Name Partial Solution *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Thursday, 9 May 2024 17:30-19:30 CDT

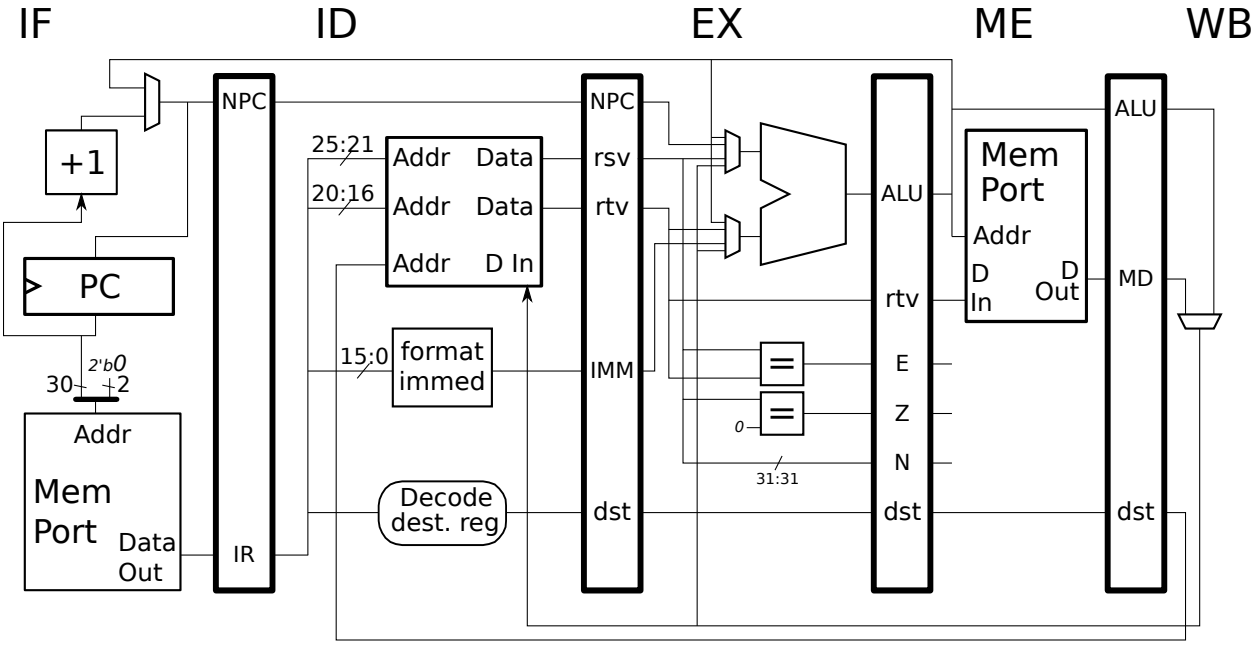
Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(8 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias Naturally Solved

Good Luck! Thank you for your effort in EE 4720!

Staple This Side

Problem 1: (20 pts) Appearing below are MIPS implementations and code fragments. Show execution (a pipeline execution diagram) of the code on the accompanying implementations.

