CS 240 Arch Assignment [103 points] **ID Number:**

About how many hours did you spend actively working on this assignment? _________

Q1 Universal Muxification of Gates [10 points] *Time spent on Q1:* ___________

1.1 NOT A (one 2:1 mux) **[1]**	**1.2. A AND B** (one 2:1 mux) **[1]**	**1.3 A NOR B** (two 2:1 muxes) **[2]**

1.4 A XOR B (two 2:1 muxes) **[Independent] [3]**	**1.5 A XOR B** (one 2:4 decoder and one 2:1 mux) **[Independent] [3]**

Q2 vALUe Judgement [28 points]
Draw circuits on next page, text answers (except **2.1b**) go here. *Time spent on Q2:* ___________

2.1 Condition Flags [5]
(draw circuits for **(a)** and give explanation for **(b)** on the next page)

2.2 Result of the ALU when *Invert A = 1, Negate B = 1,* and *Operation ID = 10*. [4]

(a) [2] Result =

(b) [2] Derivation of Result:

2.3 (a) [3 points] A, B with correct result
(multiple answers shown; you only needed one)

A	B	A - B	sign(A-B)	Is A < B?
positive	positive			
negative	negative			
different signs	different signs			

2.3. (b) [2 points] A, B with incorrect result

A	B	A - B	sign(A-B)	Is A < B?
positive				
negative				

2.3. (c) [1 point] Key reason(s) why 2.3(b) examples are incorrect.

2.3 (d) [3 points] Draw your circuit for the `Less-Than Flag` on the next page.

2.3. (e) [1 point] Control lines for `Less-Than Flag`:
Invert A = *Negate B =* *Operation =*

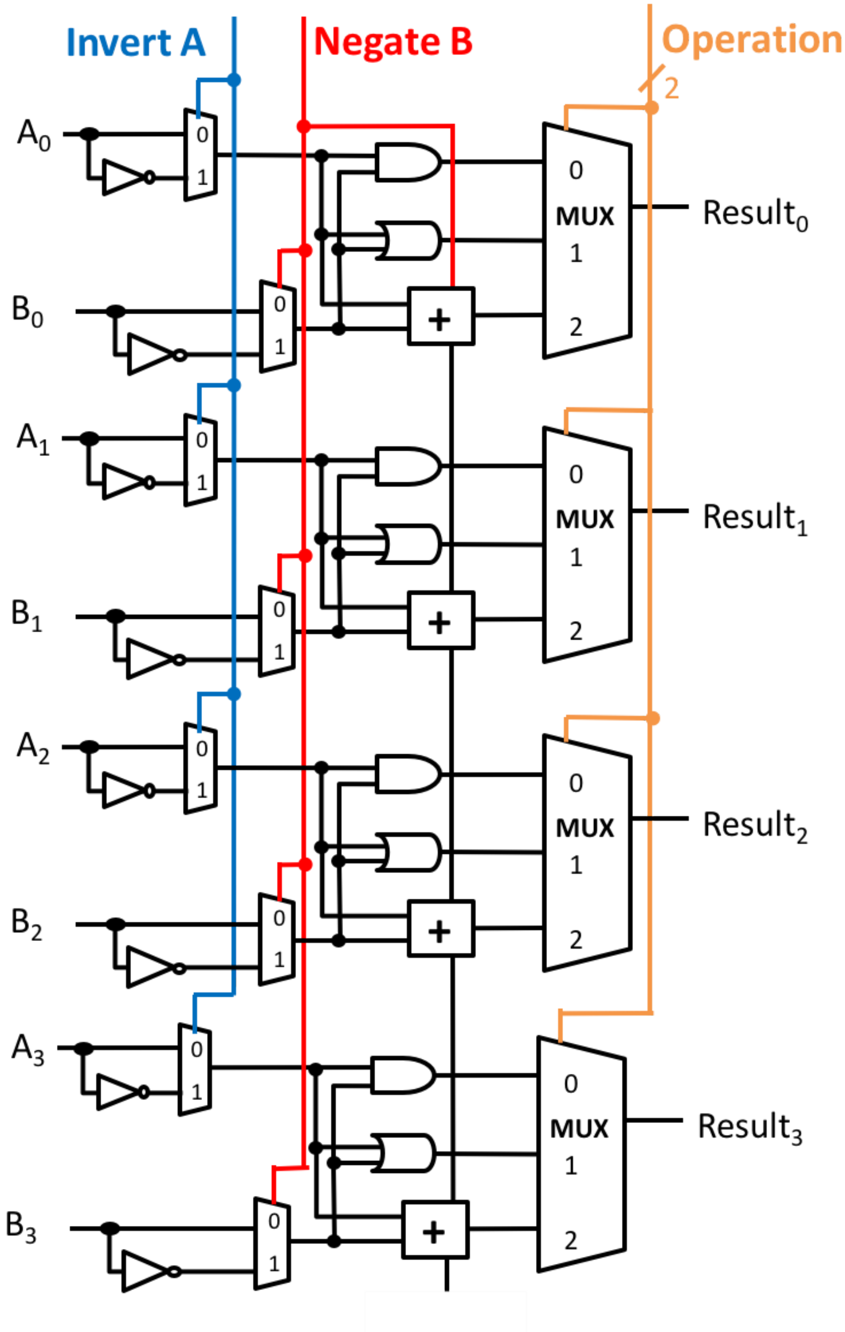
2.3. (f) [2 points] Explain why your `Less-Than Flag` circuit on the next page gives the correct result.

