CSM 61C Spring 2020

Single Cycle Datapath

Exam Question Compilation

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

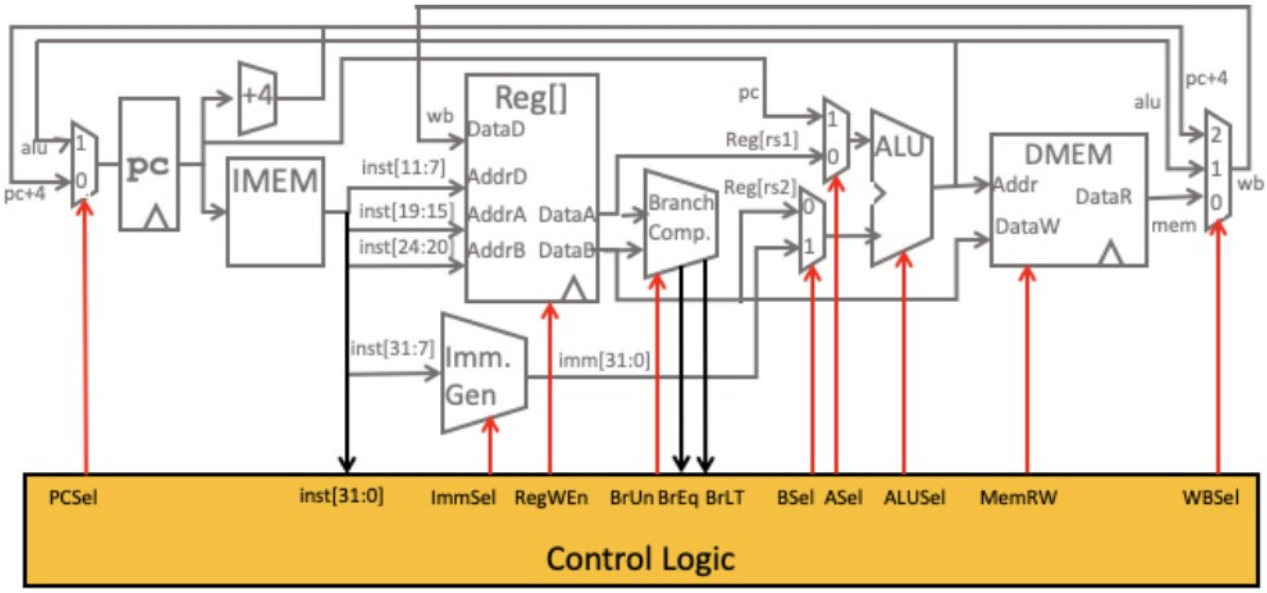
- Fall 2019 Midterm Q6
- Summer 2018 Midterm 2 Q3
- Summer 2019 Midterm 2 Q4
- Fall 2019 Final Q6
- Fall 2017 Final Q6
- Spring 2018 Final Q8
- Spring 2018 Midterm 2 Q3
- Fall 2018 Final F3A+F3B

Q6) *comp a0, RISC-V, <3* (18 pts = 5*1 + 7*1 + 4 + 2)

You add a new R-Type *signed* compare instruction called **comp**, into the RISC-V single-cycle datapath, to compare `R[rs1]` and `R[rs2]` and set `R[rd]` appropriately. The RTL for it is shown on the right.

```
comp rd, rs1, rs2

if   R[rs1] >  R[rs2]: R[rd] = 1
elif R[rs1] == R[rs2]: R[rd] = 0
else: do nothing
```



a) You want to change the datapath to make this work. You start by adding two more inputs (`0x00000000` and `0x00000001`) to the rightmost `WBSel` MUX. What else is **required** to make this instruction work?

◯ **True** ◯ **False** Modify `Branch Comp`
◯ **True** ◯ **False** Modify `Imm. Gen.`
◯ **True** ◯ **False** Modify the `ALU` and `ALUSel` control signals
◯ **True** ◯ **False** Modify the control logic for `RegWEn`
◯ **True** ◯ **False** Modify the control logic for `MemWEn`

b) You realize you can also implement this with **NO** changes to the datapath! **From this point until the end of the page**, let's assume that's what we're going to do. Fill in the control signals for it. We did the first one, `COMP`, which is a new boolean variable within the control logic that is only set to 1 when we have a **comp** instruction.

	COMP	**PCSel**	**BrUn**	**BSel**	**ASel**	**ALUSel**	**MemRW**	**WBSel**
`comp x1, x2, x3`	● 1 ◯ 0	◯ ALU ◯ PC+4	◯ 1 ◯ 0	◯ 1 ◯ 0	◯ 1 ◯ 0	◯ ADD ◯ SUB ◯ OTHER	◯ Read ◯ Write	◯ PC+4 ◯ ALU ◯ MEM

c) The control signal `RegWEn` can be represented by the Boolean expression "`add+addi+sub+...`" (where `add` is only 1 for `add` instructions, `addi` is only 1 for `addi` instructions, etc.). What new Boolean expression should we add (i.e., Boolean logic "or") to the original `RegWEn` expression to handle the **comp** instruction? Select ONE.

◯`COMP` ◯`COMP*BrLT` ◯`COMP*BrEq` ◯`COMP*!BrLT` ◯`COMP*!BrEq`
◯`COMP*(!BrLT+!BrEq)` ◯`COMP*(BrLT+!BrEq)` ◯`COMP*(!BrLT+BrEq)` ◯`COMP*(BrLT+BrEq)`

d) Select all of the stages of the datapath this instruction will use. Select all that apply.

▫Instruction fetch (IF) ▫Instruction decode (ID) ▫Execute (EX) ▫Memory (MEM) ▫Writeback (WB)

SID: _____

Question 3: Maddi-o and Lui-p (20 pts)

As discussed in lecture, the standard RISC-V approach for loading in a 32-bit immediate into a register is by using the following **`lui-addi`** pair of instructions:

