Name _____

# Computer Architecture and Engineering
# **CS152 Quiz #1**
# February 19th, 2013
# Professor Krste Asanovic

## Name: _____<ANSWER KEY>_____

This is a closed book, closed notes exam.
80 Minutes
20 pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

| | | |
|---|---|---|
| Writing name on each sheet | _____ | 1 Point |
| Question 1 | _____ | 18 Points |
| Question 2 | _____ | 25 Points |
| Question 3 | _____ | 18 Points |
| Question 4 | _____ | 18 Points |
| TOTAL | _____ | 80 Points |

Name _____

# Question 1: Microprogramming [18 points]

In this question, you are going to implement a compare-and-swap (CAS) instruction in microcode. A CAS instruction compares the contents of a memory location to a given value and, only if they are the same, modifies the content of that memory location to a new given value. The instruction will also write the old value read from the memory location (not the value written to it) to the register file. This instruction is often used for implementing synchronization primitives. The instruction has the following format:

**CAS** rd, rs1, rs2

CAS performs the following operation:

value ← M[R[rs1]]
if (value == R[rd]) M[R[rs1]] ← R[rs2]
R[rd] ← value

Given a bus-based microarchitecture in Appendix A (the same bus-based microarchitecture from the online material), fill in worksheet Q1-1 with the microcode for CAS. Use don't cares (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any ISA-visible registers (except the PC register and the rd register) in the course of executing the instruction. You will receive credit for elegance and efficiency.

Finally, make sure that your microcode sequence fetches the next instruction in program order (i.e., by doing a microbranch to FETCH0 as discussed in the handout).

Name _____

| State | PseudoCode | ld IR | Reg Sel | Reg Wr | en Reg | ld A | ld B | ALUOp | en ALU | ld MA | Mem Wr | en Mem | Imm Sel | en Imm | μBr | Next State |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FETCH0: | MA <- PC;<br>A <- PC | 0 | PC | 0 | 1 | 1 | * | * | 0 | 1 | * | 0 | * | 0 | N | * |
| | IR <- Mem | 1 | * | * | 0 | 0 | * | * | 0 | * | 0 | 1 | * | 0 | S | * |
| | PC <- A+4 | 0 | PC | 1 | 1 | * | * | INC_A_4 | 1 | * | * | 0 | * | 0 | D | * |
| . . . | | | | | | | | | | | | | | | | |
| NOP0: | microbranch back to FETCH0 | 0 | * | * | 0 | * | * | * | 0 | * | * | 0 | * | 0 | J | FETCH0 |
| CAS: | MA <- R[rs1] | 0 | rs1 | 0 | 1 | * | * | * | 0 | 1 | * | 0 | * | 0 | N | * |
| | A <- Mem | 0 | * | * | 0 | 1 | * | * | 0 | 0 | 0 | 1 | * | 0 | S | * |
| | B <- R[rd] | 0 | rd | 0 | 1 | 0 | 1 | * | 0 | 0 | * | 0 | * | 0 | N | * |
| | If (A-B != 0) goto SKIP | 0 | * | * | 0 | 0 | * | SUB | *(0/1) | 0 | * | 0 | * | 0 | NZ | SKIP |
| | Mem <- R[rs2] | 0 | rs2 | 0 | 1 | 0 | * | * | 0 | 0 | 1 | 1 | * | 0 | S | * |
| SKIP: | R[rd] <- A; goto FETCH0 | * | rd | 1 | 1 | * | * | * | 0 | * | * | 0 | * | 0 | J | FETCH0 |

Worksheet Q1-1

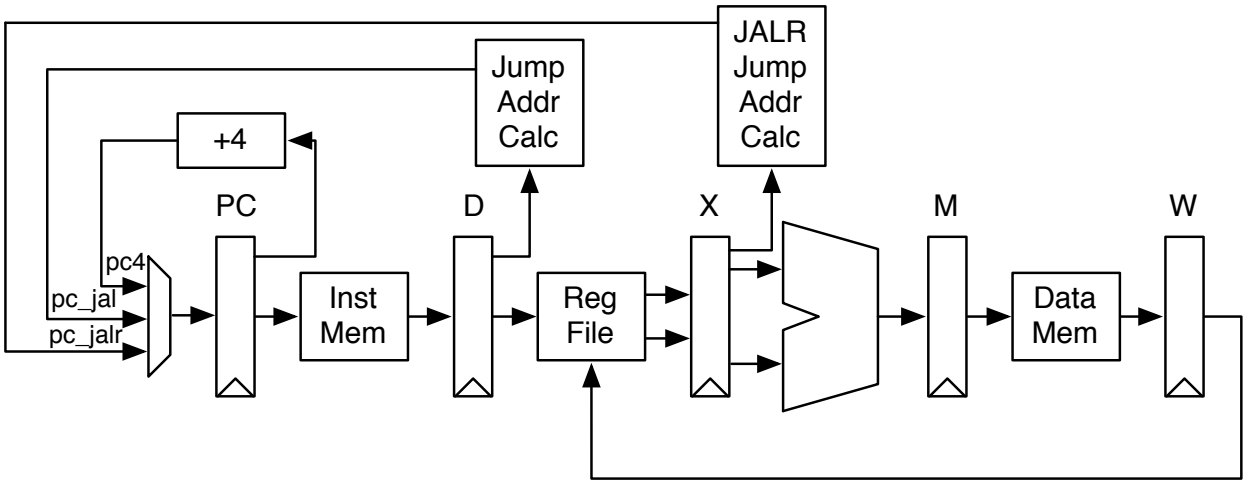
There are many valid answers to this question. The pseudocode was worth 10 points, and the control signals were worth 8 points. The correctness of the pseudocode was examined first. The problem description said to use "elegance and efficiency". With that in mind, one point was deducted per extra cycle your implementation took. Correctness issues cost three or more points.

