Name _____ *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 _____

Good Luck!

Staple This Side

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

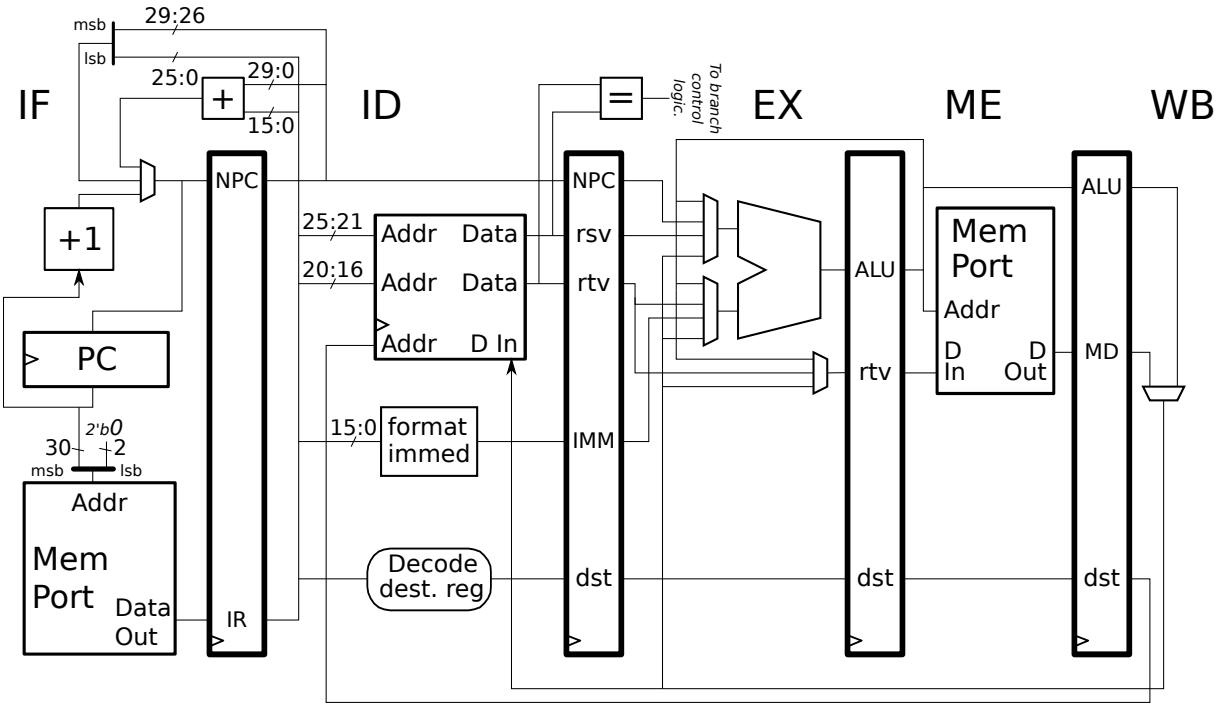
☐ Show the execution of this code, ▷ including the instructions before `LOOP`, with ▷ the `beq` **never** taken and ▷ the `bne` **always** taken, on the MIPS implementation on the facing page for ☐ enough iterations to compute instruction throughput (IPC).

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

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

```
 addi $a0, $a0, -1

 lb $t0, 0($a0)

 ori $t4, $0, 41

LOOP:
  # beq never taken.
  beq $t0, $t5, fo_comma

  addi $a0, $a0, 1

  # bne always taken.
  bne $t0, $t4, LOOP

  lb $t0, 0($a0)
```


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.

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

                      IF ID EX ME WB

                         IF ID EX ME WB

                            IF ID -> EX ME WB  # 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.

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

                      IF ID EX ME WB

                         IF ID EX ME WB

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.

