Name Solution *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)

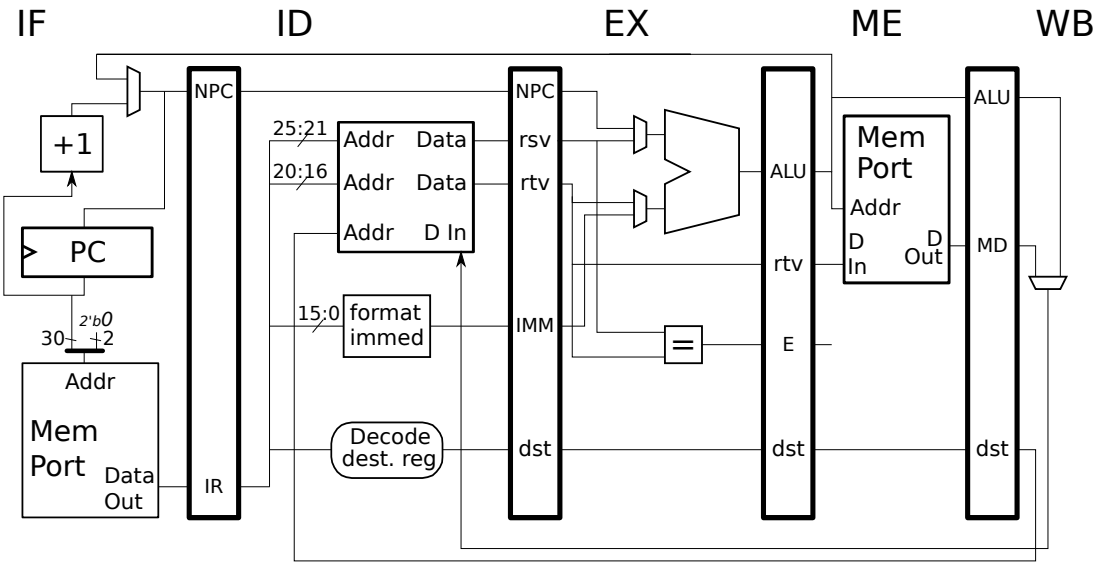
Alias _____

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.

The solution appears below. To help in understanding the solution registers carrying dependencies that result in stalls are shown in uppercase and **bold**, for example as R5 instead of `r5`.

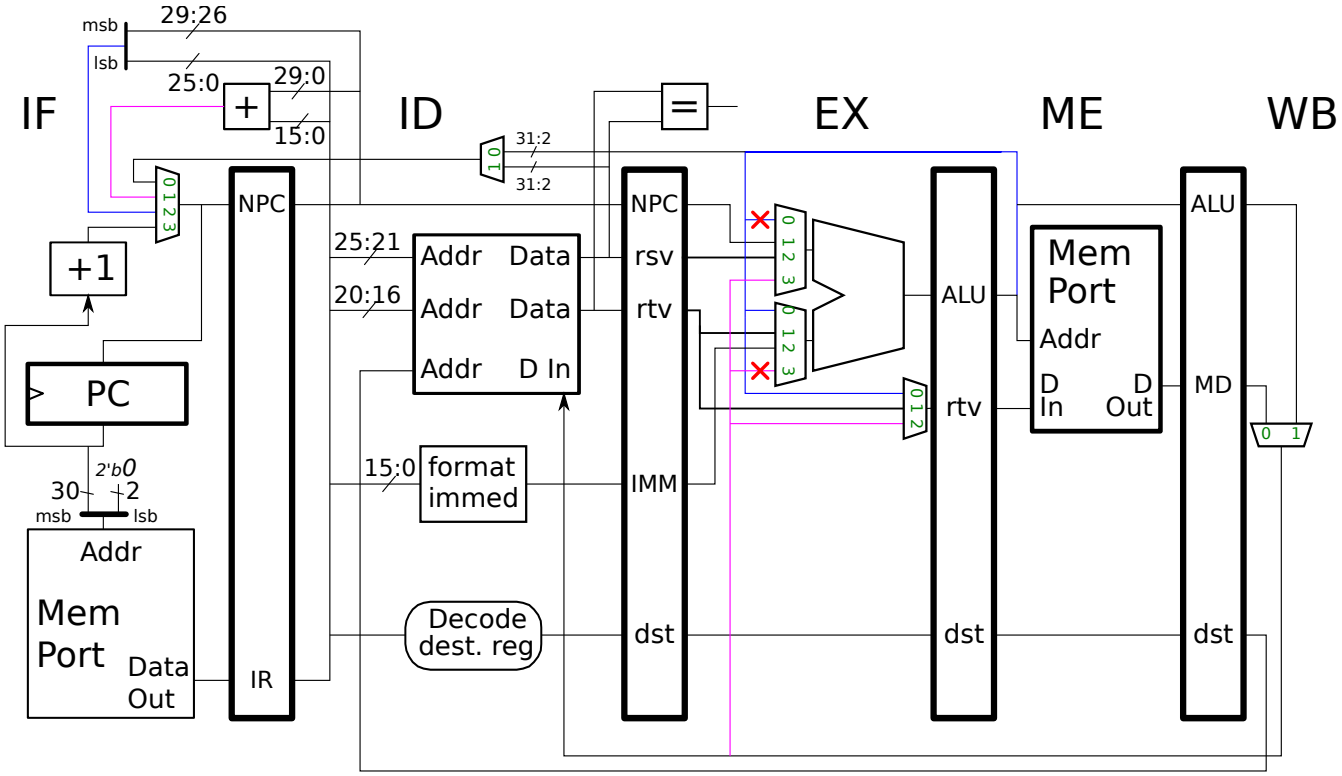
Notice that in this implementation there are no bypass paths, which is why ALU-to-ALU dependencies (such as between `sll` and `add`) are two cycles. If this does not make sense then please please please study more carefully and ask for help if you don't get it!

