Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Computer Architecture
LSU EE 4720
Final Examination

Monday, 8 May 2023 10:00-12:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

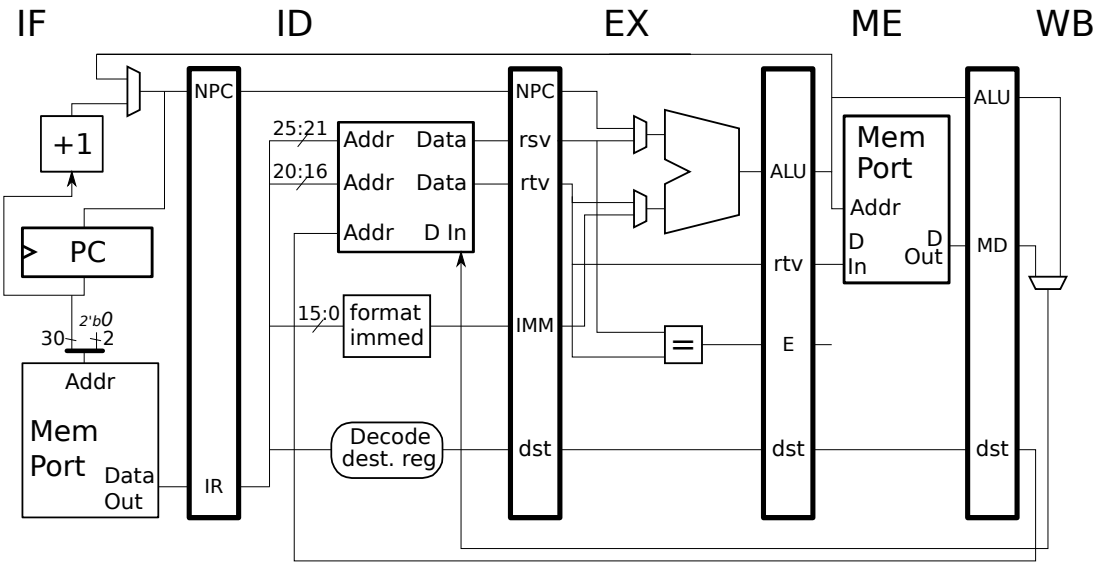
Alias _____

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

Staple This Side

Problem 1: (25 pts) Show the execution of the code fragments on the following implementations. In each case the branch is taken.

(*a*) Show the execution on this basic MIPS implementation.



☐ Show execution for the case where the branch is taken. ☐ Check for dependencies. ☐ Base execution on hardware shown. ☐ Pay close attention to branch behavior.