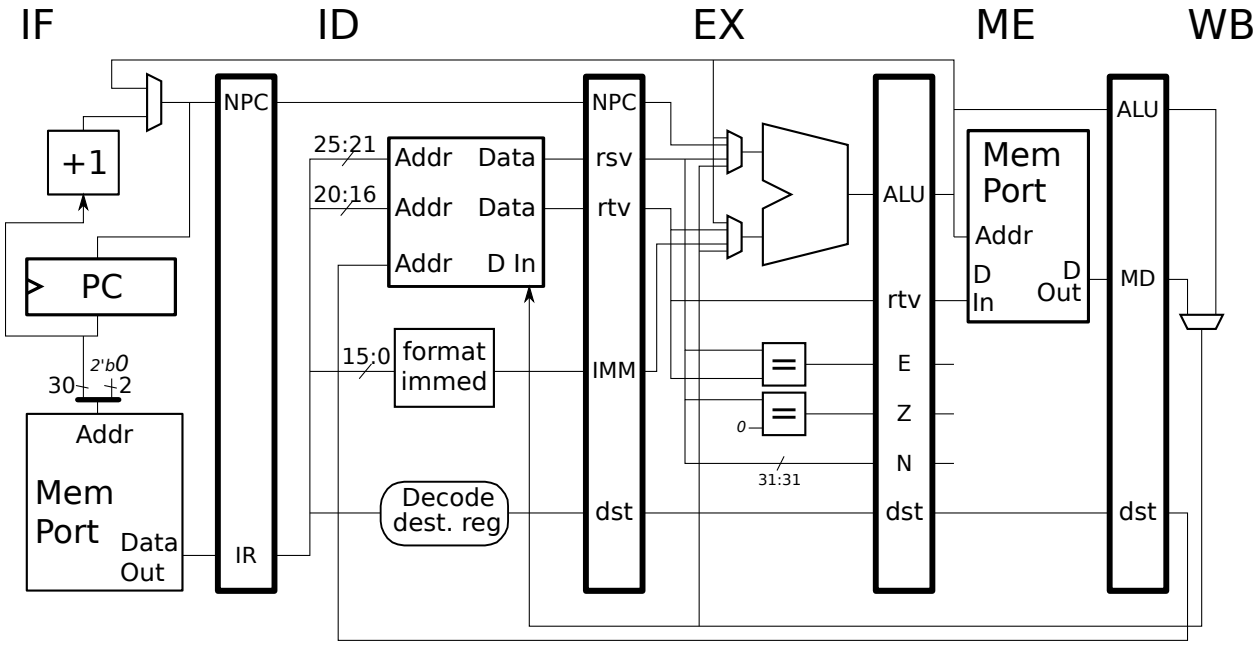


☑ Show execution of the code below on the implementation above. ☑ Check for dependencies.

The solution appears below. The implementation above can bypass to the ALU, which is why `lw` and `sub` do not stall. But, it can't bypass to the input of the ME-stage memory port, which is why the `sw` stalls.

```
# SOLUTION
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11
add r6, r7, r8   IF ID EX ME WB
lw r9, 0(r6)        IF ID EX ME WB
add r1, r2, r9         IF ID -> EX ME WB
sub r4, r1, r5            IF -> ID EX ME WB
sw r4, 0(r5)                    IF ID ----> EX ME WB
# Cycle          0  1  2  3  4  5  6  7  8  9  10 11
```


