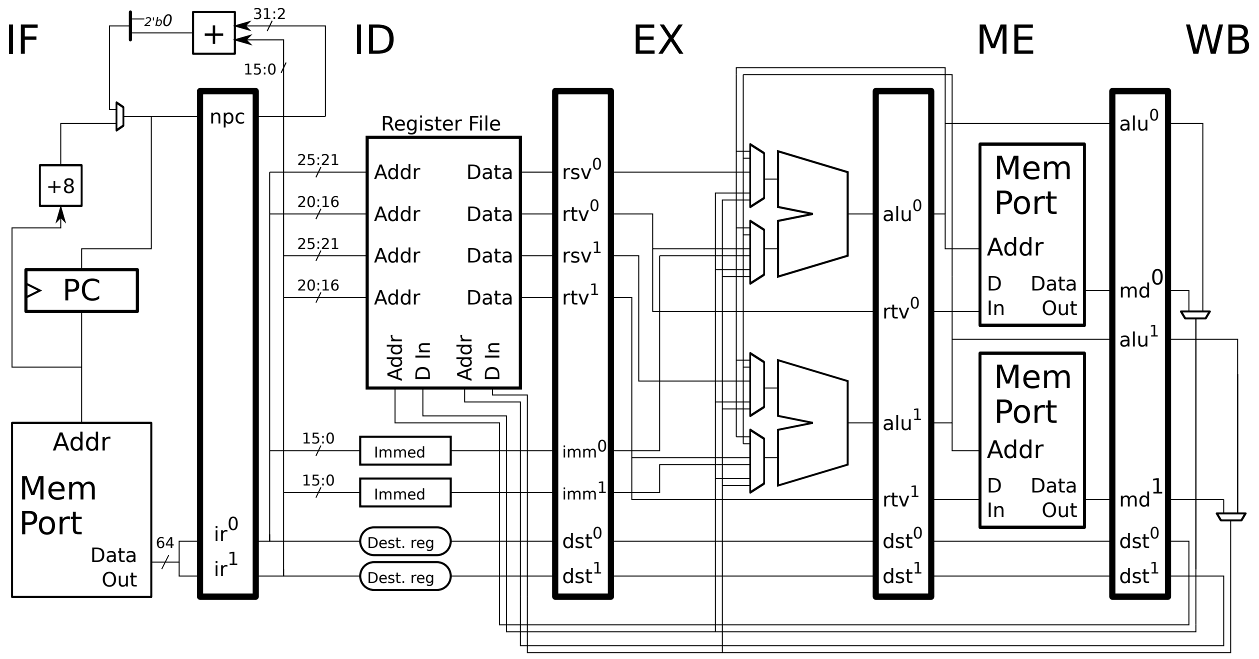
2 May 2018, 15:00–17:00 CDT

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.