Then, the control signals were examined to see if it reflected your pseudocode. The problem description said to use don't cares (*). Up to two points were deducted for not using don't-cares everywhere possible. The don't care on the enALU of the SUB operation wasn't enforced. Correctness issues cost one point each.

Name _____

# Question 2: BTBs and Subroutine Return Stacks [25p]

For this question, consider a fully bypassed 5-stage RISC-V processor (as shown in Lecture 4, and used in Lab 1). We have reproduced the pipeline diagram below (bypasses are not shown). The fetch stage always speculates that the next PC is PC+4. ***For this problem, we will ignore conditional branches, and only concern ourselves with unconditional jumps.***




For a JAL instruction (J-type instructions), the 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the pc to form the jump address. Note we have an adder to calculate the jump address for JAL instructions in the decode stage. JAL stores the address of the instruction following the jump (pc+4) into register x1.

For a JALR instruction (I-type instruction), the jump address is obtained by sign-extending the 12-bit immediate then adding it to the address contained in register rs1, and hence is only known in the execute stage. The address of the instruction following the jump (pc+4) is written to register rd. Note that the MIPS instruction "jr ra" is equivalent to a RISC-V instruction "jalr x0,x1,0".

To summarize,

| Instruction | Taken known? | Target known? |
|---|---|---|
| JAL | Decode stage | Decode stage |
| JALR | Decode stage | Execute stage |

Name _____

**Q2.A Motivating BTBs and Subroutine Return Stacks [3 points]**

To understand the pipeline behavior, please fill out the following instruction/time diagram for the following set of instructions until the pc becomes 0x2010. The first two instructions have been done for you.

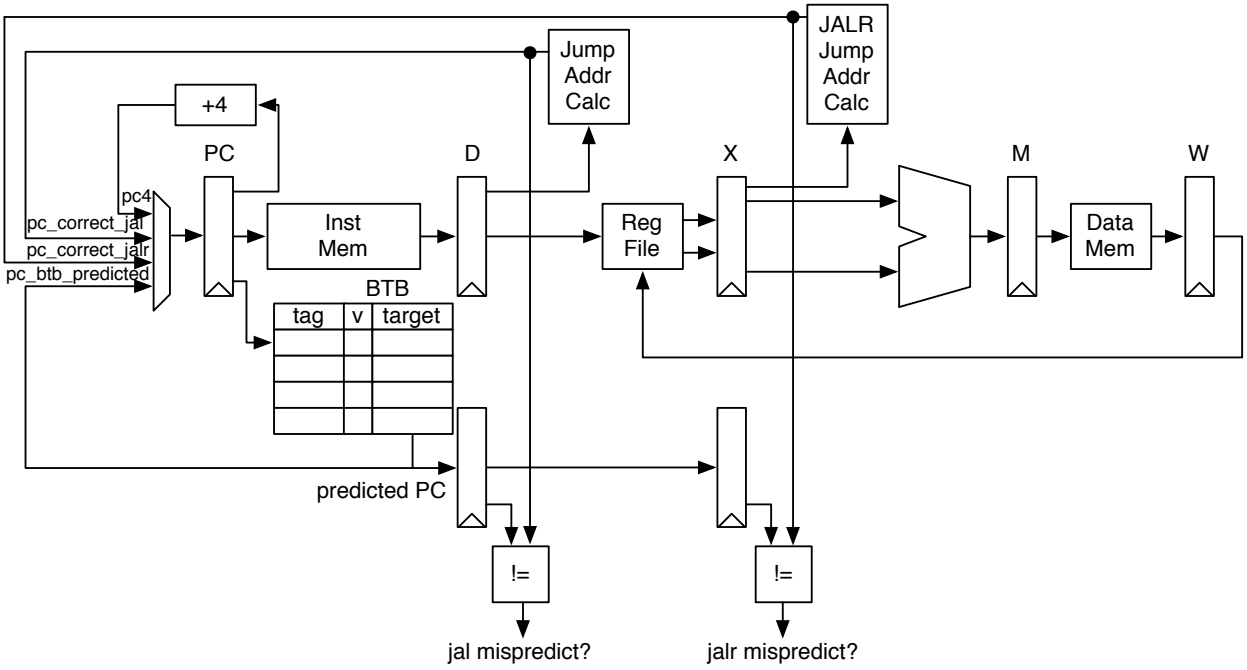
```
     0x2000: jal foo
     0x2004: addi x3,x0,3
     0x2008: jal foo
     0x200c: sub x4,x5,x7
     0x2010: lw x7,4(x6)
     ...

foo: 0x4000: and x10,x11,x12
     0x4004: jalr x0,x1,0
bar: 0x4008: xor x20,x21,x22
     0x400c: or x24,x25,x26
     0x4010: jalr x0,x1,0
```