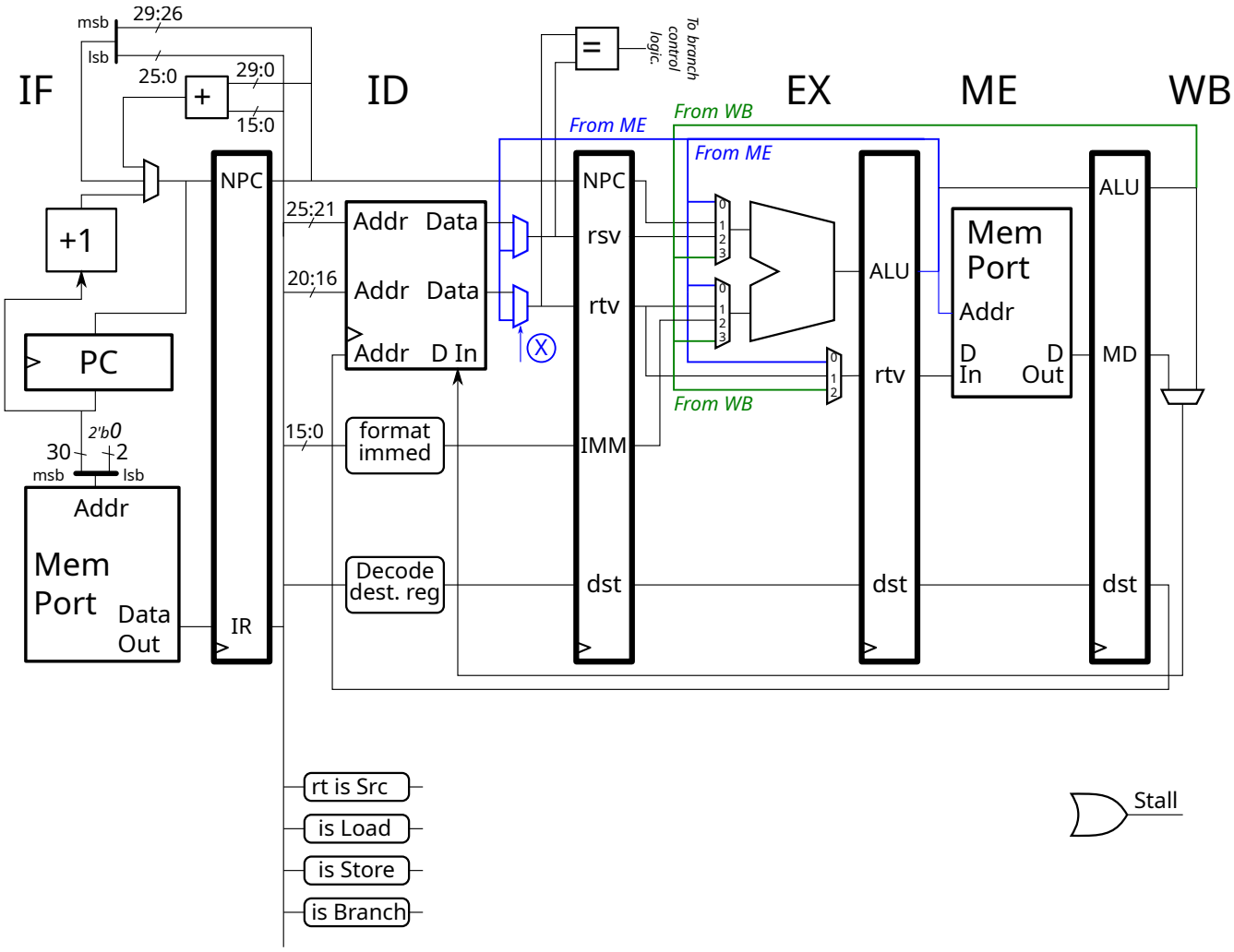
(*c*) Design the following control logic:

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

☐ 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.

☐ 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.

Staple This Side



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.

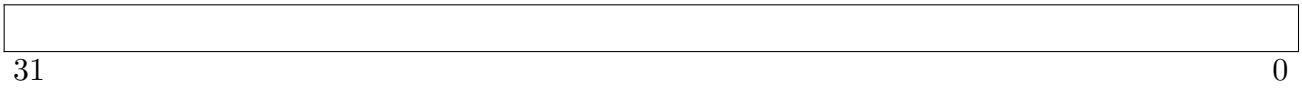
(*a*) Show encodings of the following instructions:

☐ 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.

