Name Solution

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 Does it have to be good?

Good Luck!

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.

Solution and discussion on the next page.

```
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.

Solution and discussion on the next page.

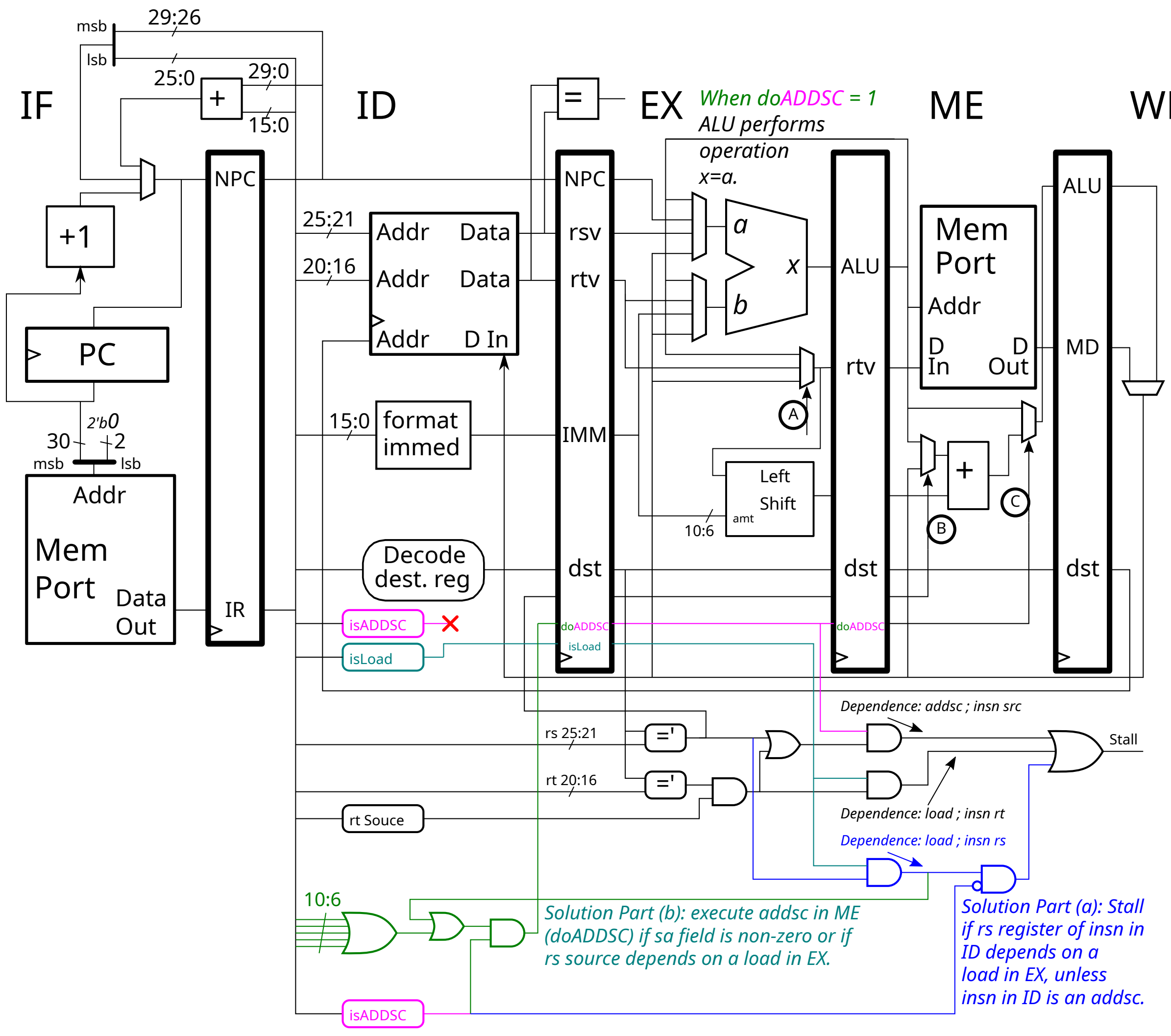
```
# Cycle                    0  1  2  3  4  5  6  7  Fragment b1 - addsc adds in ME.
lw R2, 8(r9)               IF ID EX ME WB
addsc r1, R2, r3, 0           IF ID EX ME WB
or    r5, r10, r6                IF ID EX ME WB