Name _____

| PC | Instruction | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x2000 | jal | F | D | X | M | W | | | | | | | | | | | | | | | | |
| 0x2004 | addi | | F | - | - | - | - | | | | | | | | | | | | | | | |
| 0x4000 | and | | | F | D | X | M | W | | | | | | | | | | | | | | |
| 0x4004 | jalr | | | | F | D | X | M | W | | | | | | | | | | | | | |
| 0x4008 | xor | | | | | F | D | - | - | - | | 1 point for getting jalr right | | | | | | | | | | |
| 0x400c | or | | | | | | F | - | - | - | - | | | | | | | | | | | |
| 0x2004 | addi | | | | | | | F | D | X | M | W | | | | | | | | | | |
| 0x2008 | jal | | | | | | | | F | D | X | M | W | | | | | | | | | |
| 0x200c | sub | | | | | | | | | F | - | - | - | - | | 1 point for getting jal right | | | | | | |
| 0x4000 | and | | | | | | | | | | F | D | X | M | W | | | | | | | |
| 0x4004 | jalr | | | | | | | | | | | F | D | X | M | W | | | | | | |
| 0x4008 | xor | 1 point for getting jalr right | | | | | | | | | | | F | D | - | - | - | | | | | |
| 0x400c | or | | | | | | | | | | | | | F | - | - | - | - | | | | |
| 0x200c | sub | | | | | | | | | | | | | | F | D | X | M | W | | | |
| 0x2010 | lw | | | | | | | | | | | | | | | F | D | X | M | W | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |

**Q2.B Adding a BTB [4 points]**

Let's assume we have added a 4-entry fully associative BTB to the Fetch Stage. The BTB is fully searched in the Fetch Stage to see if the PC matches any valid tags. If there's a match, the BTB makes a prediction (i.e., redirects the PC to the target PC recorded in the BTB). The new pipeline diagram is shown below.




For a JAL instruction, (1) the jump address calculation logic in the Decode stage detects a mispredict, and if a misprediction is detected, (2) kills the appropriate stages, (3) fixes up the BTB, and (4) starts the Instruction Fetch using the correct jump address (pc_correct_jal).

For a JALR instruction, (1) the jump address calculation logic in the Execute stage detects a mispredict, and if a misprediction is detected, (2) kills the appropriate stages, (3) fixes up the BTB, and (4) starts the Instruction Fetch using the correct jump address (pc_correct_jalr).

Remember, the Fetch Stage is still predicting PC+4 every cycle, unless either the BTB makes a prediction (has a matching and valid entry for the current PC), or the jump address calculation logic in the Decode and Execute stage corrects for a misprediction (pc_correct_jal, pc_correct_jalr).

Show how this pipeline will execute the same code segment shown in Q2.A (repeated below as well) by filling in the following pipeline diagram until the pc becomes 0x2010. Assume the BTB first allocates invalid entries, and then starts kicking out least-recently-used entries.

Initially, the BTB contains:

| Tag | Valid | Target PC |
|---|---|---|
| 0x2000 | 1 | 0x4000 |
| 0x4004 | 1 | 0x2004 |
| 0x2008 | 0 | 0x4008 |
| 0x4010 | 0 | 0x2010 |

Please fill in the BTB's final state below after running the code.

| Tag | Valid | Target PC |
|---|---|---|
| 0x2000 | 1 | 0x4000 |
| 0x4004 | 1 | 0x200c |
| 0x2008 | 1 | 0x4000 |
| - | 0 | - |

2 points for the BTB. -1 point for an incorrect BTB entry.
2 points for the timing diagram. -1 point for an incorrect bubble.

We have copied over the same code segment shown in Q2.A for your convenience.

```
     0x2000: jal foo
     0x2004: addi x3,x0,3
     0x2008: jal foo
     0x200c: sub x4,x5,x7
     0x2010: lw x7,4(x6)
     ...

foo: 0x4000: and x10,x11,x12
     0x4004: jalr x0,x1,0
bar: 0x4008: xor x20,x21,x22
     0x400c: or x24,x25,x26
     0x4010: jalr x0,x1,0
```