Staple This Side

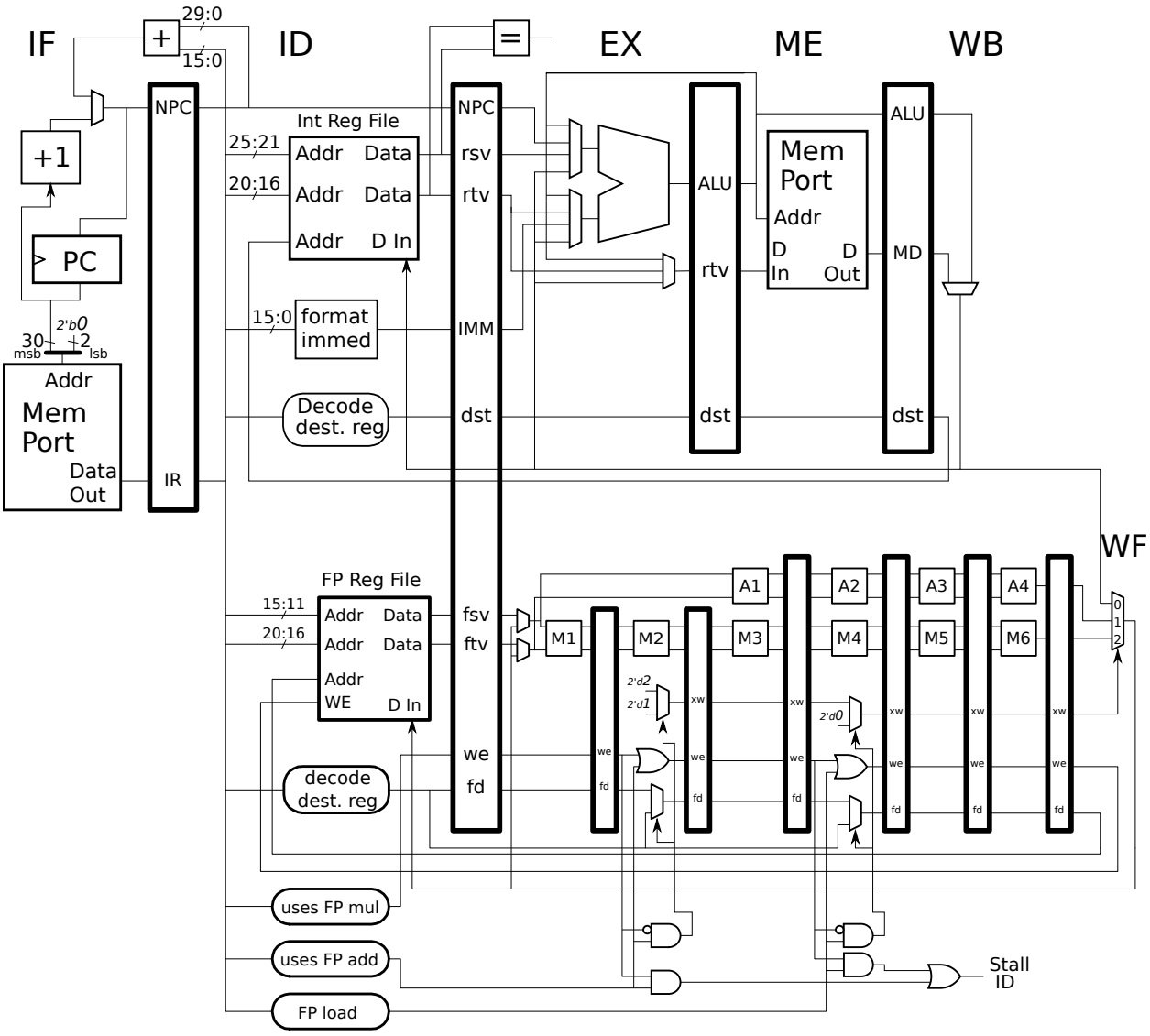


☑ Show execution of the code below on the implementation above ☑ until the second fetch of `lw`. ☑ Show when and where instructions are squashed (with an `x`). ☑ The branch must be taken. ☑ Pay close attention to branch behavior. ☑ Check for dependencies.

The solution appears below. In the implementation above the branch resolves in ME, so the branch target is fetched when the branch is in WB, which is cycle 6 in the execution below.

In MIPS the delay-slot instruction (`addi r1` below) executes whether or not the branch is taken. Because the branch resolves in ME two instructions had to be squashed.

```
 # SOLUTION
LOOP: # Cycle      0  1  2  3  4  5  6  7
 lw r3, 0(r4)      IF ID EX ME WB    IF
 addi r4, r4, 4       IF ID EX ME WB
 bne r1, r2, LOOP        IF ID EX ME WB
 addi r1, r1, r3            IF ID EX ME WB
 sw r3, 0(r9)                  IF IDx
 sw r4, 4(r9)                     IFx
 sw r2, 8(r9)
#       Cycle      0  1  2  3  4  5  6  7
```



☐ Complete the execution of the code below for the implementation above. ☐ Check for dependencies, ☐ don't overlook the first two `mul.s` instructions.

```
# Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12 ...
mul.s f1, f2, f3   IF   ID   M1   M2   M3   M4   M5   M6   WF
mul.s f4, f5, f6        IF   ID   M1   M2   M3   M4   M5   M6   WF


add.s f8, f15, f16


add.s f7, f4, f9


lwc1 f11, 0(r1)


add.s f10, f11, f1

# Cycle            0    1    2    3    4    5    6    7    8    9    10   11   12...
```