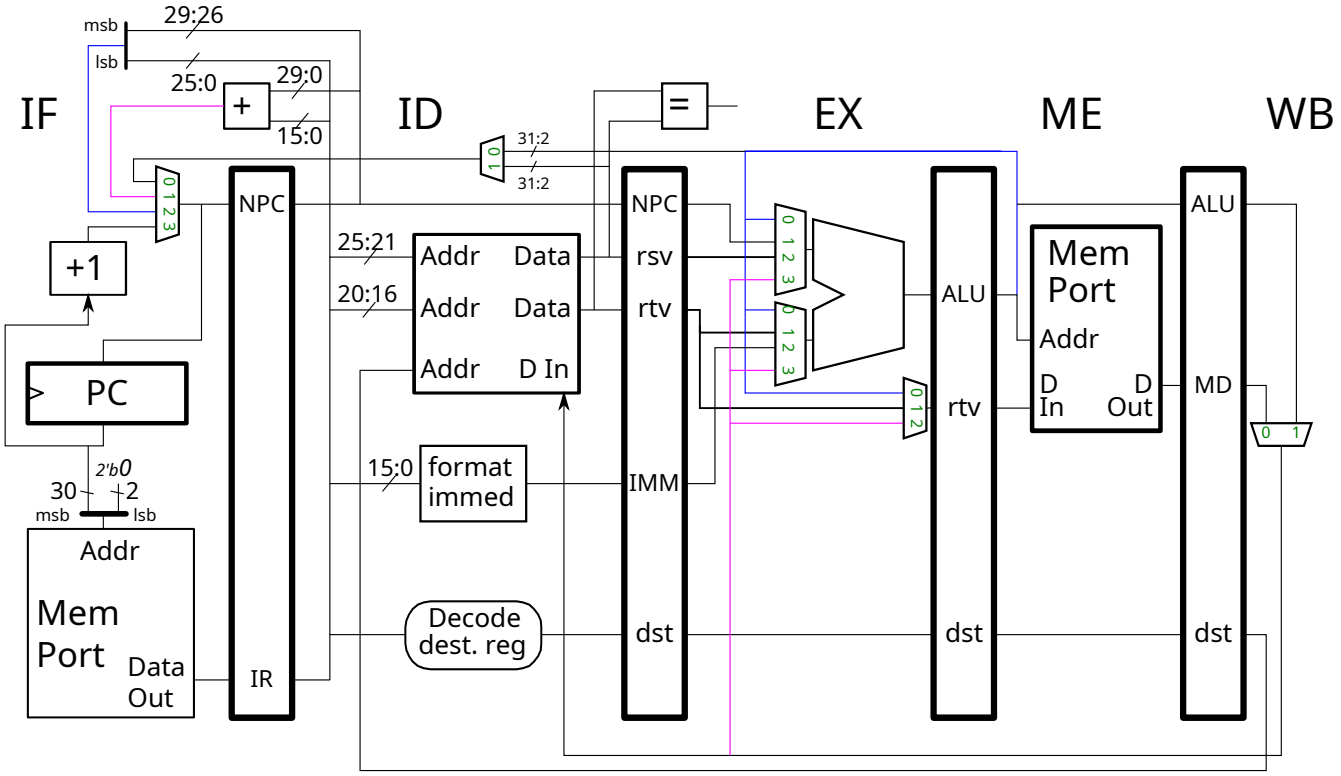
```
 addi r6, r6, 1

 lui r5, 0xf00d

 lw r3, 0x81b4(r5)

 bne r1, r2, TARG

 or r8, r3, r6

 sw r8, 0x8120(r5)

 lw r3, 0x8200(r5)

 addi r5, r5, 16

TARG:
 sll r10, r3, 8

 add r11, r10, r12
```


Staple This Side

This page left blank to provide extra space for the solution.

Staple This Side

(*b*) Appearing below is a MIPS implementation and an **incorrect** execution of a code fragment on that implementation. The code executes more slowly than it would on the implementation. Modify **the implementation** so that the execution is correct. Your modifications will reduce the cost of the implementation.