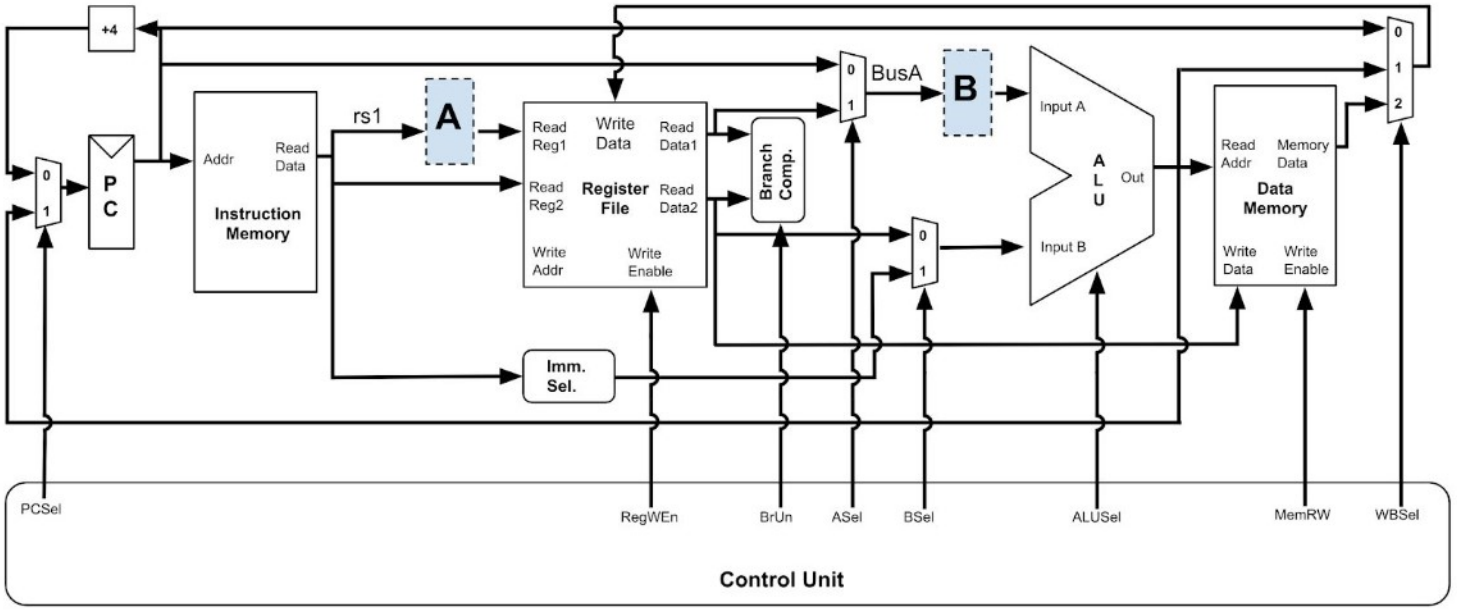
```
lui   rd HIGH_IMM
addi  rd rd LOW_IMM
```

We will explore an alternative method of storing a 32-bit immediate into a register by implementing a new **U-type** instruction named **`luip`,** which will load a 20-bit immediate into the upper 20 bits of the `rd` register and **preserve** the lower 12 bits of the `rd` register. This way, we can use an `addi-luip` pair to load a 32-bit immediate into a register, as shown on the left. On the right is a description of what **`luip`** does:

```
addi  rd x0 LOW_IMM
luip  rd HIGH_IMM
```

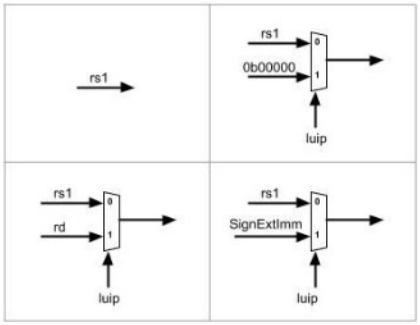
```
R[rd] = R[rd] [11:0] + {imm, 12b'0}
```

Below is the existing RISC-V datapath as presented in lecture, except there are two boxes labeled “A” and “B” which are areas in the datapath that you will fill in with the necessary hardware to implement **`luip`**.

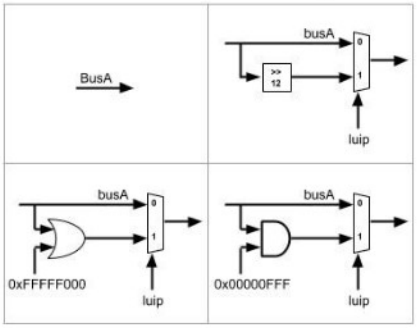


For Parts A and B, please **clearly circle** the correct hardware that would fill their respective dashed boxes above. For Part C, **circle** all the ALUSel operations that could be used by the **`luip`** instruction in the ALU unit.

A.



B.



C.

AND	OR
ADD	SUB
XOR	BSEL

SID: _____

D. Fill out the following control signals that must be generated by the control unit in order to correctly implement **`luip`**. Fill in the bubble for "X" if the exact value of the control signal does not matter (e.g. a value of ⓪ or ① is incorrect when the signal could instead be Ⓧ).

⓪ Signal = 0 ① Signal = 1 ② Signal = 2
Ⓡ Write Disabled Ⓦ Write Enabled Ⓧ Don't Care

`luip`	PCSel	RegWEn	BrUn	ASel	BSel	MemRW	WBSel
1	⓪ ① Ⓧ	Ⓡ Ⓦ Ⓧ	⓪ ① Ⓧ	⓪ ① Ⓧ	⓪ ① Ⓧ	Ⓡ Ⓦ Ⓧ	⓪ ① ② Ⓧ

E. Now that we've added **`luip`** to our instruction set architecture, we need to give it an opcode. We now have 3 U-type instructions: **`auipc`**, **`lui`**, and **`luip`**. The opcode for `auipc` is 0b001 0111 and the opcode for `lui` is 0b011 0111. For this question, we refer to the most significant (leftmost) opcode bit as bit 6 and the least significant (rightmost) opcode bit as bit 0.

You are given the following truth table whose two inputs are two of the opcode bits, and the output is which U-type instruction is being identified by the two opcode bits. Fill out all possible opcode bits that could be the input to the truth table in order to generate the specified outputs (**an output of X means there is no defined instruction with that opcode in our ISA**).

(Hint: the other five bits of the opcode are compared to a **constant** value which identifies the opcode as being a U-type opcode—your task is figuring out which bits can identify which U-type instruction it is.)

Opcode bit(s): ⑥ ⑤ ④ ③ ② ① ⓪	**Opcode bit(s):** ⑥ ⑤ ④ ③ ② ① ⓪	Instruction
0	0	X
0	1	`luip`
1	0	`auipc`
1	1	`lui`

Question 4: I'm afraid of datapaths, so iarrn away - 29 pts

Morgan notices much of the assembly code she writes involves iterating through arrays of integers. Instead of using several instructions to calculate the address of the next element, she proposes a new instruction,