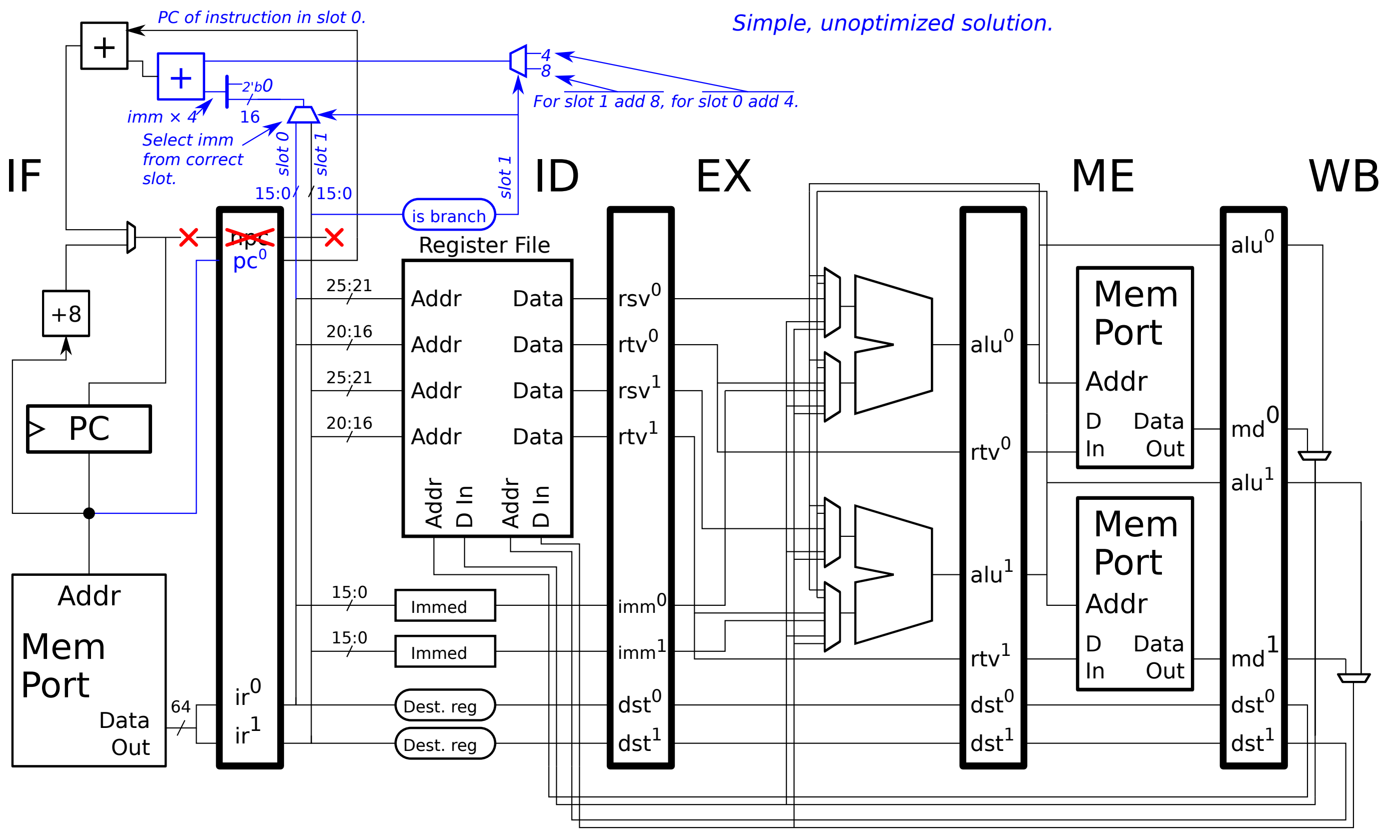
In both solutions the PC, rather than the `NPC` (next program counter) is passed to `ID`. The PC is the address of the slot-0 instruction so it is labeled `pc0` in the diagram. If the branch is in slot 0 then the branch target is computed in the usual way, `pc0 + 4 + imm*4`. If the branch is in slot 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 $\boxed{\text{is branch}}$ 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 $\boxed{\text{is branch}}$ 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.

