Name Solution

Computer Architecture

LSU EE 4720

Final Examination

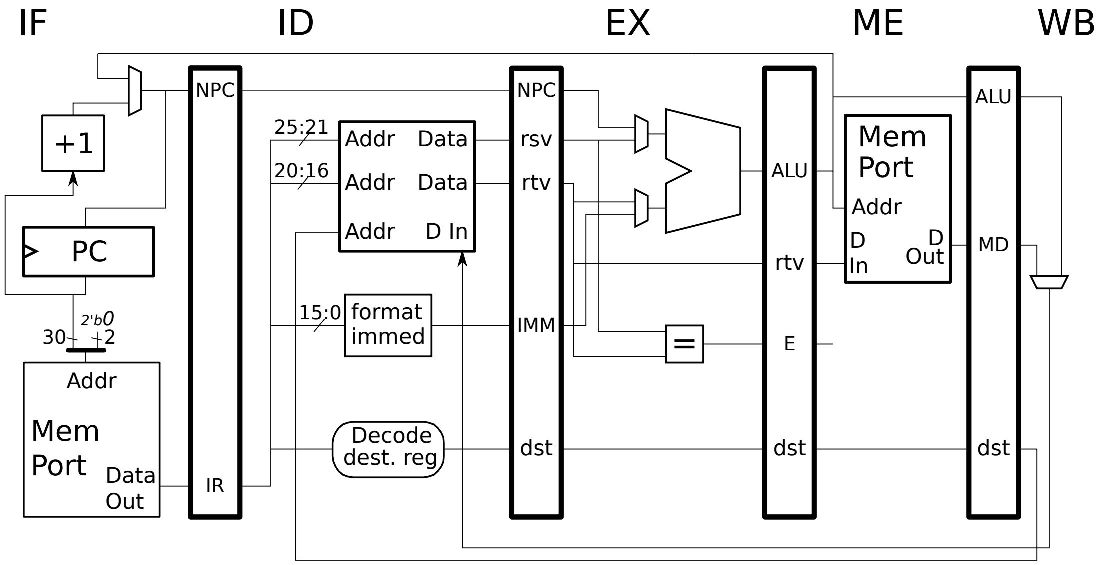
Monday, 9 May 2022 10:00-12:00 CDT

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.



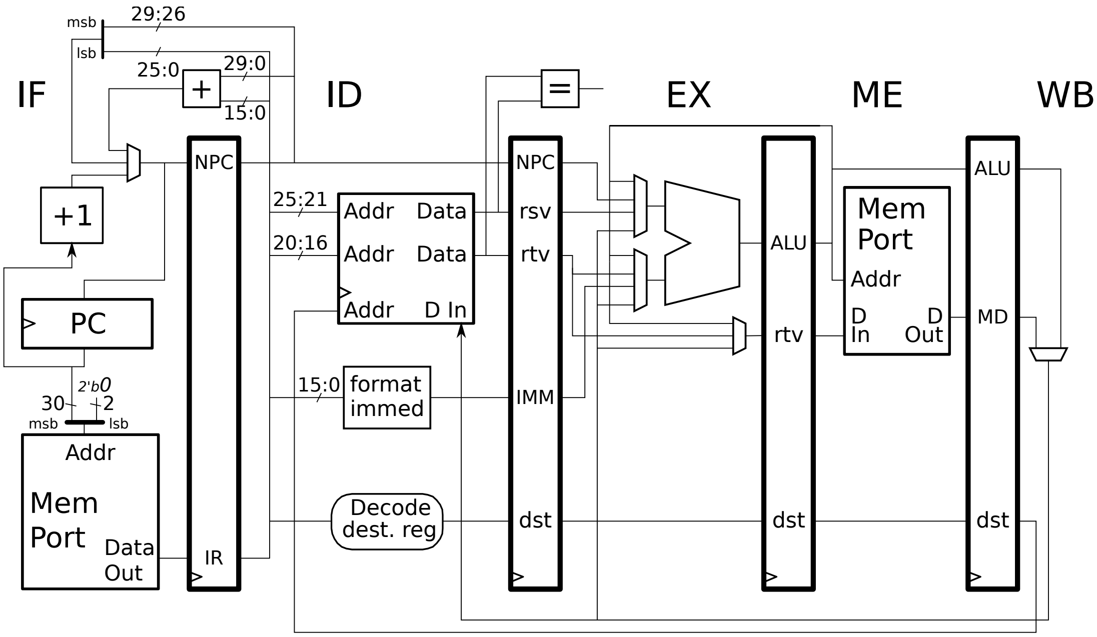
☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