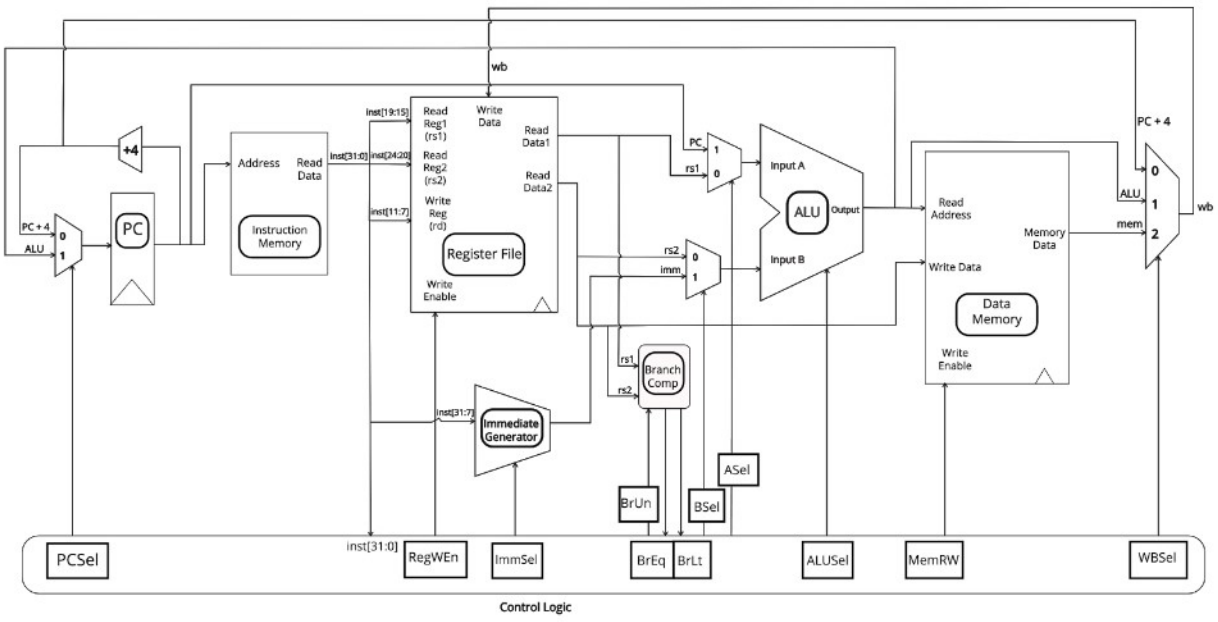
```
iarrn rd rs1 rs2
```

which places into rd the address of the rs2-th element of the array pointed to by rs1. This instruction does not do bounds checking and it assumes the size of an integer is 4B (32 bits). Do *not* assume this instruction belongs to a specific type.

In verilog, the instruction is described as follows:

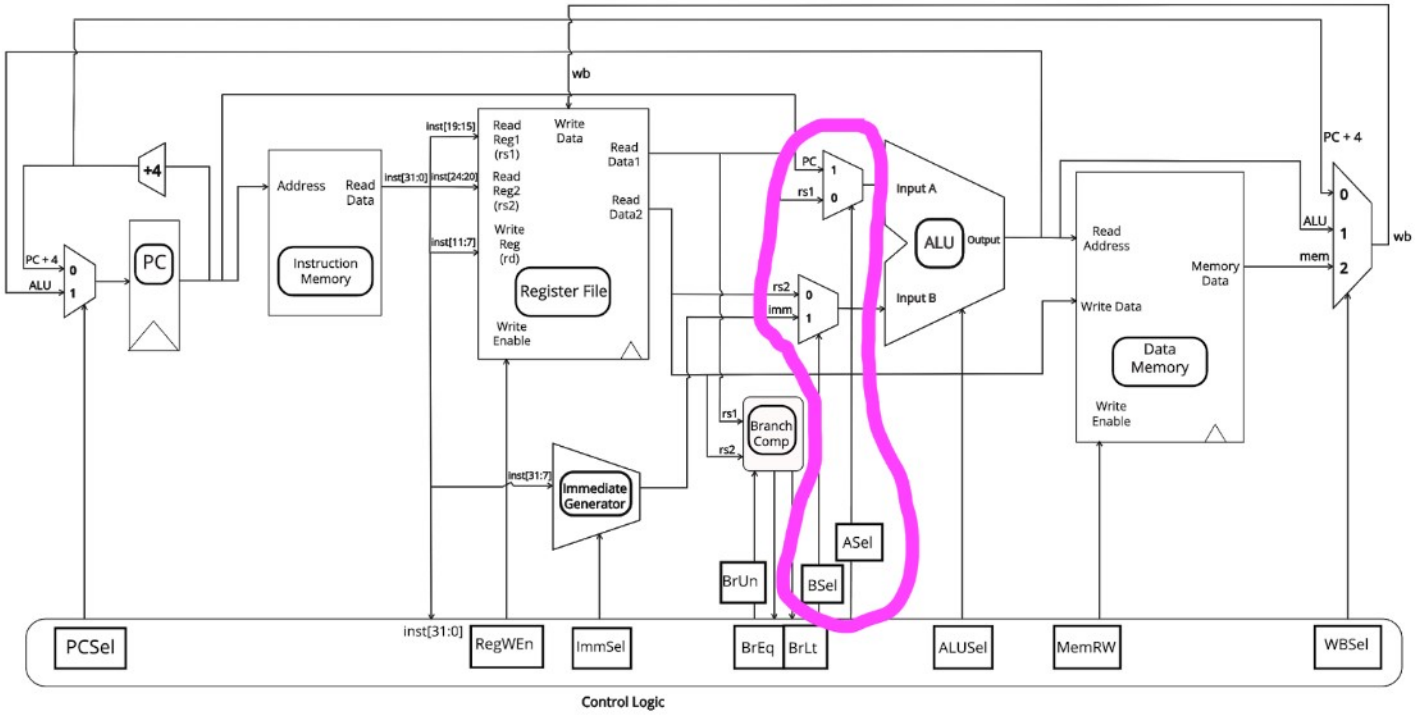
```
R[rd] = R[rs1] + (4 * R[rs2])
```

Morgan is interested in modifying our RISC-V datapath to support this instruction. Assume we have introduced a new control bit "IArrN" which is 1 when the current instruction is `iarrn` and 0 otherwise. Using the datapath below, fill in the following table with the rest of the control bits for this instruction. If the control bit can be set to "*", please draw an X in the table below.



IArrN	PCSel	RegWEn	MemRW	WBSel	BrUn	ALUSel
1						ADD

Morgan notices this instruction involves changing a few hardware pieces on the datapath in addition to changing control bits above. She proposes modifying the ASel and BSel muxes, and their associated control bits (circled below).

