Name Solution

*Formatted For 2-Sided Printing*

*Staple This Side*

# Computer Architecture
# LSU EE 4720
# Midterm Examination

Friday, 20 March 2026 9:30-10:20 CDT

| | | |
|---|---|---|
| Problem 1 | _____ | (20 pts) |
| Problem 2 | _____ | (30 pts) |
| Problem 3 | _____ | (30 pts) |
| Problem 4 | _____ | (20 pts) |
| Exam Total | _____ | (100 pts) |

Alias Don't bet on it.

*Good Luck!*

*Staple This Side*

Problem 1: [20 pts] The MIPS code below is based on the solution to Homework 1.

☑ Show the execution of this code, ▷ including the instructions before `LOOP`, with ▷ the `beq` **never** taken and ▷ the `bne` **always** taken, on the MIPS implementation on the facing page for ☑ enough iterations to compute instruction throughput (IPC).

☑ Check for dependencies, including branch sources. ☑ Base branch and bypass behavior on the implementation shown.

The solution appears below. Notice that in the first iteration the `lb` before the loop causes the `beq` to stall but in the second iteration the `lb` from the first iteration stalls `beq`, and because it is closer the stall is for two cycles.

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

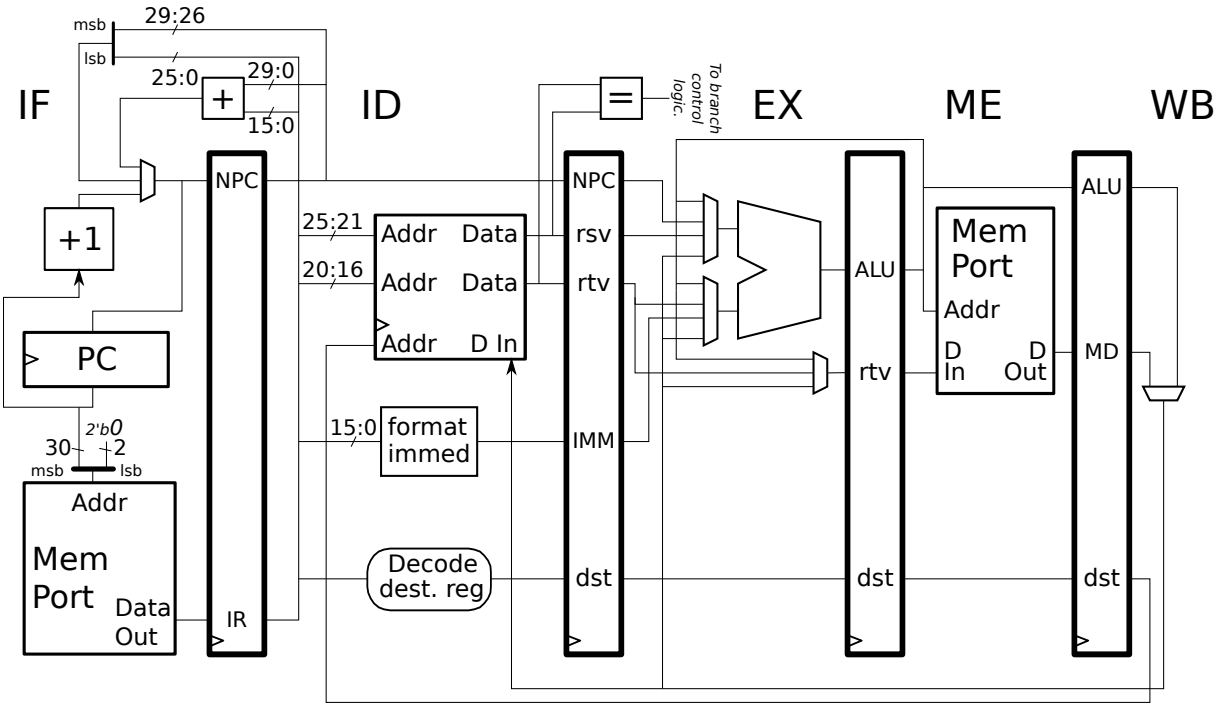
*Full-Credit Answer:* The throughput is $\frac{4\text{ insn}}{(14-8)\text{ cyc}} = \frac{2}{3}$ insn/cycle.

