Department of Electrical and Computer Engineering
The University of Texas at Austin

ECE 306 Fall 2023
Instructor: Yale N. Patt
TAs: Chester Cai, Jaeyoung Park, Sophia Jiang, Anna Guo, Asher Nederveld, Edgar Turcotte, Nadia Houston, Varun Arumugam, Ali Mansoorshahi
Exam 1
October 11, 2023

Name: _____Solution_____

Problem 1 (20 points): _____

Problem 2 (15 points): _____

Problem 3 (15 points): _____

Problem 4 (25 points): _____

Problem 5 (25 points): _____

Total (100 points): _____

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Note: Please be sure your name is recorded on each sheet of the exam.

Please read the following sentence, and if you agree, sign where requested:
I have not given nor received any unauthorized help on this exam.

Signature: _____

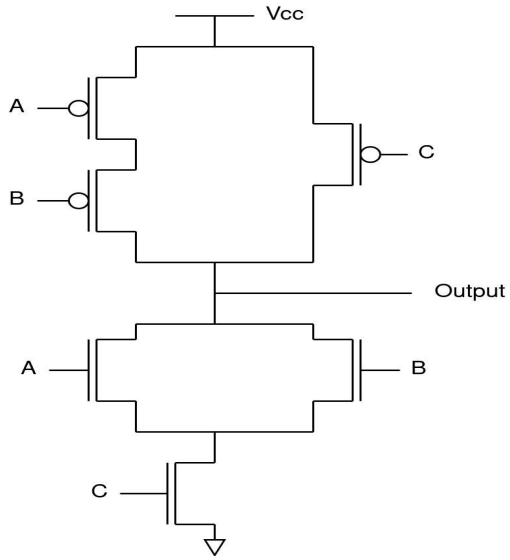
GOOD LUCK!

Name: _____

Question 1 (20 points): Answer the following questions.
Note: For each of the four answers below, if you leave the box empty, you will receive one point.
Part a (5 points): The TRAP instruction provides a mechanism for a user program to ask the Operating System to do something on behalf of the user program. How does the Operating System know what the user program wants it to do (in 15 words or fewer)?

Trap vector

Part b (5 points): What logic function is performed by the following transistor diagram? Complete the truth table for that logic function.



A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Part c (5 points): You know from constructing programming logic arrays that with a sufficient number of AND, OR, and NOT gates, one can implement every logic function, regardless how many input variables there are. If you only have AND and NOT gates, can you still implement every logic function, regardless how many input variables there are. (**Circle one:** YES/NO) Explain in 15 words or fewer:

Yes. You can use AND and NOT to build OR gates

Part d (5 points): The computer has just executed the instruction A:

0101 010 010 1 00000

and the computer is now at state 18, about to start executing the next instruction B, which is:

0000 011 000011111

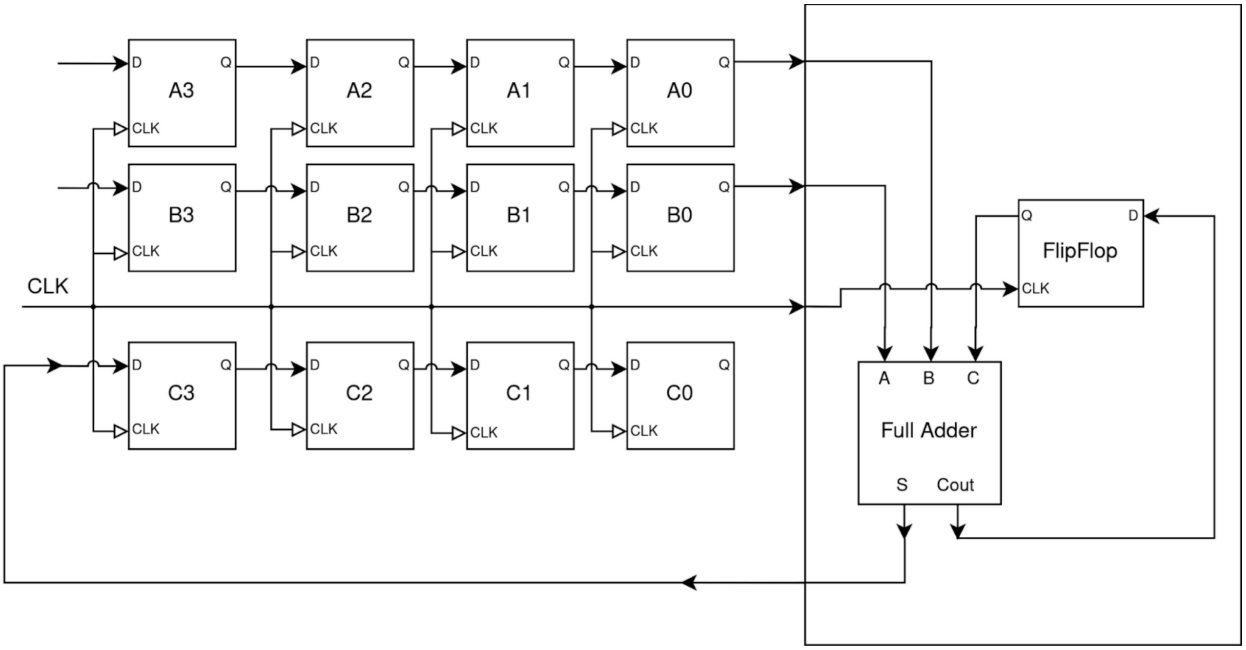
List the states that the microarchitecture will execute, starting with state 18 when processing instruction B.

18, 28, 30, 32, 0, 22
Note that the previous instruction sets the Z bit in the condition code, so it's a taken branch

Name: _____

Question 2 (15 points): We wish to add two unsigned integers with the logic blocks shown below. You can assume that A and B are each stored in four flip flops A3-A0 and B3-B0 (least significant bit in A0/B0) and that the sum is to be stored in C3-C0 after four clock cycles.

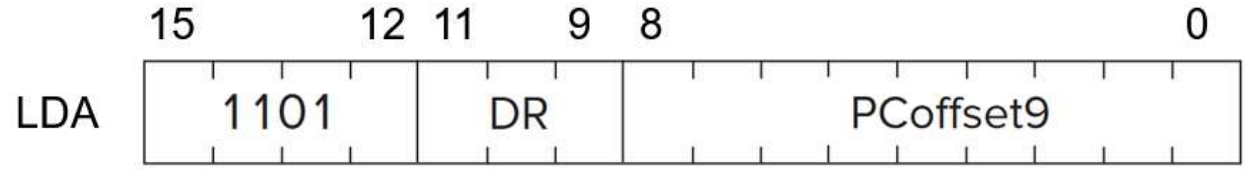
Your job: Add the additional logic and connections to the figure to accomplish this. Some inputs and outputs have already been drawn for you. You may add more, or leave some unused if you wish. Note that you only have one Full Adder, and can not add any more. You are allowed to add Constant Values, NANDs, NORs, D-Latches, and FlipFlops. You are not allowed to construct additional Full Adders with NANDs and NORs. You can assume that any added flip flops or latches start with value "0".



