Staple This Side

Name ______________________ *Formatted For 2-Sided Printing*

Computer Architecture
LSU EE 4720
Midterm Examination

Friday, 21 March 2025 9:30-10:20 CDT

Problem 1	______	(30 pts)
Problem 2	______	(30 pts)
Problem 3	______	(40 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Staple This Side

Problem 1: [30 pts] Appearing on the facing page is the MIPS implementation that includes the addsc from Homework 3.

(*a*) The first four code fragments below will execute as shown with the illustrated control logic (from the Homework 3 solution), but the logic won't generate the stall for the last fragment.

☐ Add control logic to the implementation so that *all* of the code fragments execute as shown. That is, add logic to generate the stall signal ☐ for the last fragment ☐ without changing whether the others stall.

```
lw R5, 8(r2)             IF ID EX ME WB         # Correctly stalls with existing logic.
addsc r3, r4, R5, 7         IF ID -> EX ME WB

lw R4, 8(r2)             IF ID EX ME WB         # Correct with existing logic.
addsc r3, R4, r5, 7         IF ID EX ME WB

xori R4, r2, 8           IF ID EX ME WB         # Correct with existing logic.
and r6, R4, r5              IF ID EX ME WB

lw R5, 8(r2)             IF ID EX ME WB         # Correctly stalls with existing logic.
xor r6, r4, R5              IF ID -> EX ME WB

# Cycle                  0  1  2  3  4  5  6
lw R4, 8(r2)             IF ID EX ME WB         # Should stall but doesn't with existing logic.
xor r6, R4, r5              IF ID -> EX ME WB
```

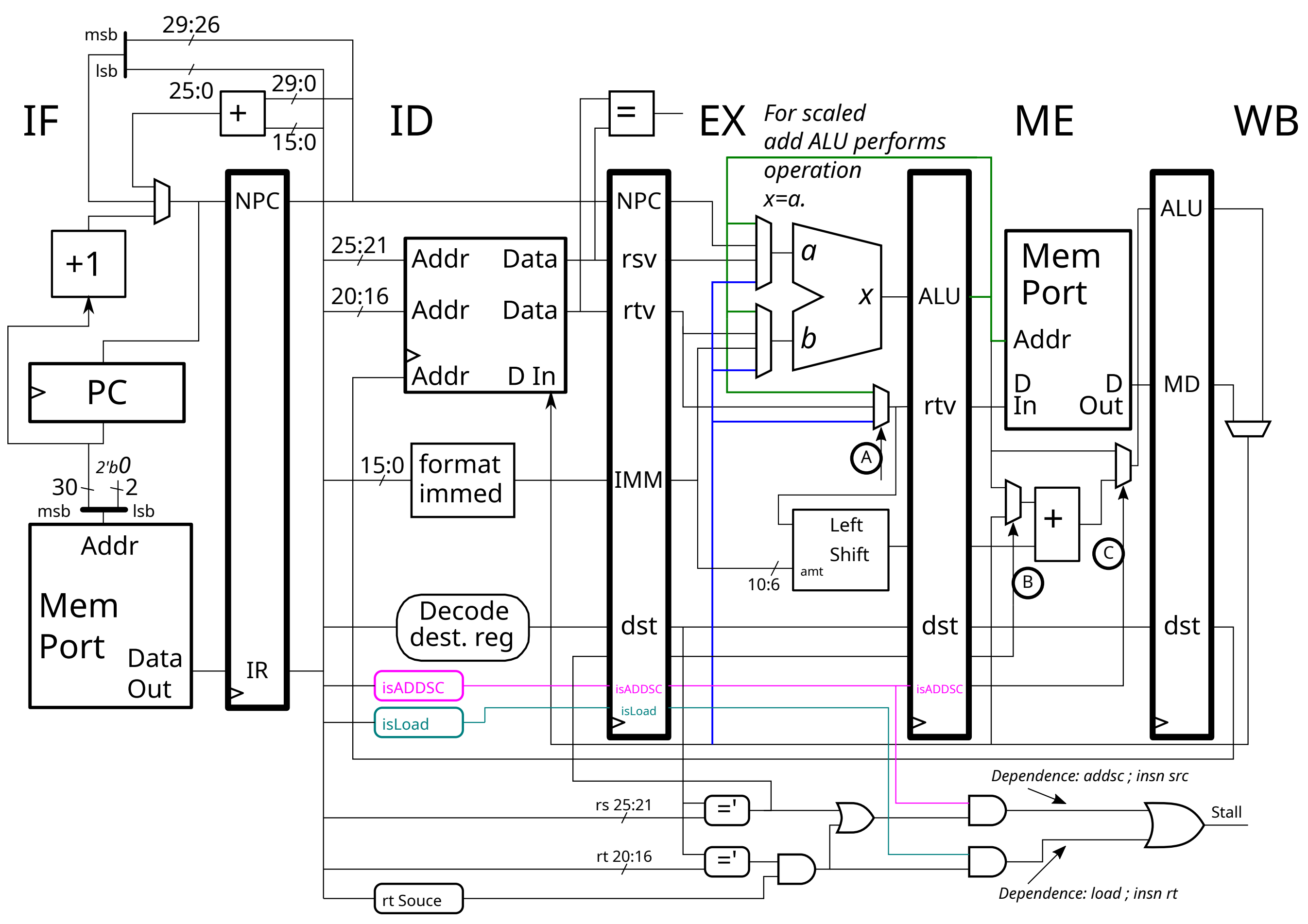
(*b*) Notice that in the first two fragments below the addsc shift amount is zero, and so those instructions just add. In the first fragment addsc executes in ME due to the load dependence, but in the second fragment it executes in EX so it can avoid stalling the or. *Note: The material about* doADDSC *described below was not in the original exam.*