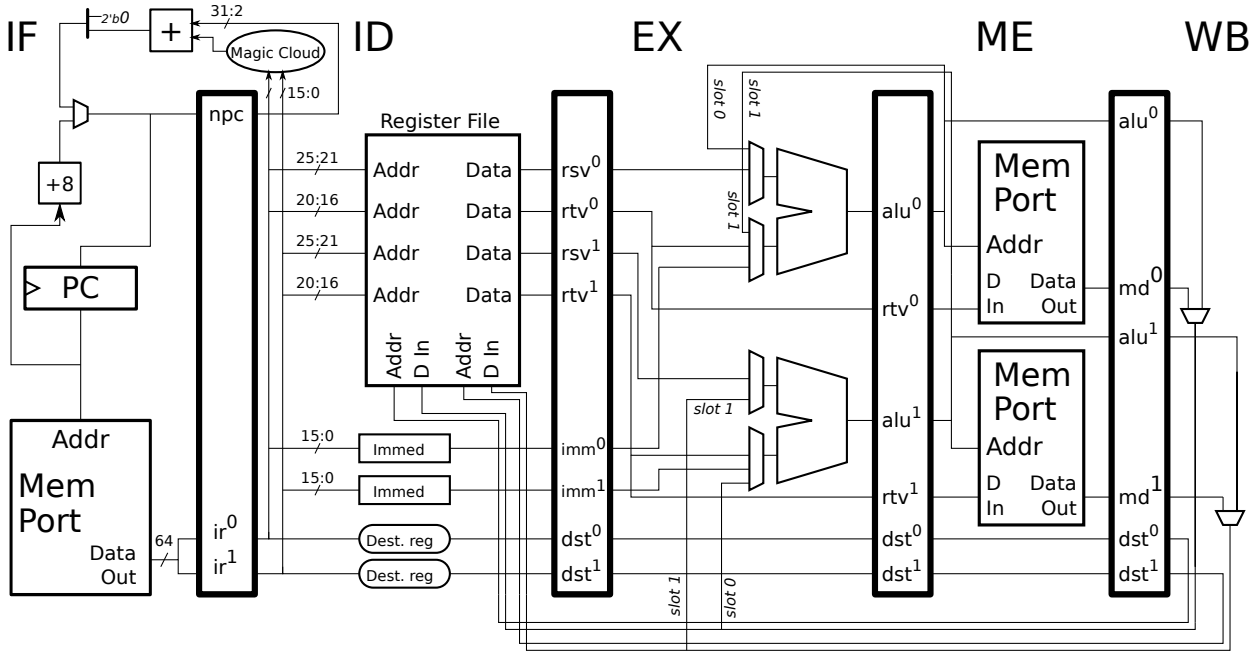

Staple This Side

This page left mostly blank to provide room for the pipeline execution diagram.

Staple This Side

Problem 2: (15 pts) Answer the following questions on superscalar MIPS implementations.

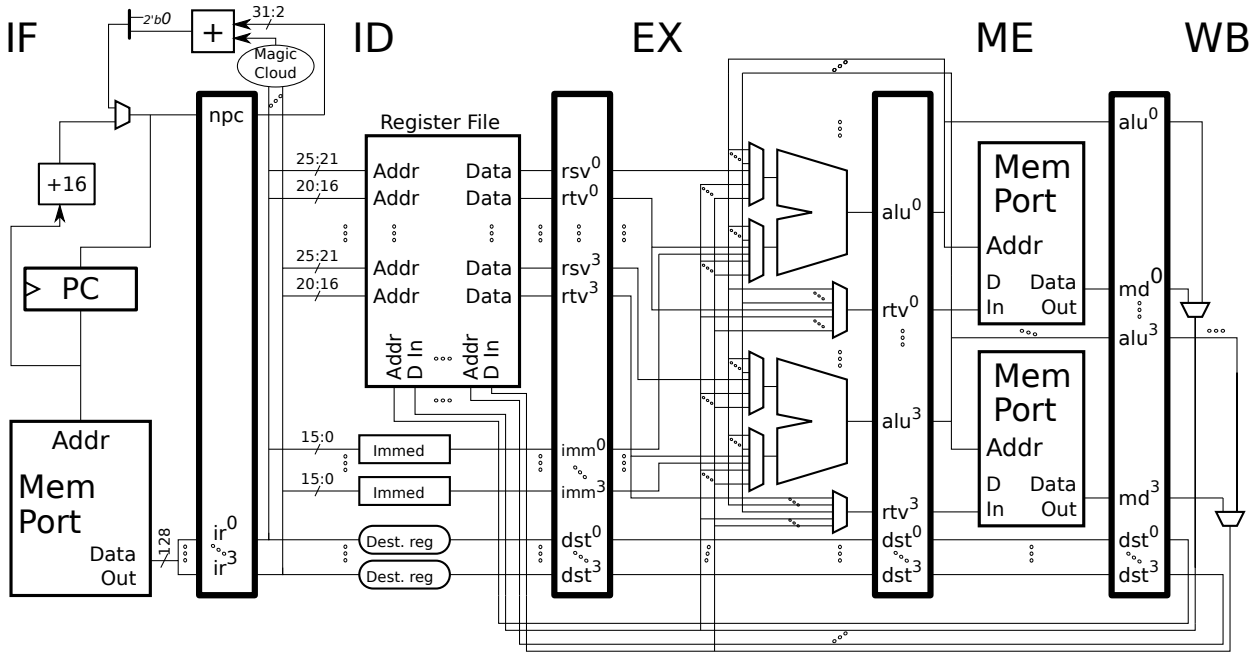
(*a*) The superscalar MIPS implementation below has only four bypass paths, one per ALU multiplexor. The paths have labels, slot 0 and slot 1 (showing where they originate).