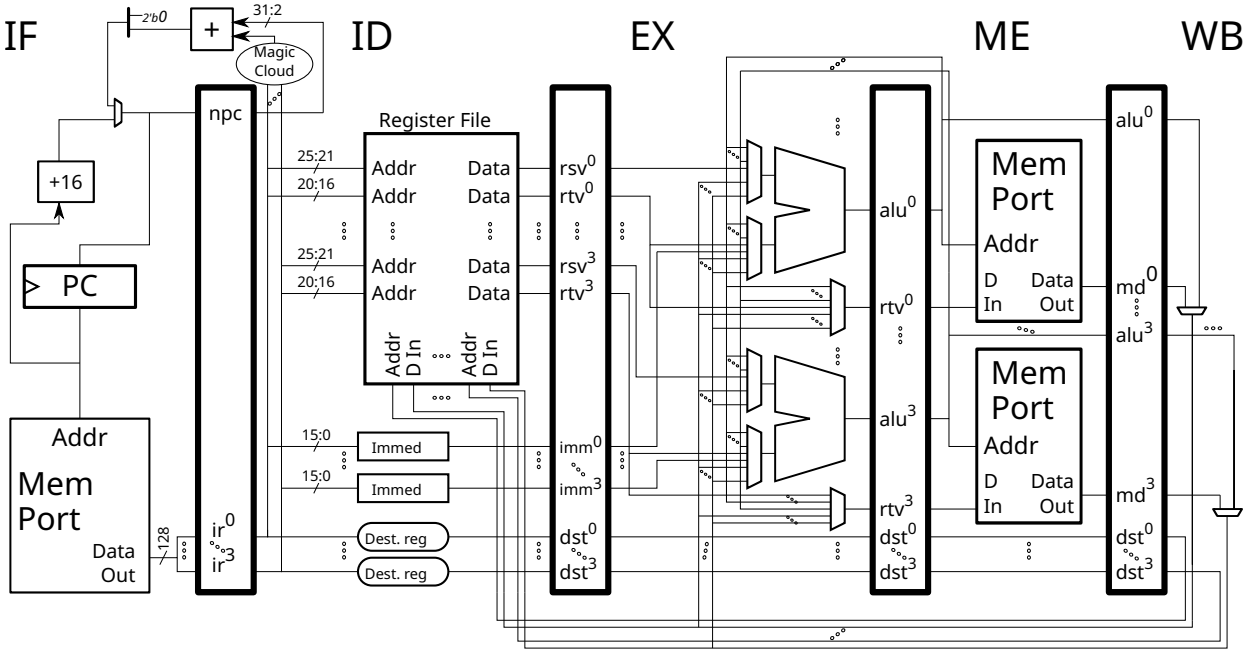
☐ Modify the implementation above so that the code below executes as shown.

☐ **Make as few** changes as possible. For example, remove a bypass path from a mux input, rather than the entire bypass path.

```
# Cycle            0  1  2  3  4  5  6  7  8  9  10
addi r6, r6, 1     IF ID EX ME WB
lui r5, 0xf00d        IF ID EX ME WB
lw r3, 0x81b4(r5)        IF ID -> EX ME WB
bne r1, r2, TARG            IF -> ID EX ME WB
or r8, r6, r3                     IF ID -> EX ME WB
# Cycle            0  1  2  3  4  5  6  7  8  9  10
```


Staple This Side

(*c*) Appearing below is a **4-way** superscalar MIPS implementation.

