Your Name (first last) _____ SID _____

UC Berkeley CS61C
Fall 2019 Midterm

← Name of person on left (or aisle) _____

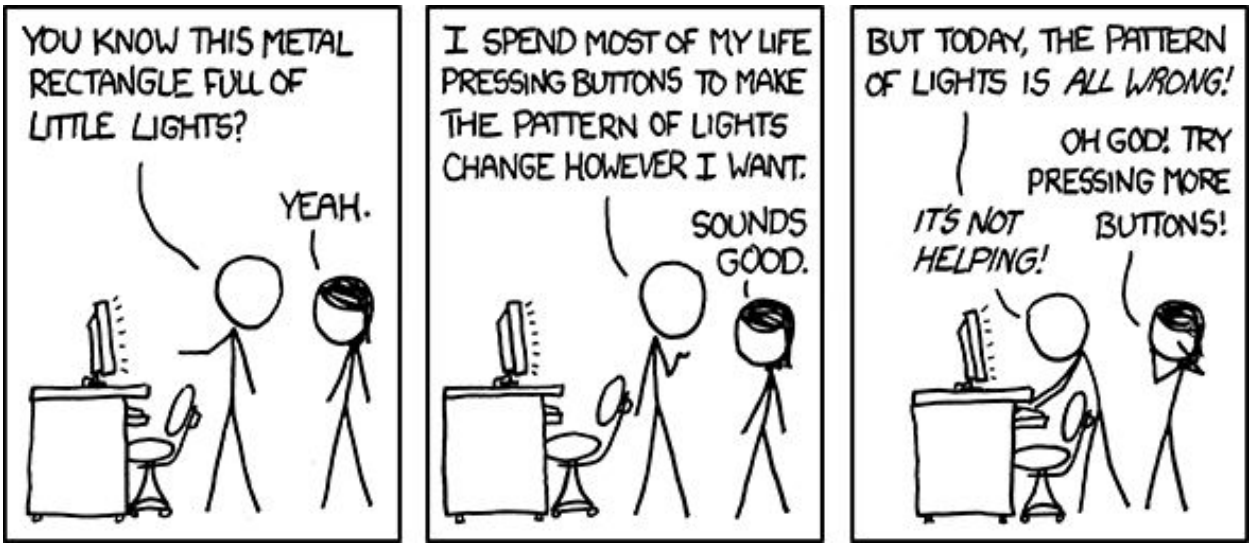
TA name _____

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.



This page has been intentionally left blank

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*?
0x		

SHOW YOUR WORK FOR PARTS (a,b,c) HERE

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.	◯	◯	◯	◯
b) Pseudoinstructions are removed	◯	◯	◯	◯
c) Outputs assembly language code	◯	◯	◯	◯
d) The symbol table is read by	◯	◯	◯	◯
e) Copies arguments passed to the program onto the stack	◯	◯	◯	◯

Q3) ***I thought I needed to do a 2s but it was really just a sign-mag?!*** **(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>				

SHOW YOUR WORK FOR PART (h) HERE

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, ____________________________ ### Node *second = node->next;

    ____________________________________ ### node->next = prev

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

    ____________________________________

    ____________________________________

    addi sp, sp, -4

    ____________________________________

    jal ra reverse                      ### return reverse(second, node);

    ____________________________________

    ____________________________________

returnnode:

    ____________________________________
```

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**_____

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**_____

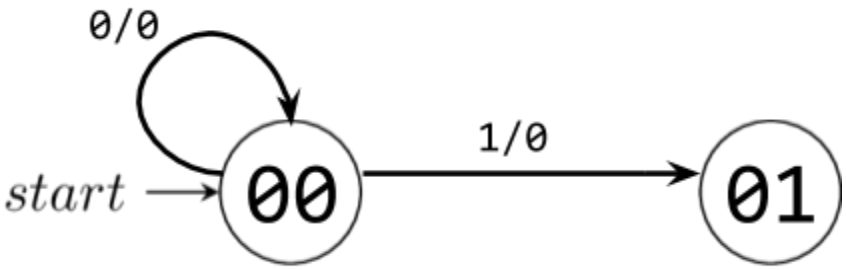
SHOW YOUR WORK FOR PART (b,c) HERE

Q5) ***What kind of Algebra do ghosts like? Booooooolean 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`

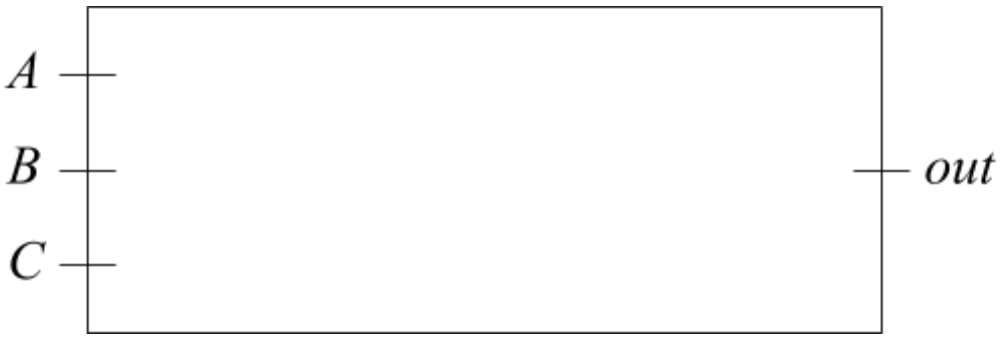
a) Fill in the following FSM with the correct transitions and outputs. Format state changes as (input / output); we've done two for you. This is the **minimum** number of states; you may not add any more.



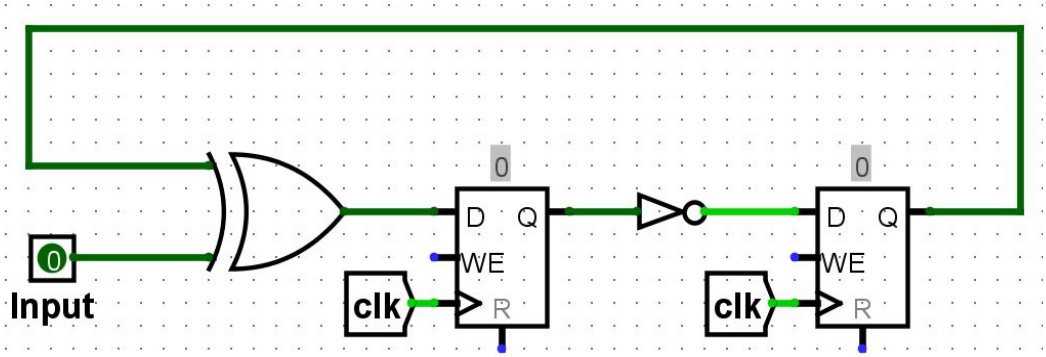
b) Draw the **FULLY SIMPLIFIED** (*fewest* number of primitive gates) circuit for the equation below.
You may use the following primitive gates: AND, NAND, OR, NOR, XOR, XNOR, and NOT.

SHOW YOUR WORK FOR PART (b) BELOW

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



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}

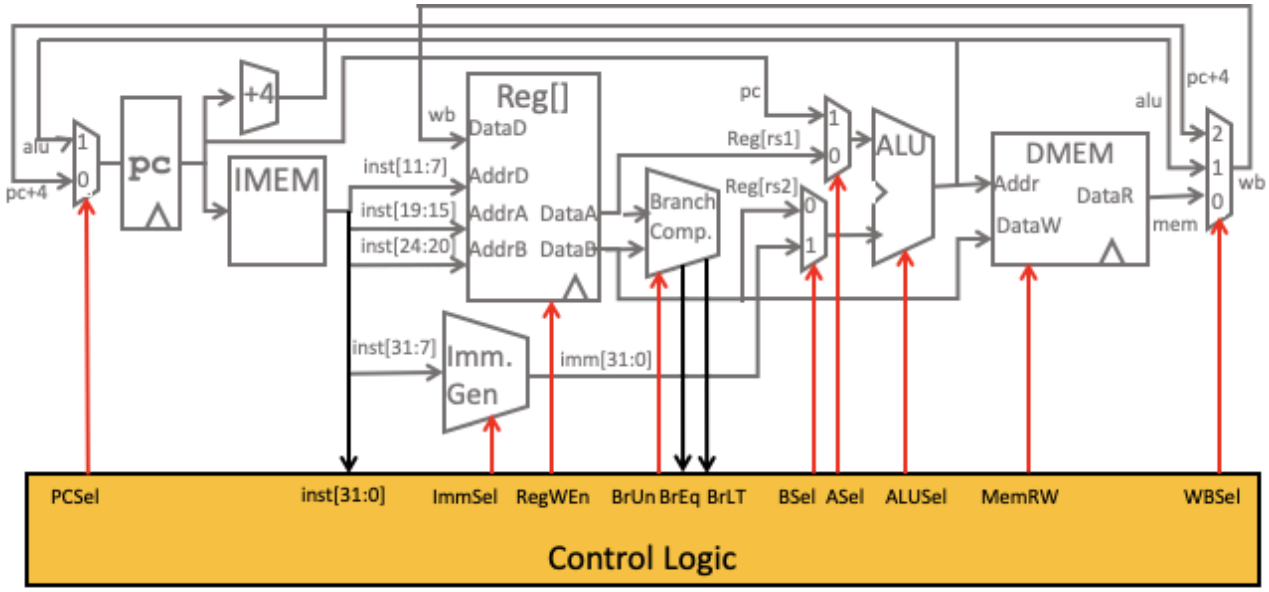


Q6) *comp a0, RISC-V, <3* **(18 pts = 5*1 + 7*1 + 4 + 2)**

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)