Also notice that we can tell that the branch resolves in ME because a connection to the multiplexor in the IF stage comes from the ME stage. Because the branch resolves in ME the target will be fetched in the next cycle, when the branch is in WB, which is cycle 9 in the example below. (In most five-stage MIPS implementations used in class the branch resolved in ID.)

```
 # SOLUTION
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 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, R3, r6                        IF ID -> EX ME WB
 sw r8, 0x8120(r5)                       IF ->x
 lw r3, 0x8200(r5)
 addi r5, r5, 16
TARG:
 sll R10, r3, 8                                  IF ID EX ME WB
 add r11, R10, r12                                  IF ID ----> EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

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.



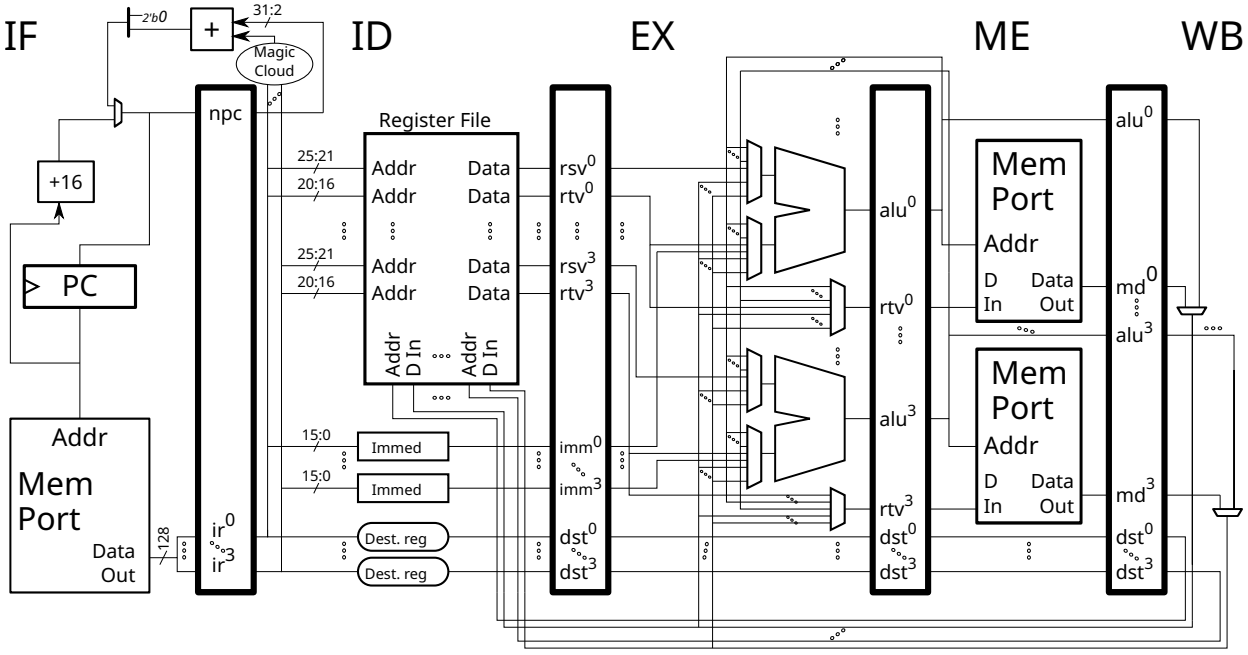
☑ Modify the implementation above so that the code below executes as shown.

☑ **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
```

