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

ECE 306 Fall 2023
Instructor: Yale N. Patt
TAs: Chester Cai, Sophia Jiang, Ali Mansoorshahi, Jaeyoung Park, Anna Guo, Asher Nederveld, Edgar Turcotte, Nadia Houston, Varun Arumugam,
Final Exam
Dec. 8th, 2023

Name and EID: Solution

Part A:	**Part B:**
Problem 1 (10 points): _______	Problem 6 (20 points): _______
Problem 2 (10 points): _______	Problem 7 (20 points): _______
Problem 3 (10 points): _______	Problem 8 (20 points): _______
Problem 4 (10 points): _______	Problem 9 (20 points): _______
Problem 5 (10 points): _______	
Part A Total: _______________	Part B Total: _______________

Total (130 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.

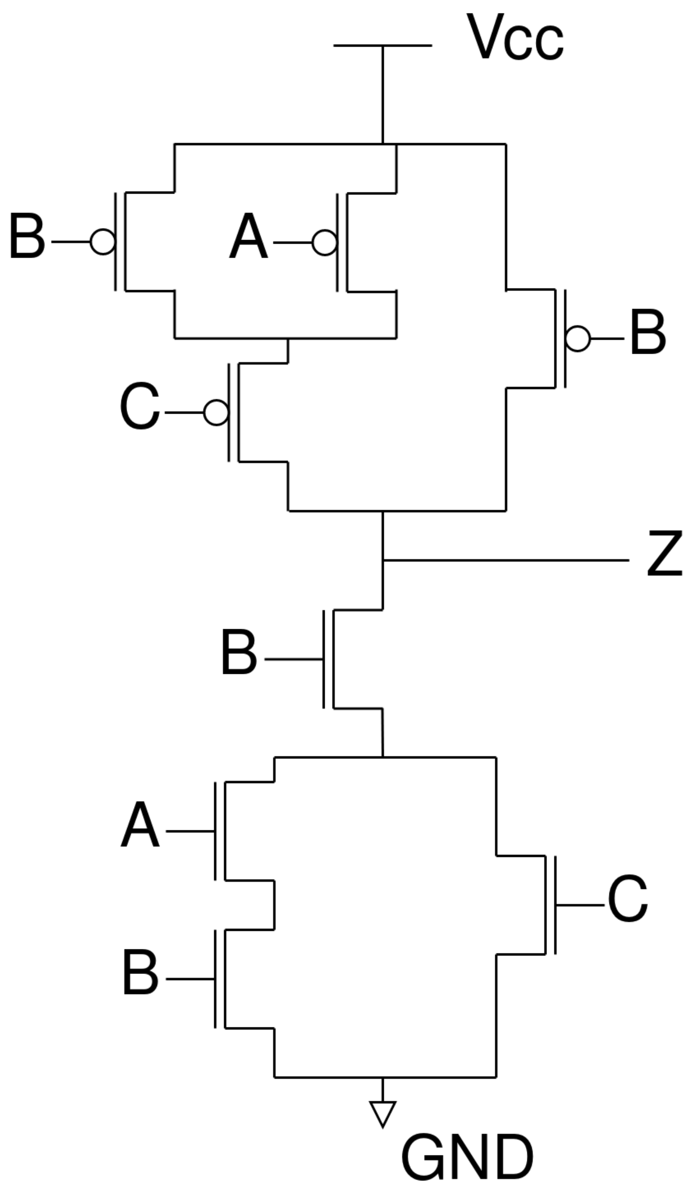
Signature: ____

GOOD LUCK!
(Have a good winter break)

Name: ____

Question 1 (10 Points): Shown below is a transistor circuit with three inputs (A, B, C) and one output (Z). Also shown is the truth table for this circuit. The outputs of the truth table are not shown.

Your job: Complete the truth table. Every input combination produces an output of either 0 or 1.