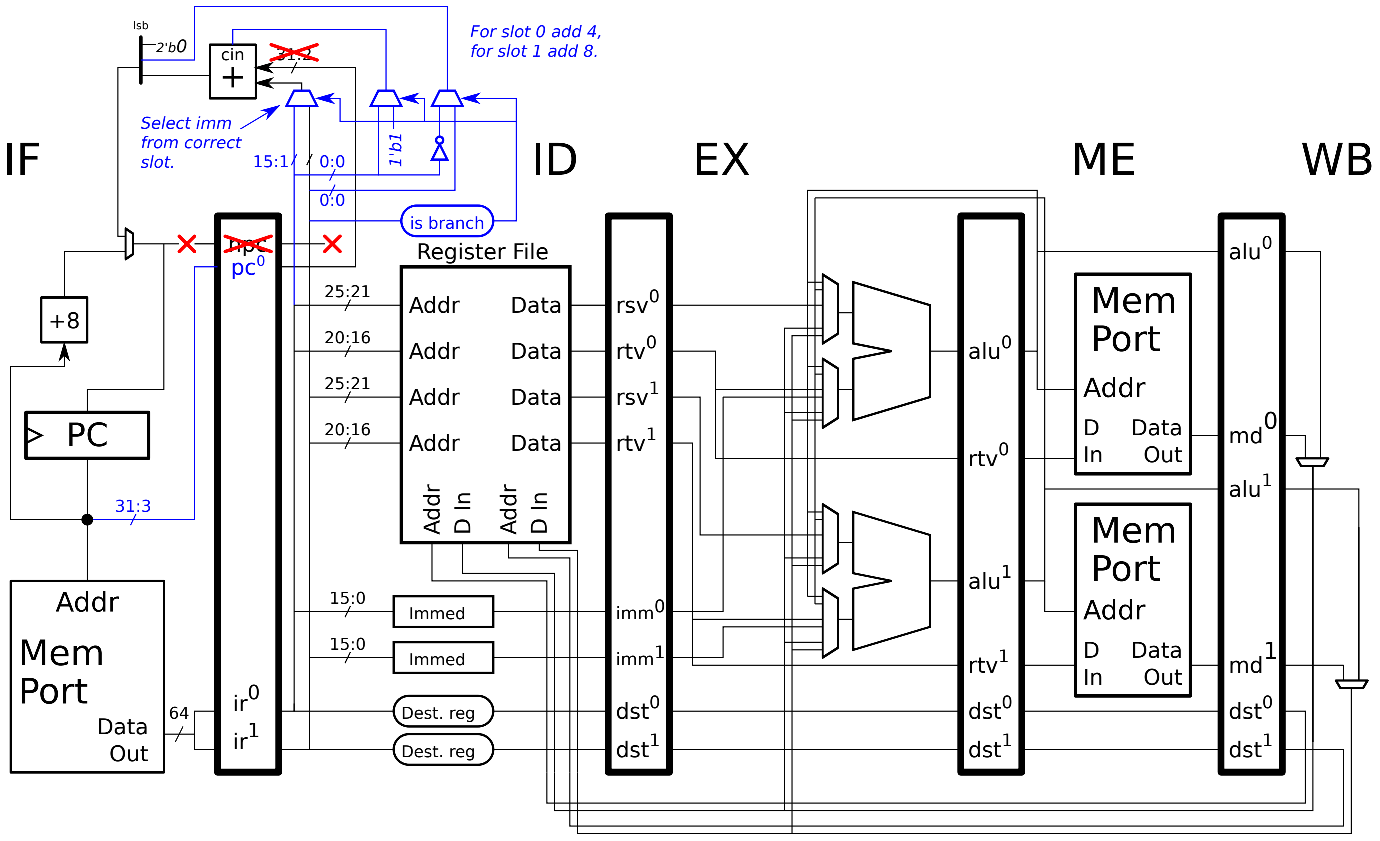


The cost of the solution above can be reduced by eliminating the adder above the `imm` $\times$ 4 label. Notice that the upper input to this adder is either a 4 or an 8. Also notice that `pc0` must be a multiple of 8. Based on these we can eliminate the adder, and instead use the carry-in input to the remaining adder to perform the $+4$ or $+8$.

