Consider two competing 8-bit floating point formats. Each contains the same fields (sign, exponent, significand) and follows the same general rules as the 32-bit IEEE standard (denorms, biased exponent, non-numeric values, etc.), but allocates its bits differently. To save you time, *you only need to complete and circle the (LEFT or RIGHT) blank whose value is closest to zero*, that's the only one we'll grade! (If they're the same value, write the answer in both, & circle both). E.g,. The number represented by `0x00` was 0 for both, so we circled both. But for "exponent bias", just from the # of `EE…E` bits in each, we know |LEFT's bias| < |RIGHT's bias|, so there's no need to calculate or write the answer on the RIGHT.

"LEFT" format: | S | EE | MMMM |

scratch space (show all work here)

EE=00 → denorm, 0.MMMM * $2^{-(BIAS-1)}$ =
0.MMMMEE=01 → 1.MMMM * $2^{1-BIAS=0}$ =
1.MMMMEE=10 → 1.MMMM * $2^{2-BIAS=1}$ =
1M.MMMEE=11 → Inf, NaNs

Number represented by `0x00`: 0

Exponent Bias: 1

Numbers ($0 \le n < 1$): (32)#

($1 \le n < 2$): 32# Numbers

c) Difference between two smallest positive values: 2^{-5}

d) Difference between two biggest non-∞ values: (2^{-4})

e) Positive Integer closest to 0 it cannot represent: (4)

"RIGHT" format: | S | EEEEEE | M |

scratch space (show all work here)

E…E=000000→denorm, 0.M * $2^{-(BIAS-1)}$ = 0.M * 2^{-30}
E…E=000001→ 1.F * $2^{1-BIAS=-30}$ = 1.M * 2^{-30}#=1 →
1.0*2^0 → M=0,E…E-31=0 → E…E=31SE…EM =
$00111110_2 = 62_{10}$E…E=111110→ 1.M * $2^{62-BIAS=31}$ =
1.M * 2^{31}E…E=111111→ Inf, NaNs

Number represented by `0x00`: 0

Exponent Bias: 31 (not graded, so no need to write)

Numbers ($0 \le n < 1$): 62#

Numbers ($1 \le n < 2$): (2)#

Difference between two smallest positive values: (2^{-31})

Difference between two biggest non-∞ values: 2^{30}

Positive Integer closest to 0 it cannot represent: 5

f) Which implementation is better for *approximating* π, (LEFT) or RIGHT ? (circle one)

1) For a 12-bit integer represented with two's complement, what is the:

a) Most positive value (in decimal): _____2047_____

b) Binary representation of that number: _0b011111111111__

c) Most negative value (in decimal): _____-2048_____

d) Hex representation of that number: _____0x800_____

e) In general, for an n-bit, two's complement integer:

i) What is the largest value you can represent, in decimal? ___2^(n-1) - 1___

ii) What is the smallest value you can represent, in decimal? _____-2^(n-1)_____

2) Fill in the blank below so that the function `mod16` will return the remainder of `x` when divided by 16. The first blank should be a bitwise operator, and the second blank should be a single decimal number:

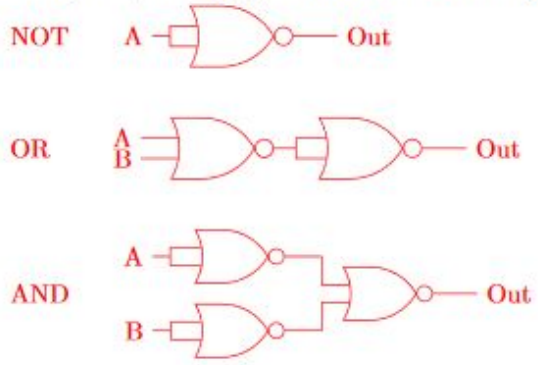
```
unsigned int mod16(unsigned int x) {
    return x ___&____ _______15_______;
}
```

Connect the definition with the name of the process that describes it.

a) Compiler
b) Assembler
c) Linker
d) Loader

1) Outputs code that may still contain pseudoinstructions. __a__
2) Takes binaries stored on disk and places them in memory to run. __d__
3) Makes two passes over the code to solve the "forward reference" problem. __b__
4) Creates a symbol table. __b__
5) Combines multiple text and data segments. __c__
6) Generates assembly language code. __a__
7) Generates machine language code. __b__
8) Only allows generation of TAL. __b__
9) Only allows generation of binary machine code. __c__

(b) 2-input NOR gates are said to be complete because any Boolean function can be computed with them. Prove this fact. Hint: implement a subset of the standard gates (AND, NOT, OR, NOR, NAND, XOR, XNOR) using just NOR gates, then apply a standard boolean algebra technique using these gates.



