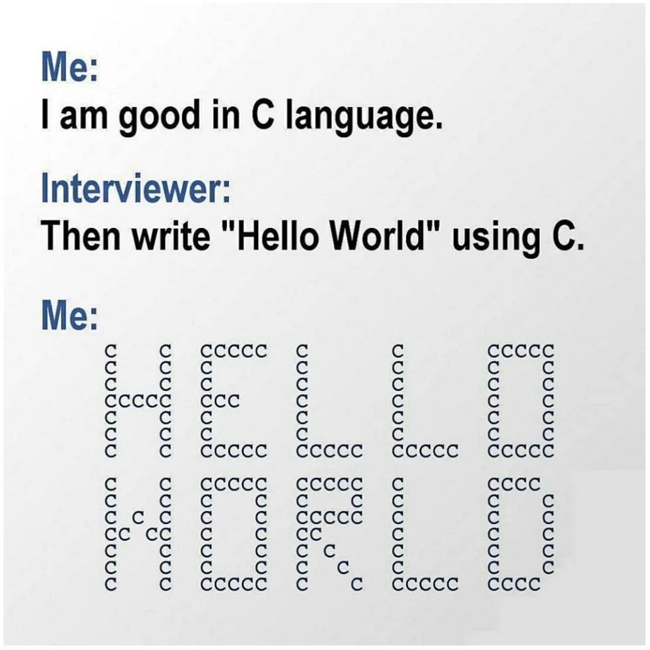
Your Name (first last)	# UC Berkeley CS61C Fall 2019 Midterm	SID
← Name of person on left (or aisle)	TA name	Name of person on right (or aisle) →

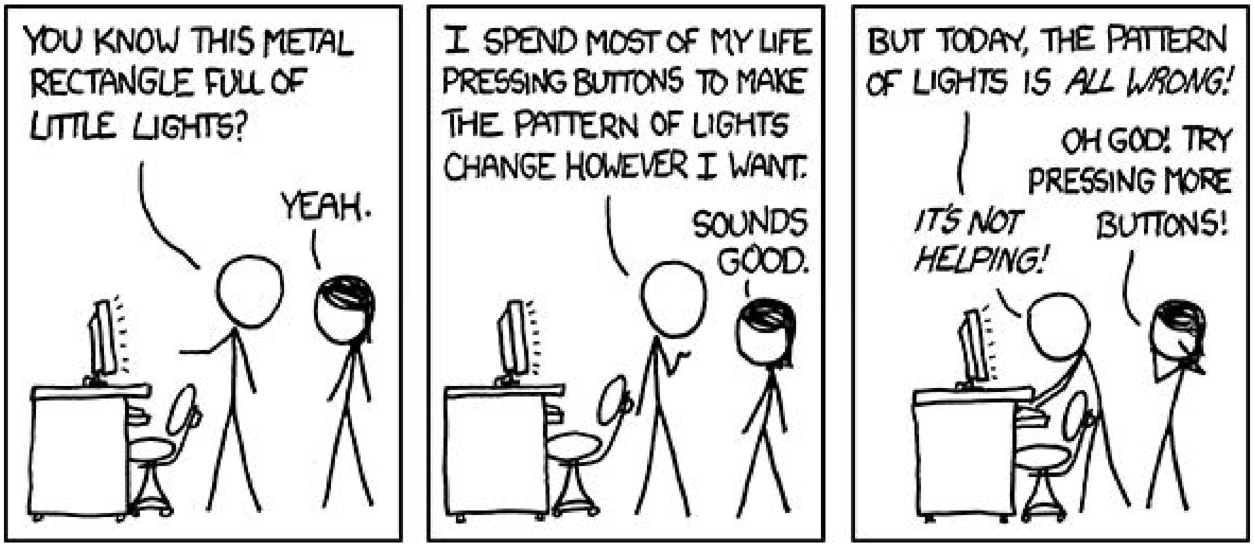
Fill in the correct circles & squares completely…like this: ● (select ONE), and ■(select ALL that apply)

Quest-clobber question: Q3

When you see SHOW YOUR WORK, that means a correct answer *without work* will receive NO CREDIT, and your work needs to show how you were led to the answer you reached. If you find that there are multiple correct answers to a "select ONE" question, please choose just one of them.