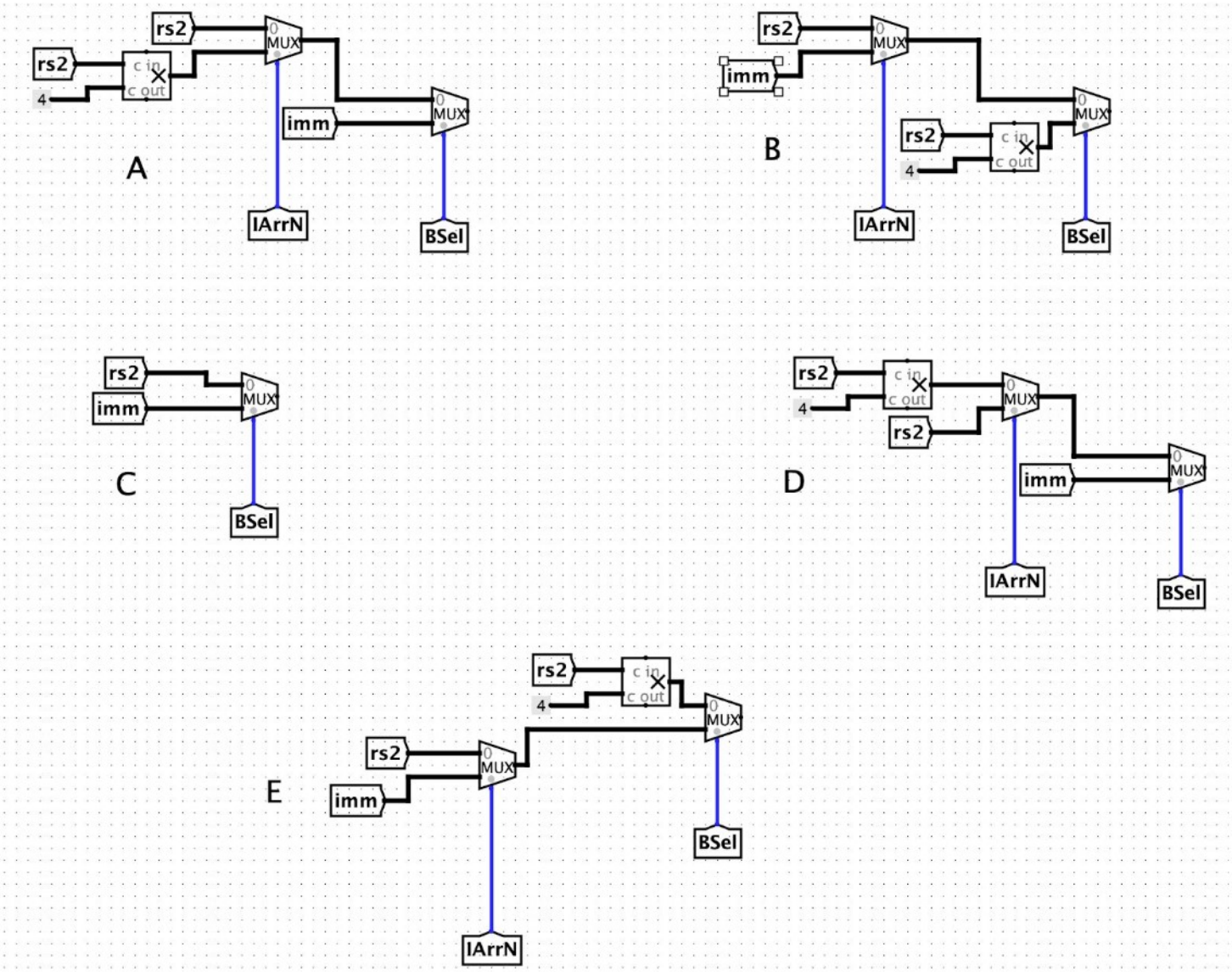


(Questions on next page)

How should we change BSel to allow our new instruction, and **all other RISC-V instructions**, to execute correctly?

Ⓐ Option A Ⓑ Option B Ⓒ Option C Ⓓ Option D Ⓔ Option E

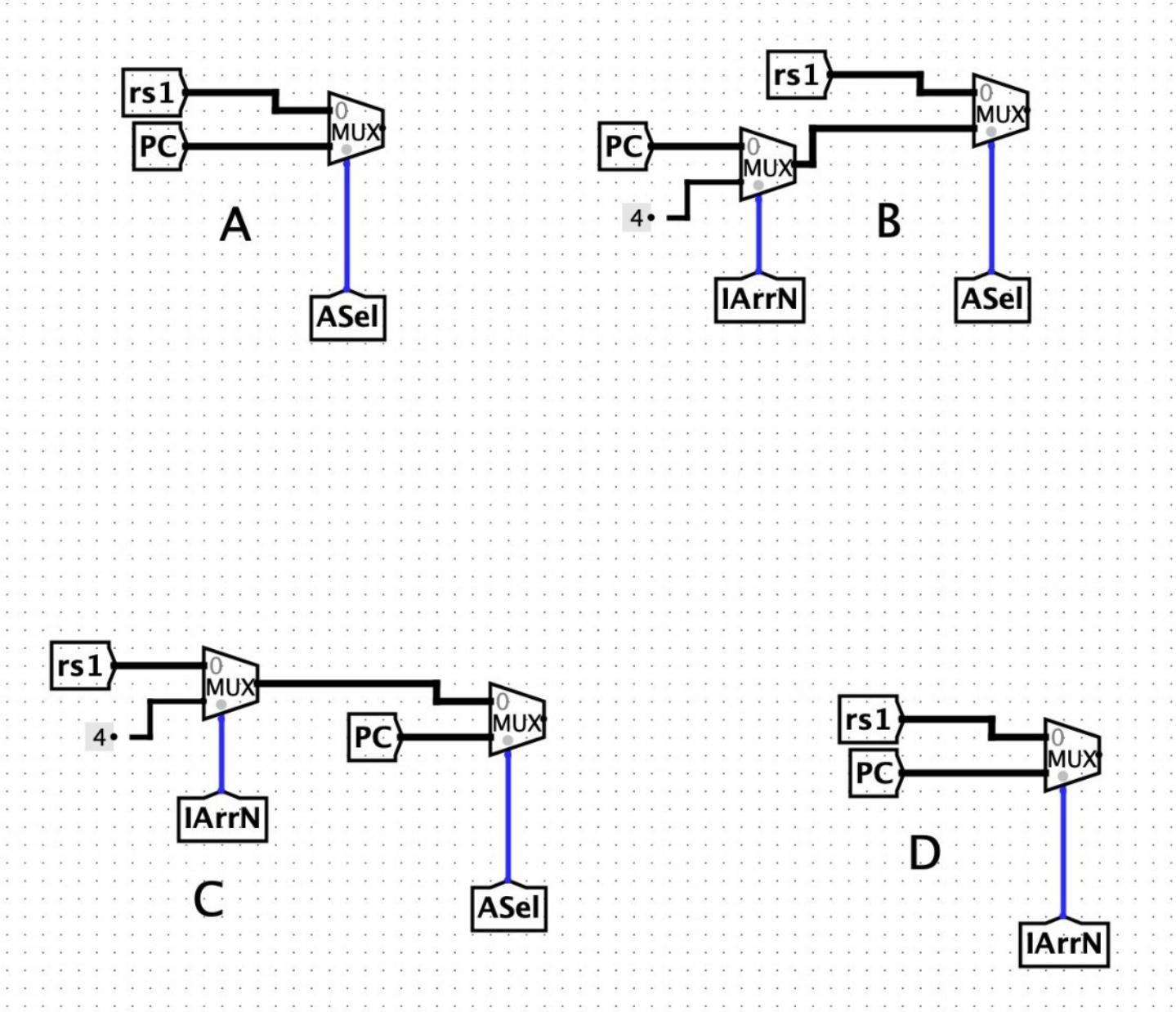
NOTE: some options showcase original hardware from the datapath. If you believe no changes are necessary, you should select this option. Assume our ALU, RegFile, and memory units remain unchanged internally.



How should we change ASel to allow our new instruction, and all other RISC-V instructions, to execute correctly?