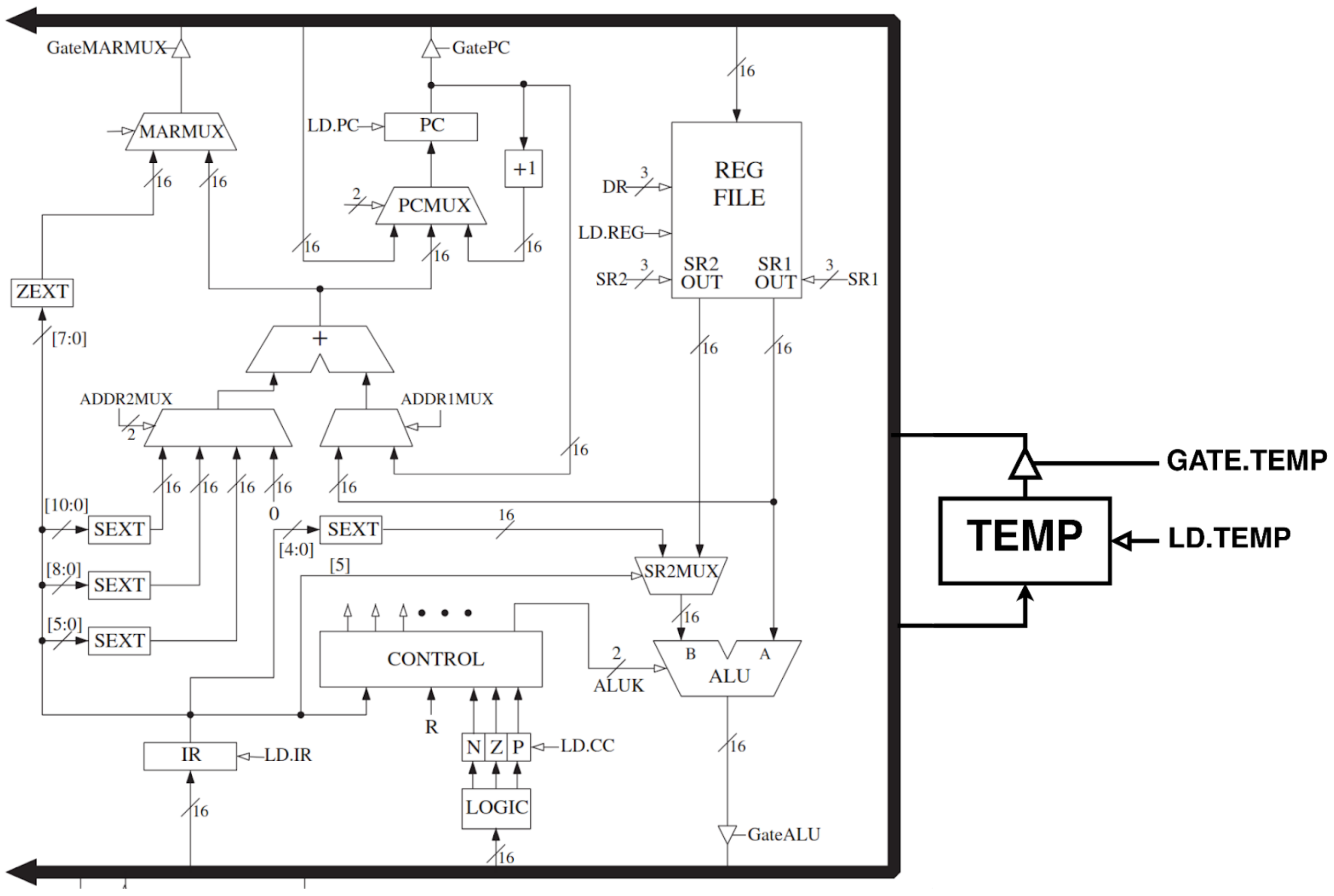
A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Name: ____________________________________

Question 2 (10 Points): We wish to implement a new instruction that swaps 2 values, one in a register and one in memory. The instruction's format is shown below. SR1 contains a value to be swapped and SR2 contains an address to a location in memory that has the other value to be swapped.

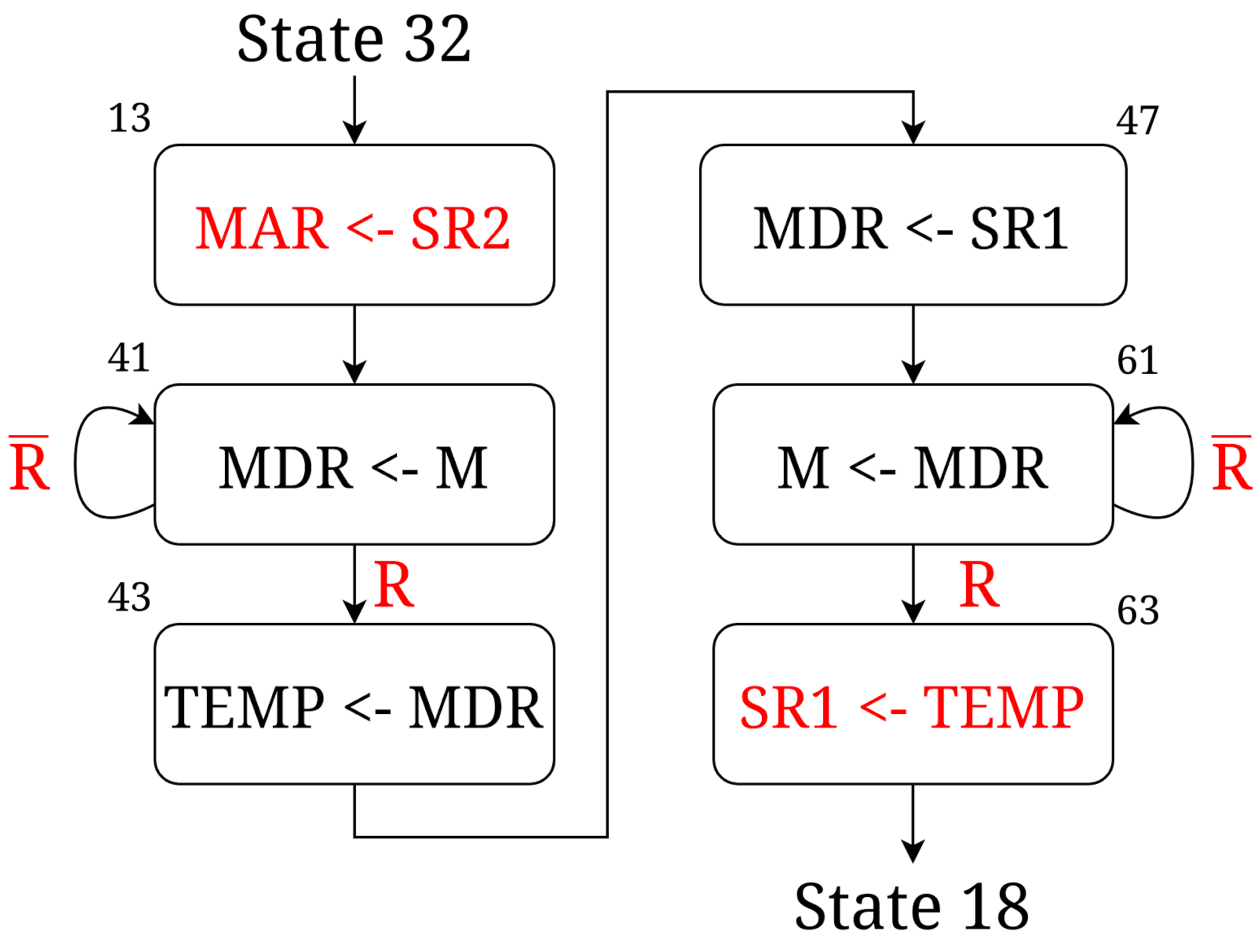
15 12	11 9	8 6	5 0
1 1 0 1	S R 1	S R 2	0 0 0 0 0 0