Solution shown above in red in the EX stage. The crossed out input on the upper mux is for the bypass from ME that would have provided the value of R5 needed by the `lw`. The crossed out lower mux input would have been used by the `or` instruction to bypass the in WB from the `lw` instruction.

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



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

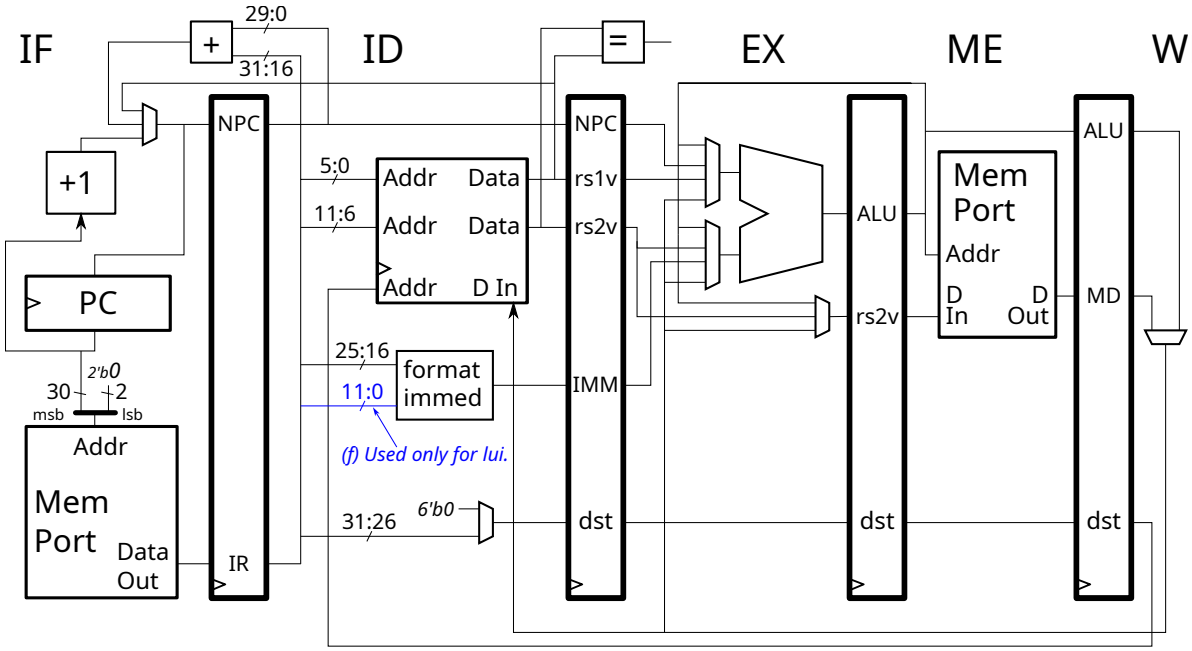
☑ Pay attention to ☑ branch behavior and ☑ the order of instructions within a stage.

Solution appears below. Notice that the branch target, `sll`, is fetched when the branch is in `EX`.