Using these three gates and applying the sum of products technique we can compute any boolean function.

(c) We want to implement a very simple finite state machine that determines its next state by the result of and AND operation on the current state and the input. The output is always the current state. Assume registers have a CLK to Q delay of 5ns, a setup time of 2ns, and a hold time of 3ns. To achieve a clock rate of 25MHz, what is the maximum propogation delay that a NOR gate could have, assuming we are implementing AND as a combination of one or more of the gates built in part (b)?

Max Delay = CLK-to-Q + CL + Setup Time

Max Freq = 1/Max Delay

$$\frac{1}{25\text{MHz}} = 40\text{ns}$$

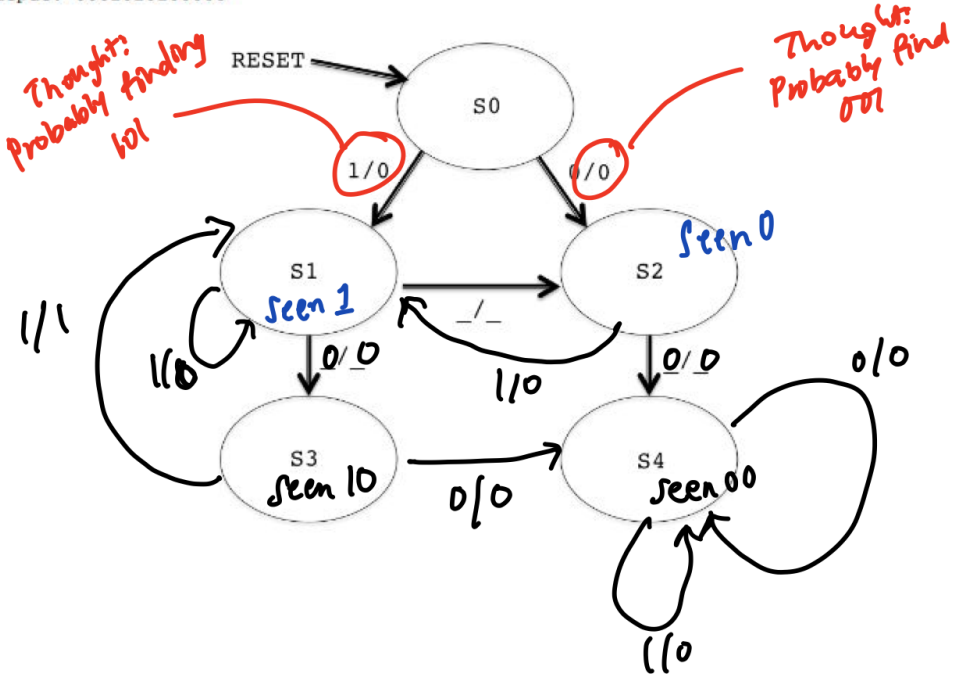
$$40\text{ns} = 5\text{ns} + x + 2\text{ns}$$

$$x = 33\text{ns}$$

Our implementation of AND has a critical path of 2 NOR gates, so each NOR gate must have a delay less than or equal to 33/2 = 16.5ns

(d) Complete the state diagram for a finite state machine that outputs 1 if and only if it has just seen the input sequence 101 and it has never seen the input sequence 001. You may add more arrows or more states as you see fit. Provide a brief description of each state.

```
Example
Input : 1101010100101
Output: 0001010100000
```



Considering the standard 32-bit RISC-V instruction formats, convert `lw t5, 17(t6)` to machine code:

(a) **Solution:** 0x011FAF03

Prof. Wawrzynek decides to design a new ISA for his ternary neural network accelerator. He only needs to perform 7 different operations with his ISA: XOR, ADD, LD, SW, LUI, ADDI, and BLT. He decides that each instruction should be 17 bits wide, as he likes the number 17. There are no funct7 or funct3 fields in this new ISA.