☐ Complete the code fragment below (by adding registers) so that it ☐ uses all four bypass paths **in cycle 4**.

```
# Cycle          0  1  2  3  4  5  6
add              IF ID EX ME WB
sub              IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
or                  IF ID EX ME WB
xor                 IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
and                    IF ID EX ME WB
slt                    IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6
```

(*b*) Show the execution of the code below on the following 4-way superscalar MIPS implementation. As we have been doing this semester, instruction fetches are not aligned (the address can be any multiple of 4).



☐ Show execution on this ☐ **4-way** superscalar MIPS implementation, with ☐ the **branch taken**. ☐ Show all squashes with an `x`. ☐ Check for dependencies and ☐ pay attention to branch behavior.

```
 beq r2, r3 SKIP

 addi r1, r1, 8

 add r2, r2, r5

SKIP:
 sw r2, -4(r1)

 addi r1, r1, r5

 lw r2, 0(r1)

 xori r2, r2, 0xaa

 slt r8, r1, r9
```

Problem 3: (25 pts) When the modifications to the MIPS implementation on the facing page are complete an `add.s` instruction will not have to stall to avoid a structural hazard with preceding `mul.s` instructions. Here an `add.s` instruction can pass through the same number of stages as a `mul.s`, so there is no possibility of a stall due to a structural hazard at `WF`. In the execution below `add.s f14` avoids such structural hazard stalls by passing through the two extra stages, `a5` and `a6`. But to avoid the necessity of *always* having to pass through those two extra stages an `add.s` can use *hop* multiplexors to skip ahead to `WF` early. The `add.s f10` skips over two stages, and `add.s f7` skips over one stage.

To keep the problem description from getting too long the following interesting material was not included in the original final exam. Hopping ahead this way is probably not the best way to deal with structural hazards, even if the avoided structural hazard stalls justified the cost of the pipeline latches. The reason is that those two new hop multiplexors could instead be used to implement bypass paths from `a5` *and* `a6` *into the* `M1` *and* `A1` *functional units. (See the Fall 2006 final exam.) With such bypasses possible there would no longer be a need to write back early. Hmmm, this may turn into a Fall 2025 question.*

New and important hardware is shown in blue. The hardware generating signals `xw`, `we`, and `fd` is from the old design and needs to be modified. Hardware for `lwc1` has been removed, it is not part of this problem. When solved correctly code should execute as shown below:

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
mul.s f1, f2, f3   IF ID M1 M2 M3 M4 M5 M6 WF
mul.s f4, f5, f6      IF ID M1 M2 M3 M4 M5 M6 WF
add.s f14, f15, f16      IF ID A1 A2 A3 A4 a5 a6 WF
sub r1, r2, r3              IF ID EX ME WB
add.s f7, f8, f9               IF ID A1 A2 A3 A4 a5 WF
or r4, r5, r6                     IF ID EX ME WB
add.s f10, f11, f12                  IF ID A1 A2 A3 A4 WF
a4/a5.wa                                  1     1     1
a5/a6.wa                                     1     1     0
a6/WF.wa                                        1     0     0
# Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

In cycle 11 the `add.s f7` uses the lower input of Hop6 to hop from `a6` to `WF` (which is why `a6` is not shown). In cycle 12 `add.s f10` uses the upper input of Hop6 to hop from `a5` to `WF`.