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
```



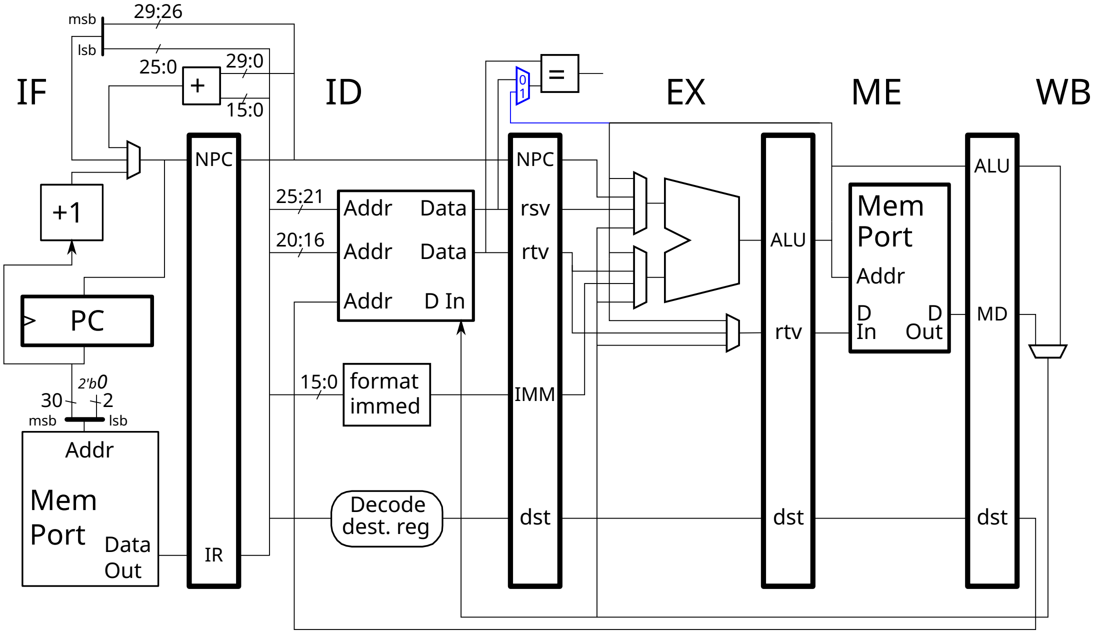
☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

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: $\frac{2\,\text{insn}}{(6-2)\,\text{cyc}} = \frac{2}{4}$ 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
```