Name _____

| PC | Instruction | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x2000 | jal | F | D | X | M | W | | | | | | | | | | | | | | | | |
| 0x4000 | and | | F | D | X | M | W | | | | | | | | | | | | | | | |
| 0x4004 | jalr | | | F | D | X | M | W | | | | | | | | | | | | | | |
| 0x2004 | addi | | | | F | D | X | M | W | | | | | | | | | | | | | |
| 0x2008 | jal | | | | | F | D | X | M | W | | | | | | | | | | | | |
| 0x200c | sub | | | | | | F | - | - | - | - | | | | | | | | | | | |
| 0x4000 | and | | | | | | | F | D | X | M | W | | | | | | | | | | |
| 0x4004 | jalr | | | | | | | | F | D | X | M | W | | | | | | | | | |
| 0x2004 | addi | | | | | | | | | F | D | - | - | - | | | | | | | | |
| 0x2008 | jal | | | | | | | | | | F | - | - | - | - | | | | | | | |
| 0x200c | sub | | | | | | | | | | | F | D | X | M | W | | | | | | |
| 0x2010 | lw | | | | | | | | | | | | F | D | X | M | W | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |

Name _____

**Q2.C BTB Misprediction [4 points]**

When a JAL instruction hits in the BTB, can the target PC be incorrect? Explain.

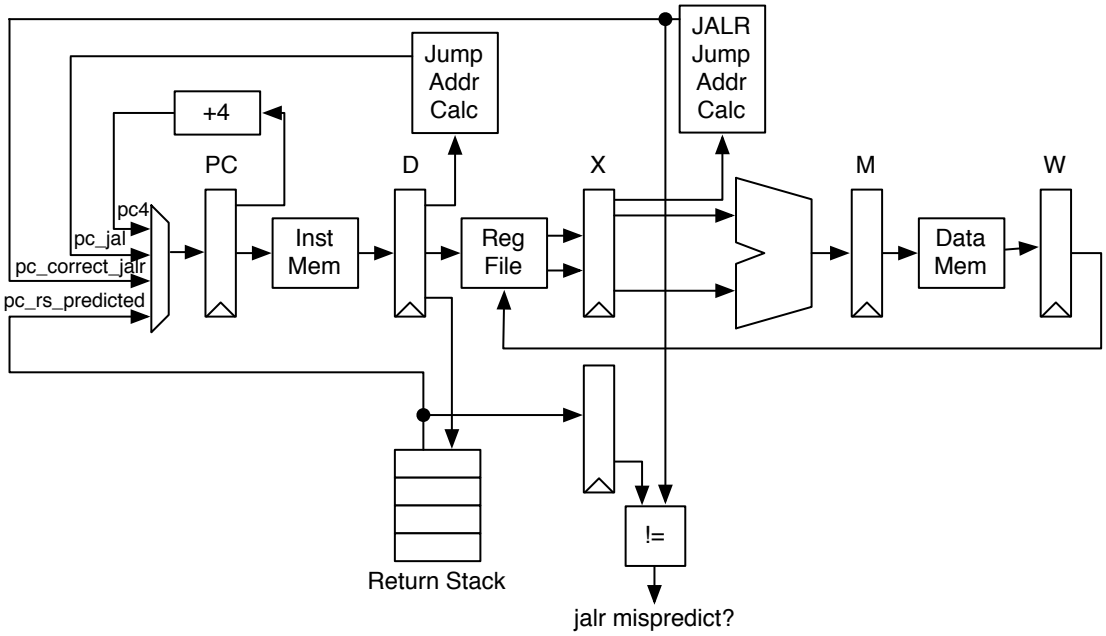
No, JAL's target addresses cannot change; the target PC is statically encoded in the instruction.
2 points. 1 point for Yes/No, 1 point for reasoning.

When a JALR instruction hits in the BTB, can the target PC be incorrect? Explain.

Yes, JALR's target addresses can change; the target PC is read from a register.
2 points. 1 point for Yes/No, 1 point for reasoning.

**Q2.D Adding a return stack [4 points]**

Let's assume we have added a 4-entry subroutine return stack to the Decode Stage. Note we have removed the BTB from the pipeline, and will only be using the return stack for this question. Once a JAL instruction is decoded, pc+4 is pushed into the return stack. Once a JALR instruction is decoded, the return address is popped off from the return stack, and the PC is redirected to that address (pc_rs_predicted). The predicted address is pushed down to the Execute stage, where it is gets checked for a misprediction.




Remember, the Fetch Stage is still predicting PC+4 every cycle, unless the return stack makes a prediction, a JAL instruction redirects the PC in the decode stage (pc_ jal), or the jump address calculation logic in the Execute stage corrects for a misprediction (pc_correct_jalr).

Show how this pipeline will execute the same code segment shown in Q2.A by filling in the following pipeline diagram until the pc becomes 0x2010.