Grading Note: A common mistake (in 2024 when this appeared in Homework 5) was to mistake `lui` for some kind of load. Remember that `lui` is the load upper immediate instruction, it puts the immediate into the 16 high bits of the destination register and zeros the low 16 bits.

```
 # SOLUTION
 # Cycle            0  1  2  3  4  5  6  7  8  9
 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, r3, r6         IF -> ID -> EX ME WB
 sw r8, 0x8120(r5)     IF ->x
 lw r3, 0x8200(r5)     IF ->x
 addi r5, r5, 16       IF ->x
 # Cycle            0  1  2  3  4  5  6  7  8  9
TARG:
 sll r10, r3, 8           IF -> ID EX ME WB
 add r11, r10, r12        IF -> ID -> EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9
```

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?

Full-Credit Answer: ANRI has 64 registers, while MIPS has 32, and the ANRI immediate size is 10 bits while MIPS' is 16 bits.

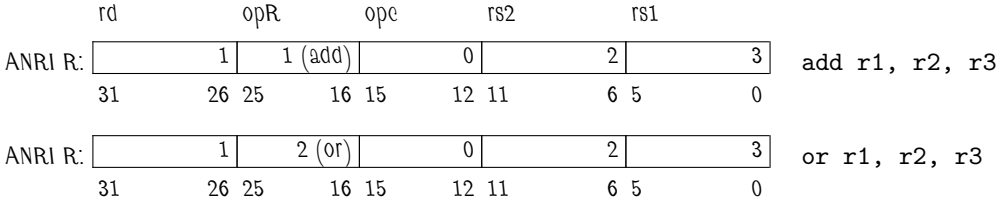
Explanation: The number of registers (integer [general-purpose] registers to be exact) is determined by looking at the number of bits used as an address in the ports of the register file (in the `ID` stage). The width of each of the `Addr` inputs is 6 bits, and so the number of registers is $2^6 = 64$. The size of the immediate can be determined by looking at the number of bits at the input to the original (black in the diagram) `format immed` unit in `ID`, that's 10 bits. (The input labeled (f) is part of the solution to part (f).)

(*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.)

Solution:



Staple This Side

Explanation: The position of the `rs1` and `rs2` fields can be determined by looking at the read-port `Addr` inputs of the register file (in the `ID` stage). Similarly, the `rd` field can be determined by looking at the bits, `31:26`, that travel through the pipeline `dst` latches on their way to the register file write `Addr` port.

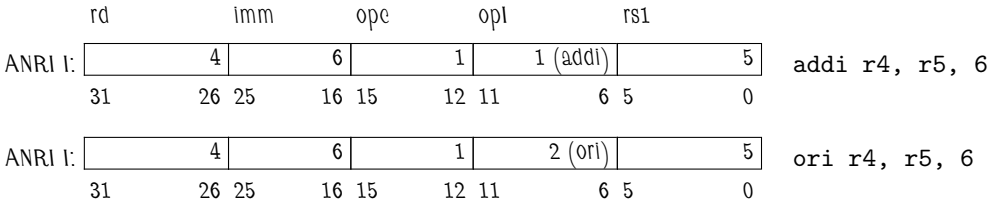
The position of the opcode field needs to be inferred. The opcode field must appear in the same place in every instruction. Bits `25:16` are used for the immediate, and so they can't be used for the opcode field. The only remaining bits are `15:12`, and so those must be the location of the opcode field. A 4-bit opcode field, with only 16 distinct values, is not enough to encode every instruction. So like MIPS an opcode extension field will be needed, that's called `opR` (opcode extension for format R) here. Also like MIPS, the opcode field is set to zero for type-R integer instructions. Values of 1 and 2 in the `opR` field are used for `add` and `or` instructions.

Note: MIPS uses field names `opcode, rs, rt, rd, sa, func, immed, ii`. There is no reason to use exactly the same names in ANRI. Most ISAs use `rd` for a destination and `opcode` for the opcode, so that's retained in this solution. But MIPS' ambiguously named `rt` does not apply because `rd` is always a destination and the two sources are always sources (if used at all). For that reason, the sources were named `rs1` and `rs2` (which is how many other ISAs name sources). Many ISAs use an immediate field to specify a constant shift amount, MIPS is an exception using a dedicated `sa` field. Given that no shift unit was shown there was no reason to expect a dedicated shift amount field.

(*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`.