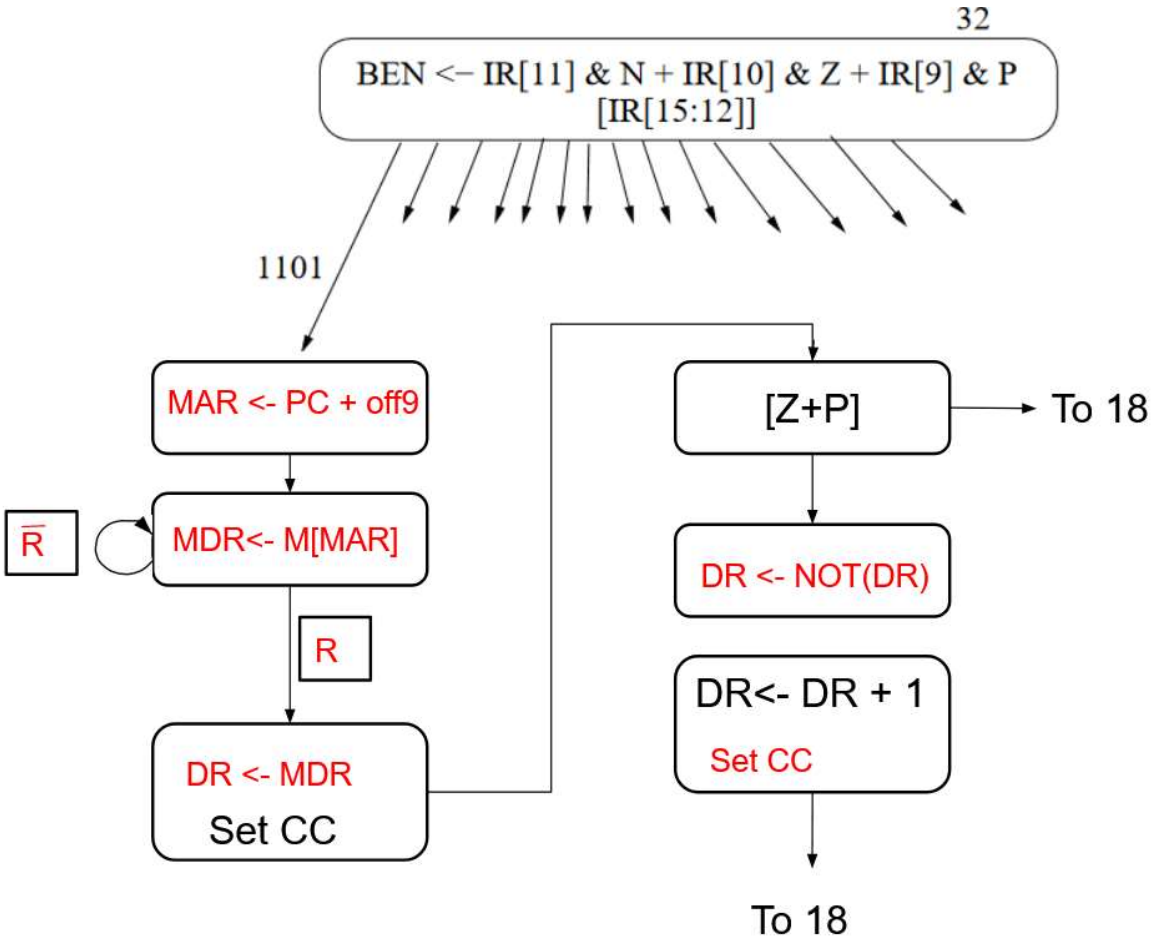
Name: _____

Problem 3. (15 points) We wish to use the unused opcode to add a new instruction Load absolute (LDA) to the LC-3 instruction set. LDA will take a 16-bit 2's complement signed integer stored in memory, form its absolute value, load the absolute value into the register specified by bits [11:9] of the instruction, and set the condition codes based on the absolute value. The addressing mode for determining the address of the memory location where the 2's complement integer is stored is PC + offset9. This is the same addressing mode as used for LD.

The new instruction has the following format:



Your Job: Fill out the state machine below to properly implement the LDA instruction. Please note that some states have already been completely or partially filled out.