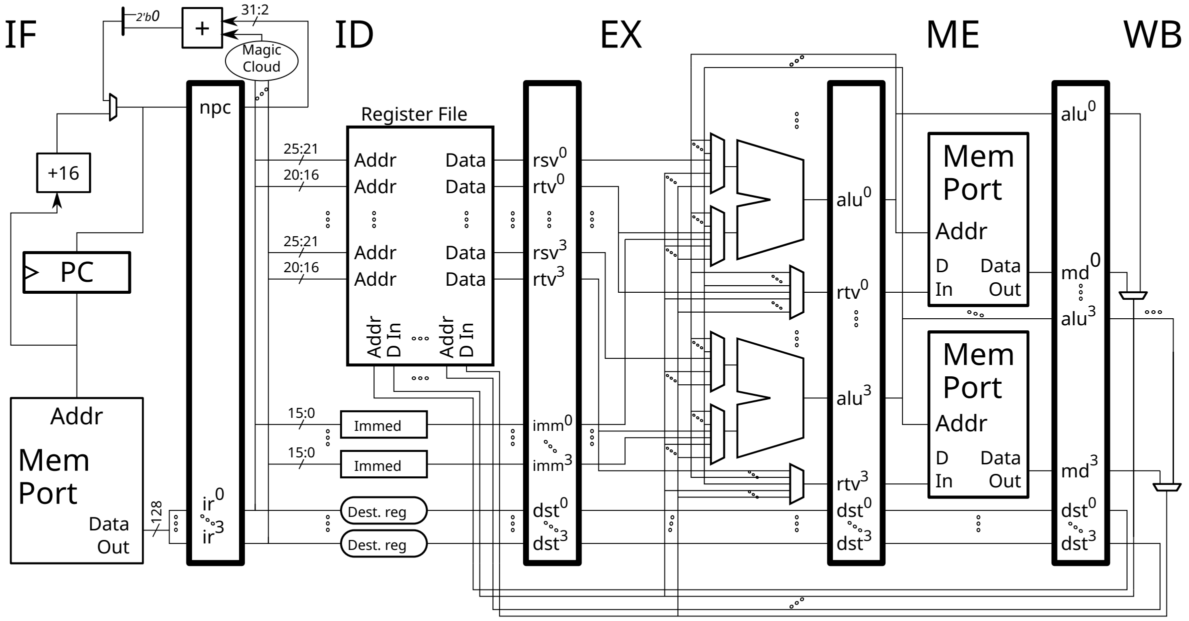


☑ Show execution and ☑ determine instruction throughput (IPC) based on a large number of iterations.