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

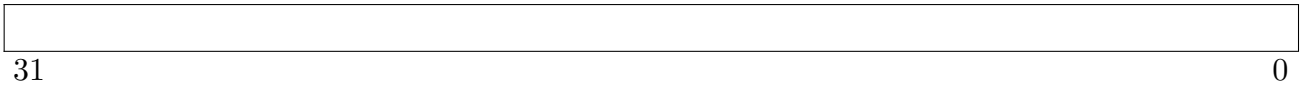
☐ Show the encoding of the instruction below, ☐ include bit positions and ☐ field names and ☐ field values that can be determined by the diagram and the instruction.

```
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.

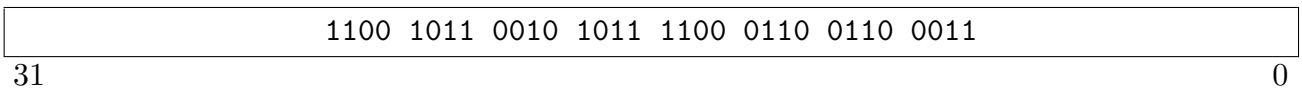
```
sh r9, 0x896(r11)  # Note: 0x896 = Binary 1000,1001,0110
```



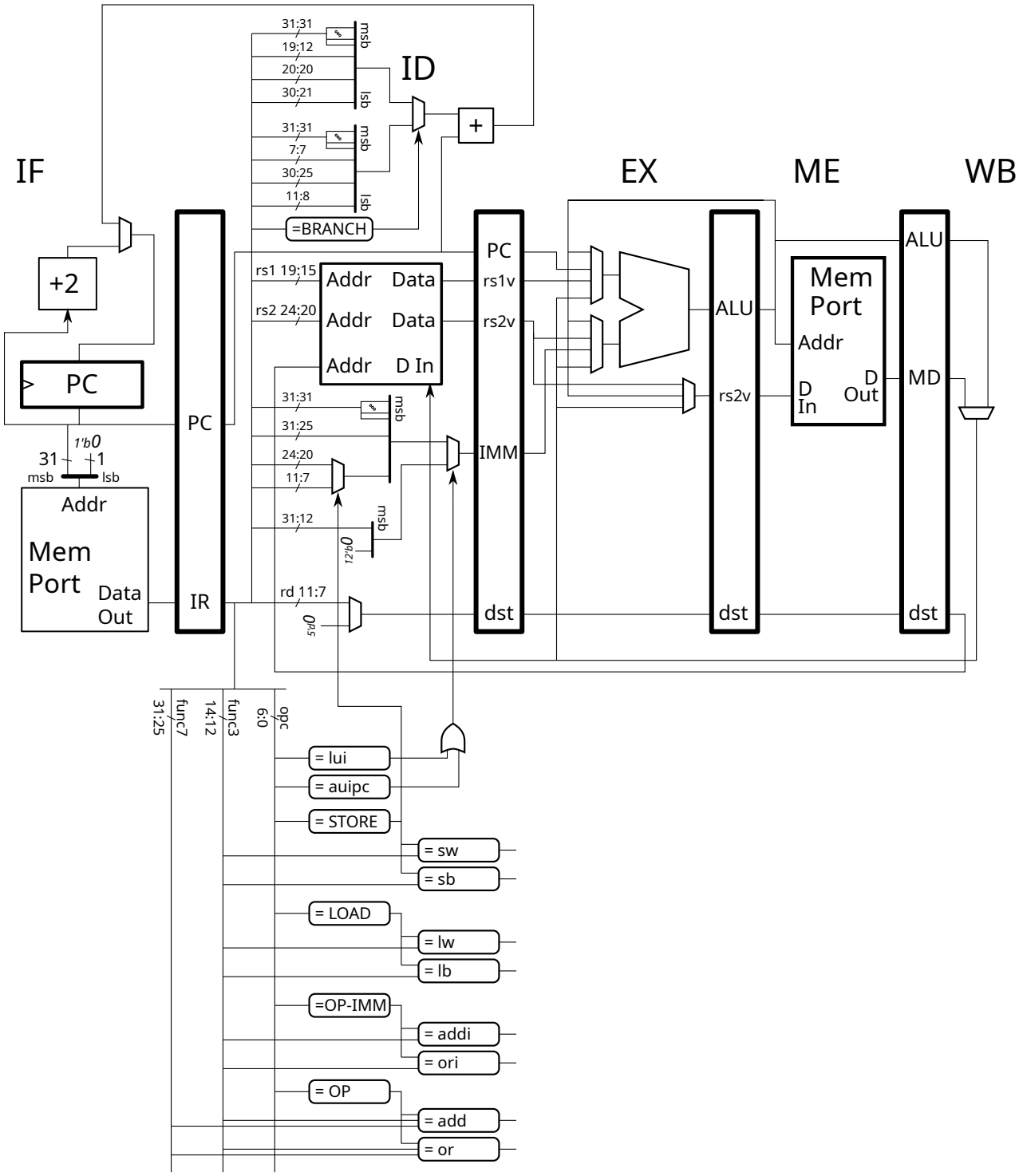
(*b*) Based on the encoding below determine the registers and destination of the `blt` instruction.

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

```
0x1000000: blt    ,      , TARG   #  Fill in registers.

# Much further ahead.
TARG: add r10, r11, r12          #  TARG =
```



Staple This Side



(*c*) Answer the questions below.

☐ Based on the diagram, how many kinds of load instruction can have the `LOAD` opcode?

☐ 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?

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?

☐ 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?

☐ 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?

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

```
lw r1, 0(r10)
mtc1 r1, f3
lwc1 f2, 4(r10)
cvt.s.w f4, f2
add.s f5, f3, f4
```

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

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

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

☐ Re-write the code fragment using fewer instructions.

Staple This Side

Staple This Side

(*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.

```
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 r10, 42(r10)
```

Staple This Side