Name: _____

Question 4 (25 points): An Aggie, browsing the bookstore, stumbled across Patt's 306 textbook, and said, "Hey, I guess anyone can program with this!," and proceeded to generate the following program that does nothing useful:

Assembly	**Machine Code**
.ORIG x_____	x_____
AND R1, R1, #0	x5260
LD R2,A	x2402
BRz B	x0402
ADD R2,R2,R2	x1482
A ST R2,C	x3401
B TRAP x25	xF025
C ADD R3,R1.R2	x1642
.END	- -

We decided to analyze his program. The table in part a shows a number of clock cycles, **and for each clock cycle, the value(s) written at the END of that clock cycle.** Some of the values are already in the table. A dash in an entry means the value is not being updated in that particular cycle. The first instruction in the program starts in state 18 in cycle 1.

Also note that the LC-3 we are running this program on was built by an Aggie, so the number of clock cycles required to perform a memory access may not be five.

Name: _____

Part a (15 points): Fill out all missing entries in the table below, including the cycle # on the last row. Provide the value **ONLY** if it was updated in **THAT CYCLE**, use a "-" to indicate the value has not been modified in **THAT CYCLE**.

Cycle #	PC	MAR	MDR	IR	BEN	NZP	R0	R1	R2	R3
7	-	-	-	-	-	010	-	x0000	-	-
8	x4429	x4428	-	-	-	-	-	-	-	-
14	-	x442B	-	-	-	-	-	-	-	-
19	x442A	x4429	-	-	-	-	-	-	-	-
24	-	-	-	-	0	-	-	-	-	-
38	-	-	-	-	0	-	-	-	-	-
40	-	-	x6802	-	-	-	-	-	-	-
48	-	-	-	xF025	-	-	-	-	-	-

Part b (4 points): Recall that in our lectures this semester, Dr. Patt has said that memory accesses take 5 clock cycles each. In our measurements, we discovered that the LC-3 we are using took a different number of clock cycles to access memory. How many?

3 cycles. The first AND finished on cycle 7, which means you spent 1 cycle in state 18, 3 cycles in state 28, 1 cycle in state 30 and 1 cycle in state 32, and 1 cycle in state 1

Part c (4 points): You will note that .ORIG does not have a value after it. What value should go there?

x4427. On cycle 8, the second instruction puts x4428 in MAR, which is the PC for the second instruction. So the PC for the first instruction must be x4427

Part d (2 points): In executing the Aggie's program, what value will the LC-3 store in **R3**?

R3 is not being written to during the execution of the program. It halted before the last instruction

Name: _____

Question 5 (25 points): R0 and R1 contain 16-bit bit vectors. The program in the next page determines if rotating R1 by *n* bits to the left produces the same bit vector that is in R0. If yes, the program stores the value *n* in M[x3070]. If not, the program stores -1 to M[x3070].

Rotating left a bit vector one bit consists of left shifting the bit vector one bit, and then loading into bit[0] the bit that was shifted out of bit[15].

For example, rotating left **1**1110000111100000 one bit produces 1110000111100001.

Rotating left *n* bits is simply rotating left one bit, but doing it *n* times.

For example, rotating left **111**1000011110000 3 bits produces 1000011110000**111**.

An example of the program's execution is shown:

R0 = 1000 0000 0000 0110
R1 = 1101 0000 0000 0000
→ The program will store 3 to M[x3070]

Part a (5 points)

For each iteration, the program compares R0 with R1 and rotates R1 one bit to the left if those two registers don't match. What is the minimum number of 1-bit left rotations the program has to make to guarantee that the contents of R0 and R1 will never be the same? Add a brief explanation why.

15. Rotating 15 times checks every possibility. 16 rotations is equal to the original R1.

(16 is also accepted because the program below performs the comparison and rotation process a minimum of 16 times.)

Name: _____

Part b (20 points)

Your job: Complete the program below by supplying the missing instructions so it stores *n* in location M[x3070] if rotating left R1 *n* bits produces the bit vector in R0, and store -1 if it is not possible to produce the bit vector of R0 by rotating R1.

Note: The Comments section of the table is strictly for your use. It will not be considered in the grading.