*Explanation:* Instruction throughput is the rate of executing instructions in the units of instructions per cycle (IPC). The number of instructions in an iteration is four. An iteration starts when the first instruction, `beq`, is in `IF`. The start time for the first three iterations are cycle 3, cycle 8, and cycle 14. The number of cycles for the first iteration is $8 - 3 = 5$ and the number of cycles for the second iteration is $14 - 8 = 6$. The state of the pipeline at the start of the first and second iterations is clearly different. (In cycle 3 the pipeline contains the three instructions before the loop, but in cycle 8 in contains a loop iteration.) But the state is the same in cycles 8 and 14, so we can conclude that the third iteration will take as much time as the second. So the throughput is $\frac{4\text{ insn}}{(14-8)\text{ cyc}} = \frac{2}{3}$ insn/cycle.

```
# SOLUTION
# Cycle              0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 addi $a0, $a0, -1    IF ID EX ME WB
 lb $t0, 0($a0)          IF ID EX ME WB
 ori $t4, $0, 41            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
  beq $t0, $t5, fo_comma       IF ID -> EX ME WB             # First Iteration
  addi $a0, $a0, 1                IF -> ID EX ME WB
  bne $t0, $t4, LOOP                    IF ID EX ME WB
  lb $t0, 0($a0)                           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
  beq $t0, $t5, fo_comma                      IF ID ----> EX ME WB  # 2nd Iter
  addi $a0, $a0, 1                               IF ----> ID EX ME WB
  bne $t0, $t4, LOOP                                      IF ID EX ME WB
  lb $t0, 0($a0)                                             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
  beq $t0, $t5, fo_comma                                        IF   # 3rd Iter
```




Staple This Side



Problem 2: [30 pts] The MIPS implementation to the right is similar to our bypassed MIPS, but with bypass multiplexors in the ID stage and changes in the bypass connections from WB.

(*a*) The bypass from WB, in green, is different from the *Problem 1 (P1)* implementation on the previous page.

☑ In the execution diagram below show a code fragment that stalls in this implementation but not the P1 implementation ☑ due to the differences in the bypass from WB. ☑ Don't forget to show registers. ▷ A correct solution can use fewer than four instructions.

The solution appears below. Note that unlike the P1 implementation, here a value cannot be bypassed from WB.MD to EX. To produce the stall a load instruction is placed so that it is in WB in cycle 5.

```
# Cycle            0  1  2  3  4  5  6  7  8
nop                IF ID EX ME WB             # Only 3 instructions needed.

lw R1, 0(r2)          IF ID EX ME WB          # Solution

xor r3, r4, r5           IF ID EX ME WB       # Solution

add r6, r7, R1              IF ID -> EX ME WB # Solution. Stalls here, but not P1.
# Cycle            0  1  2  3  4  5  6  7  8
```

☑ In the execution diagram below show a code fragment in which a branch does not stall on this implementation (thanks to the changes in ID), but would stall on the P1 implementation. ☑ Don't forget to show registers. ▷ A correct solution can use fewer than four instructions.

The solution appears below. The bypasses into the ALU don't help the branch instructions, but those new ME to ID bypasses do. So, while the beq is in ID, in cycle 4, place an instruction that writes r2 so that it is ME in cycle 4, that's the add.

```
# Cycle            0  1  2  3  4  5  6  7  8
                   IF ID EX ME WB

add R2, r3, r4        IF ID EX ME WB          # Solution

xor r5, r6, r7           IF ID EX ME WB       # Solution

beq r1, R2, TARG            IF ID EX ME WB    # No stall here, but would in P1
# Cycle            0  1  2  3  4  5  6  7  8
```

(*b*) With the bypass multiplexors in ID it is possible to eliminate inputs to the multiplexors in EX.

☑ In EX cross out multiplexor inputs (to the left of the little numbers) that are no longer needed because of the ID-stage multiplexors.

Solution shown in red.

(*c*) Design the following control logic:

☑ Design control logic for select signal X (in ID). ☑ A correct solution needs only a small amount of hardware.

The solution shown in the diagram. The X signal is set to 1 when the rt register number of the instruction in ID matches the destination of the instruction in EX. Though it's not necessary for this part, the logic also makes sure that the instruction in ID uses rt register as a source.