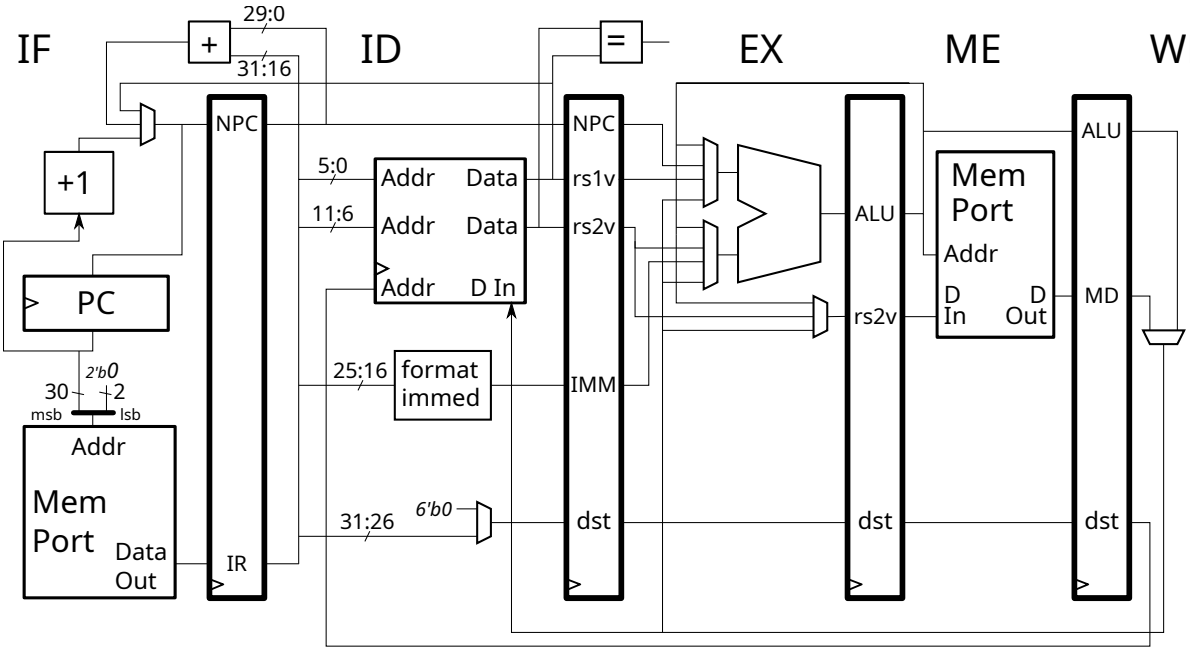


☐ Show execution on the 4-way superscalar implementation ☐ with branch taken.

☐ Pay attention to ☐ branch behavior and ☐ the order of instructions within a stage.

```
 addi r6, r6, 1

 lui r5, 0xf00d

 lw r3, 0x81b4(r5)

 bne r1, r2, TARG

 or r8, r3, r6

 sw r8, 0x8120(r5)
 ...

TARG:
 sll r10, r3, 8

 add r11, r10, r12
```

Problem 2: (25 pts) The implementation of *ANRI*, a new RISC ISA, appears below. Many instructions are similar to those of MIPS, though they differ in format and other features. Like MIPS, ANRI registers are named `r0`, `r1`, ...



(*a*) First, an easy question:

☐ How does ANRI differ from MIPS in the ☐ number of registers and ☐ the immediate size?

(*b*) Like MIPS, ANRI's Format R is used for three-register instructions such as `add r1, r2, r3`. Show a possible ANRI Format R consistent with the hardware.

☐ Show the bit positions and the name of each field in ANRI Format R based on reasonable guesses.

☐ Show possible field values for `add r1, r2, r3` ☐ and for `or r1, r2, r3`. (Two instructions for a reason.)

Staple This Side

Staple This Side

(*c*) Like MIPS, ANRI's Format I is used for immediate instructions such as `addi r4, r5, 6`. Assume that like MIPS, **each of the dozens of ANRI arithmetic and logical instructions has an immediate variant**.

☐ Show the bit position and name of each instruction field in ANRI Format I based on reasonable guesses and ☐ heeding the bold text above. ☐ Show possible field values for `addi r4, r5, 6` ☐ and for `ori r4, r5, 6`.

(*d*) Consider load and store instructions.

☐ Show encoding of `lw r7, 8(r9)` and ☐ `sw r10, 12(r11)` in ANRI Format I, or something similar. Don't forget to ☐ base the encoding on the hardware.

(*e*) Consider procedure call instructions.

☐ Based on the implementation, why does it appear that ANRI would lack the equivalent of MIPS `jal SomeProcedure` though it could still encode the equivalent of `jalr r1, r2`.

(*f*) Based on the hardware above, ANRI would lack a means of loading an arbitrary 32-bit constant into a register using two instructions. Modify the hardware so that ANRI could encode an instruction like MIPS `lui`, one that could be used to load an arbitrary 32-bit constant into a register using two instructions.

☐ Modify hardware to implement an instruction to help loading a 32-bit constant.

☐ Show the format and encoding of this new instruction. ☐ The format must fit in as much as possible with existing formats.