How would you write a C program to prevent titanic from sinking?
float titanic;




YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

Q1) *Float, float on...* **(7 pts = 2 + 3 + 2)**

You notice that floats can generally represent much larger numbers than integers, and decide to make a modified RISC instruction format in which all immediates for jump instructions are treated as 12-bit floating point numbers with a mantissa of 7 bits and with a standard exponent bias of 7. *Hint: Refer to reference sheet for the floating point formula if you've forgotten it...the same ideas hold even though this is only a 12-bit float…*

a) To jump the farthest, you set the float to be the most positive (not ∞) integer representable. *What are those 12 bits* (in hex)?	b) What is the *value* of that float (in decimal)?	c) Between 0 and (b)'s answer (inclusive), *how many integers are not representable*?
0x77F	255	0

Part A

Since you want the most positive float, the **sign bit** should be 0. For the **exponent**, you want the second biggest possible exponent, as the biggest possible exponent is always used for NaN and infinity. With 4 exponent bits, the largest possible number is 1111 or 15, so the second largest is 1110, or 14 (as an unsigned number; with the bias, this becomes 2^7).

With an exponent of 2^7 and 7 mantissa bits, when you multiply 1.<mantissa> by the exponent, the decimal point will end up right after the last mantissa bit. This means *every* number representable with an exponent of 2^7 is an integer, so you just need the largest one (which will be when you have all 1's). Putting the three parts together, 0 -- 1110 --1111111 is 0x77F.

Part B

The value of the float above will be $(-1)^0 * 2^7 * 1.1111111 = 11111111.0$ in binary, or 255.

Part C

For an exponent of 2^7 with 7 mantissa bits you have a step size of $2^{-7} * 2^7 = 2^0 = 1$. This means every integer between 0 and 255 is representable.

Q2) *CALL me maybe?* (5 pts)

For each of the following questions, determine what stage of **CALL** the following actions can happen. Select ONE per row.	**C**ompiler	**A**ssembler	**L**inker	**L**oader
a) The imm in `jal LABEL` gets replaced with its *final value.* Note that `LABEL` lives in a different file than the `jal LABEL` instruction.	○	○	●	○
If the label lived within the same file the assembler would be able to resolve it. Since it does not, the Linker must use the symbol tables of the different objects is linking to resolve it.				
b) Pseudoinstructions are removed	○	●	○	○
The assembler generates machine code which means we need to remove pseduo-instructions and replace them with their TAL equivalents to do this.				
c) Outputs assembly language code	●	○	○	○
Remember the compiler outputs assembly language code. The assembler outputs machine code.				
d) The symbol table is read by	○	○	●	○
The symbol table is generated by the assembler. The linker uses the symbol tables of the files it is linking together to resolve jal labels.				
e) Copies arguments passed to the program onto the stack	○	○	○	●
The loader takes the linked programs file which was generated by the linker and puts it into your computer's memory. The loader also sets up the different memory spaces including passing in arguments to the stack.				

<u>Q3) *I thought I needed to do a 2s but it was really just a sign-mag?!*</u> (20 pts = 7*2 + 6)

You recover an array of critical 32-bit data from a time capsule and find it was encoded in sign-magnitude! Write the **`ConvertTo2sArray`** function in C that converts all the data to 2s complement. You are told that **`0x00000000`** was never used to record any *actual data*, and is the array terminator (just as you do for strings). **`ConvertTo2s`** does the actual conversion for each number. Select ONE per letter; for **`<h>`** fill in the blank.

```
void ConvertTo2sArray( <a> A ) {
    while ( <b> ) {
        if ( <c> )
            ConvertTo2s( <d> );
        <e> ;
    }
}

void ConvertTo2s( <f> B ) {
    <g> = <h> ;
}
```

`<a>`	`int32_t`		[✓ `int32_t *`]	
`<b>`	`true`	`false`	`A`	[✓ `*A`]
`<c>`	`A < 0`	[✓ `*A < 0`]	`A`	
	`A > 0`	`*A > 0`	`*A`	
	`A <= 0`	[✓ `*A <= 0`]	`true`	
	`A >= 0`	`*A >= 0`	`false`	
`<d>`	`&A`	[✓ `A`]	`*A`	
`<e>`	[✓ `A = A + 1`]		`*A = *A + 1`	
`<f>`	`int32_t`		[✓ `int32_t *`]	
`<g>`	`&B`	`B`	[✓ `*B`]	

`<h>`	Some valid answers:
	`~(*B & 0x7FFFFFFF)+1;`
	`-(*B & 0x7FFFFFFF);`
	`-(*B + (1<<31));`
	`(*B - 1) ^ 0x7FFFFFFF;`
	`~((*B<<1)>>1)+1;`
	`(*B ^ 0x7FFFFFFF) + 1;`			

With Sign and magnitude (SM), the most significant bit (MSB, aka leftmost bit) represents sign, and the remaining bits represent magnitude. A positive SM number has a MSB of 0, negative SM has MSB of 1. With 2's complement (2's), positive numbers look identical to how they would with SM. However, to get negative 2's numbers, remember we take the positive number representation, flip each bit, and then add 1. A key observation is that positive SM and 2's numbers are represented identically (thus our condition in <c> isn't called for these positive numbers). Thus the chief job of the function `ConvertTo2s` is to take a negative SM number and convert it to a negative 2's number.