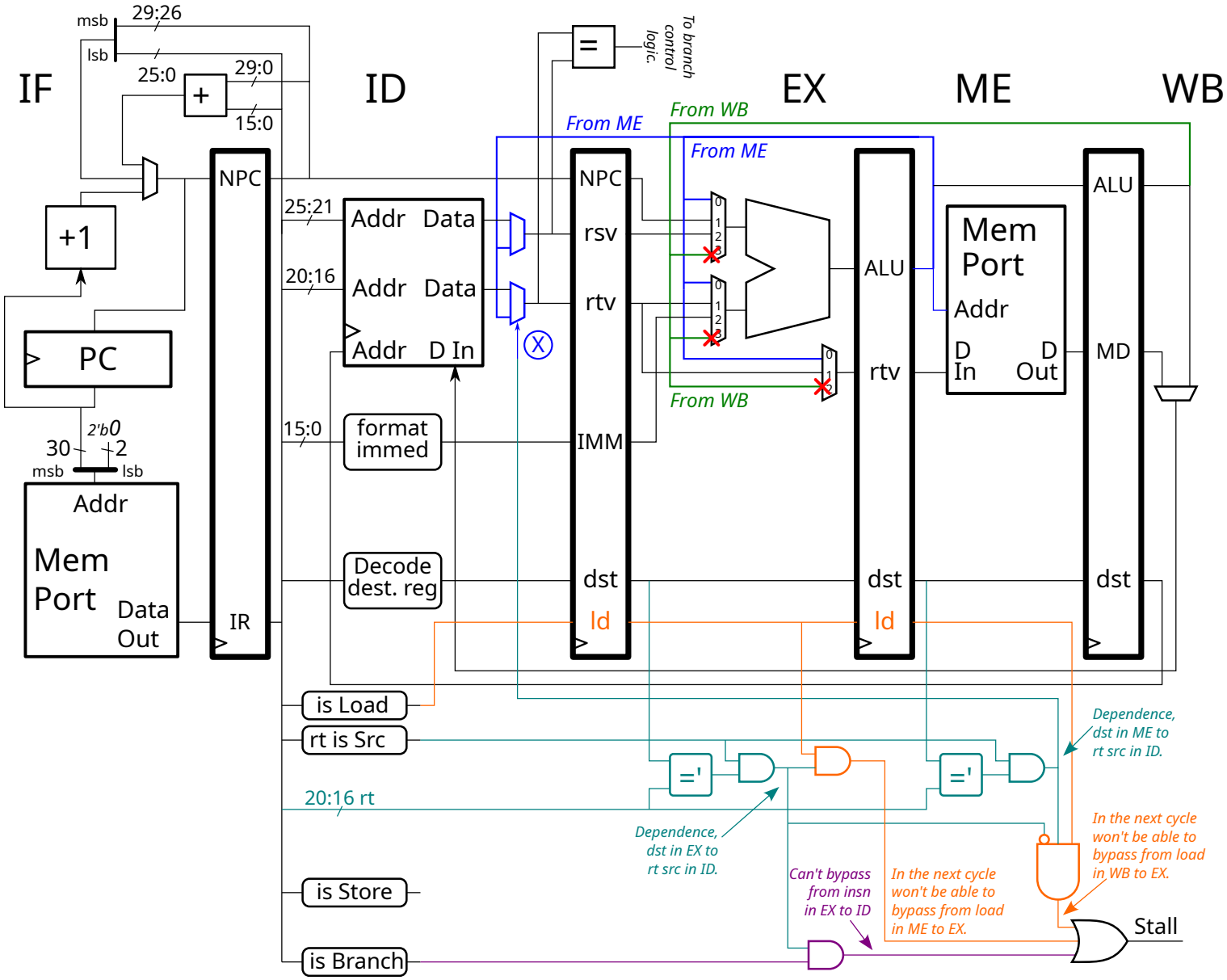

Staple This Side

☑ Show hardware for a stall signal due to unbypassable load value dependencies ▷ with `rt` sources. That hardware should be an input to the OR gate on the right.