# 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
```

The solution to Part a appears below in blue. The last code fragment stalls because of a dependence between a load, `lw` in particular, and the rs register of an ALU instruction, `xor`. The stall occurs in cycle 2, when the `xor` is in ID and the `lw` is in EX. (The `xor` is also in ID during cycle 3, but it is not stalled, it's free to leave in the next cycle.) For this discussion refer to cycle 2 in the last code fragment. A new AND gate checks for the dependence, its output is commented "Dependence: load; insn rs." There should *not* be a stall if the instruction in ID is an `addsc` because it is possible to bypass the loaded value without a stall. That case is handled by the second blue AND gate which suppresses the stall if there is an `addsc` in ID.

The solution to Part b appears in green. The goal is to execute `addsc` in EX in some cases and ME in other cases. When `addsc` is executed in EX it is treated like an ordinary `add` instruction. The control signal `isADDSC` that appeared in the pipeline latches has been renamed to `doADDSC`. When `doADDSC=0` an `addsc` is treated like an `add` instruction, executing in EX. When `doADDSC=1` the ALU will execute the `x=a` operation and the C select signal will route the ME-stage adder output to the `WB.ALU` pipeline latch (otherwise it routes the `EX.ALU` value to the `WB.ALU` latch).

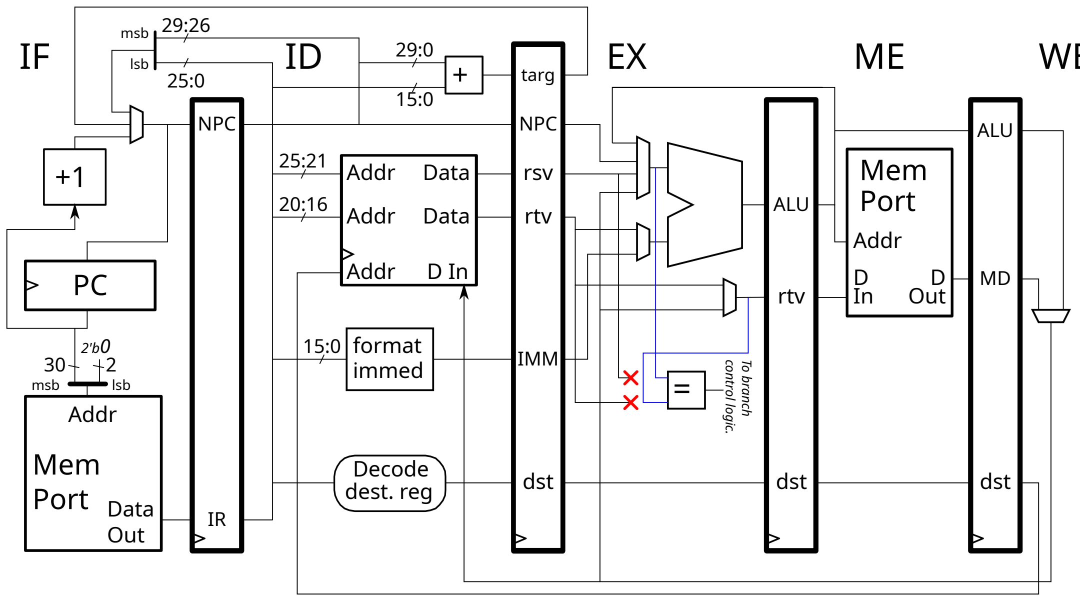
The value of `doADDSC` is set to 1 if there is an `addsc` in ID and either the `sa` field is non-zero (and so it must use the ME-stage adder) or the `rs` source of the `addsc` depends on a load in EX (and so uses the ME-stage adder to avoid a stall). The logic computing this is on the lower left.

Grading Note: Many students mistakenly thought that a stall signal was needed. The problem as given did not mention the `doADDSC` *signal which might have contributed to this mistake.*



IF
ID
EX
ME
WB
When doADDSC = 1 ALU performs operation x=a.
msb
lsb
29:26
25:0
29:0
15:0
+
+1
PC
NPC
IR
Addr
Mem Port
Data Out
30
2
2'b0
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
doADDSC
isLoad
a
b
x
A
Left Shift
amt
10:6
ALU
rtv
dst
Mem Port
Addr
D In
D Out
B
C
ALU
MD
dst
rs 25:21
rt 20:16
='
rt Souce
Dependence: addsc ; insn src
Dependence: load ; insn rt
Dependence: load ; insn rs
Stall
10:6
Solution Part (b): execute addsc in ME (doADDSC) if sa field is non-zero or if rs source depends on a load in EX.
Solution Part (a): Stall if rs register of insn in ID depends on a load in EX, unless insn in ID is an addsc.
isADDSC

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!

Solution appears below. The `and` stalls because its `rt` register, `R4`, could not be bypassed in cycle 4 since in this particular implementation there is no bypass path to the lower ALU input. The other dependencies are with the `rs` register, and those can be bypassed (to the upper ALU input).