(b) What is the minimum number of bits required for the opcode field?

Solution: $\lceil log_2 7 \rceil = 3$

(c) Suppose Prof. Wawrzynek decides to make the opcode field 6 bits. If we would like to support instructions with 3 register fields, what is the maximum number of registers we could address?

Solution: $\lfloor (17-6)/3 \rfloor = 3$ bits per register field which means 8 registers we could address

Assume we have two arrays `input` and `result`. They are initialized as follows:

```
int *input = malloc(8*sizeof(int));
int *result = calloc(8, sizeof(int));
for (int i = 0; i < 8; i++) {
    input[i] = i;
}
```

You are given the following RISC-V code. Assume register `a0` holds the address of `input` and register `a2` holds the address of `result` when `MAGIC` is called by `main`.

```
main:
    ...
    # Start Calling MAGIC
    addi a1, x0, 8
    jal ra, MAGIC      # a0 holds input, a2 holds result
    # Checkpoint:  finished calling MAGIC
    ...
exit:
    addi a0, x0, 10
    add a1, x0, x0
    ecall      # Terminate ecall
MAGIC:
    # TODO: prologue.  What registers need to be stored onto the stack?
    mv s0, x0
    mv t0, x0
loop:
    beq t0, a1, done
    lw t1, 0(a0)
    add s0, s0, t1
    slli t2, t0, 2
    add t2, t2, a2
    sw s0, 0(t2)
    addi t0, t0, 1
    addi a0, a0, 4
    jal x0, loop
done:
    mv a0, s0
    # TODO: epilogue.  What registers need to be restored?
    jr ra
```

(a) Consider the function `MAGIC`. The prologue and epilogue for this function are missing. Which registers should be saved/restored in `MAGIC`'s prologue/epilogue? Select all that apply.

- ○ t0
- ○ t1
- ○ t2
- ● s0
- ○ a0
- ○ a1
- ○ a2
- ○ ra
- ○ x0

(b) Assume you have the prologue and epilogue correctly coded. You set a breakpoint at "Checkpoint: finish calling `MAGIC`" and call `main`. What does `result` contain when your program pauses at the breakpoint? Please write the 8 numbers starting at `result` in the blanks below.

Solution: 0 1 3 6 10 15 21 28

(c) Translate `MAGIC` into C code. You may or may not need all of the lines provided below.

Solution:

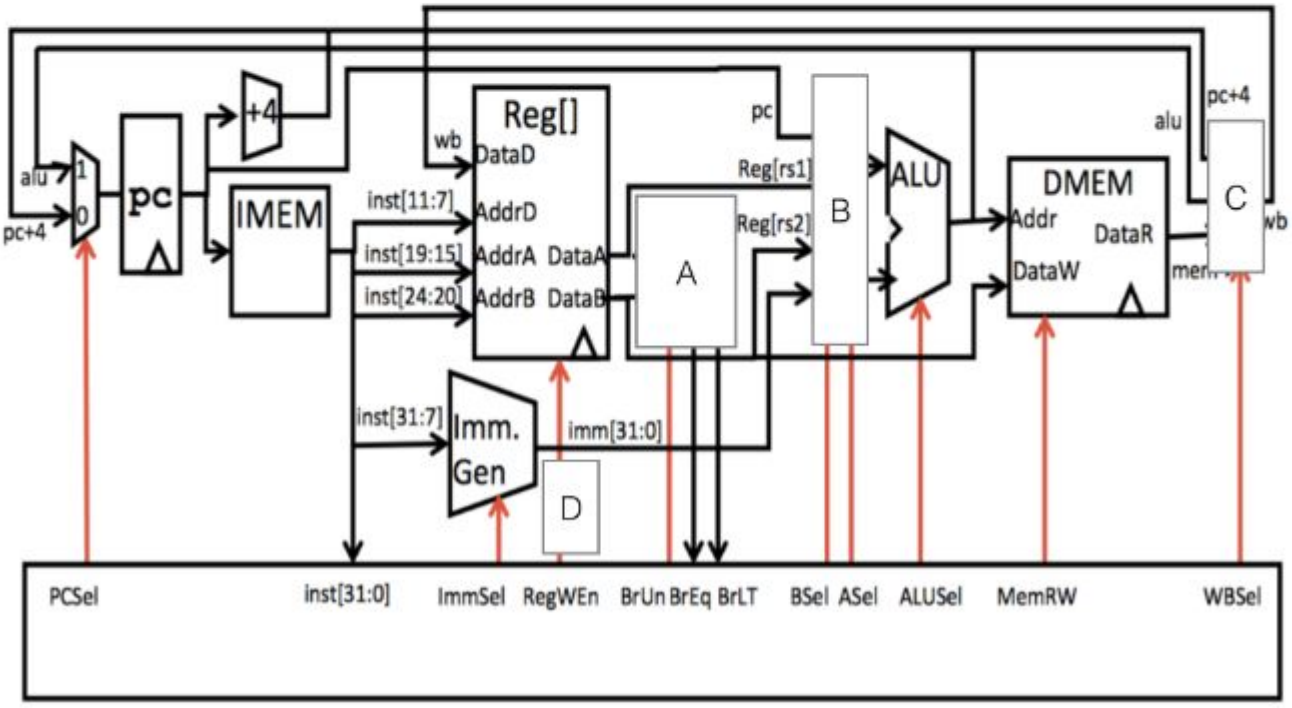
```
// sizeof(int) == 4
int MAGIC(int *a, int b, int *c) {
    int sum = 0;
    for (int i = 0; i < b; i++) {
        sum += a[i];
        c[i] = sum;
    }
    return sum;
}
```