The solution is shown in orange. The `AND` gate with three inputs produces the stall that is needed here but not on the `P1` implementation. The output of that three-input gate is `1` during cycle 4 of the code fragment from part a. The orange AND gate connecting to the `EX` stage generates stalls that would occur in this and the `P1` implementation. Since the load produces the value, we need to check for its presence in `EX` and `ME`, and for that the `is Load` value is sent down the pipeline.

☑ Show hardware for a stall signal due to unbypassable branch `rt` sources. (The branch would be stalling.) That hardware should be another input to the OR gate on the right.

Solution appears in purple. Here the branch is the source, so we check for its presence in the `ID` stage.



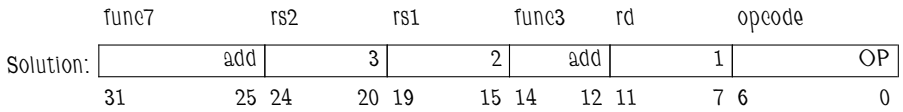
Problem 3: [30 pts] Appearing to the right is a RISC-V RV-32i implementation in which some instruction decoding logic is shown. Use the diagram to help answer the encoding questions below.

(*a*) Show encodings of the following instructions:

☑ Show the encoding of the instruction below, ☑ include bit positions and ☑ field names and ☑ field values that can be determined by the diagram and the instruction.

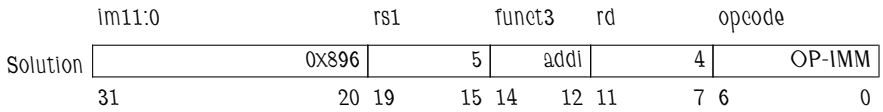
Endcodings shown below. Field values are based on what can be determined from the instruction and the diagram. For `add` the diagram reminds us that the opcode is `OP`, and we use instruction names for `func7` and `func3` for the `add`. For the `sh` we know that the opcode must be `STORE`, so the `func3` field must provide additional info: that the size is a half.

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



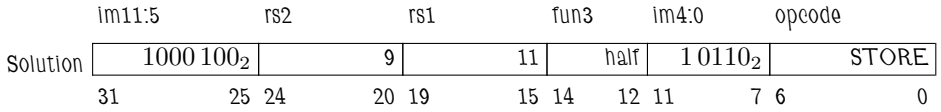

☑ Show the encoding of the instruction below, ☑ include bit positions and ☑ field names and ☑ field values that can be determined by the diagram and the instruction.

```
addi r4, r5, 0x896  # Note: 0x896 = Binary 1000,1001,0110
```




☑ Show the encoding of the instruction below, ☑ include bit positions and ☑ field names and ☑ field values that can be determined by the diagram and the instruction.

```
sh r9, 0x896(r11)  # Note: 0x896 = Binary 1000,1001,0110
```





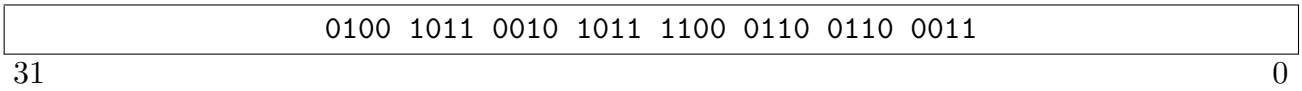


Staple This Side

(*b*) Based on the encoding below determine the registers and destination of the `blt` instruction.

☑ Determine ☑ the registers and ☑ the address of the destination (`TARG`). ☑ Show work for determining `TARG`.

```
0x1000000: blt r23  , r18      , TARG   # ☑ Fill in registers.
# Much further ahead.
TARG: add r10, r11, r12          # ☑ TARG = 0x10004ac
```




First group the bits into fields:

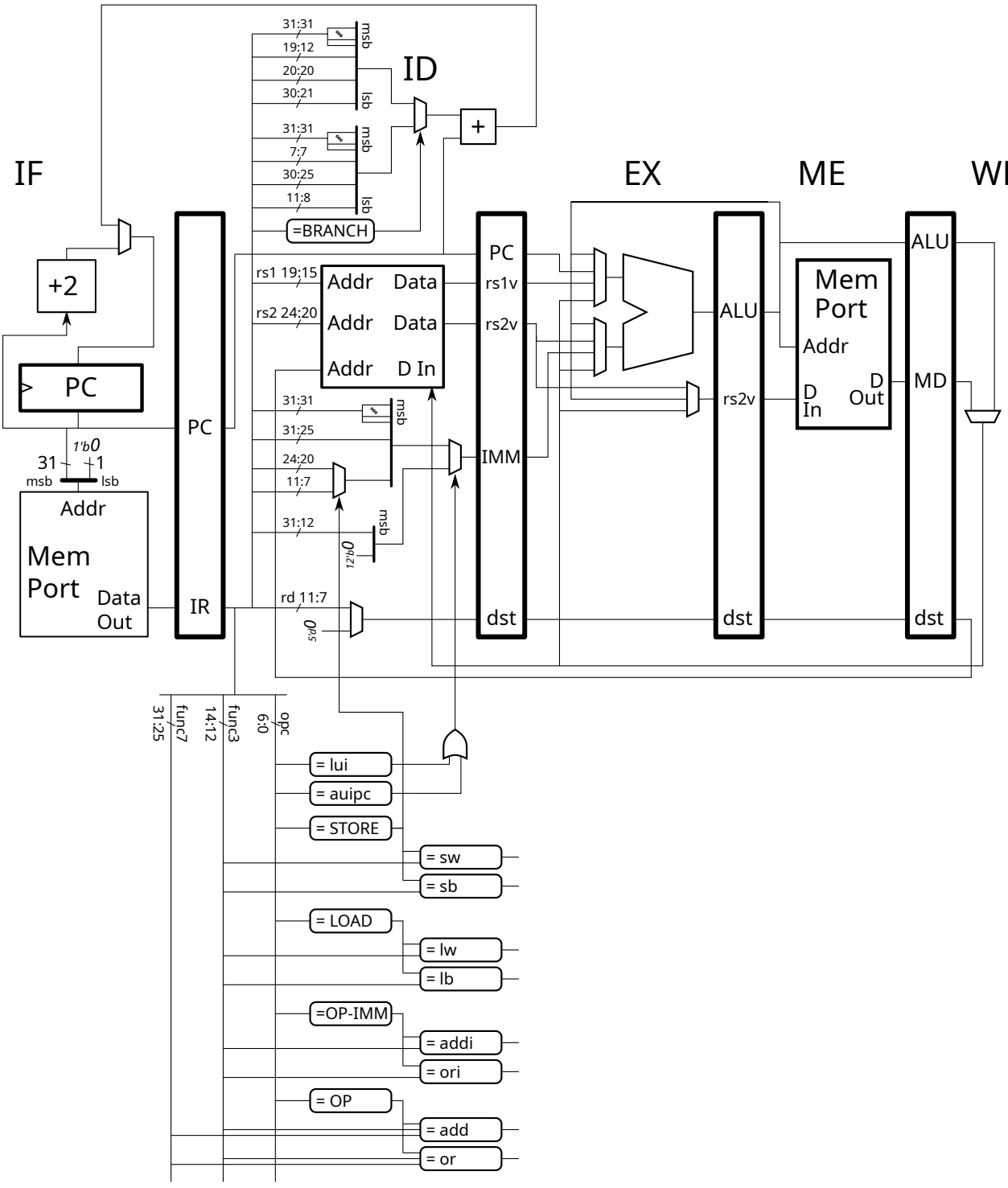
| | im12 | im10:5 | rs2 | rs1 | fun3 | im4:1 | im11 | opcode |
|---|---|---|---|---|---|---|---|---|
| Solution | $0_2$ | $10\,0101_2$ | $1\,0010_2 = 18_{10}$ | $1\,0111_2 = 23_{10}$ | $100_2$ | $0110_2$ | $0_2$ | $110\,0011_2$ |
| | 31 | 30 25 | 24 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 |

The branch target is formed by adding the address of the branch, $100\,0000_{16}$, to the displacement. The displacement is $2\times$ the immediate value. As one can tell from the implementation diagram below the immediate is in four pieces and not in order. The immediate labels on the encoding above uses RISC-V's convention of labeling immediate fields with the position of those bits in the immediate. For example, the field labeled `im12` holds the value to be placed at bit position 12 in the immediate and `im11` is to be placed at bit position 11. The displacement is formed by arranging the field values in numerical order by their names (such as `im12` to the left of `im11`), and putting a `0` in the least-significant position (there is no `im0`): `im12 im11 im10:5 im4:1 0` → `0 0 100101 0110 0` → `0 0100 1010 1100` → `0x14ac`. The value of `TARG` is then the branch address plus the displacement: $100\,0000_{16} + 4ac_{16} = 100\,04ac_{16}$.