☐ Connect logic to the inputs of AND gates W5 and W6 so that the `wa` (write add) signal will be `0` for an instruction that hopped out of a stage, preventing a hopping `add.s` from writing back twice. See sample `wa` values in the execution above.

☐ Add select signals to the two hop multiplexors. ☐ *Little or no logic is required. For partial credit assume* `wa` *is correct.*

☐ Modify the design so that `fd` is correct for ☐ `mul.s` and ☐ the hopping `add.s` instructions.

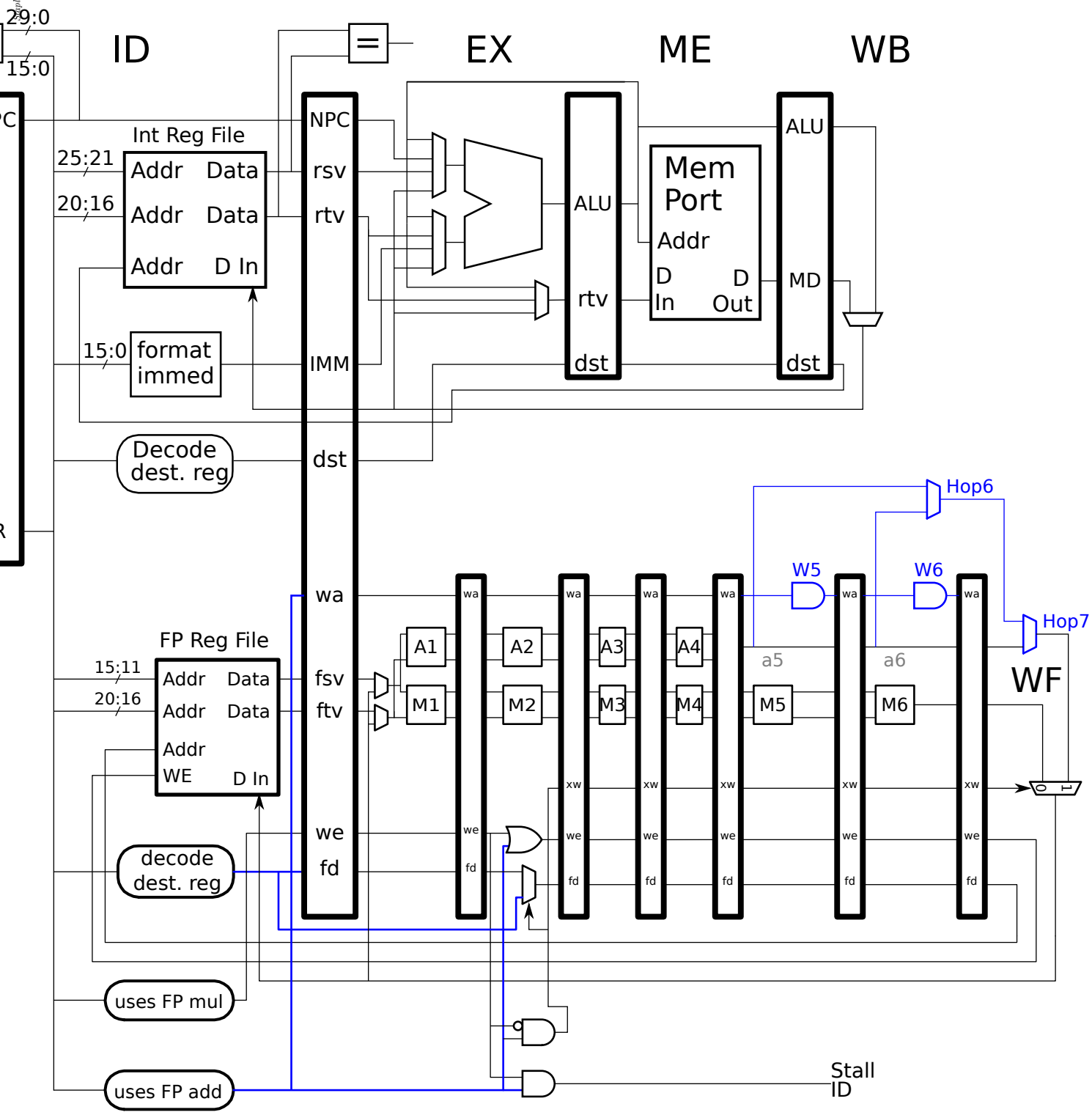
☐ Modify the `xw` logic so that it works for hopping `add.s` instructions and the `mul.s`.