Address	Value	Comments
x3000	0101 010 010 1 00000	; AND R2 R2 #0
x3001	1001 000 000 1 11111	; NOT R0 R0
x3002	0001 000 000 1 00001	; ADD R0 R0 #1
x3003	0001 011 010 1 10001	; ADD R3 R2 #-15
x3004	0000 001 000001010	; BRp x300F
x3005	0001 011 001 0 00000	; ADD R3 R1 R0
x3006	0000 010 000001010	; BRz to x3011
x3007	0001 010 010 1 00001	; ADD R2 R2 #1
x3008	0001 001 001 1 00000	; ADD R1 R1 #0
x3009	0000 100 000000010	; BRn to x300C
x300A	0001 001 001 0 00001	; ADD R1 R1 R1
x300B	0000 111 111110111	; BR x3003
x300C	0001 001 001 0 00001	; ADD R1 R1 R1
x300D	0001 001 001 1 00001	; ADD R1 R1 #1
x300E	0000 111 111110100	; BR to x3003
x300F	0101 010 010 1 00000	; AND r2 r2 #0
x3010	0001 010 010 1 11111	; ADD R2 R2 #-1
x3011	0011 010 001011110	; ST R2 to x3070
x3012	1111 0000 00100101	; HALT

Name: _____

Step 1. x3000 clears R2. This implies that R2 is going to be used.

Step 2. x3001~x3002 negates what is inside of R0. During the class, we went over how to know if two registers match. We negated one register and added to the one we are comparing it to. So these two lines imply that negated R0 will be added to R1 in the future to check whether two bit vectors are the same or not.

Step 3. x3003 is the hardest part. Students need to skip this part and move on to get more information.

Step 4. x3004 tells a lot about x3003. If you follow where the branch leads to, you will notice that x300F and x3010 clears what is in r2 and stores -1 to r2. This implies 3 things.

1. x3003 somehow checks that the two bit vectors are never going to match
2. Since we are clearing R2, R2 will be used to count the number of iterations.
3. Also, since we just made R2 to -1 and x3011 is the next instruction just before HALT, x3011 should be a store instruction that stores R2 to M[x3070]

So we figured out what x3011 is.

Step 5. x3005 is clearly checking whether (-R0) + R1 is 0 or not. So x3006 needs to use this result to branch to somewhere. Since when the result is zero we have a match, we should branch to x3011 which we just filled out to store *n* and HALT the program.

Step 6. We can see that x3007 uses R2 as a counter(*n*).

Step 7. x3008 checks whether the R1 is negative or not. Why? From the description above we stated that what was inside bit[15] goes to bit[0]. What can cause the problem? We could move on for now.

Step 8. x300A and x300B are for positive R1. All students should be familiar with x300A. We learned this through Lab 1 part b that it left shifts the register by one bit. x300B goes back to x3003.

Step 9. Still couldn't figure out x3003 so let's go to x300C which is the destination from a branch when R1 is negative. Looking at x300C it still does the left shift. However, the x300D and x300E are left blank. Since x300F is clearing R2 x300E should be a branch back to x3003 just like what we have noticed in Step 8.

Step 10. So now we have to think about what is so special about a register being negative. MSB==1 in 2's complement. Correct reasoning would be thinking of the problem that arises when we are left shifting a register when MSB==1 and then realizing that we should do an additional step when the register is "negative".

Step 11. Since LSB needs to be a 1, we just add 1 at x300D.

Step 12. Now all we are left with is x3003. If you rotate your bit vector 16 times, it is the same as the initial bit vector so it is meaningless to go further. Thus halt if (R2-15)>0. However, since the program is comparing *n* before comparing R0 and R1, it does rotate the bit vector 16 times.

Name: _____

This page is left blank intentionally. Feel free to use it for scratch work.
You may tear the page off if you wish.
Nothing on this page will be considered for grading.