*In the original exam the most significant bit of the instruction,* `im12`*, was a* `1` *and as a result the displacement was negative. Here is how the question should have been answered for that case:*

*The displacement is formed by duplicating* `im12` *so that it fills 20 bits, appending the remaining field values in numerical order by their names, and placing a* `0` *in the least significant position:* `im12 im12` $\cdots$ `im12 im11 im10:5 im4:1 0` → `1111 1111 1111 1111 1111 100101 0110 0` → `1111 1111 1111 1111 1111 0100 1010 1100` → `0xffff f4ac` → `-0xb54`. *Note that* $\text{ffff f4ac}_{16}$ *is a 2's complement signed number (and it's negative), its sign-magnitude value is* $-\text{b54}_{16}$*. The value of* `TARG` *can be computed using addition of the two's complement representation* $100\,0000_{16} + \text{ffff f4ac}_{16} = \text{ff f4ac}_{16}$ *or by subtraction of the sign-magnitude representation* $100\,0000_{16} - b54_{16} = \text{ff f4ac}_{16}$.

The RISC-V implementation and more problems on next page.








Staple This Side

(*c*) Answer the questions below.

☑ The diagram shows two instructions with the `LOAD` opcode. Based on the diagram, what are the maximum number of instruction that can have the `LOAD` opcode? *Note: The "two instructions" sentence did not appear in the original exam.*

Load instructions use the `opc` and `func3` fields. Since the `func3` field is 3 bits, the maximum number of instructions is $2^3 = 8$.

☑ Some immediate bits for store instructions are in a different place than they are for loads and immediate instructions such as `addi`. What is the benefit of that extra complication?

*Full-Credit Answer:* The benefit is having register fields in the same place in all instructions. In particular, the second source register is in 24:20 for all instructions that need one, and the destination register is in 11:7 for all instructions that need one.

*Long Answer:* Most instructions that use an immediate have just one source register. Since they don't need to use the `rs2` field those bit positions, 24:20, can be used for the immediate. But store instructions are an exception: the second source register is the value to store. Since they don't have a destination register the five immediate bits that are placed in 24:20 (`rs2`) in other immediate instructions are placed in 11:7 (`rd`) in store instructions.

Problem 4: [20 pts] Answer each question below.

(*a*) Since the 1960s the accepted practice has been to first define an ISA, and then to have many implementation of that ISA.

☑ Is the number of stages a feature of an ISA or of an implementation?

*Full-Credit Answer:* It is a feature of the implementation.

*More Info:* An ISA (instruction set architecture) describes the result of executing instructions, including how registers and memory are changed. An implementation of an ISA is a piece of hardware that executes instructions *as defined by the ISA*. "Number of stages" refers to hardware, and so is a feature of the implementation. Note that the number of stages an implementation might have, or whether it's pipelined at all, will affect cost and performance, but it does not affect what instructions actually do (assuming that the implementation is correct).

☑ For a company, which wants to make money, why is it better to have many implementations of one ISA, than to define a new ISA for each implementation?

*Full-Credit Answer:* It rewards customer loyalty since they don't need to re-compile or re-purchase software when replacing old implementation $A$ with new implementation $B$.

*Explanation:* This answer assumes that the company's competitors products implement different ISAs, and so a customer switching to a competitor's product will always have to re-compile software. The reality is a bit more complicated because some ISAs are licensed by their owners to multiple companies, such as ARM, and because US copyright and patent law allows ISAs to be essentially copied, as AMD sort of did (I'm not a lawyer) with Intel 64 and once upon a time Amdahl did with IBM 360. There are also free or open ISAs, such as RISC-V or once upon a time Sun SPARC (originally proprietary. *Remember, this is not part of a full credit answer, just some background.*

☑ The number of bits in a register is considered an ISA feature. What would be the harm of making the number of bits in a register an implementation feature instead?