Name _____

| PC | Instruction | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | T17 | T18 | T19 | T20 | T21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x2000 | jal | F | D | X | M | W | | | | | | | | | | | | | | | | |
| 0x2004 | addi | | F | - | - | - | | 1 point for getting jal right | | | | | | | | | | | | | | |
| 0x4000 | and | | | F | D | X | M | W | | | | | | | | | | | | | | |
| 0x4004 | jalr | | | | F | D | X | M | W | | | | | | | | | | | | | |
| 0x4008 | xor | | | | | F | - | - | - | - | | 1 point for getting jalr right | | | | | | | | | | |
| 0x2004 | addi | | | | | | F | D | X | M | W | | | | | | | | | | | |
| 0x2008 | jal | | | | | | | F | D | X | M | W | | | | | | | | | | |
| 0x200c | sub | | | | | | | | F | - | - | - | - | | 1 point for getting jalr right | | | | | | | |
| 0x4000 | and | | | | | | | | | F | D | X | M | W | | | | | | | | |
| 0x4004 | jalr | | | | | | | | | | F | D | X | M | W | | | | | | | |
| 0x4008 | xor | 1 point for getting jalr right | | | | | | | | | | F | - | - | - | - | | | | | | |
| 0x200c | sub | | | | | | | | | | | | F | D | X | M | W | | | | | |
| 0x2010 | lw | | | | | | | | | | | | | F | D | X | M | W | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |

**Q2.E More on return stacks [4 points]**

When would a return stack not work effectively? Briefly explain when, and provide a minimal code sequence that will result in return stack mispredictions.

2 points for reasoning, 2 points for minimal code sequence.

more than 4 jals -> overflows the stack
jalr with an immediate
jalr with a different register
Math on x1, before used in jalr x0,x1,0
jalr used for a switch statement
jalr used for indirect function calls
long jumps
return stacks also wouldn't work well with exceptions

**Q2.F Return stack in the Fetch Stage [6 points]**

For further benefits, you could implement a return stack in the Fetch stage. Why would that help, and how would you build that? Explain.

Why (3 points): By moving the return stack into the Fetch stage, we are able to get rid of the one cycle penalty when executing JALR instructions (see timing diagram in Q2.D).

How (3 points): Make a BTB like structure to tag PCs of a JAL/JALR instruction. With this BTB like structure, the Fetch stage can detect JAL/JALR instructions and manipulate the return stack without increasing the critical path.

Some people suggested to add some decode logic in the fetch stage, but that scheme would likely increase the critical path (1 point). People who simply said "add a BTB like structure" without any explanation got 2 points.

Name _____

# Question 3: Load-Value Speculation [18 points]

In the conventional fully bypassed 5-stage pipeline discussed in class, the main remaining data hazard is the load-use hazard. For example, in the following instruction sequence:

```
lw x1, 16(x2)
xor x3, x1, x5
```

The xor instruction will experience a one-cycle stall in the decode stage, while the lw propagates to the memory stage. For the remainder of this question, we will only consider 32-bit loads (lw). ***You may ignore branch and jump instructions in this problem.***

```
F   D   X   M   W                       lw
    F   D   D   X   M   W               xor
```

**Load-value speculation**

---

One approach to removing the hazard is to speculate on the load value returned (*load-value speculation*). In lecture, we briefly described one scheme, a *load-zero predictor* where the value was predicted to be zero. An instruction that would otherwise stall in the decode stage waiting for a memory value was provided the value 0 instead. When the load instruction reaches the memory stage, the value is checked and if not zero, the pipeline is flushed and the dependent instruction is replayed as shown.

```
F   D   X   M   W                           lw
    F   D   X   -   -                       xor flushed
        F   D   -   -   -                   flushed
            F   -   -   -   -               flushed
                F   D   X   M   W           xor correct value
```

**Q3.A [3 points]** Assuming the pipeline is flushed as shown on a mispredict, calculate the minimum accuracy required for the load-zero predictor to improve performance. Note the predictor is only used when an instruction would otherwise stall in decode waiting for a load value.

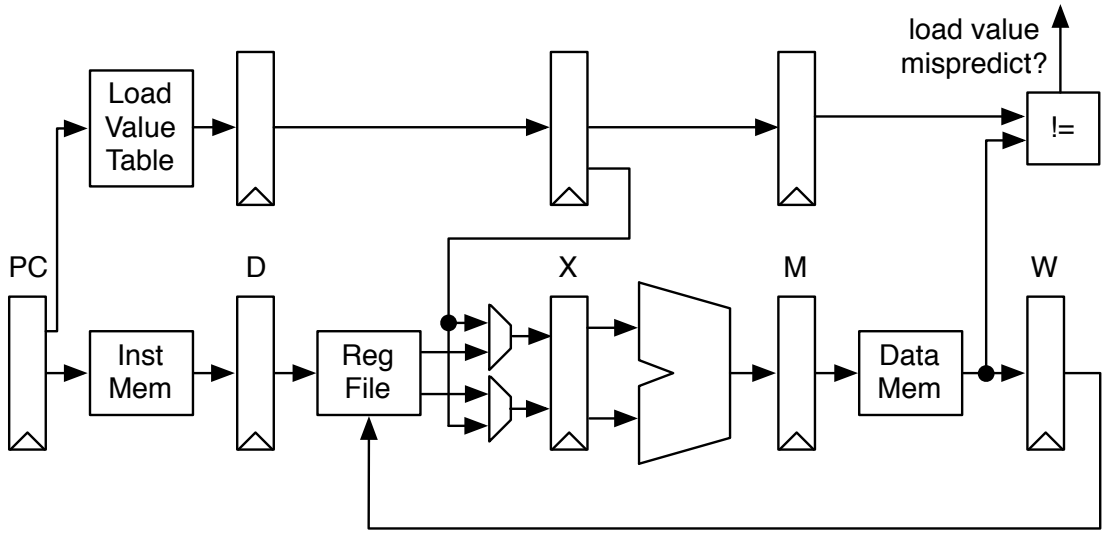
Mispredict penalty of a predicted load-zero is 3 cycles. The load-use delay is 1 cycle. The predictor should be at least 66.7% accurate to improve performance.