Staple This Side

Problem 3: (20 pts) In the incomplete MIPS implementation to the right the FP multiply unit has its own write port to the FP register file, shown in blue and labeled `WM` in several places. Because of this new `MW` port the `sub.s` instruction in the execution below does not stall, both the `sub.s` and `mul.s` can write back in cycle 8. The control logic has not yet been updated for `MW`.

```
# Cycle           0  1  2  3  4  5  6  7  8  9
mul.s f1, f2, f3  IF ID M1 M2 M3 M4 M5 M6 WM       # Uses MW, the mult-only write port.
add.s f4, f5, f6     IF ID A1 A2 A3 A4 WF
sub.s f7, f8, f9        IF ID A1 A2 A3 A4 WF       # No stall!
lwc1 f10, 0(r11)           IF ID ----> EX ME WF    # Stall due to WF str hazard.
add.s f12, f1, f14            IF ----> ID -> A1... # Stall due to dep with mul.s
```

(*a*) With the illustrated hardware a result cannot be bypassed from a `mul.s` to another instruction. The last `add.s` suffers a stall because of that. Add bypass hardware for such cases.

☐ Add the bypass hardware. ☐ Try to keep cost down by using one mux.

(*b*) Modify the control logic so that it no longer stalls instructions that would write back through `WF` at the same time as a preceding `mul.s`. *Hint: This is just a matter of crossing things out.*

☐ Modify logic to eliminate stalls due to `mul.s` ☐ **but retain stalls** for instructions contending for `WF`, such as `lwc1` in execution above.

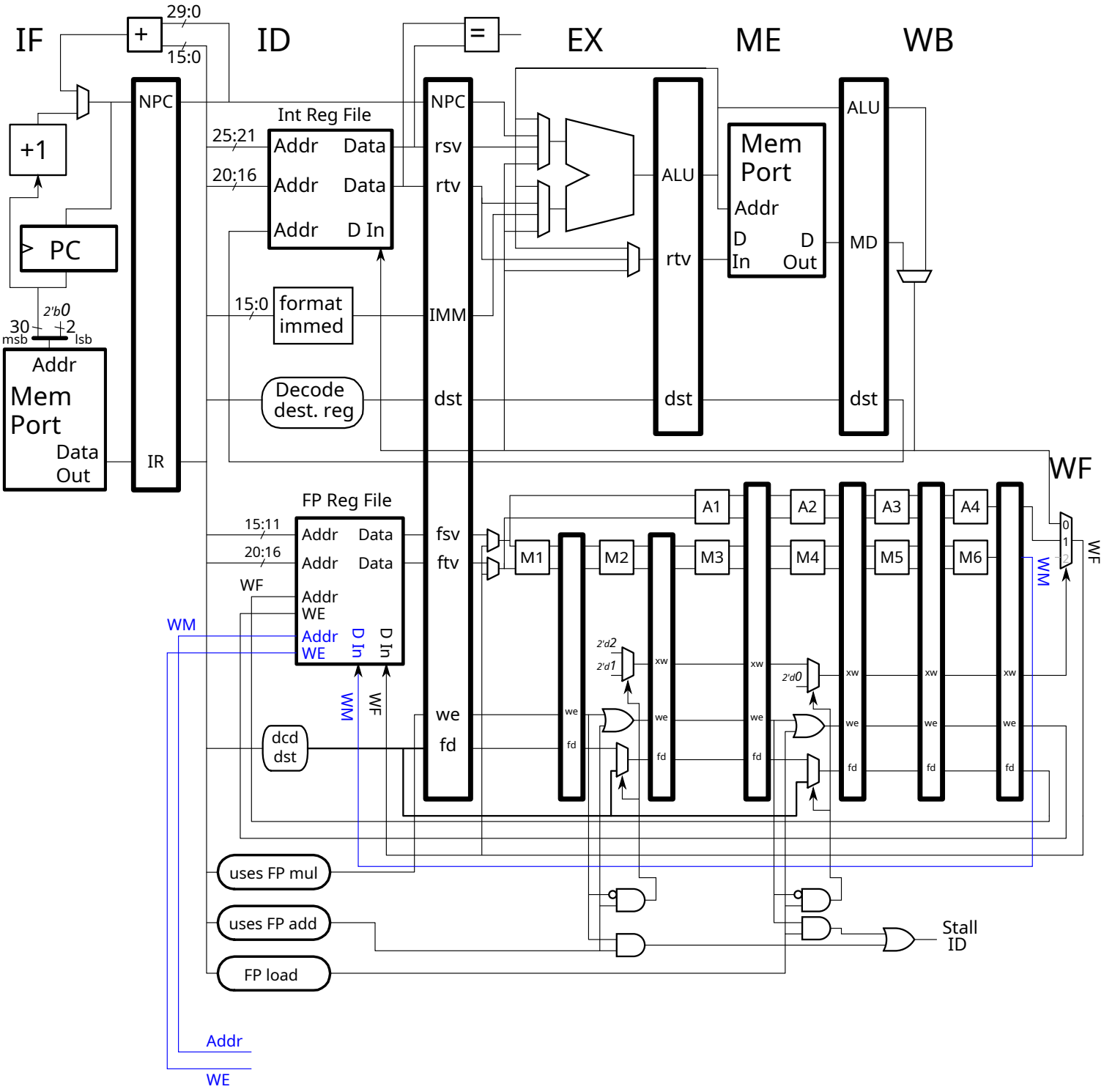
(*c*) Provide the correct `Addr` and `WE` signals to the `WM` and `WF` ports of the FP register file. Note that the `WF` ports are connected, but based on the original version. The `WM` port wires are shown unconnected on the lower-left of the diagram.