Solution:

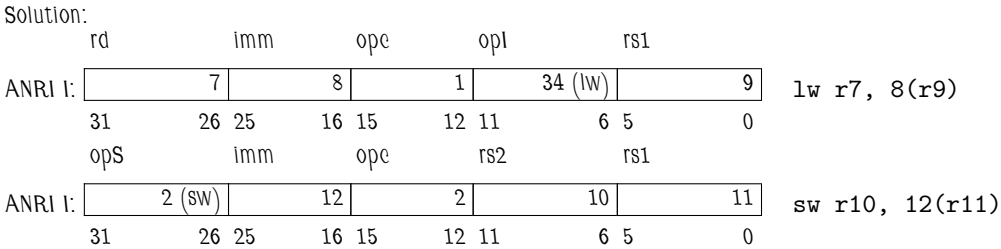


Explanation: The type I instruction uses an `imm` field, whose position is determined by the input to the `format immed` unit in `ID`. Because there are dozens of type I instructions (see **the bold text**) the 4-bit opcode field will not be sufficient to code them all. For that reason bits `11:6` are used as an opcode extension field, called `opI`, occupying the same position as `rs2`, which is not used in type-I instructions.

(*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.

Solution:



Explanation: The `lw` is encoded the same way as the other type I ANRI instructions. However, that approach won't work for `sw` because the `rs2` field is needed for the register holding the value to be written to memory (`r10` in the example). Therefore for store instructions bits `31:26` are used for an opcode extension field, called `opS` here.

(*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`.

Full-Credit Answer: Because the only inputs to the `dst` mux are zero and `rd`, so there is no way to encode an implicit destination register such as `r31`.

Explanation: The MIPS `jal` instruction writes the return address in `r31`. The instruction has only an opcode and an immediate field, the 31 is not encoded. The ID-stage mux selecting the writeback register (`dst`) has only two inputs, a zero or the `rd` field. Therefore every instruction in ANRI that writes a register must use the `rd` field to specify the destination register, explicitly. That makes it impossible to have an implicit register such as `r31` in MIPS. (Note that `r0` must be the zero register because the register file lacks a write-enable input.)

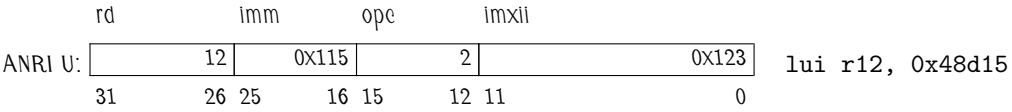
Grading Note: Many solutions to 2024 Homework 2, based on this problem, incorrectly answered that there was no way for an ANRI `jal` to write a return address into a register. The return address in MIPS is `PC + 8` (the same as `NPC+4`). If that were the case for ANRI, then a return address could easily be saved by having the ALU add 4 (or maybe 1) to the `NPC` value available in the upper input.

(*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.

The encoding appears below and the hardware modifications appear in the diagram in blue. The format is called type U, following RISC-V terminology. Since a `lui` instruction does not need source registers the `rs1` and `rs2` fields are used for an immediate extension field, called `imxii` here. With the 10-bit `imm` field and the 12-bit `imxii` field the ANRI `lui` can accommodate 22-bit immediates. That is sufficient to load an arbitrary 32-bit contents using the instruction pair `lui r1, hi(0x12345678); ori r1, ri, lo(0x12345678);`, where assembler macros `hi` and `lo` extract the high 22 bits and low 10 bits of their arguments. Because ordinary type-I instructions use `imm` for their entire immediate it would be more efficient hardware-wise to use `imm` for the least-significant 10 bits of the immediate in `lui` instructions and `imxii` for the remaining 12 bits. In this example `hi(0x12345678)` evaluates to `0x048d15` or $00\,0100\,1000\,1101\,0001\,0101_2$. So the `imm` field would be set to the lower ten bits, $01\,0001\,0101_2 = 115_{16}$. The `imxii` would be set to the high 12 bits, $0001\,0010\,0011_2 = 123_{16}$.



`lui r12, 0x48d15`

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.

Solution appears in green. Notice that the bypass can either be from the multiply unit or the value headed to WF, but not both. Because of this stalls are still possible, which might be the topic of a followup question in 2024.

(*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.