2.4. (a) [3 points] Draw your `Equals Flag` design on the next page.

2.4. (b) [1 point] Explain why your `Equals Flag` circuit correctly calculates A == B.

2.4. (c) [1 points] Control lines for the `Equals Flag`
Invert A = *Negate B =* *Operation =*

2.1(a) Condition Flag circuits;, **2.1(b)** explanation why Overflow Flag circuit is correct; **2.3(d)** Less-Than Flag circuit; **2.4(a)** Equals Flag circuit. **Label all outputs clearly.**

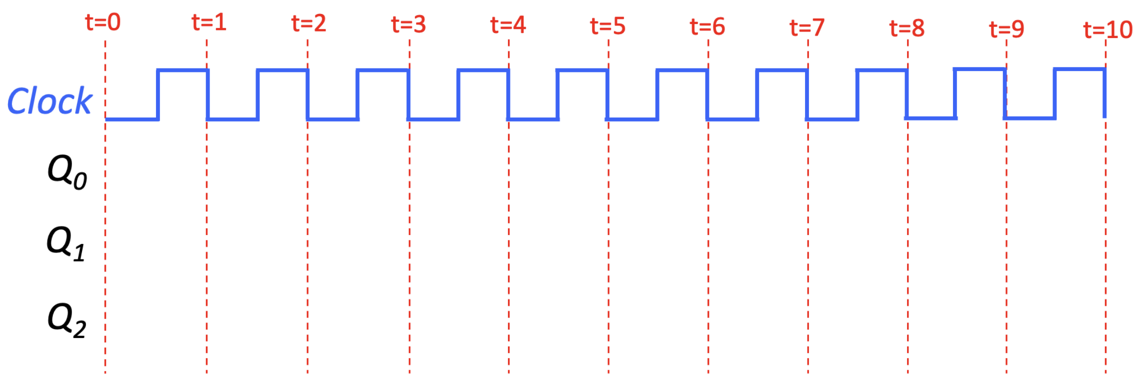


2.1(b) Explain why the Overflow Flag circuit is correct

Q3. Flop-Flip-Flopping [10 points]

Time spent on Q3: ____________

3.1 Waveform diagrams (Timing diagrams)



3.2. Cycles Completed	Q_2	Q_1	Q_0
0 (initial)	0	0	0
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

3.3. Explanation

Q4 Some Loopy Programs [22 points]

Time spent on Q4: ____________

4.1 [8 points] Execution Table for program P1

PC	*Instruction*	*State Changes*

4.2 [3 points] Final contents	**R2:**	**R3:**	**R4:**

4.3 [4 points] C statements equivalent to P1:

```
// Alice starts with these C statements for program P1
int R0 = 0;
int R1 = 1;
int R2 = R0+R1;
// Below, fill in the remaining C statements to complete program P1:
```

4.4 (a) Execute this program **P2**, assuming **R2** holds 5 and **R3** holds 4. Below, indicate the final register values when the code reaches **HALT**. (Do *not* show the step-by-step execution)

```
0x0:  AND R2, R2, R4
0x2:  AND R3, R3, R5
0x4:  BEQ R5, R0, 3
0x6:  SUB R5, R1, R5
0x8:  ADD R4, R4, R4
0xA:  JMP 2
0xC:  HALT # Stops execution.
```

Final register contents after executing P2 [3 points]	R2:	R3:	R4:	R5:)

4.4(b) [2 points] C line for P2

Single line of C code equivalent to the HW ISA code for **P2**, assuming variables R2 and R3 can hold any integer values. Use only basic C operations (no conditionals, loops, or function calls).

```
R4 =
```

4.3(b) [2 points] Explanation why C line in 4.4(b) calculates the same result as program P2