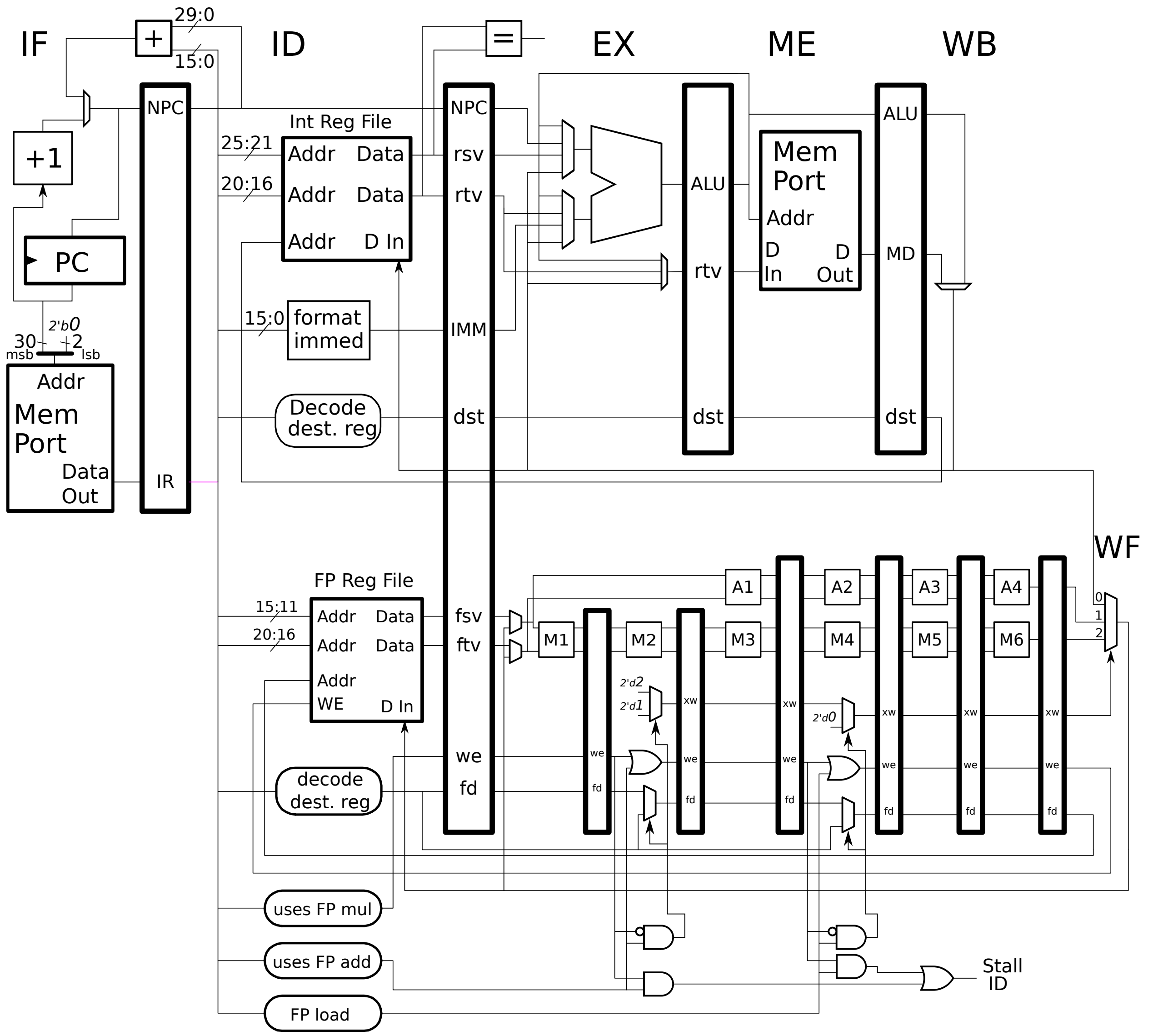
Solution continued on next page.

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.)