Gates shown crossed out with red exes. The exed-out gates checked for an instruction in M2 to avoid a structural hazard on WF. The only instruction that can be in M2 is a multiply, and that now uses its own write port, so there is no need to check.

(*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.

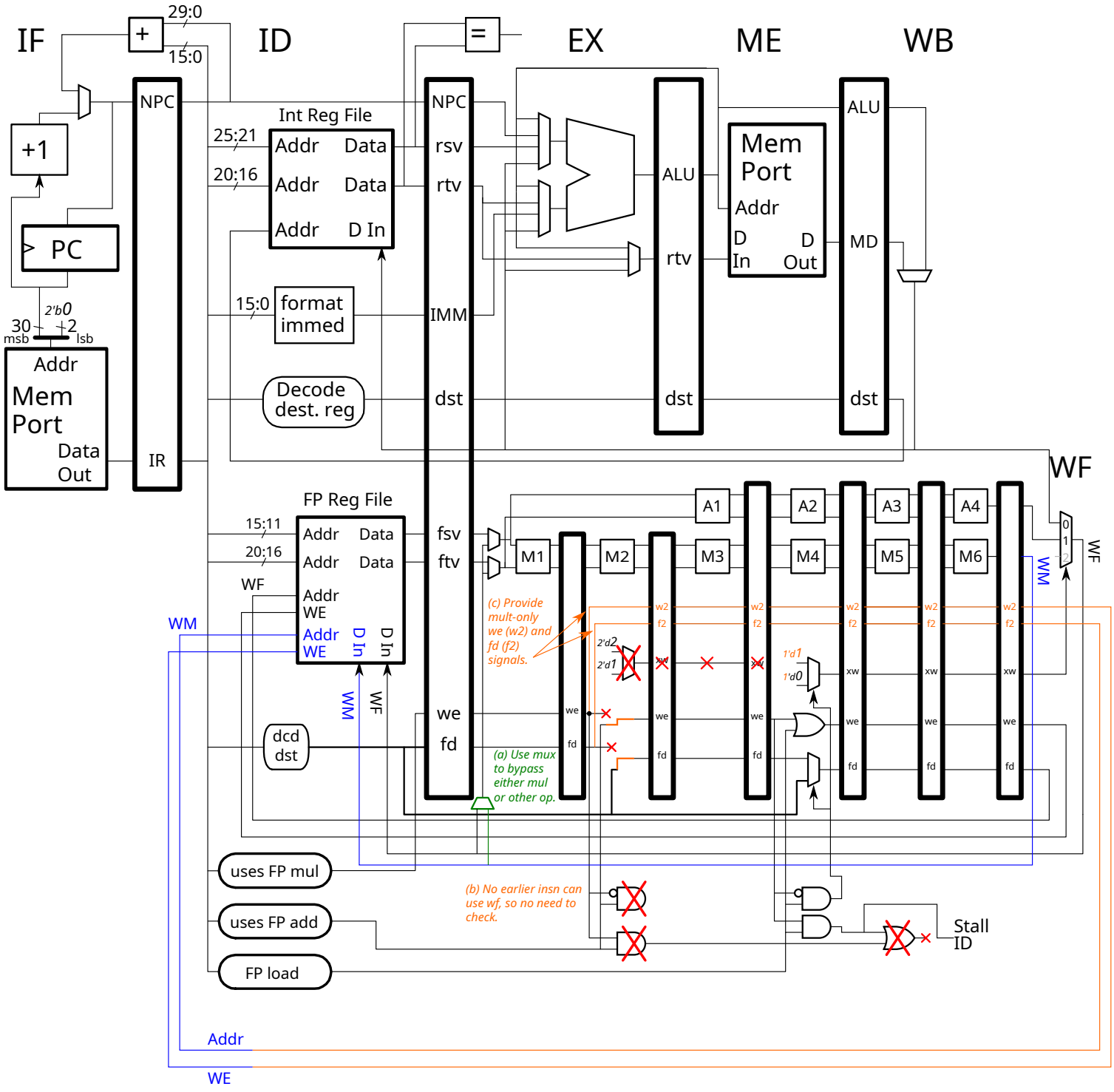
☑ Add hardware for the `MW Addr` and `WE` signals, ☑ and make changes to the `WF Addr` and `WE` signals.