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
```



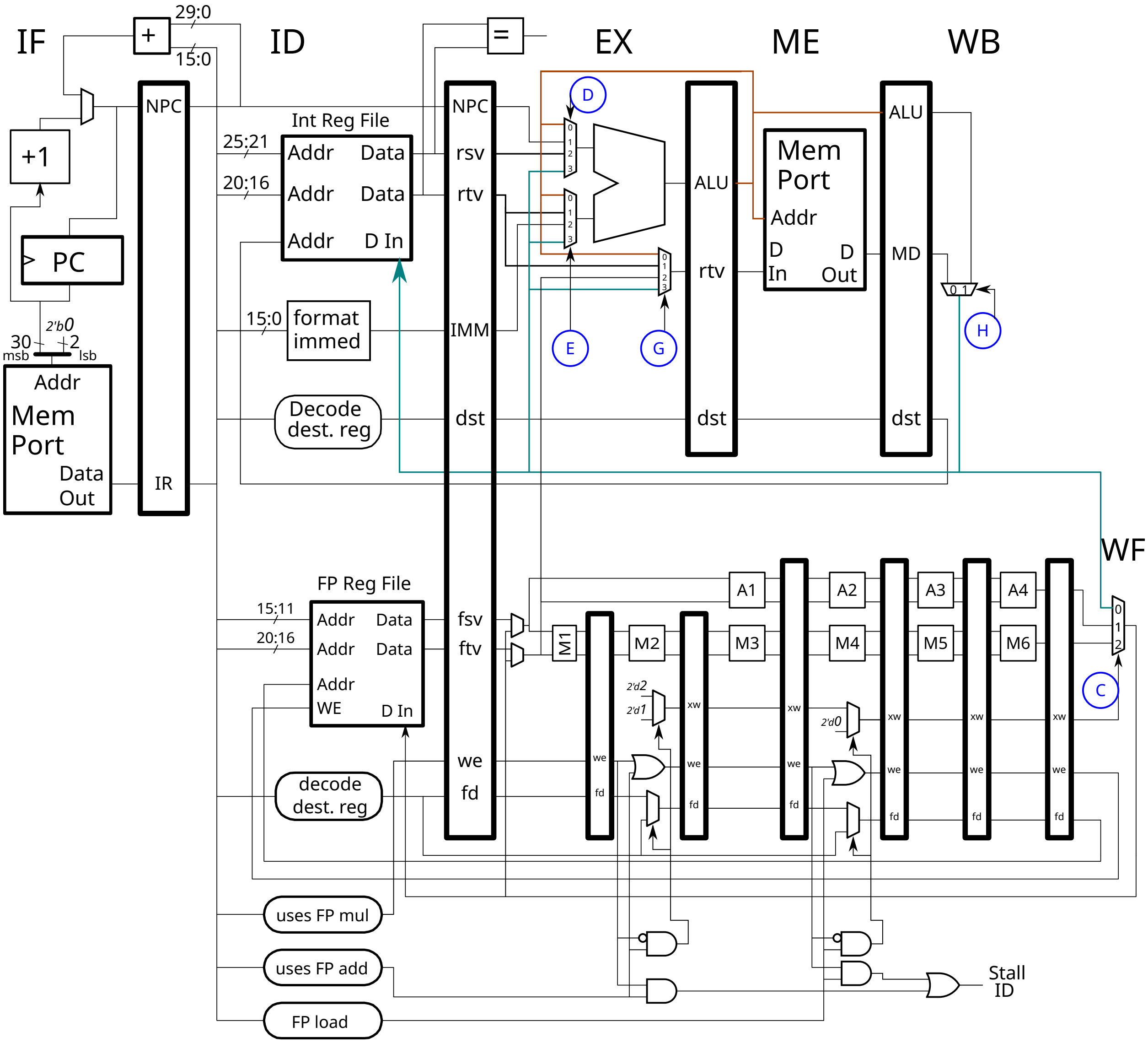
☑ For the **4-way** superscalar MIPS above show execution until the fetch of the `lw r1` in the second iteration.
☑ Show instruction throughput (IPC) assuming a large number of iterations.

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.

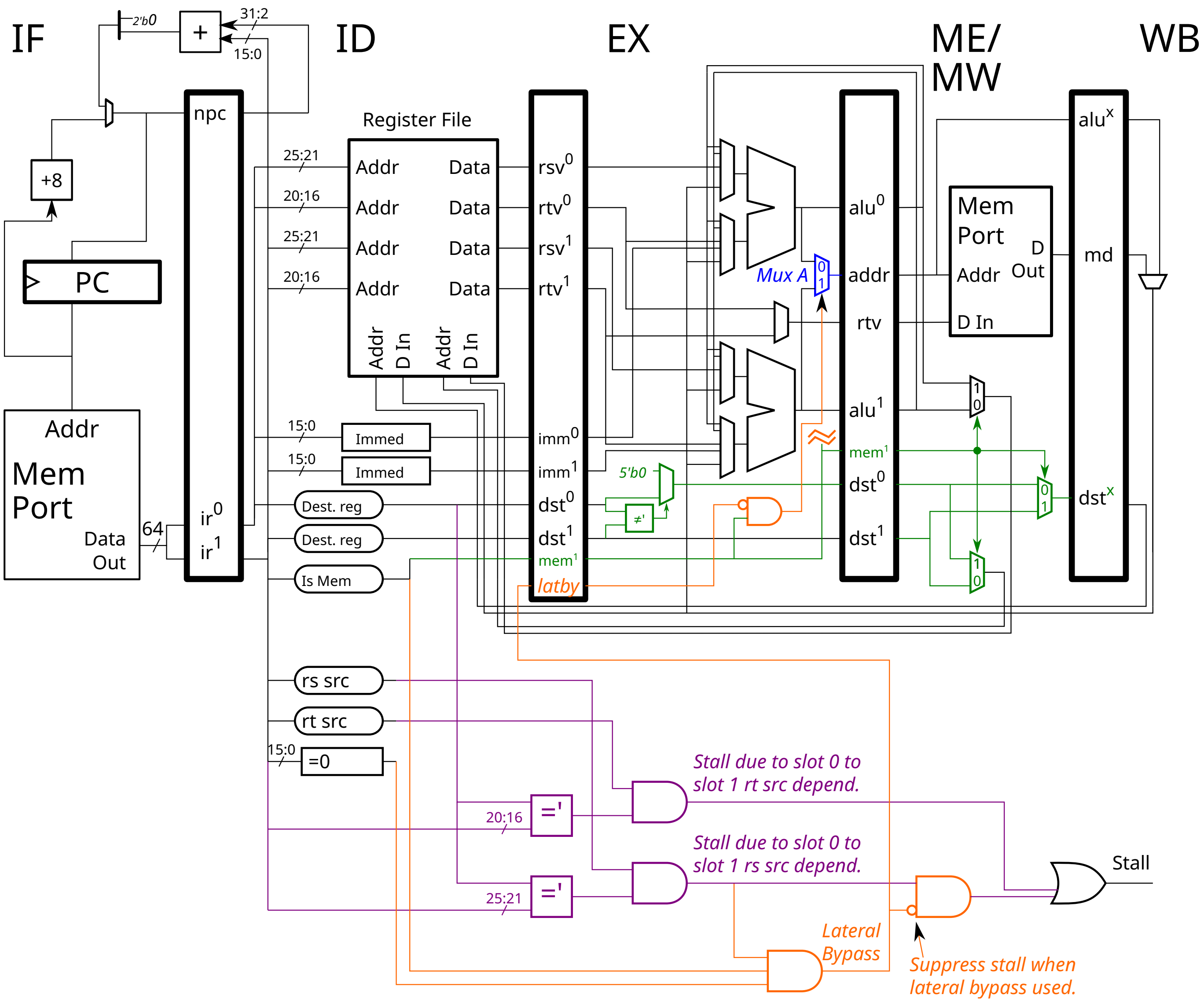
☑ As engineers always do, pay attention to cost and performance.

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.

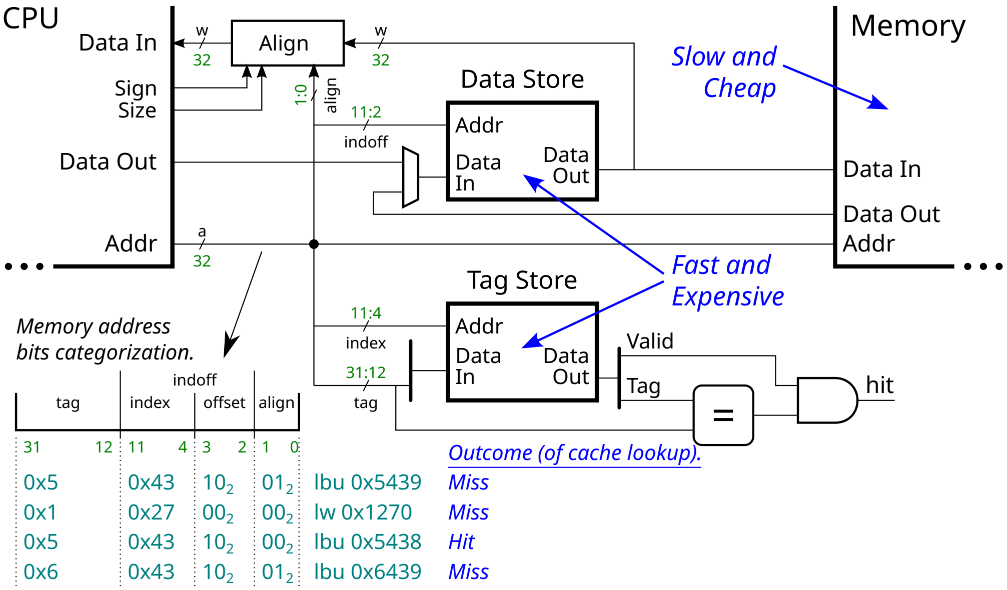
```
          All N’s          All T’s             All N’s          All T’s     SOLUTION WORK
      ---------------  --------------      ---------------  --------------  ...
    0  0  0    0  0  0  1  2   3  3  3      2  1      0  0  1  2  3   3  3
B2:   N  N  ...  N  N  T  T  ... T  T      N  N  N. ... 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, ..`).



tag	index	offset	align		Outcome (of cache lookup).
		indoff			
31 12	11 4	3 2	1 0		
0x5	0x43	10_2	01_2	lbu 0x5439	*Miss*
0x1	0x27	00_2	00_2	lw 0x1270	*Miss*
0x5	0x43	10_2	00_2	lbu 0x5438	*Hit*
0x6	0x43	10_2	01_2	lbu 0x6439	*Miss*

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.

	Opcode	RS	RT	Immed
MIPS I:	4	2	6	2
	31 26	25 21	20 16	15 0

☑ 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.

	Opcode	RS	RT	Immed
MIPS I:	0x23	3	1	4
	31 26	25 21	20 16	15 0

(*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.