```
## SOLUTION
#   Cycle          0  1  2  3  4  5  6  7  8  9
add R1, r2, r3    IF ID EX ME WB
sub R4, R1, r5       IF ID EX ME WB
and R6, r7, R4          IF ID ----> EX ME WB
sw  r1, 0(R6)              IF ----> ID EX ME WB
#   Cycle          0  1  2  3  4  5  6  7  8  9
```

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

Solution appears below. There is a dependency from the `addi` to the two `sw` instructions. The EX-stage mux routing a value to the `ME.rtv` input can bypass a value from WB but not ME, and so the first `sw` must stall one cycle. Note that there is no problem for the second `sw` because it is in ID when `addi` is in WB and so it can get the value from the register file.

```
## SOLUTION
#   Cycle           0  1  2  3  4  5  6  7
addi R1, r1, 1     IF ID EX ME WB
sw R1, 0(r2)          IF ID -> EX ME WB
sw R1, 4(r2)             IF -> ID EX ME WB
```

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

There are no stalls here because each `sw` can use the bypass from `WB`.

```
## SOLUTION
lw r1, 0(r2)   IF ID EX ME WB
lw r4, 4(r2)      IF ID EX ME WB
sw r1, 0(r3)         IF ID EX ME WB
sw r4, 4(r3)            IF ID EX ME WB
```

✓ 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.

Solution appears below. The `beq` stalls two cycles because there are no bypass paths to the `EX`-stage comparison unit needed by the branch. Also notice that the branch is resolved in `EX`, and so the correct target is not fetched until the branch is in `ME`. As a result the `and` instruction is fetched and then squashed.

```
## SOLUTION
#  Cycle          0  1  2  3  4  5  6  7  8  9  10
add R1, r2, r3    IF ID EX ME WB
beq R1, r4, TARG     IF ID ----> EX ME WB
sw R1, 0(r8)            IF ----> ID EX ME WB
and r5, r1, r6                   IFx
ori r5, r5, 0x6
sw r5, 4(r8)

TARG:
lw r1, 8(r8)                        IF ID EX ME WB
#  Cycle          0  1  2  3  4  5  6  7  8  9  10
```

✓ 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.

Solution appears on the diagram in blue. The `rs` value is taken from the mux at the upper ALU input. That works here because the ALU is not computing the branch target. The `rt` value is taken from the mux for the store value, which can bypass from `WB`. In contrast the lower ALU mux can't bypass at all.

With these changes the branch would not stall. However, because it is resolved in `EX` there would still be a squashed instruction.

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.

# SOLUTION
#
# Let x denote the value in r4.
# At this point r4 holds an integer representation of x.
#
# Let y denote the value in f1.
# At this point f1 holds a single-precision floating point representation of y.
#

mtc1 f4, r4         # Move x to f4.

# At this point f4 holds an integer representation of x.

cvt.s.w f5, f4      # Convert x to a single-precision FP representation.

# At this point f4 still holds an integer representation of x ...
# ... but f5 now holds a single-precision FP representation of x.

mul.s f9, f1, f5    # Perform the multiplication.
```

(*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.