☐ Add hardware for the `MW Addr` and `WE` signals, ☐ and make changes to the `WF Addr` and `WE` signals.

☐ Cross out unneeded hardware and ☐ simplify remaining hardware where possible.

Attention perfectionists: Get the SVG source for the hardware at `https://www.ece.lsu.edu/ee4720/2023/fe-ill-fp-2wr.svg` *and edit it yourself!*

Staple This Side



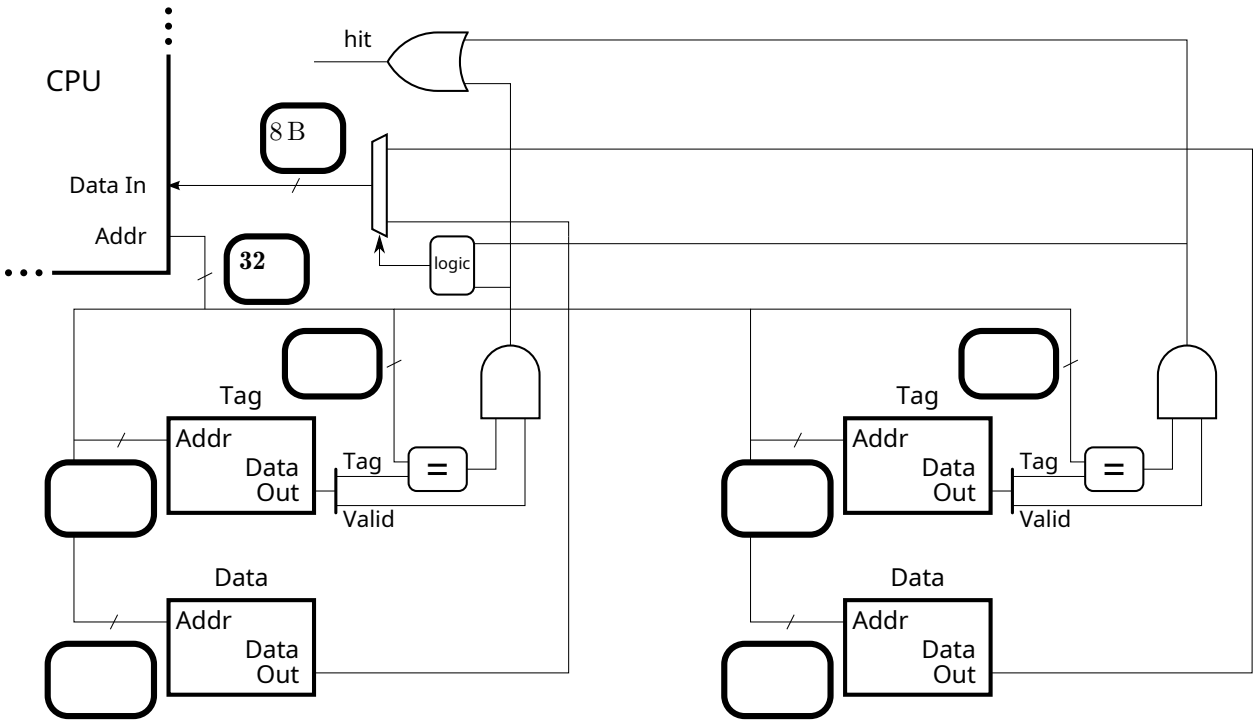
Staple This Side

Staple This Side

Problem 4: (10 pts) The diagram below is for a 4 MiB two-way set-associative cache with a line size of 128 B. The character size is the usual 8 bits. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $128 = 2^7$.

(*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

Staple This Side

The code in the problem belows run on a cache with a line size of 128 B (which is 2^7 B). The code fragment starts with the cache cold (empty); consider only accesses to the array.

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

```
float sum = 0;
bfloat16_t *a = 0x2000000; // sizeof(bfloat16_t) == 2
int ILIMIT = 1 << 11;    // = 2^11

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

Problem 5: (20 pts) Answer each question below.

(*a*) How does the ARM A64 `fcvtzs` instruction differ from MIPS `trunc.w.s` instruction? These were the instructions used in `sum_thing_unusual` from Homework 5.

☐ The difference between `fcvtzs` and `trunc.w.s` is:

(*b*) With the SPEC CPU benchmarks it is the testers responsibility to compile and run the benchmarks.

☐ A brand-new implementation has many more bypass paths than the old implementation. Why might the results of a tester-compiles test (like SPEC CPU) show better performance on the new implementation than on the old implementation, while with a pre-compiled test the old and new implementations would show the same performance?