Solution appears in orange. Because the multiply is the only instruction that uses the WM port, multiply-only pipeline latches are provided for the write-enable and destination register signals. The multiply's write-enable signals are labeled `w2` and the multiply's destination register signals are labeled `f2`.

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

Because a multiply has its own write-enable and destination register pipeline latches, there is no need for the `fd` multiplexor, the `xw` multiplexor, and OR gate in the M2 stage. Also notice that the `xw` signal can be made 1 bit since the WF mux now has two inputs.

Staple This Side

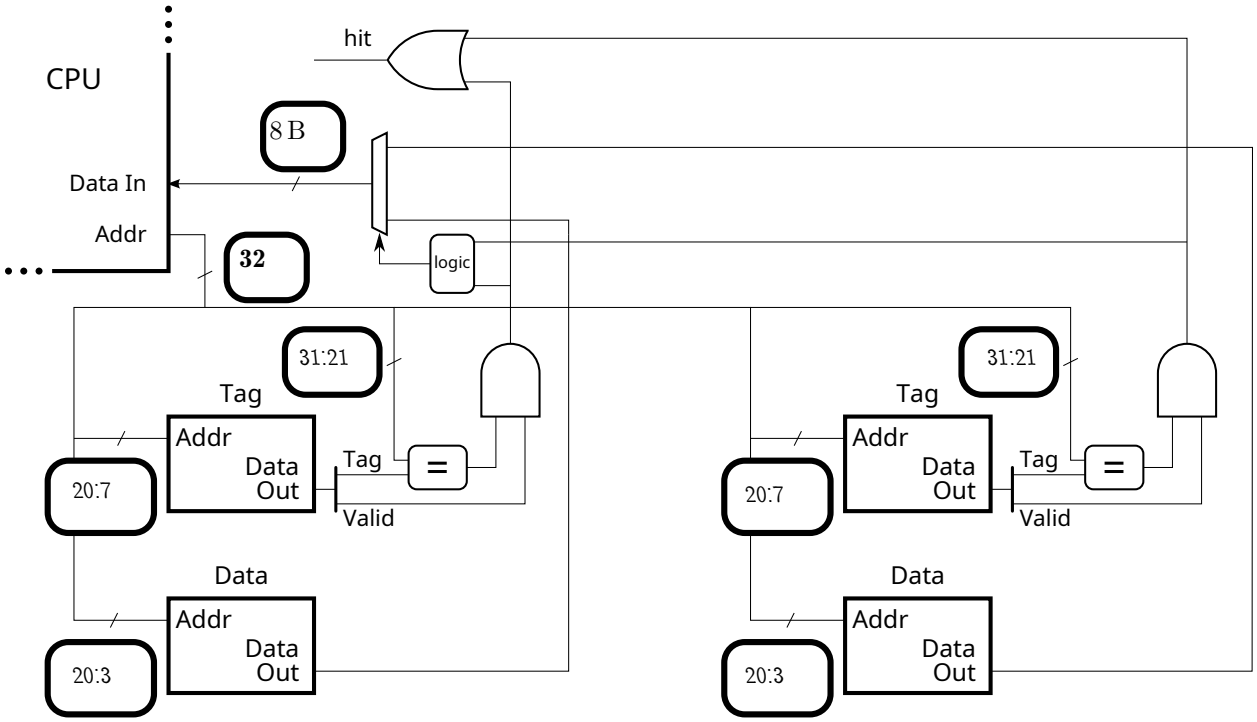


Problem 4: (10 pts) The diagram below is for a 4 MiB two-way set-associative cache with a line size of 128 B. The character size is the usual 8 bits. Helpful facts: $4\,\text{MiB} = 2^{22}\,\text{B}$, $128 = 2^7$.

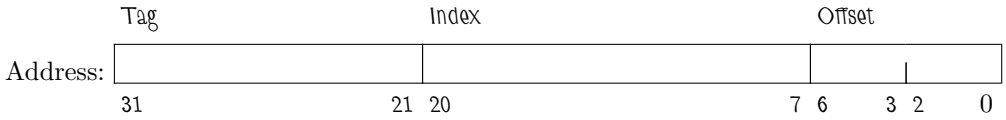
(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants.