The most intuitive strategy for this is to take the negative SM number, set the MSB to 0 to give us the positive magnitude, and then convert this positive magnitude to 2's by flipping all bits and adding 1. *B gives us the number we want to work with, and 0x7FFFFFFF is a mask of a single 0 bit followed by thirty-one 1 bits (note MSB for this mask is 0). 'and'ing with such a mask will set the MSB to 0 (any bit 'and'ed with 0 equals 0), and leaves all other bits unchanged (any bit 'and'ed with 1 is unchanged). Thus (*B & 0x7FFFFFFF) will flip the MSB of the negative SM number, giving us just the positive magnitude. We can then use the formula ~(positive magnitude)+1 to convert the positive magnitude to a negative 2's number, giving us the full expression of ~(*B & 0x7FFFFFFF)+1. Check out this article for a more concrete example explaining this process: http://cseweb.ucsd.edu/classes/fa99/cse141l/ass1update.htm

Alternatively we can just apply the unary negation operator '-' on the positive magnitude to get the negative 2's number: -(*B & 0x7FFFFFFF), since C uses 2's to store signed ints, so the '-' operator invokes 2's rules of ~(...)+1 to negate.

Another method is to cause the negative SM number to overflow: (*B + (1 << 31)). The (1 << 31) just sets the mask MSB to 1, and 'add'ing this mask effectively 'add's 1 to the SM number's MSB of 1, overflowing the resulting MSB to 0, giving us the positive magnitude. We now can use the unary negation operator '-' to get the negative 2's number. This approach gives a final formula of -(*B + (1 << 31)).

Q4) *!noitseuq V-CSIR taerg a s'ereH* (20 pts = 12 + 4 + 4)

a) Below you will find the standard definition for a linked-list node. The recursive C code below reverses a linked list *with at least one node*. (For the initial call, the head of the list would be the first parameter, and the second parameter would be `NULL`) Your project partner translated this to nice RISC-V 32-bit code which honors the RISC-V calling conventions. Unfortunately, you spilled boba on it rendering it much of unreadable, and now you need to reconstruct it. Our solution used every line, but if you need more lines, just write them to the right of the line they're supposed to go after and put semicolons between them (like you would do in the C language). ***Don't waste time trying to understand the algorithm*** *for* `reverse`*, just compile it line-by-line.*

`struct node_struct {`
`    int32_t value;`
`    struct node_struct *next;`
`}`
`typedef struct node_struct Node;`
	`Node *reverse(Node *node, Node *prev) { // Requires: node != NULL`
	`    Node *second = node->next;`
	`    node->next  = prev;`
	`    if (second == NULL) { return node; }`
	`    return reverse(second, node);  }`

```
reverse:

    lw t0, 4(a0)              ### Node *second = node->next; (node->next addr = 4(a0) )

    sw a1, 4(a0)              ### node->next = prev (node->next addr = 4(a0), prev = a0)

    beq x0, t0, returnnode    ### if (second == NULL) { return node; }

    mv a1, a0                 ### 2ndarg = node

    mv a0, t0                 ### 1starg = second

    addi sp, sp, -4           ### prologue: move stack down

    sw ra, 0(sp)              ### prologue: store ra

    jal ra reverse            ### return reverse(second, node);  (recurse, ret val in a0)

    lw ra, 0(sp)              ### epilogue: restore ra

    addi sp, sp, 4            ### epilogue: restore the stack

returnnode:

    jr ra                     ### return
```

Now assume all blanks above contain a single instruction (no more, no less).

b) The address of **`reverse`** is **`0x12345678`**.

What is the hex value for the machine code of **`beq x0, t0, returnnode`**? **`0x`**<u>`02500063`</u>

c) The user adds a library and this time the address of **`reverse`** is **`0x76543210`**.