Name _____

**Load-value table**

---

One possible way to improve the performance of load-value speculation, is to add a lookup table to remember the previous value returned by a given load instruction. We call this scheme a *load-value table*. The lookup table is indexed by the PC of the load instruction, and returns a predicted load value in parallel with instruction fetch. The predicted load value is used when an instruction would otherwise stall waiting for the load.




As before, if the predicted value is different than the actual load value, the pipeline is flushed, the table is updated, and the dependent instruction is replayed.

**Q3.B [5 points]** Discuss whether it is ever possible for the load-value table to be less accurate than the load-zero predictor.

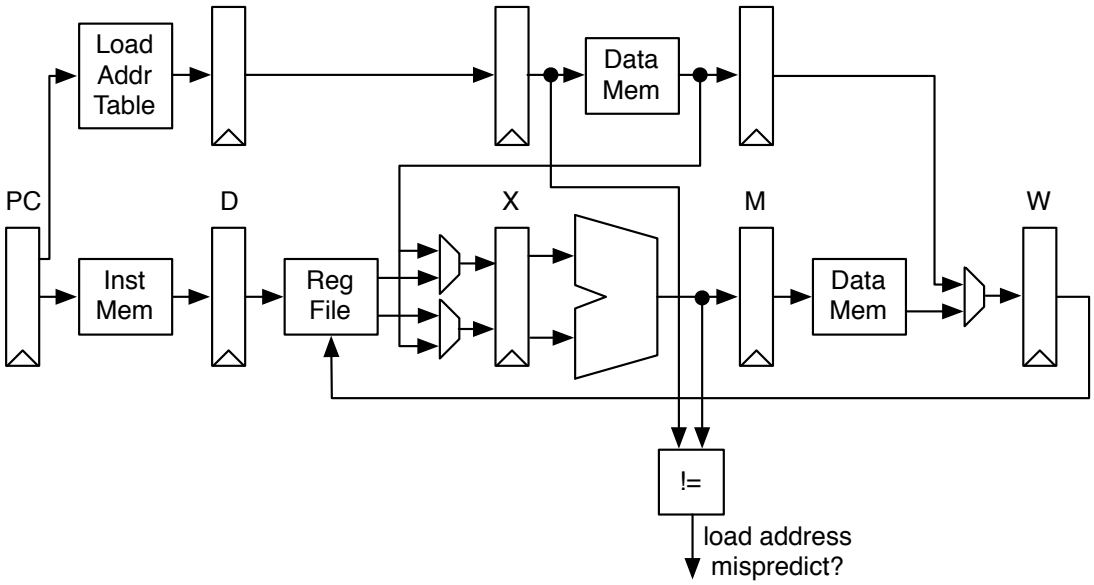
Yes, when the same load instruction returns alternating values including zeros. For example, if an array returned [0,1,0,1,0,1,…], a load-zero predictor would be correct half the time. A load-value table would always predict a wrong value. This would happen in the following code sequence:

```
loop:
  lw x2,0(x3)
  addi x2,x2,1
  andi x2,x2,1
  sw x2,0(x3)
  j loop
```

Name _____

**Load-address table**

Another approach is to replace the load-value table with a *load-address table*. The load-address table is also indexed by the PC of the load instruction, but now returns a prediction of the address that the load instruction will access. If the pipeline control logic detects a load in the execute stage, with a dependent use in the decode stage, the predicted address is now used to access the data memory in the execute stage, allowing the value to be returned from memory one cycle earlier, hence avoiding the stall. The execute stage is still used to calculate the address for the load, and this is checked against the predicted address. If the predicted and actual addresses match, the load instruction does not need to access the data memory in the memory stage (the load value is passed down the pipeline to writeback). If there is an address mismatch, the load accesses the data memory again in the memory stage and the pipeline behind the load is stalled for a cycle.




**Q3.C [5 points]** For the load-address table scheme, give a short instruction sequence that shows the occurrence of a structural hazard in accessing the data memory. Indicate clearly for each load instruction whether its address is correctly or incorrectly predicted.

lw x2,0(x3)
lw x4,0(x1)
addi x3,x4,1

sw x2,0(x3)
lw x4,0(x1)
addi x3,x4,1

Note that there's a structural hazard only when there's a dependent instruction (addi). Otherwise the second load instruction will not speculatively access the data memory.

lw x2,0(x3)
lw x4,0(x2)
add x3,x4,1

If the second load is dependent on the first load, there's a structural hazard only when the first load mispredicts.

**Q3.D [5 points]** Let's now assume a load instruction will not access memory in the execute stage if a structural hazard would be present on the data cache, but will instead access the data cache in the memory stage as in the original pipeline. The dependent instruction will have to stall to wait for the bypassed value in this case. Calculate the minimum accuracy needed to improve performance in this case.