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

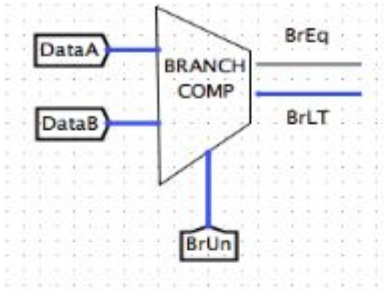
```
if (R[rs2] == 0):
    R[rd] = R[rs1]
```

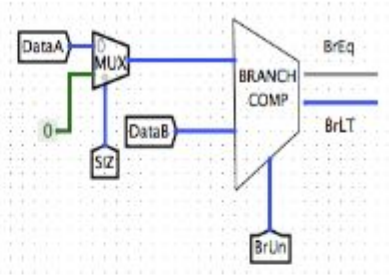
Given the single cycle datapath below, select the correct modifications in parts (a) - (d) such that the datapath executes correctly for this new instruction (and all core instructions!). You can make the following assumptions:

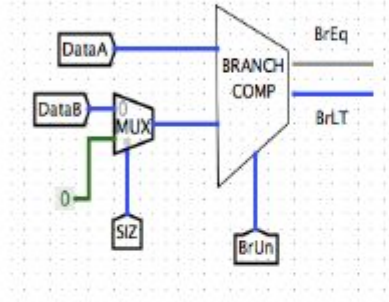
- the `SIZ` signal is 1 if and only if the instruction is `SIZ`
- `ALUSel` is **ADD** when when we a have `SIZ` instruction.
- **the immediate generator outputs ZERO** when we have a `SIZ` instruction.

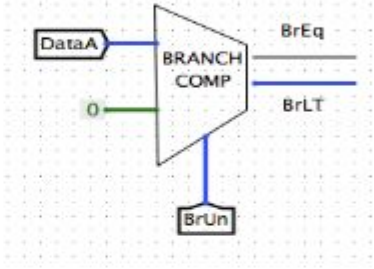


(a) Consider the following modifications to the branch comparator inputs. Which configuration will allow this instruction to execute correctly without breaking the execution of other instructions in our instruction set?