Ⓐ Option A Ⓑ Option B Ⓒ Option C Ⓓ Option D

NOTE: some options showcase original hardware from the datapath. If you believe no changes are necessary, you should select this option. Assume our ALU, RegFile, and memory units remain unchanged internally.



Morgan later discovers she spends a lot of time writing assembly that increments the contents of integer arrays. She proposes another new instruction

```
iarrinc rd rs1 rs2
```

which reads the element pointed at by rs1, increments it by rs2, and writes the result to rd. In verilog:

```
R[rd] = M[R[rs1]] + R[rs2]
```

Morgan notices this instruction will *not* execute in our current datapath because it requires a memory access before the execute (ALU) stage. She proposes the following re-orderings. For each option, mark whether it would allow iarrinc to execute correctly and/or whether all other RISC-V instructions would execute correctly (given proper control is added).

Assume stages marked with numbers (ie. EX2) are duplications of the original stage. They may be idle or busy depending on the instruction's needs. Assume branch comparison happens in EX1. Assume all standard execution happens in EX1, and assume that EX2 is used only if an instruction requires additional ALU computation.

1. IF ID EX MEM WB
   [ ] iarrinc can execute
   [ ] all other RISC-V instructions can execute

2. IF ID MEM EX WB
   [ ] iarrinc can execute
   [ ] all other RISC-V instructions can execute

3. IF ID MEM1 EX MEM2 WB
   [ ] iarrinc can execute
   [ ] all other RISC-V instructions can execute

Q6) RISCV Exam-isim Debug – Single Cycle **(12 pts = 2 x 6)**

For your CPU project, you followed the datapath diagram we gave you exactly and built a single-cycle CPU. However, something is not working correctly. *All instructions besides some of the I-types and SB-types are working.* You start by testing with an **`addi a0 a0 -3`** instruction. The **`a0`** register initially holds a value of **7** and all other registers initially hold **0**. This instruction is stored in IMEM at address **`0x00000004`**. DMEM reflects the initial IMEM. Undefined ImmSel outputs an I-type imm. You put a probe at the data read from IMEM and find the instruction is correct. You next put a probe at **wb**, and see the output is `0b0000_0000_0000_0000_0001_0000_0000_0100` (**`0x00001004`**).

a) Since the output is incorrect, what errors alone would cause the erroneous behavior? (select all that apply)

- ☐ The RegWEn is set to false.
- ☐ The Immediate Generator is not sign extending.
- ☐ Read Reg1 and Read Reg2 are flipped.
- ☐ The writeback MUX is selecting DMEM.
- ☐ PCSel is set to PC + 4.
- ☐ The writeback MUX is selecting PC + 4.
- ☐ MemRW is set to write.
- ☐ BSel is selecting rs2 and not imm.

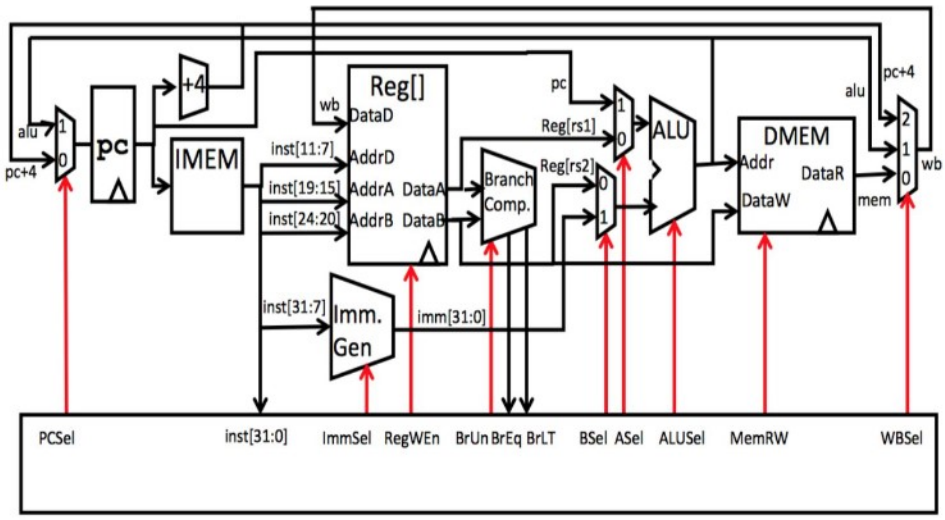
You fix that issue. You then test **`beq x0 a0 label`** but something is still not working. This instruction is at address **`0xbfffff00`** and **`label`** is at address **`0xbfffff40`**. The register **`a0`** holds **0** and all other registers hold **1**. Assume that we get the correct instruction machine code for **`beq x0 a0 label`** when we probe it. You put a probe before the PC register and see this incorrect output: **`0xbfffff20`.**
Note: All other instruction types are working.

b) What errors alone would cause the erroneous behavior? (select all that apply)

- ☐ WBSel is incorrect.
- ☐ ImmSel is Incorrect.
- ☐ PCSel is set to PC + 4.
- ☐ There is an error in the Read Data1/rs1 wire.
- ☐ The ImmGen is not correctly padding w/ extra 0.
- ☐ The inputs to the ASel MUX are flipped.
- ☐ The inputs to the BSel MUX are flipped.
- ☐ Read Reg1 and Read Reg2 are flipped.

Q6: Game of Signals

The following canonical single-cycle datapath for a RISC-V architecture supports the base RISC-V ISA (RV32I).



For the following questions, we want to expand our instruction set to support some new instructions. For each of the proposed instructions below, choose ONE of the options below.

A. Can be implemented without changing datapath wiring, only changes in control signals are needed.(i.e. change existing control signals to recognize the new instruction)
B. Can be implemented, but needs changes in datapath wiring, only additional wiring, logical gates and muxes are needed.
C. Can be implemented, but needs change in datapath wiring, and additional arithmetic units are needed (e.g. comparators, adders, shifters etc.).
D. Cannot be implemented.

Note that the options from A to D gradually add complexity; thus, selecting B implies that A is not sufficient. You should select the option that changes the datapath the least (e.g. do not select C if B is sufficient). You can assume that necessary changes in the control signals will be made if the datapath wiring is changed.

1) Allowing software to deal with 2's complement is very prone to error. Instead, we want to implement the negate instruction, neg rd,rs1, which puts -R[rs1] in R[rd].

2) Sometimes, it is necessary to allow a program to self-destruct. Implement `segfault rs1, offset(rs2)`. This instruction compares the value in `R[rs1]` and the value in `MEM[R[rs2]+offset]`. If the two values are equal, write 0 into the PC; otherwise treat this instruction as a NOP.

Problem 8 ***[M2-4] Datapath*** **(10 points)**

Recall the standard 5-stage, single cycle datapath contains stages for Instruction Fetch, Decode, Execute (ALU), Memory, and Write-back. Datapath designers are interested in reducing the phases necessary for execution such that instead of accessing both the Execute (ALU) phase and the Memory phase, instructions access either one or the other, but not both. This would create a 4-stage, single cycle datapath with the following stages: Instruction Fetch, Decode, Execute OR Memory, and Write-back.