What is the hex value for the machine code of **`beq x0, t0, returnnode`**? **`0x`**<u>`02500063`</u>

<u>Part B</u>

`beq x0, t0, returnnode` moves the PC (program counter) forward by 8 instructions, each of which is 1 word or 4 bytes. This means the immediate = 8 * 4 = 32.

In binary, we represent 32 as 0b100000. However, since we know branch/jump immediates are always even numbers, we don't store bit 0.

imm[12]	imm[10:5]	rs2	rs1	func3	imm[4:1]	imm[11]	opcode
31	30 - 25	24 - 20	19 - 15	14 - 12	11 - 8	7	6 - 0
0	000001	00101	00000	000	0000	0	1100011

0000	0010	0101	0000	0000	0000	0110	0011
0	2	5	0	0	0	6	3

<u>Part C</u>

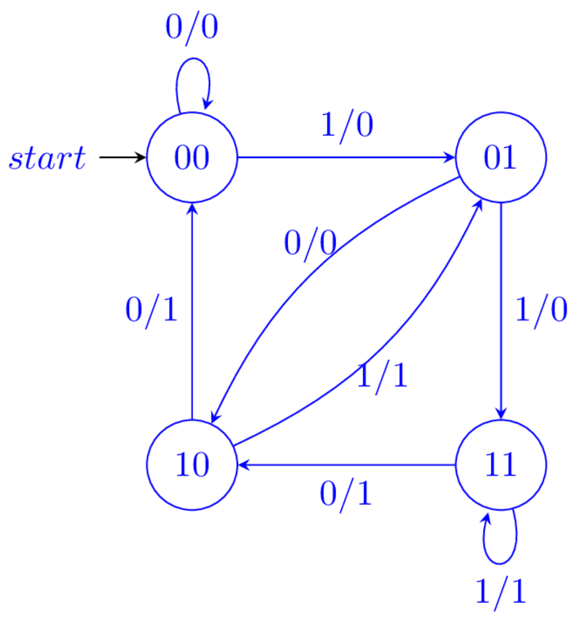
Since the number of instructions between the branch and its label does not change, the immediate value should not change regardless of the location since branches' immediates are relative to the current instruction.

Q5) *What kind of Algebra do ghosts like? Boooooolean Algebra!* (20 pts = 7 + 7 + 6)

Write an FSM that takes in an n-bit binary number (starting with the MSB, ending with the LSB) and performs a **logical right shift by 2** on the input. E.g., if our input is `0b01100`, then our FSM should output `0b00011`.

Input (MSB → LSB)	0	1	1	0	0
Output	0	0	0	1	1

a) Fill in the following FSM with the correct transitions and outputs. Format state changes as (input / output); we've done two for you. This is the **minimum** number of states; you may not add any more.



State 00 means 'two most recent inputs are 00', so an output FROM this state will always be 0.
State 01 means 'two most recent inputs are 01', so an output from this state will always be 0.
State 10 means 'two most recent inputs are 10', so an output from this state will always be 1.
State 11 means 'two most recent inputs are 11', so an output from this state will always be 1.

Note that the output from any state is always the second most recent input for that state. Based on the next input, we transition to the according state, and use this the aforementioned output rule to determine what the transition's output value is.

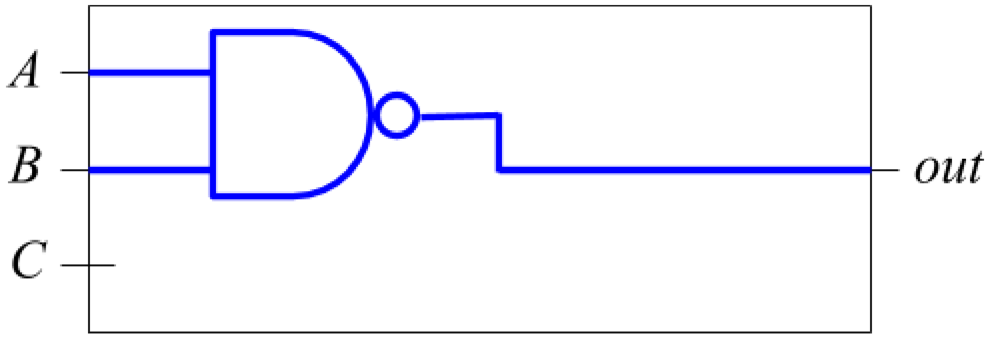
An aside: it is important to note that in general, the binary name of a state (in this case 00/01/10/11) has NO association with what that state represents - it is up to you, the FSM designer, to determine what each state represents. We could have equivalently named the four states something like 11/10/01/00 respectively and kept the same state meanings (state 11 means 'most recent are 00', ...), and still have a perfectly valid FSM. We provided state names of 00/01/10/11 as a helpful/useful hint towards what the meanings of the four states should be. To summarize, for any FSM with N states, we cleverly give each of these N states a meaning, and then can somewhat arbitrarily give these N states a binary name. Remember, we can use these binary names for states to convert the FSM into a truth-table/circuit where there's an input, register storing current state (by its binary name) and some combinational logic to determine truth-table/circuit output and update the register with the new state; this combinational logic is determined by the transitions in the FSM.