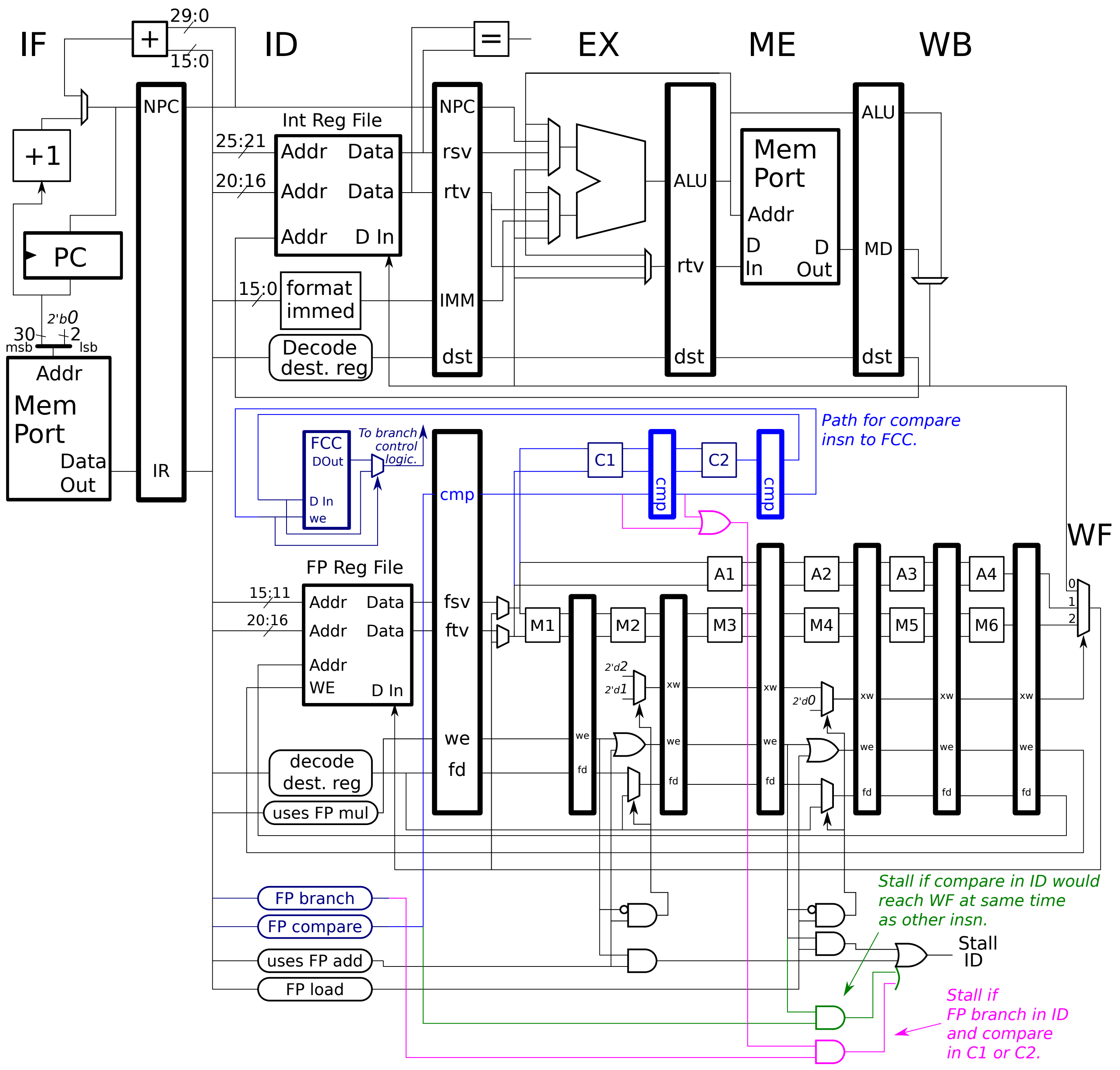
☑ Modify the control logic so that the `Stall ID` signal is asserted for dependencies from compare to branches, such as occurs above with the `bc1f`.