Q5 Taking Control [9 points] *Time spent on Q5:* ___________

Control Unit Truth Table

Instruction Name	Opcode$_{[3:0]}$ (4 bits)	Reg Write (1 bit)	ALU Op$_{[3:0]}$ (4 bits)	Mem Store (1 bit)	Mem Load (1 bit)	Branch (1 bit)	Jump (1 bit) **7.2 [1]**
LW	`0000`	`1`	`0010`	`0`	`1`	`0`	
SW							
ADD							
SUB							
AND							
OR							
BEQ							
NAND **6.2(b) [3]**							
JMP **7.3 [3]**							

Q6 Instruction Not Missing [12 points] *Time spent on Q6:* ___________

6.1 [4 points]

6.1(a) [1 point] Give a definition of ~X in terms of X and signed two's complement arithmetic:

~X =

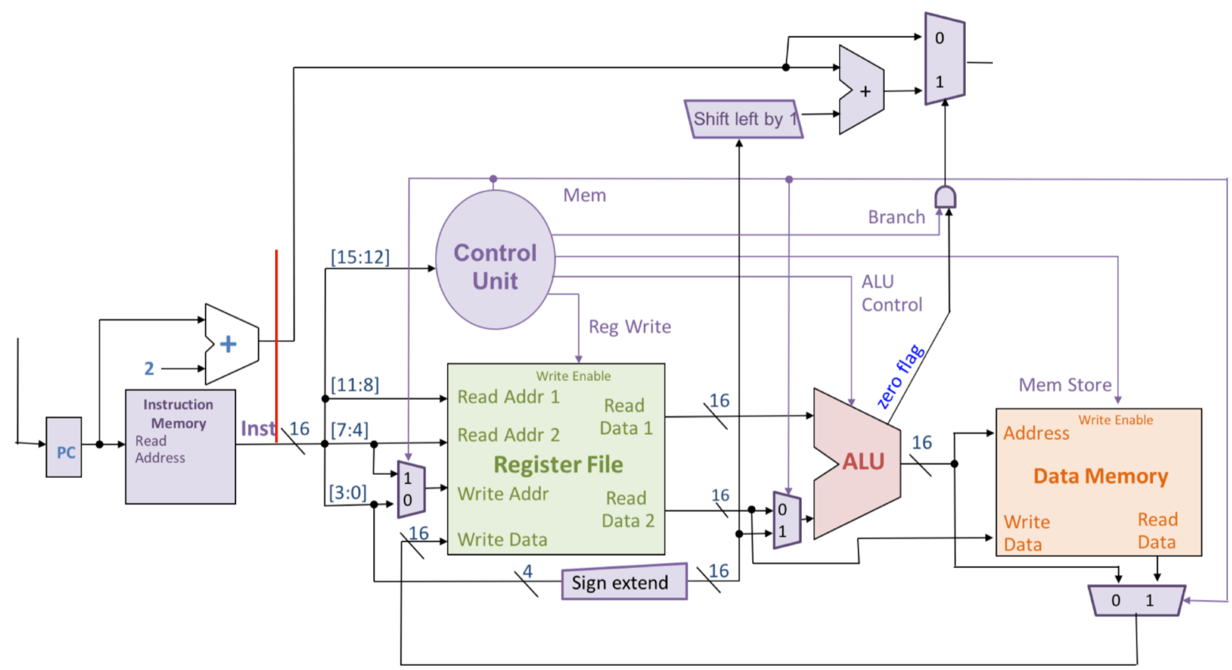
6.1(b) [3 points] Based on the previous subpart, the instruction **`NOT Rs,Rd`** can be emulated by running the following instructions instead:

------------------- 16-bit encoding ------------------

Assembly	Meaning	Opcode [15:12]	Rs [11:8]	Rt [7:4]	Rd [3:0]
6.2(a) [3 points] **`NAND Rs,Rt,Rd`**	***R[d] ← ~(Rs & Rt)***				
6.3 [2 points] **`NOT Rs,Rd`**	***R[d] ← ~Rs***				

Q7 Jumping into the Unknown [12 points] *Time spent on Q7:* ___________

7.1(a) [8 points]. Below, add a `Jump` output wire from the Control Unit and modify logic to use it to implement `JMP` instruction. Note: if you use the new red write split off from `Inst`, be sure to label which range ([?, ?]) of bits you use.



7.2 [1 point] *For this part, fill out the Jump column in the Control Unit Truth Table in Q5.*

7.3 [3 points] *For this part, fill out the JMP row in the Control Unit Truth Table in Q5.* ***When the bits in a cell don't matter (they can be anything), you must explicitly write this!***