☐ Modify the control logic so that an addsc with a zero shift executes as shown below. Do so by ☐ relabeling the isADDSC pipeline latches to doADDSC. Set this signal to 1 only if there is an addsc in ID that needs to execute in ME. ☐ The logic should not break correct behavior for other cases, such as the ones above.

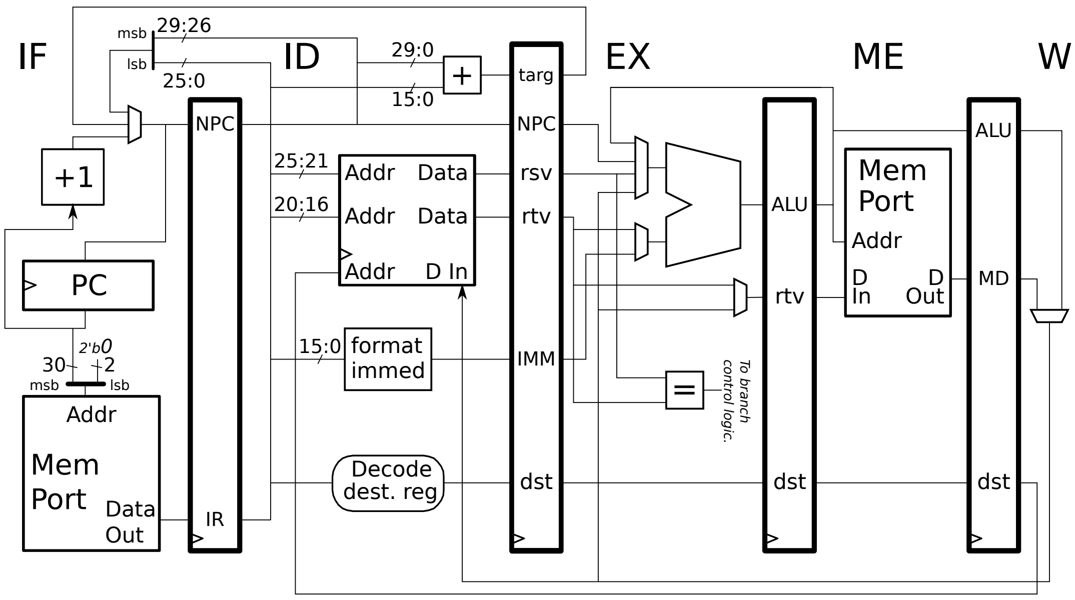
```
# Cycle                     0  1  2  3  4  5  6  7  Fragment b1 - addsc adds in ME.
lw R2, 8(r9)                IF ID EX ME WB
addsc r1, R2, r3, 0            IF ID EX ME WB
or    r5, r10, r6                 IF ID EX ME WB

# Cycle                     0  1  2  3  4  5  6  7  Fragment b2 - addsc adds in EX
andi R2, r9, 8              IF ID EX ME WB
addsc R1, R2, r3, 0            IF ID EX ME WB
or    r5, R1, R6                  IF ID EX ME WB

# Cycle                     0  1  2  3  4  5  6  7  Fragment b3 - addsc adds in ME.
lw R6, 8(r9)                IF ID EX ME WB
addsc R1, r2, r3, 4            IF ID EX ME WB
or    r5, R1, R6                  IF ID -> EX ME WB
```