b) Draw the **FULLY SIMPLIFIED** (*fewest* number of primitive gates) circuit for the equation below. You may use the following primitive gates: AND, NAND, OR, NOR, XOR, XNOR, and NOT**.**

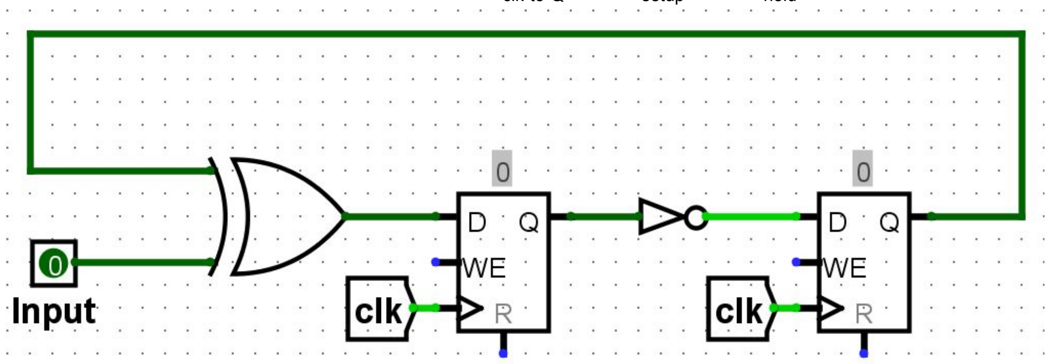
SHOW YOUR WORK FOR PART (b) BELOW

$$out = \overline{(A + \overline{B}B)} + \overline{(B + \overline{A})(A + BC)}$$

$out = \overline{(A)} + \overline{(B + \overline{A})(A + BC)}$	**Apply inverse law to** $\overline{B}B$
$out = \overline{(A)} + \overline{(BA + \overline{A}A + BBC + \overline{A}BC)}$	**Apply distributive law**
$out = \overline{(A)} + \overline{(BA + BC + \overline{A}BC)}$	**Apply inverse + idempotent laws**
$out = \overline{(A)} + \overline{(B(A + C + \overline{A}C))}$	**Apply distributive law**
$out = \overline{(A)} + \overline{(B)(A + C))}$	**Apply absorption law**
$out = \overline{(A)} + \overline{(B)} + \overline{(A + C)}$	**Apply DeMorgan's law**
$out = \overline{(A)} + \overline{(B)} + (\overline{A})(\overline{C})$	**Apply DeMorgan's law**
$out = \overline{(A)} + \overline{(B)}$	**Apply absorption law**
$out = \overline{(A)(B)}$	**Apply DeMorgan's law**



c) Assume `Input` comes from a register, and that there are no hold time violations. What's the fastest **frequency** you can run your clock for this circuit so that it executes correctly? Write your answer as a mathematical expression (you can also use `min()`, `max()`, `abs()`, and other simple operations if needed) using these variables: **X** = XOR delay, **N** = NOT delay, **C** = $t_{clk\text{-}to\text{-}Q}$, **S** = t_{setup}, **H** = t_{hold}



$$\frac{1}{max(C+X+S,\, C+N+S)} = \frac{1}{C+S+max(X\,,N)}$$

Explanation: we start by looking for the longest path in our circuit. There are three paths to consider: (1) Input ⇒ XOR ⇒ Register, (2) Register ⇒ XOR ⇒ Register, and (3) Register ⇒ NOT ⇒ Register. We know input comes from a register, so (1) and (2) are equivalent.

The time it will take (2) to execute properly is clk-to-Q + XOR delay + setup time, and the time it will take (3) to execute properly is clk-to-Q + NOT delay + setup time. We don't know whether N > X, so we take the maximum of both values.