Instr Fetch	Instr Decode	Execute (ALU)	Memory	Write-back
100ps	150ps	200ps	350ps	150ps

(a) Given the table above and the described datapath above, what is the time it takes for a single instruction that utilizes all stages to execute on the typical 5-stage, single cycle datapath?

(b) What is the time it takes for a single instruction that utilizes all stages to execute on the new 4-stage, single cycle datapath?

(c) If the designers go ahead with this modification, which instructions will NOT function correctly? Why? Please limit your answer to two sentences or less.

(d) Propose a program-level modification that will fix the issue. Do NOT propose a modification to the datapath. Please describe your modification in two sentences or less.

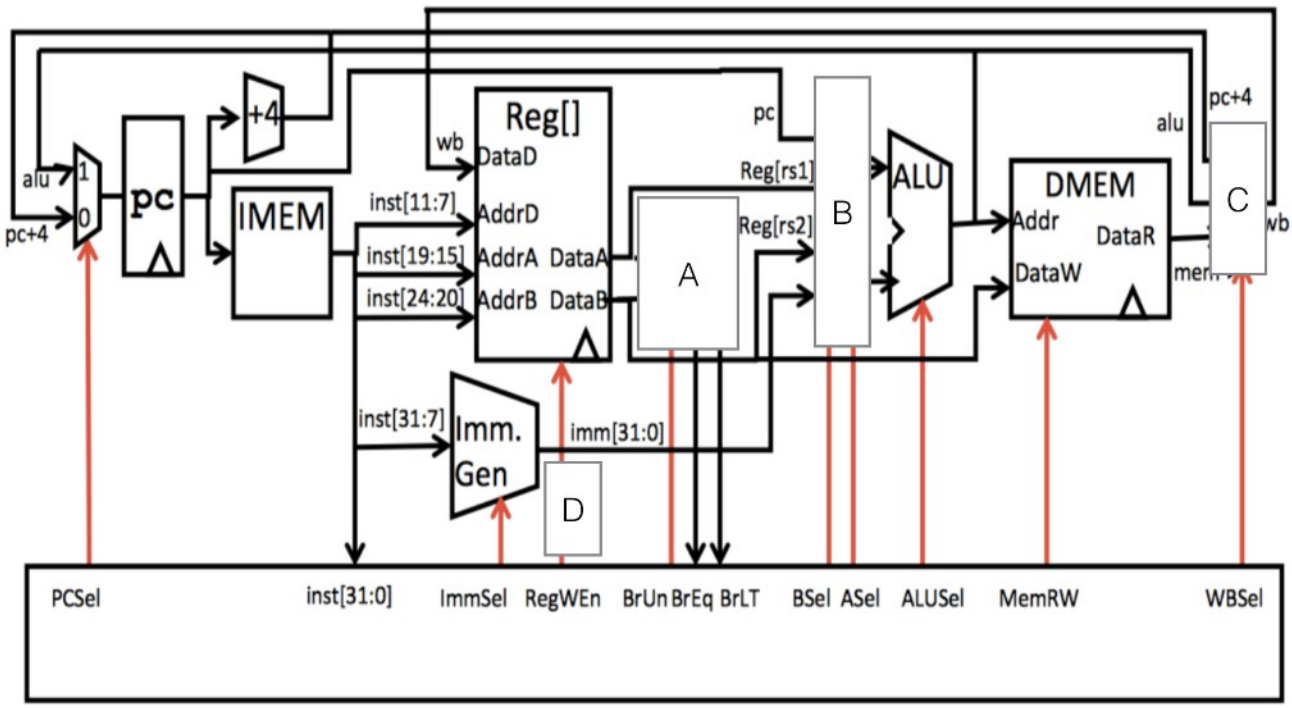
Problem 3 ***Set ... If Zero*** **(19 points)**

We wish to introduce a new instruction into our single-cycle datapath. The instruction **SIZ** (set if zero) works as follows:

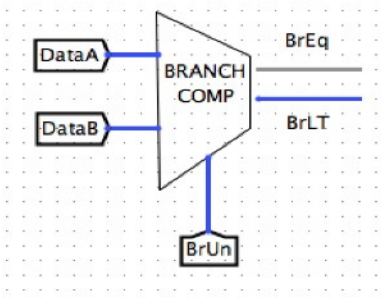
```
if (R[rs2] == 0):
    R[rd] = R[rs1]
```

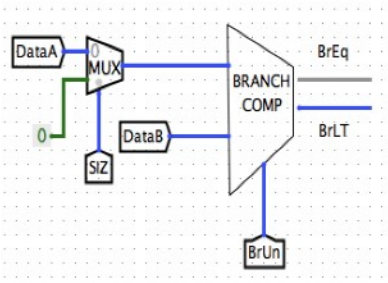
Given the single cycle datapath below, select the correct modifications in parts (a) - (d) such that the datapath executes correctly for this new instruction (and all core instructions!). You can make the following assumptions:

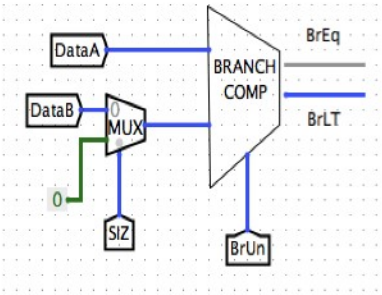
- the `SIZ` signal is 1 if and only if the instruction is `SIZ`
- `ALUSel` is **ADD** when when we a have `SIZ` instruction.
- **the immediate generator outputs ZERO** when we have a `SIZ` instruction.

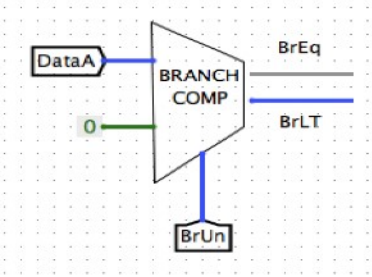


(a) Consider the following modifications to the branch comparator inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set?

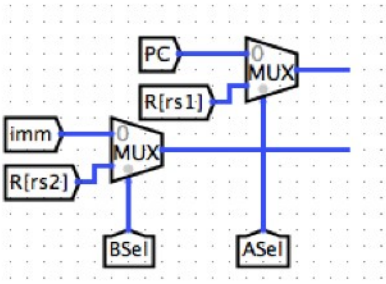
○ 

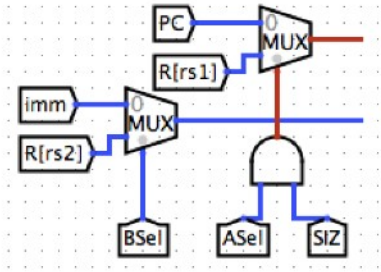
○ 

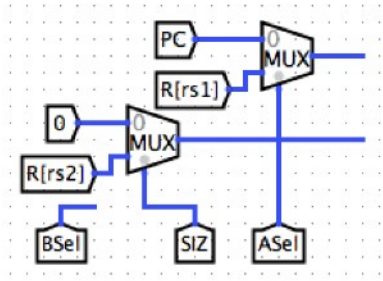
○ 

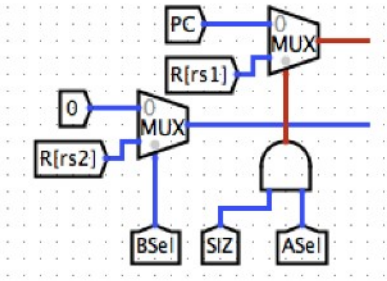
○ 

(b) Consider the following modifications to the ALU inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set? Select the configuration that requires **minimum** modifications to the original datapath. Notice in the bottom left choice `BSel` is unused.