In case of a mispredict, the load instruction will access the data cache the next cycle, wasting one cycle. Assuming we don't have load prediction, there's already a cycle that we need to stall for the load-use delay. Assuming that we don't increase the critical path, improvement is guaranteed. Minimum accuracy needed = 0%.

Name _____

# Question 4: Iron Law of Processor Performance [18 points]

Mark whether the following modifications will cause each of the three categories to **increase**, **decrease**, or whether the modification will have **no effect**. ***Explain your reasoning*** to receive credit.

Assume the initial machine is pipelined. Also assume that any modification is done in a way that preserves correctness and maintains efficiency, but that the rest of the machine remains unchanged.

| | | Instructions/Program | Cycles/Instruction | Seconds/Cycle |
|---|---|---|---|---|
| A | Improve branch predictor accuracy | No effect<br><br>No changes to the program. | Decrease<br><br>Branch prediction normally works well, decreasing the number of stall cycles on a branch, and hence reduces CPI. | Same<br><br>Should stay the same. Could increase if circuit is large, but usually make sure not to impact cycle time. |
| B | Provide a new instruction that sums three source registers and writes result to destination register | Decrease<br><br>Can replace two add instructions with one three-way add instruction. | Increases<br><br>The number of instructions will be reduced, but the number of stall cycles will be mostly unchanged.<br><br>-0.5 if said would increase because the instruction was implemented to take multiple cycles with only 2 read ports (not a great implementation)<br><br>-1.5, if said would stay the same | Could increase, or maybe same<br><br>The instruction might require a slightly long cycle time, either because register file has an extra read port or because the 3-way adder takes slightly longer. But might be same if critical path somewhere else (e.g., memory stage). |

| | | | | |
|---|---|---|---|---|
| C | Change load/store instructions to only use the address in a register (i.e., no offset). | Increase<br><br>More instructions are required to replace base+offset calculation in load/store instructions. | Decrease<br><br>As all address arithmetic instructions that are added will take fewer cycles than original base+offset loads (number of instructions increases, but number of stalls stays the same).<br><br>Also, pipeline can be simplified to have memory stage in parallel with ALU stage, reducing some load-use delays, further reducing CPI. | Likely improves, might stay the same<br><br>Removing one pipeline stage removes a number of bypasses which are often on critical path.<br><br>-0.5 for saying same, as most likely will reduce.<br><br>-1 for not seeing change in pipeline structure and fewer bypasses. |
| D | Remove hardware floating-point instructions and instead use software subroutines for floating-point arithmetic | Increase<br><br>More instructions required to emulate floating-point instructions. | Decrease<br><br>Integer instructions have much lower CPI than floating-point instructions in general.<br><br>-1.5 for No effect | No effect<br><br>Usually stays the same as FPU is pipelined to not be on critical path<br><br>-0.5 for Decrease |

Name _____

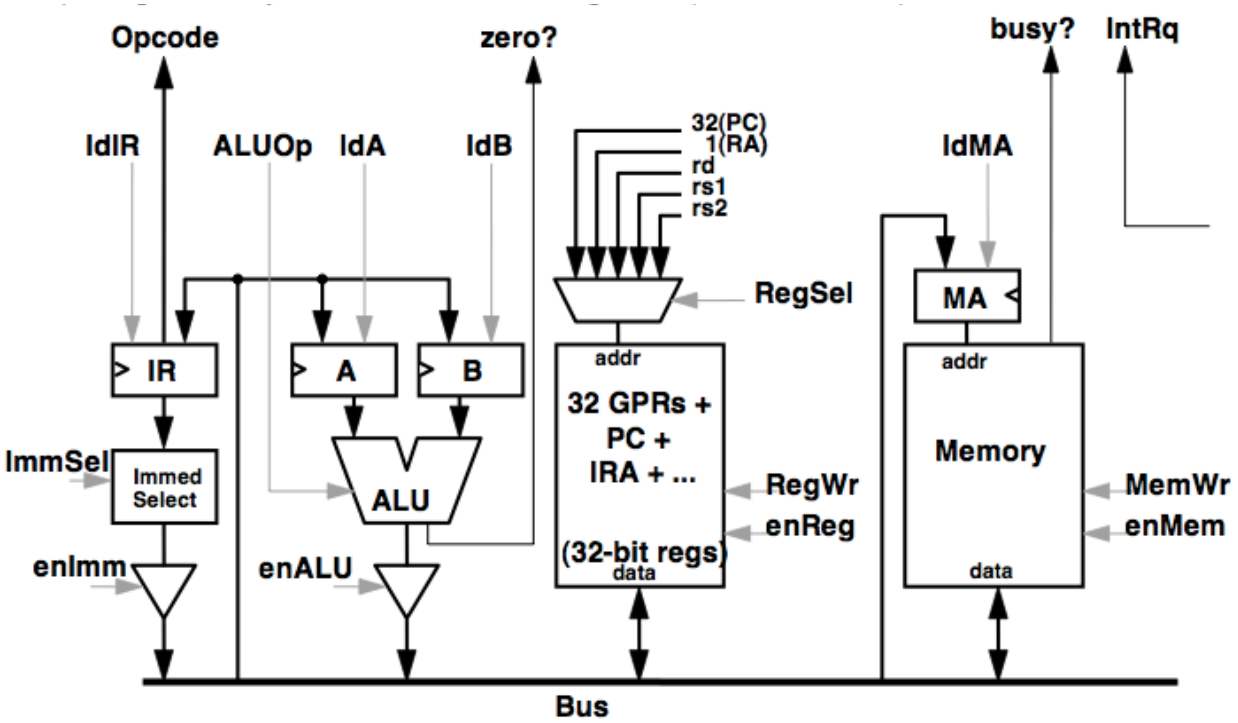
## Appendix A. A Cheat Sheet for the Bus-based RISC-V Implementation