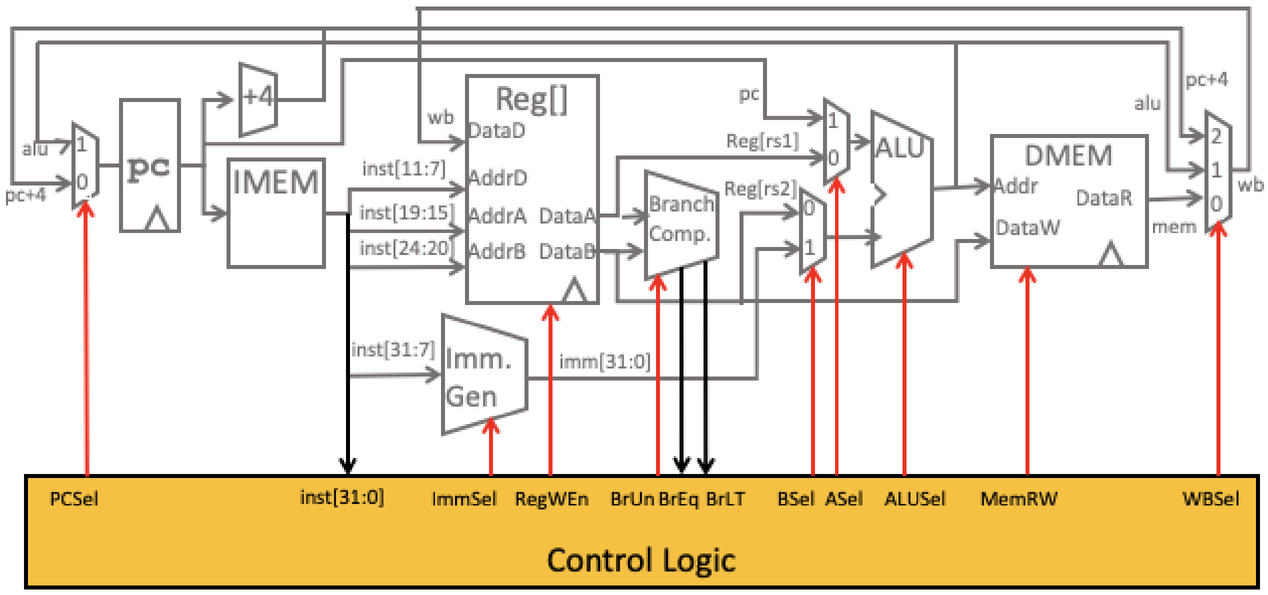
Because we're looking for **frequency**, we want to know how many times this happens **each second**, which is why we have an expression under 1.

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`

The only change necessary on top of adding new inputs to the `WBSel MUX` is to change the logic for `RegWEn` so that for `COMP` instructions, it is 1 only if `R[rs1] >= R[rs2]`.

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

- `comp` is not a branch or jump, so **`PCSel`** should be `PC + 4` -- after `comp`, we always want to execute the next instruction in sequence

- **`BrUn`** should be 0, because in the case if `R[rs1] > R[rs2]`, we want to do a signed comparison. If we were only looking at `R[rs1] == R[rs2]`, it wouldn't matter what BrUn was
- We'll be doing the comparison in the ALU, so **`ASel`** and **`BSel`** should be 0, so that both inputs are registers
- **`ALUSel`** should be other, since we're adding a new operation to the ALU, which will output 0, 1, or don't care (when `comp` doesn't write) for the input `R[rs1]` and `R[rs2]`
- **`MemRW`** should be read -- only operations that modify memory (e.g. store word) should have this set to 1 or Write
- **`WBSel`** should be ALU, since we're getting our modified output of 0 / 1 from the ALU Note: We are fine with outputting a 0 / 1 into the Regfile since we would disable the RegWEn if the comparison was less.

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)`

A `comp` instruction writes back to the RegFile in two cases: (1) if `R[rs1] > R[rs2]` and (2) if `R[rs1] == R[rs2]`. This is essentially equivalent to when `R[rs1]` is NOT less than `R[rs2]`. This is where we get the answer **`COMP*!BrLT`**.

For `comp` in particular, it's also the case that we always write when **`BrEq`** is true -- this means there's an additional (if less efficient) correct answer: **`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)]

Every instruction uses the IF, ID, and EX stages -- every instruction must be fetched and decoded, and every instruction uses the ALU for some purpose, either to perform an arithmetic/logical operation or to calculate an address.

Only load and store instructions use the MEM stage; comp does not modify memory or retrieve data from memory, so this stage isn't needed.

Some instructions (specifically store and branch instructions) do not modify the RegFile; they have no destination register (`rd`). `comp` does have a destination register, and should use the WB stage.