Staple This Side

Staple This Side

(*c*) Appearing below are some hypothetical CISC instructions.

```
 # Some Hypothetical CISC Instructions
I1: add r1, r2, 0x12345678          # r1 = r2 + 0x12345678
I2: add (r1), r2, 0x1234            # Mem[r1] = r2 + 0x1234
I3: add (r1), 0xff04(r2), 0x1234    # Mem[r1] = Mem[r2+0xff04] + 0x1234
I4: add r1, (r2), 8(r3)             # r1 = Mem[r2] + Mem[r3+8]
I5: add (r1), r2, ((r3))            # Mem[r1] = r2 + Mem[ Mem[r3] ]
```

☐ Which of these instructions could not easily be included in an ISA with 32-bit fixed-length instructions? ☐ Explain.

☐ Which of these instructions would be difficult to implement in a pipelined implementation, even if `IF` and `ID` could easily handle variable-length instructions? ☐ Explain.

(*d*) Consider a 4-way superscalar implementation of a conventional ISA, like MIPS, that has **just one memory port** in the `ME` stage. Also consider *Hy4VI*, a hypothetical 4-slot VLIW ISA in which only slot 1 can contain a load or store instruction.

☐ Why might the memory port and related hardware in the `ME` stage of a Hy4VI implementation cost less than that hardware in the `ME` stage in the 4-way RISC implementation?

☐ Why might code written in the conventional ISA enjoy an advantage over code in Hy4VI when running on respective **future** implementations even when the performance of that code is the same on current implementations? The reason given ☐ must have something to do with load and store instructions.

Staple This Side