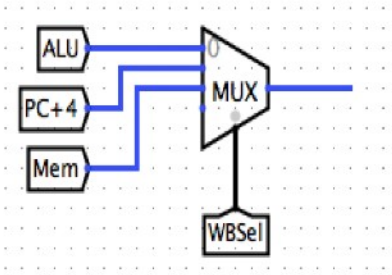
○ 

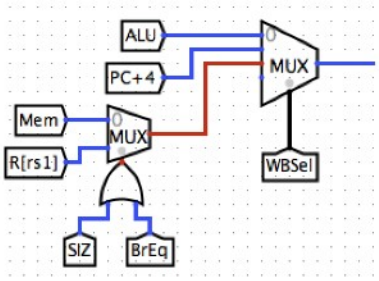
○ 

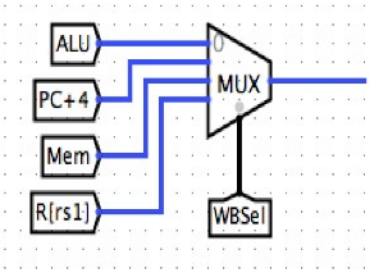
○ 

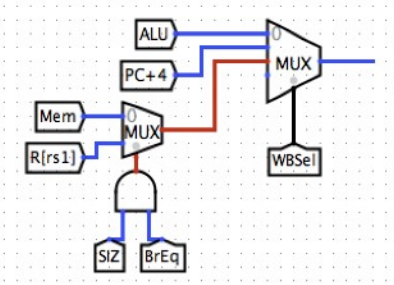
○ 

(c) Consider the following modifications to the WB mux inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set? Select the configuration that requires **minimum** modifications to the original datapath.

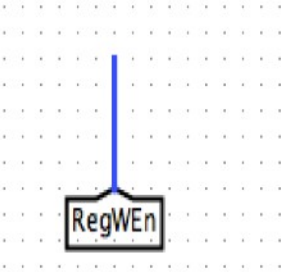
○ 

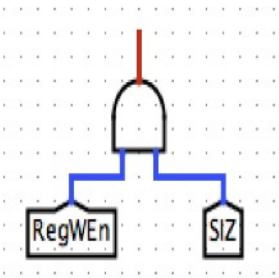
○ 

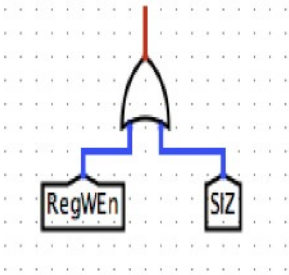
○ 

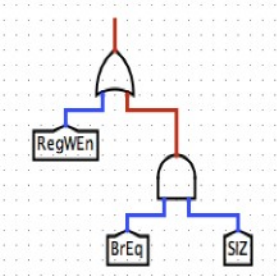
○ 

(d) Consider the following modifications to the RegWEn inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set?

○ 

○ 

○ 

○ 

(e) Given your selections above, decide the rest of the control signals for this instruction based on the diagram given at the beginning of the problem. Select `X` when a signal's value doesn't matter. You can assume:

- the `SIZ` signal is 1 if and only if the instruction is `SIZ`
- `ALUSel` is **ADD** when when we a have `SIZ` instruction.
- the immediate generator outputs **ZERO** when we have a `SIZ` instruction.

1. `PCSel`:

   ○ 1 ○ 0 ○ X

2. `RegWEn`:

   ○ 1 (Enable) ○ 0 (Disable) ○ X

3. `BrUn`:

   ○ 1 (Signed) ○ 0 (Unsigned) ○ X

4. `BSel`:

   ○ 1 ○ 0 ○ X

5. `ASel`:

   ○ 1 ○ 0 ○ X

6. `MemRW`:

   ○ 1 (Enable) ○ 0 (Disable) ○ X

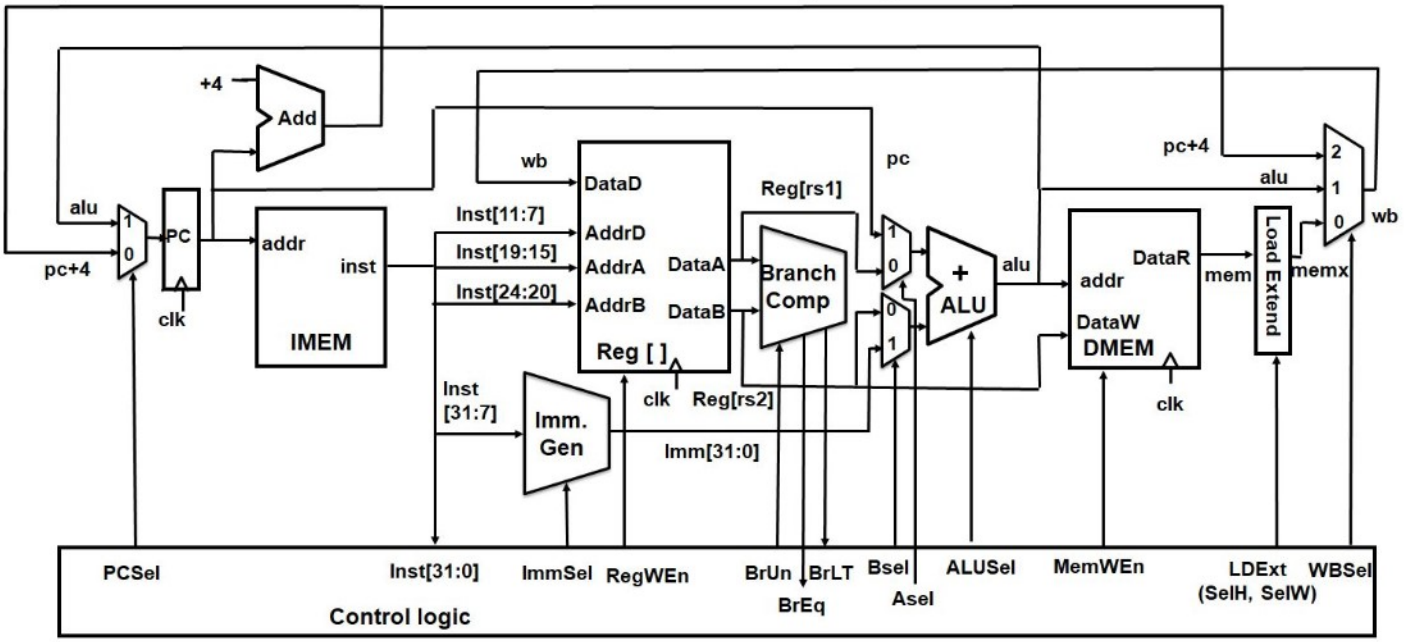
7. `WBSel`

   ○ ALUOut ○ MemOut

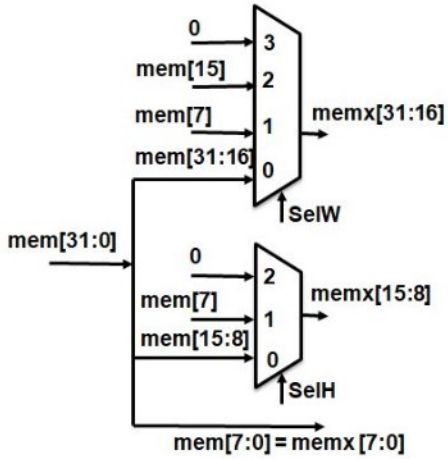
   ○ PC + 4 ○ Other: Please specify:_____

F3) *Datapathology* [this is a 2-page question] (20 points = 4+10+6, 30 minutes)

The datapath below implements the RV32I instruction set. We'd like to implement sign extension for loaded data, but our loaded data can come in different sizes (recall: **`lb`**, **`lh`**, **`lw`**) and different intended signs (**`lbu`** vs. **`lb`** and **`lhu`** vs **`lh`**). Each load instruction will retrieve the data from the memory and "right-aligns" the LSB of the byte or the half-word with the LSB of the word to form **`mem[31:0]`**.



a) To correctly load the data into the registers, we've created two control signals **SelH** and **SelW** that perform sign extension of **`mem[31:0]`** to **`memx[31:0]`** (see below). **SelH** controls the half-word sign extension, while **SelW** controls sign extension in the two most significant bytes. What are the Boolean logic expressions for the four (0, 1, 2, 3) **SelW** cases in terms of Inst[14:12] bits to handle these five instructions (**`lb`**, **`lh`**, **`lw`**, **`lbu`** and **`lhu`**)? **SelH** has been done for you. In writing your answers, use the shorthands "I14" for Inst[14], "I13" for Inst[13] and "I12" for Inst[12]. You don't have to reduce the Boolean expressions to simplest form. (Hint: green card!)



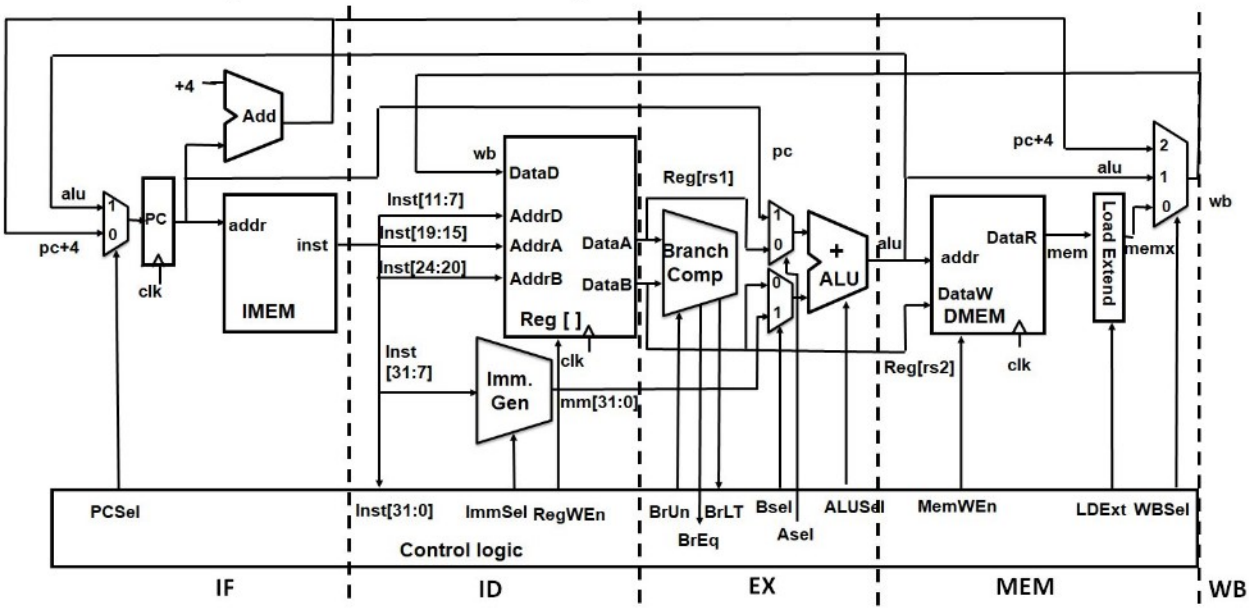
SelW=3	
SelW=2	
SelW=1	
SelW=0	

SelH=2	I14 • ~I13 • ~I12
SelH=1	~I14 • ~I13 • ~I12
SelH=0	I13 + I12

(Single-bit values **`mem[7]`** and **`mem[15]`** are wired to 8 or 16 outputs)

F3) *Datapathology, continued* (20 points = 4+10+6, 30 minutes) SID_____

(this is the same diagram as on the previous page, with five stages of execution annotated)



b) In the RISC-V datapath above, mark what is used by a **`jal`** instruction. *(See green card for its effect...)*

Select one per row	**PCSel Mux:**	◯ "pc + 4" branch	◯ "alu" branch	◯ * (don't care)	
	ASel Mux:	◯ "pc" branch	◯ Reg[rs1] branch	◯ * (don't care)	
	BSel Mux:	◯ "imm" branch	◯ Reg[rs2] branch	◯ * (don't care)	
	WBSel Mux:	◯ "pc + 4" branch	◯ "alu" branch	◯"mem" branch	◯ * (don't care)
Select all that apply	**Datapath units:**	▢ Branch Comp	▢ Imm. Gen	▢ Load Extend	
	RegFile:	▢ Read Reg[rs1]	▢ Read Reg[rs2]	▢ Write Reg[rd]	

c) If we convert the above datapath to a 5-stage pipeline with **no forwarding**, what types of hazards (S=structural, D=data, C=control) exist **after** each line in the following code; how many **`nops`** must we add? (Assume a register can be written and read in the same cycle, and that the Branch Comp is in the EX stage.)

```
start: lw    t0, 0(a0)        Hazard (circle one): S D C None   # of nop _____
       beq   t0, 0, end       Hazard (circle one): S D C None   # of nop _____
       addi  t0, t0, 2        Hazard (circle one): S D C None   # of nop _____
       sw    t0, 0(a0)        Hazard (circle one): S D C None   # of nop _____
end:
```