☐ Modify `we` so that it is correct for hopping `add.s` and `mul.s` instructions.

☐ Remove logic that is no longer needed. That can include logic for `xw` or `we`, depending on the solution.

Staple This Side

The IF stage is not shown to make space.



Problem 4: (12 pts) Answer the following branch prediction questions.

(*a*) Code producing the branch patterns shown below is to run on several systems, each with a different branch predictor. One system has a bimodal predictor, and the other uses a 3-outcome local history predictor.

Answer each question below, the answers should be for predictors that have already warmed up. Show work or provide brief explanations.

```
B1:  N  N  T  T  T  N  T    N  N  T  T  T  N  T     N  N  T  T  T  N  T   <- Outcome

B2:   N  T  N  T  N  T  N    T  N  T  N  T  N  T     N  T  N  T  N  T  N  <- Outcome
```

☐ What is the accuracy of the bimodal predictor on branch `B1`? ☐ Be sure to base the accuracy on a repeating pattern.

☐ What is the accuracy of 3-outcome local history predictor on B1 ignoring B2.

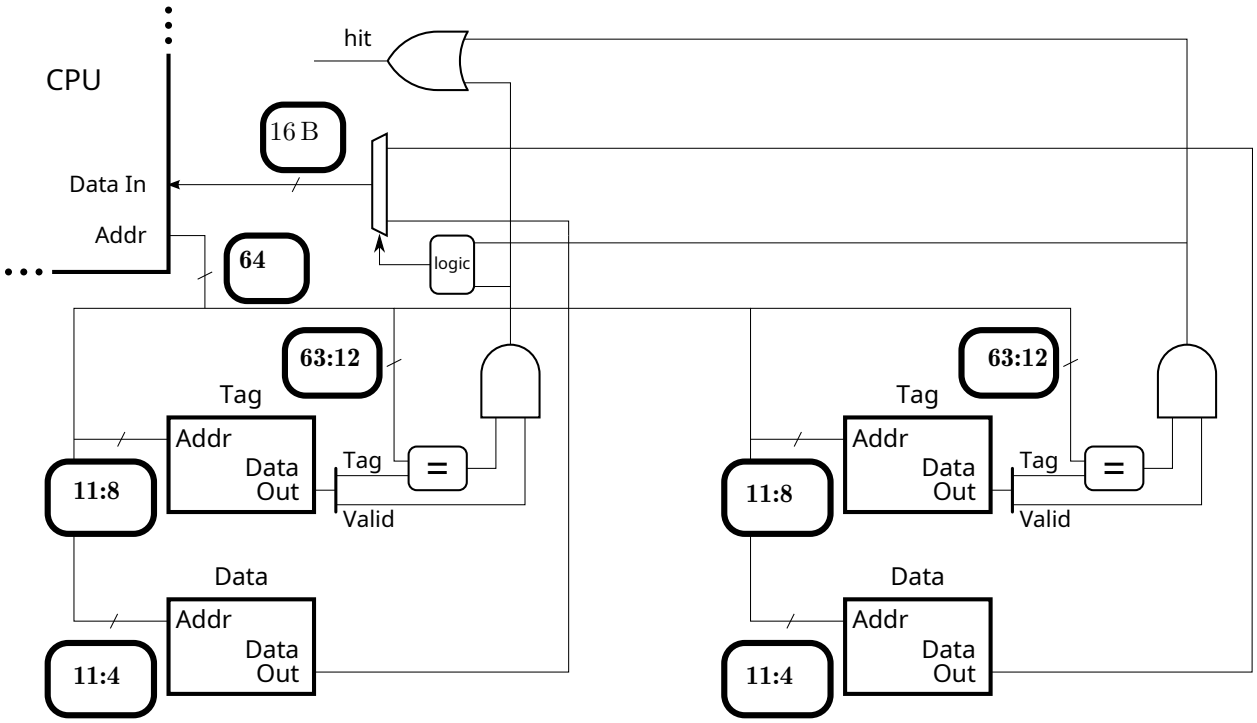
☐ What is the accuracy of 3-outcome local history predictor on B1 **taking into account** B2. ☐ Note that the B2 pattern repeats faster than the B1 pattern.