○ 

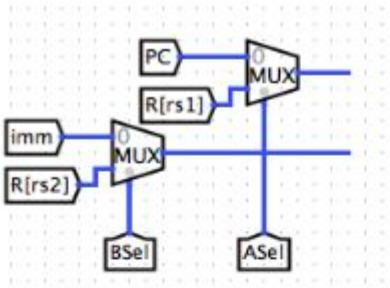
● 

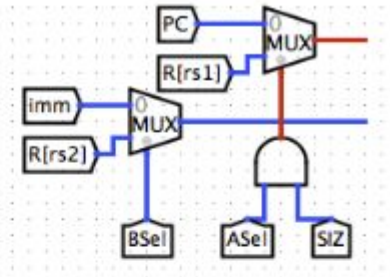
○ 

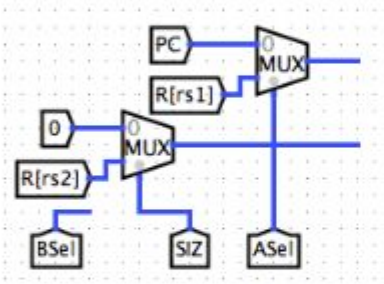
○ 

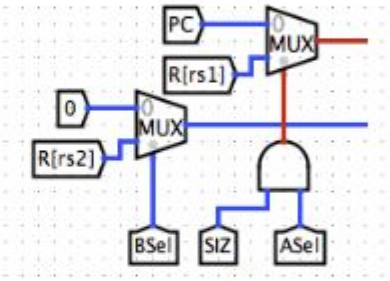
Solution: Note that for this question we have two requirements: the modification we pick must support our new instruction AND it must make it so all other instructions (those in our core instruction set) continue to execute correctly as if no changes were made. If we look at the logic describing the instruction behaviour in the first part of the question, we notice we must compare the value in register rs2 to zero. Unlike normal branch instructions, we are not comparing to another register value; we are comparing to a constant. This eliminates choice A. Noting the conditions of our modification (that it must be able to support core instructions, too) we can eliminate D because it removes the ability to compare DataA and DataB which we need for regular branch instructions. We are left with options B and C. Again, if we revisit the instruction logic, we see the item we're comparing to zero is the value in rs2; this is equal to DataB. We therefore pick option B which adds a MUX on DataA to allow us to select 0 as our second operand in the case of a SIZ instruction.

(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.

● 

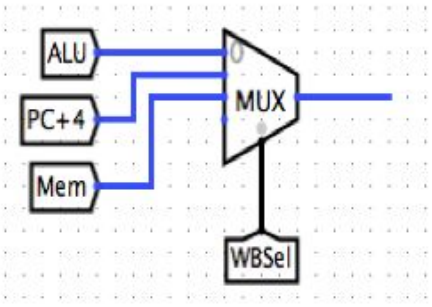
○ 

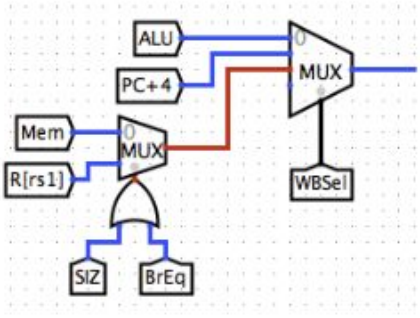
○ 

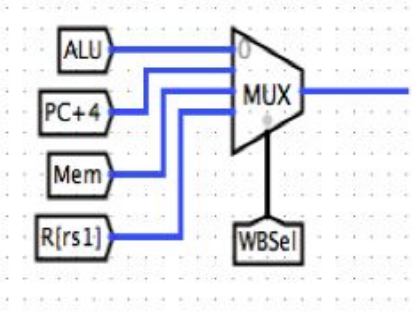
○ 

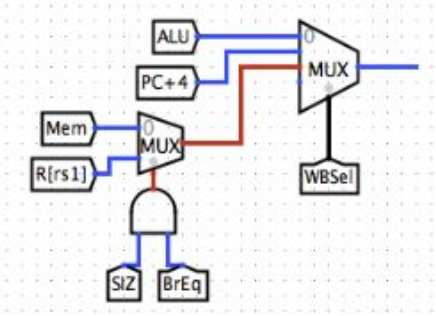
Solution: This modification must also support our new instruction while allowing other instructions to continue executing as normal. This section concerns inputs to our ALU–the output of which we will write. Looking again at the instruction logic, we can see the write we need to make is R[rd] = R[rs1]; the value in register rs1 should be written to the rd register. Therefore, the output of our ALU should be the value in register rs1. If we look at our datapath, this is already possible by manipulating existing controls (ASel, BSel), and so we do not need to make any modifications; A is the correct answer.

(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.