☑ Fill in the blanks in the diagram.

Solution appears below.



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



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.

The line size of $2^7 = 128$ bytes is given. The size of an array element, which is of type `bfloat16_t` (a 16-bit floating point type called *brain float*), is $2 = 2^1$ B, and so there are $2^7/2^1 = 2^{7-1} = 2^6 = 64$ elements per line. The first access, at `i=0`, will miss but bring in a line with 2^6 elements, and so the next $2^6 - 1 = 63$ accesses will be to data on the line, hits. The access at `i=64` will miss and the process will repeat. Therefore the hit ratio is $\frac{63}{64}$.

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:

Full-Credit Short Answer: Both read a FP register and convert the truncated FP value to an integer, the result is written to an integer register by `fcvtzs` and to a FP register by `trunc.w.s`.

Long Answer: Both instructions read a floating-point value from a floating-point register, truncate the value, and convert it from a floating-point representation to an integer representation. (Truncating a floating-point value means discarding the fractional part. For example, truncating 2.1 yields 2 and truncating -3.9 yields -3. The representation before *and after* truncation is floating point, which is way conversion to an integer representation is specifically mentioned above as something each instruction does. So that both 2.1 and 2.9 are converted to 2. Or, put another way, both `0x40066666` and `0x4039999a` are converted to `0x2`.) The ARM A64 `fcvtzs` writes the result to an integer register. In contrast, the MIPS `trunc.w.s` writes its result to a floating-point register (even though the result is in an integer representation). If the converted value is to be used by an integer instruction the MIPS `trunc.w.s` needs to be followed by a `mfc1` (move from co-processor 1) instruction.

(*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?

Suppose that the only change in the new implementation are the new bypass paths. Further, suppose the compiler used in the pre-compiled test was targeting the old implementation. (It could not target the brand new implementation because the compiler would not have been available or if it were, it would make no sense to have compiled the code for an implementation that was not yet available.) Consider an instruction pair that could make use of the new bypass path. To avoid a stall, the compiler would separate those two instructions. Suppose the compiler separates every pair of instructions that would otherwise use the new bypass paths. Then the code would run at the same rate on both processors. The bypass paths would go unused.

If the code were compiled for the new implementation there would be no reason to separate instruction pairs that could use the bypass paths. Such code would suffer stalls on the old implementation that would not occur on the new implementation.

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.

Instructions I1 and I3, because the total size of the immediate(s) in each instruction would be 32 bits, leaving no room for anything else.

☑ 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.

Short, Full-Credit Answer: Instructions I3, I4, and I5 because they each require more than one memory access. A pipelined implementation would need multiple memory ports, which would be costly.

Explanation: A typical scalar RISC pipeline has one memory port for data access, using our notation, in the ME stage. To implement these three instructions in a pipelined fashion there would need to be multiple memory ports, inflating the cost by a substantial amount. For example, to implement I4 two memory ports would be needed to read the sources. Those could be in the same stage, but for I5 they would need to be in separate stages, one to read `Mem[r3]` and one to read `Mem[ Mem[r3] ]`, and finally one to write the result to memory. It is possible to implement I3-I5 with one data memory port by using the port multiple times per instruction. But such an implementation would not be considered pipelined by most people.

What about I2? That could be implemented by our 5-stage pipeline. RISC ISAs lack such an instruction because they would be more difficult to compile code for. Difficult because sources would still have to come exclusively from registers, so dependent instructions would need to have a companion load to get the value that was written to memory.

(*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?

In the superscalar implementation the memory port would need to connect to each of the four slots in the ME stage. The multiplexors to make that connection aren't free. In Hy4VI the memory port only connects to slot 1, avoiding the multiplexors.

☑ 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.

Because sometimes code has lots of loads and stores. In a conventional ISA there's no problem with, say, eight consecutive load instructions. But in Hy4VI each load instruction would have to be accompanied by three non-load instructions. If no such useful instructions could be found those non-load instructions would be no-ops. In the current one-memory port superscalar implementation there would be stalls to limit one load at a time entering ME, and so it would have no advantage over Hy4VI. But a future superscalar implementation might have two memory ports (or even four) in ME, and so there would be fewer or no stalls and thus a performance advantage over Hy4VI.