Here *fast* will be defined as executing few instructions to complete a task. The task here is initializing an area of memory from `t5` to `t3` with the value in `t4`. (This is taken from the solution Homework 1.) So the fastest fragment is the one that executes the fewest instructions.

The number of iterations performed by the Fragment A and Fragment C loops is the same, call the number n. Because `t5` is incremented by 8 rather than 4, Fragment B runs for just $n/2$ iterations.

An iteration of Fragment A is 3 instructions, an iteration of B is 4 instructions, and an iteration of C is 6 instructions. The total number of instructions is $3n$ for A, $4\frac{n}{2} = 2n$ for B, and $6n$ for C. So B is the fastest and C is the slowest.

✓ 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.

MIPS memory accesses are aligned, meaning the memory address must be a multiple of the access size. The access size for a `sw` is 4 and so `t5` must be a multiple of 4. That restriction applies to Fragments A and B. The access size for `sb` is 1 and so Fragment C imposes no restriction on `t5`.

(*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.

In class it was explained that the most criterion for RISC is easy pipelining. Other RISC characteristics, such as fixed instruction size and restricting memory access to load and store instructions are ways of facilitating easy pipelining.

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

An instruction such as `add r1, (r3), (r4)` because that would require two memory accesses before the arithmetic operation. A pipelined implementation that could implement it would need two memory ports before the ALU. Memory ports are expensive, so it would not be worthwhile to pipeline it.

(*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.

Programs using such powerful instructions take up less space. Here is the equivalent MIPS code for the example instruction:

```
# add 4(r1), (r2), ((r3))

lw r12, 0(r2)  # Put value of (r2) into r12
lw r4, 0(r3)
lw r13, 0(r4)  # Put value of ((r3)) into r13
add r11, r12, r13
sw r11, 4(r1)
```

The MIPS equivalent uses 5 instructions taking up 20 bytes of memory. The size of the CISC instruction would be less, maybe 4 or 5 bytes.

One cannot easily claim that the *implementation* of a RISC ISA would be faster or slower than a CISC ISA. Current CISC implementations work by *cracking* CISC instructions into RISC-like *micro-ops* where they proceed through pipelined hardware. This adds to complexity (and the latency of cracking) but if your sales are high enough you can pay for the computer engineers to handle it.

(*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.)

Those Intel customers (or the customers' customers) had lots of software for IA-32 (and later Intel 64). (One notable Intel customer is IBM, using the processor in their PC. It is IBM's customers that had lots of software.) When they buy a newer implementation of IA-32 they can run their old software as is. But, if they switch to a new ISA then they must recompile or port their old software, a time-consuming distraction. To get a customer to switch ISAs the implementation of the new ISA would need to be so much better than the old ISA that its is worth the trouble.

The Apple Mac (Macintosh) did switch ISAs several times. It started with the Motorola 68020, then PowerPC, then Intel 64, and now uses Arm A64. This worked for Apple in part because many layers of the software is Apple-written, and they'd make the effort to port it to the newer ISA. Third-party software would call Apple-written libraries for many time-sensitive tasks (such as for the user interface) and the remaining parts could run under emulation until ported. (Emulated code is run by simulating the older ISA for which it was written.) And yes, there's also the reality distortion field that insured everyone would go along without complaining.

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

☑ Why is an opcode extension field needed?

The opcode field is 6 bits, allowing for only 64 instructions. The `func` field enables a greater number of instructions to be encoded.

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

In some formats, including Format I, they wanted to have as much space as possible for an immediate. If the opcode field were made larger the immediate would be smaller.