Staple This Side

Staple This Side

Problem 5: (8 pts) The diagram below is for a two-way set associative cache.
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☐ Fill in the blanks in the diagram.



☐ Complete the address bit categorization below. Label the sections appropriately. (Index, Offset, Tag.)

Address: 

0

☐ Cache Capacity, in Bytes (how much data can it cache).

☐ Line Size ☐ Indicate Unit!!:

Staple This Side

The problem on this page is **not** based on the cache from Part a. The code in the problem belows run on a cache with a line size of 64 bytes (characters). The code fragment starts with the cache cold (empty); consider only accesses to the array. Of course, $2^6 = 64$.

(*b*) Find the hit ratio executing the code below.

```
int64_t sum = 0;
int64_t *a = 0x2000000; // sizeof(int64_t) == 8
int ILIMIT = 1 << 14;    // = 2^14

for ( int i=0; i<ILIMIT; i++ ) sum += a[ i ];
```

☐ What is the hit ratio running the code above? Show formula and briefly justify.

Staple This Side

Staple This Side

Problem 6: (20 pts) Answer each question below.

(*a*) For each item below provide a code fragment exhibiting the indicated dependency. For your solving convenience one instruction is already shown.

☐ Complete the code fragment so that it exhibits a true dependence. ☐ Circle the register carrying the dependence.

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

☐ Complete the code fragment so that it exhibits an output dependence. ☐ Circle the register carrying the dependence.

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

☐ Complete the code fragment so that it exhibits an anti-dependence. ☐ Circle the register carrying the dependence.

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

(*b*) Show the encoding of the instructions below. For the FP instruction infer the encoding from the implementation diagrams in other problems.

☐ Show encoding. ☐ Label fields, and show specific values where possible.

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

☐ Show encoding. ☐ Label fields, and show specific values where possible.

```
sw r4, 5(r6)
```

☐ Show encoding. ☐ Label fields, and show specific values where possible.

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

Staple This Side

(*c*) Based on the execution below, why can't the exception raised by the `mul.s` instruction be precise? What can't the handler do after it returns that could be done if the exception were precise?

```
mul.s f4, f5, f6   IF ID M1 M2 M3 M4 M5*M6*WF
addi r6, r6, 1        IF ID EX ME WB

Handler:
sw r1, 4(fp)                              IF ID ..
```

☐ `mul.s` exception can't be precise because:

☐ If the exception were precise the handler could:

(*d*) Many early RISC ISAs avoided branch instructions that compared registers, such as `blt r1, r2, TARG`, (branch less than) because that would lower the clock frequency. Later ISAs have included them. Assume that the time needed to compute `r1 < r2` has not changed.

☐ Explain why newer ISAs can have instructions like `blt r1, r2, TARG` without slowing the clock frequency, given that comparison is no faster? ☐ In your answer indicate where branches might resolve and how penalty is avoided.

(*e*) A design team is trying to decide between including bypass path A or bypass path B. With the target workload compiled without optimization the implementation with A is faster. With the workload compiled with optimization the implementation with B is faster.

☐ Which should be used ◯ *bypass path A* or ◯ *bypass path B.* ☐ Explain.

Staple This Side

This page intentionally left blank.

Staple This Side