*Full Credit Answer:* Naieve code that ran correctly on one implementation of such an ISA might not run correctly on another that uses fewer bits per register. Calling the programmer naieve for ignoring the register size it technically correct but not good for business.

Suppose in this ISA an integer register could be between 10 and 32 bits. A special instruction returns the number of bits in an integer register. Code for this hypothetical ISA would need to check the register size, and if the register size were too small, say less than 25 bits, either refuse to run further (maybe print an error message about buying a better system) or else would need to have separate routines in which larger values are split across multiple registers. Any code that did not do that might run on one implementation but not another. None of this is a problem when the number of bits per register is an ISA feature, which is typical practice.

(*b*) Assume that the code fragment below runs correctly.

```
                    # Original program, but with comments.
lw r1, 0(r10)       # Load a value from memory into r1
mtc1 r1, f3         # Copy value in r1 into f3.  Just copy, don't change bits.
lwc1 f2, 4(r10)     # Load a value from memory into f2.
cvt.s.w f4, f2      # Convert f2 from integer (w)  to single FP (s), put in f4.
add.s f5, f3, f4    # Add, assuming f3 and f4 are single-precision FP.
```

☑ In what representation is the number loaded into `r1`?

The number loaded into `r1` must be in a single-precision floating-point representation. (Reason further below.)

☑ In what representation is the number loaded into `f2`?

The number loaded into `r1` must be in an integer representation. (Reason further below.)

☑ Why was `cvt.s.w` used on `f2` but not on `f3`.

*Full-Credit Answer:* Because the value in `f2` is in an integer representation, whereas the value in `f3` was already floating-point. We know this because the problem stated that the code was correct.

*Explanation:* The `lw` and `lwc1` load values from memory into integer and floating-point registers, respectively. The values could be in any representation, integer, floating-point, string, etc. The `add` instruction works for integers, the `add.s` for single-precision floating-point numbers. It is the programmer's responsibility to make sure that the operands are in the right format. The `lwc1` loads an integer and the `cvt.s.w` converts it to FP. That's a conventional thing to do. Using `lw` to load the FP value is unusual and requires an extra `mtc`, but it's not wrong.

☑ Re-write the code fragment using fewer instructions.

```
# Solution
lwc1 f3, 0(r10)    # Use lwc1 to directly load into f3, avoiding the mtc1.
lwc1 f2, 4(r10)
cvt.s.w f4, f2
add.s f5, f3, f4
```

(*c*) In the MIPS code below assume that the load into `r9` executes correctly. The range of memory addresses from `r10` to `r10+128` are all valid. Multiples of 4 may help in solving the problem:

$$0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, \ldots$$

☑ Put an ✗ to the right of each instruction that **would not** execute correctly given that the `r9` load is correct.

☑ Explain.

*Full-Credit Answer:* Because `lw r9` succeeds the address in `r10` must be a multiple of 4, minus 1, such as $1003_{16}$. Any odd offset would result in an address that's neither a multiple of 4 (for the `lw`) nor a multiple of 2 (for the `lh`), those instructions are marked.

*Explanation:* MIPS loads and stores are aligned, meaning that the address of the item loaded or stored must be a multiple of the size of the item. So the addresses must be a multiple of 4 for `lw` and `sw` instructions and a multiple of 2 for `lh`, `lhu`, and `sh` instructions. Any address can be used for `lb`, `lbu`, and `sb` because any address is a multiple of 1. Of course, an address that satisfies alignment might still be bad for other reasons, such as attempting to access memory that has not been allocated.

The problem states that `lw r9` is correct, and so the address in `r10` must be a multiple of 4, minus 1, such as $1003_{16}$ so the address used by `lw r9` is a multiple of 4, for the example address $37_{10} + 1003_{16} = 25_{16} + 1003_{16} = 1028_{16}$. Any even offset, such as 8 and 42, would result in an odd address, and so would not work for any `lh` or `lw`.

```
lb r1,   0(r10)
lbu r2,  1(r10)
sb r3,   2(r10)
lh r4,   8(r10)   ✗
lh r5,  17(r10)
lh r6,  22(r10)   ✗
lh r7,  27(r10)
lw r8,  32(r10)   ✗
lw r9,  37(r10)   # This load executes correctly.
lw r11, 42(r10)   ✗
```





Staple This Side