☑ 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
+
=
NPC
IR
+1
PC
2'b0
30
msb
2
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.
cmp
C1
C2
cmp
cmp
Path for compare insn to FCC.
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
M1
M2
M3
M4
M5
M6
A1
A2
A3
A4
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
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
```

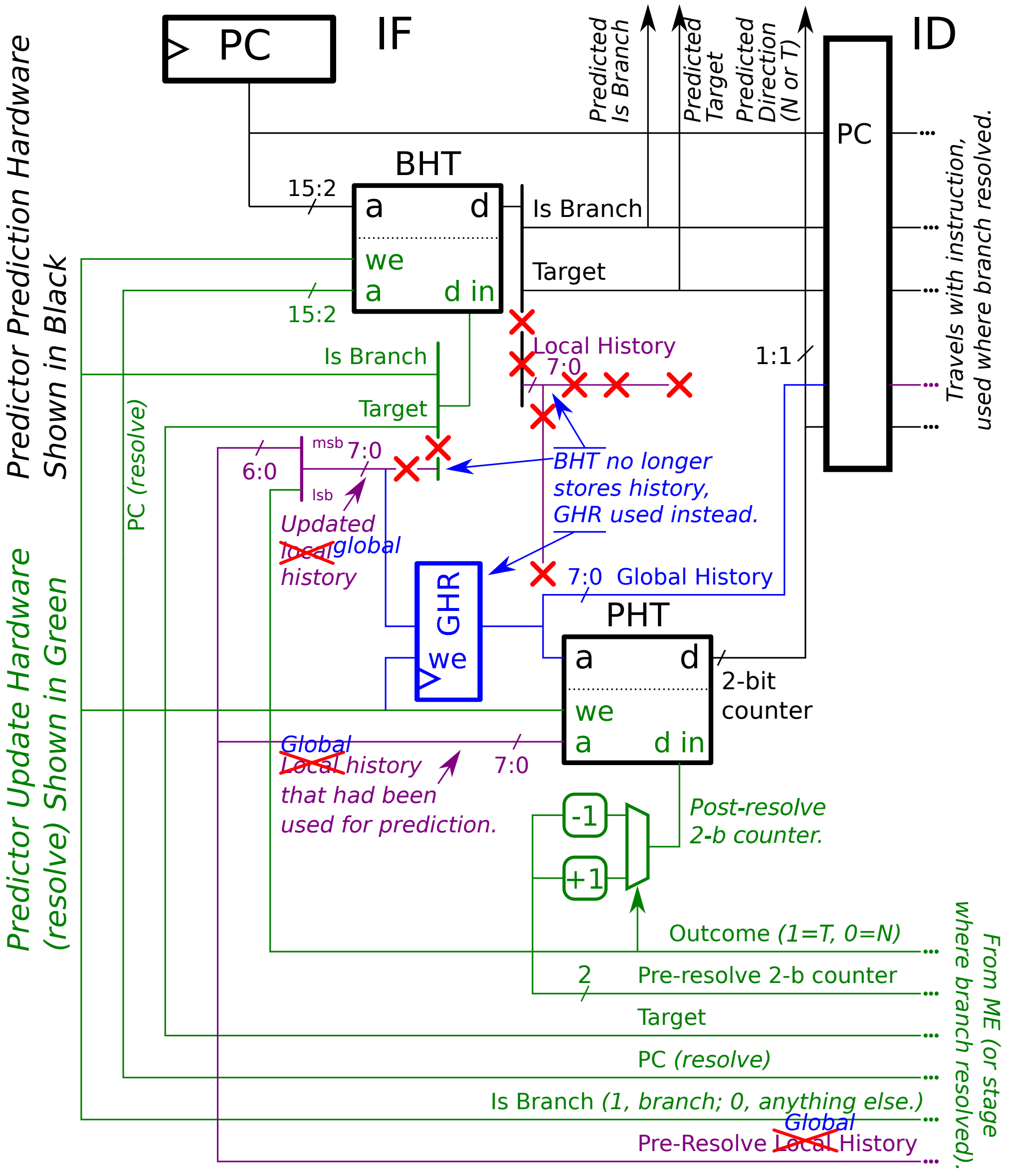
(*b*) Appearing below is a diagram of a local predictor, showing in detail the logic for predicting the instruction in `IF` and for updating the predictor for the resolving branch. Modify the diagram so that it is a global predictor with an 8-outcome global history.

☑ 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.)

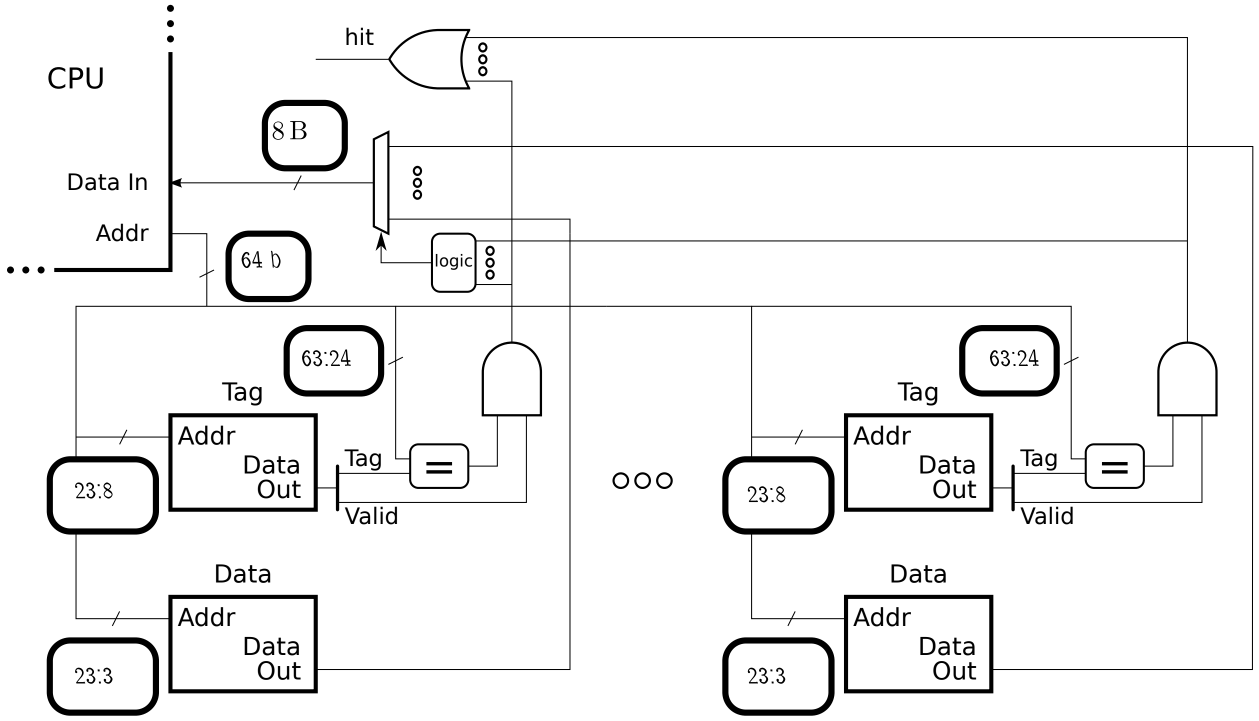


PC
IF
ID
Predicted Is Branch
Predicted Target
Predicted Direction (N or T)
PC
BHT
15:2
a
d
we
a
d in