Remember that you can use the following ALU operations:

| ALUOp | ALU Result Output |
|---|---|
| COPY_A | A |
| COPY_B | B |
| INC_A_1 | A+1 |
| DEC_A_1 | A-1 |
| INC_A_4 | A+4 |
| DEC_A_4 | A-4 |
| ADD | A+B |
| SUB | A-B |
| SLT | Signed(A) < Signed(B) |
| SLTU | A < B |

Table A1: Available ALU operations

Also remember that ⌈Br (*micro*branch) represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S. If ⌈Br is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column. If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal. If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1). If ⌈Br is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. If S, the ⌈PC spins if *busy?* is asserted, otherwise goes to (*current state* + 1).



Name _____

# Appendix B. CS152 RISC-V Instruction Table

| 31 27 | 26 22 | 21 17 | 16 15 | 14 12 | 11 10 | 9 8 | 7 6 0 | |
|---|---|---|---|---|---|---|---|---|
| jump target | | | | | | | opcode | J-type |
| rd | LUI-immediate | | | | | | opcode | LUI-type |
| rd | rs1 | imm[11:7] | imm[6:0] | | | funct3 | opcode | I-type |
| imm[11:7] | rs1 | rs2 | imm[6:0] | | | funct3 | opcode | B-type |
| rd | rs1 | rs2 | funct10 | | | | opcode | R-type |
| rd | rs1 | rs2 | rs3 | | funct5 | | opcode | R4-type |

**Control Transfer Instructions**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm25 | | | | | 1100111 | J imm25 |
| imm25 | | | | | 1101111 | JAL imm25 |
| imm12hi | rs1 | rs2 | imm12lo | 000 | 1100011 | BEQ rs1,rs2,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 001 | 1100011 | BNE rs1,rs2,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 100 | 1100011 | BLT rs1,rs2,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 101 | 1100011 | BGE rs1,rs2,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 110 | 1100011 | BLTU rs1,rs2,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 111 | 1100011 | BGEU rs1,rs2,imm12 |
| rd | rs1 | imm12 | | 000 | 1101011 | JALR.C rd,rs1,imm12 |
| rd | rs1 | imm12 | | 001 | 1101011 | JALR.R rd,rs1,imm12 |
| rd | rs1 | imm12 | | 010 | 1101011 | JALR.J rd,rs1,imm12 |

**Memory Instructions**

| | | | | | | |
|---|---|---|---|---|---|---|
| rd | rs1 | imm12 | | 010 | 0000011 | LW rd,rs1,imm12 |
| imm12hi | rs1 | rs2 | imm12lo | 010 | 0100011 | SW rs1,rs2,imm12 |

**Integer Compute Instructions**

| | | | | | | |
|---|---|---|---|---|---|---|
| rd | rs1 | imm12 | | 000 | 0010011 | ADDI rd,rs1,imm12 |
| rd | rs1 | 000000 | shamt | 001 | 0010011 | SLLI rd,rs1,shamt |
| rd | rs1 | imm12 | | 010 | 0010011 | SLTI rd,rs1,imm12 |
| rd | rs1 | imm12 | | 011 | 0010011 | SLTIU rd,rs1,imm12 |
| rd | rs1 | imm12 | | 100 | 0010011 | XORI rd,rs1,imm12 |
| rd | rs1 | 000000 | shamt | 101 | 0010011 | SRLI rd,rs1,shamt |
| rd | rs1 | imm12 | | 110 | 0010011 | ORI rd,rs1,imm12 |
| rd | rs1 | imm12 | | 111 | 0010011 | ANDI rd,rs1,imm12 |
| rd | rs1 | rs2 | 0000000 | 000 | 0110011 | ADD rd,rs1,rs2 |
| rd | rs1 | rs2 | 1000000 | 000 | 0110011 | SUB rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 001 | 0110011 | SLL rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 010 | 0110011 | SLT rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 011 | 0110011 | SLTU rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 100 | 0110011 | XOR rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 101 | 0110011 | SRL rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 110 | 0110011 | OR rd,rs1,rs2 |
| rd | rs1 | rs2 | 0000000 | 111 | 0110011 | AND rd,rs1,rs2 |
| rd | imm20 | | | | 0110111 | LUI rd,imm20 |