IF
ID
EX
ME
WB
msb
lsb
29:26
25:0
+
29:0
15:0
NPC
+1
PC
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
IR
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
Decode
dest. reg
isADDSC
isLoad
=
NPC
rsv
rtv
IMM
dst
isADDSC
isLoad
For scaled
add ALU performs
operation
x=a.
a
b
x
A
Left
Shift
amt
10:6
ALU
rtv
dst
isADDSC
Mem
Port
Addr
D
In
D
Out
+
B
C
ALU
MD
dst
Dependence: addsc ; insn src
Dependence: load ; insn rt
Stall
rs 25:21
rt 20:16
='
='
rt Souce

Problem 2: [30 pts] In the MIPS implementation below pay attention to bypass paths and how the branch is resolved.



☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
add r1, r2, r3

sub r4, r1, r5

and r6, r7, r4

sw  r1, 0(r6)
```

☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
addi r1, r1, 1

sw r1, 0(r2)

sw r1, 4(r2)
```

☐ Show the execution of this code on the implementation above. ☐ Don't forget to check for dependencies!

```
lw r1, 0(r2)

lw r4, 4(r2)

sw r1, 0(r3)

sw r4, 4(r3)
```

☐ Show the execution of the code below with ☐ the branch taken on the implementation above. ☐ Don't forget to check for dependencies! ☐ Pay attention to branch behavior.

```
 add r1, r2, r3

 beq r1, r4, TARG

 sw r1, 0(r8)

 and r5, r1, r6

 ori r5, r5, 0x6

 sw r5, 4(r8)


TARG:
 lw r1, 8(r8)
```

☐ Show how the inputs to the $\boxed{=}$ box in `EX` can be changed to eliminate stall(s) ☐ in the example above, and stalls for other kinds of ☐ dependenices. ☐ Do not add hardware, just change the inputs.

Problem 3: [40 pts] Answer each question below.

(*a*) In the routine below `r4` holds an integer, call its value x, and `f1` holds a single-precision float, call its value y. Complete the routine so that register `f9` holds $x \times y$ in single-precision floating point.

☐ Complete the routine so that `f9` is written with the product of the values of `r4` and `f1`. The solution only requires a few instructions, ☐ don't try to fill the entire page.

```
add r4, r5, r5
add.s f1, f2, f3

# At this point r4 holds an integer and f1 holds a single-precision float.
```

Staple This Side

(*b*) The three MIPS code fragments below each do the same thing, and infinite loops are not the problem.

```
loop:   # Fragment A
        sw $t4, 0($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4

loop:   # Fragment B
        sw $t4, 0($t5)
        sw $t4, 4($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 8

loop:   # Fragment C
        sb $t4, 0($t5)
        sb $t4, 1($t5)
        sb $t4, 2($t5)
        sb $t4, 3($t5)
        bne $t5, $t3, loop
        addi $t5, $t5, 4
```

☐ Which code fragment is the fastest, ◯ *Fragment A*, ◯ *Fragment B*, or ◯ *Fragment C*?

☐ Which code fragment is the slowest, ◯ *Fragment A*, ◯ *Fragment B*, or ◯ *Fragment C*?

☐ Explain choice of ☐ fastest and ☐ slowest fragment, and ☐ include a good definition of fast.

☐ Assume that the contents of `t5` and `t3` refer to a range of valid memory addresses. Which fragment(s) put a restriction on the value of `t5`? ☐ Explain. Assume that `t3` is always chosen to avoid an infinite loop.

Staple This Side

(*c*) When designing a RISC ISA what is the most important criterion when considering possible instructions based on the material presented in class?

☐ Most important factor when deciding whether an instruction should be added to a RISC ISA.

☐ Give an example of an instruction unsuitable for RISC and ☐ explain how the criterion makes it unsuitable.

(*d*) CISC ISAs have powerful instructions, such as `add 4(r1), (r2), ((r3))` or a call instruction that automatically saves registers.

☐ What is the benefit of powerful instructions, especially in the days when memory was made by people sewing wires around little metal rings.

(*e*) Intel has updated IA-32 (a.k.a. x86) since the 1980s, and later added a 64-bit variant, Intel-64. Recall that nobody actually likes IA-32.

☐ So why did Intel's customers continue to buy implementations of IA-32 and Intel 64 rather than switching to a better-designed ISA? (Note that Apple is an exception to the rule that computer makers don't switch ISAs.)

(*f*) MIPS uses the `func` field as an opcode extension field.

☐ Why is an opcode extension field needed?

☐ Why didn't they just make the opcode longer when designing MIPS?