● 

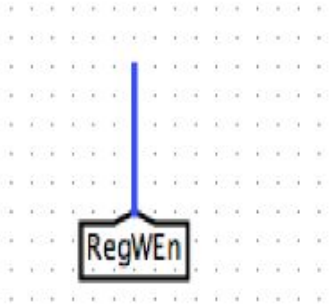
○ 

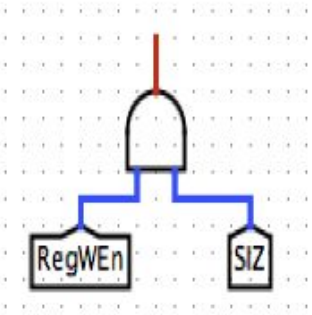
○ 

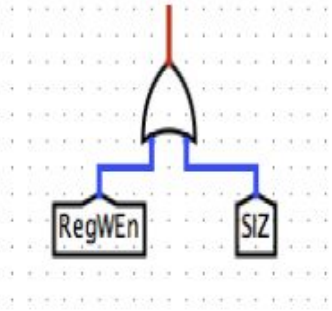
○ 

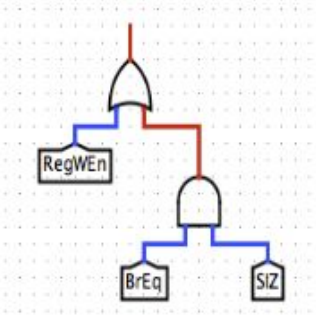
Solution: Similar to the previous question, we do not need to make a modification to the datapath and should therefore select option A. We know our ALU is emitting the value stored in register rs2. Because the standard datapath already allows us to write back the output of our ALU (and because write-back by default writes to our specified destination register rd), the value in rs2 can be written back to rd by setting our existing control bit WBSel to ALU. Again, no modifications are required.

(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?

○ 

○ 

○ 

● 

Solution: To answer this question we should look at the instruction logic to find out under what conditions the write occurs. Note that, in the SIZ instruction, we should only set R[rd] = R[rs1] in the case that R[rs2] == 0 is true. In order to check that condition, we need to make sure of two things: (a) the instruction we're writing back for should be a SIZ instruction and (b) the result of the branch equality comparison should be true (BrEq). Because we want both of these to be true before we write, we use an AND gate. To preserve existing functionality, we also want to keep our RegWEn control bit around. Because the additional logic we added for the SIZ instruction will be false for all other instructions (add, load, etc.) we use an OR gate to support write-back functionality for our core instructions.

(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

   Solution: Though this instruction is similar to other branch instructions in that it uses the branch comparator to check equality, it does not alter our control flow as a result, therefore we should select PC as PC+4 like we do normally.

2. `RegWEn`:

   ○ 1 (Enable) ● 0 (Disable) ○ X

   Solution: Our write to rd should only happen in the case that our if case is true. We added logic to support this in the previous question and, in order for that check to happen, we have to set RegWEn to false. If it were true, we would write to rd on every SIZ instruction, not just those where R[rs2] == 0.

3. `BrUn`:

   ○ 1 (Signed) ○ 0 (Unsigned) ● X

   Solution: We are doing a comparison to zero. Whether the branch comparison is signed or unsigned has no effect on the outcome.

4. `BSel`:

   ○ 1 ○ 0 ● X

   Solution: We want our ALU to produce rs1 as its output. We do not care what the value of our second operand is (because regardless, it isn't the output we want) and therefore it doesn't matter if we pass in the immediate or DataB.

5. `ASel`:

● 1 ○ 0 ○ X

Solution: Though we don't care about BSel, we do care about ASel because this is where we have the option of passing in rs1 to the ALU. If we want our write to contain the correct information, we must select DataA as our input here.

6. `MemRW`:

○ 1 (Enable) ● 0 (Disable) ○ X

Solution: This instruction makes no modifications to memory and therefore shouldn't write to memory. The reason we can't leave this control bit as an X (don't care) is because modifying memory is a state change; if we "don't care" and modify memory on some SIZ instructions and not on others, we could end up overwriting, deleting, or forcing garbage data into our memory and messing up execution of later instructions.

7. `WBSel`

● ALUOut ○ MemOut

○ PC + 4 ○ Other: Please specify:_____

Solution: See solution to part C above