15:2
Is Branch
Target
~~Local History~~
7:0
1:1
Travels with instruction, used where branch resolved.
Predictor Prediction Hardware Shown in Black
Predictor Update Hardware (resolve) Shown in Green
PC (resolve)
Is Branch
Target
msb
7:0
6:0
lsb
Updated ~~local~~ global history
BHT no longer stores history, GHR used instead.
GHR
we
7:0 Global History
PHT
a
d
2-bit counter
we
a
d in
~~Local~~ Global history that had been used for prediction.
7:0
-1
+1
Post-resolve 2-b counter.
Outcome (1=T, 0=N)
2
Pre-resolve 2-b counter
Target
PC (resolve)
Is Branch (1, branch; 0, anything else.)
Pre-Resolve ~~Local~~ Global History
From ME (or stage where branch resolved).

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 MiB $= 64 \times 2^{20}$ B $= 2^{26}$ 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:

Tag	Index	Offset
63 – 24	23 – 8	7 – 3, 2 – 0

[✓] 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.

Address:

Tag	Index	Offset
63 – 26	25 – 8	7 – 3, 2 – 0

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.

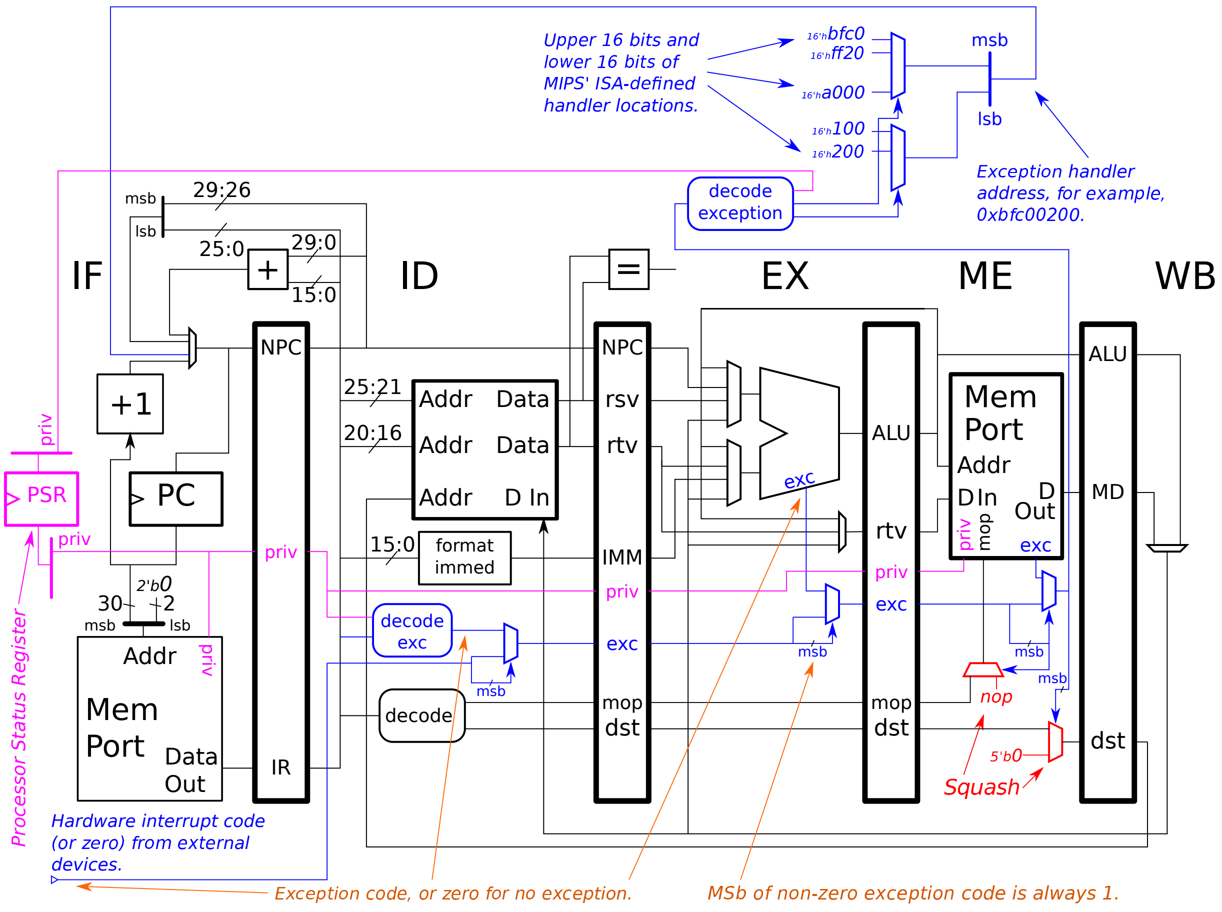
```
int sum = 0;
short *a = 0x2000000; // sizeof(short) == 2
int i;
int ILIMIT = 1 << 11;    // = 2^11

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 $\boxed{\text{hit ratio is } \frac{127}{128}}$.

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 $\boxed{\text{decode exc}}$ 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.

	Opcode	RS	RT	Immed
MIPS I:	beq = 0x4	2	3	0xC
	31 26	25 21	20 16	15 0

☑ Encoding for `add` from code fragment above:

	Opcode	RS	RT	RD	SA	Function
MIPS R:	0	8	9	7	0	add = 0x20
	31 26	25 21	20 16	15 11	10 6	4 0

☑ Encoding for `lw` from code fragment above:

	Opcode	RS	RT	Immed
MIPS I:	lw = 0x23	15	12	14
	31 26	25 21	20 16	15 0

(*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.