The changes to the datapath are bolded in the diagram below.



Name: ________________________________

Part A: Complete the state machine for the swap instruction below.



Part B: Fill out the control signals for the states listed in the table below:

State	J	GATE.MDR	GATE.TEMP	LD.TEMP	LD.MDR	LD.MAR
41	41	0	0	0	1	0
43	47	1	0	1	1	0

Name: ________________________________

Question 3 (10 Points): We want to design the state machine for an automatic bird feeder. It has two types of feed: millets and sunflower seeds. The system takes 1 bit of input, and outputs 2 bits. The input and output are described below.

Sensor Readings (inputs)

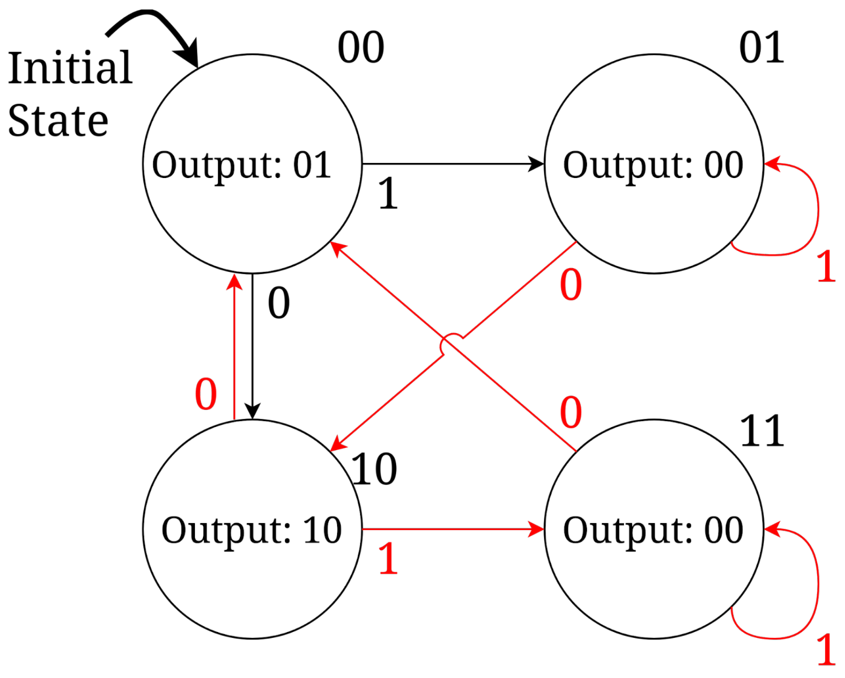
Input	Description
0	Empty
1	Not Empty

Bird Feeder Actions (outputs)

Out[1]	Out[0]	Description
0	0	Do nothing
0	1	Fill with millets
1	0	Fill with sunflower seeds

Birds need a varied diet so our bird feeder alternates between sunflower seeds and millet. **Every hour** the feeder will read the sensor. If it is empty, it will fill with the opposite food choice. If it is not empty, it will do nothing. The feeder starts empty, and should be filled with millet.

Part A: Complete the finite state machine below that allows the bird feeder to operate as described. The 2-bit number on the top-right corner of each state is the state number for that state.

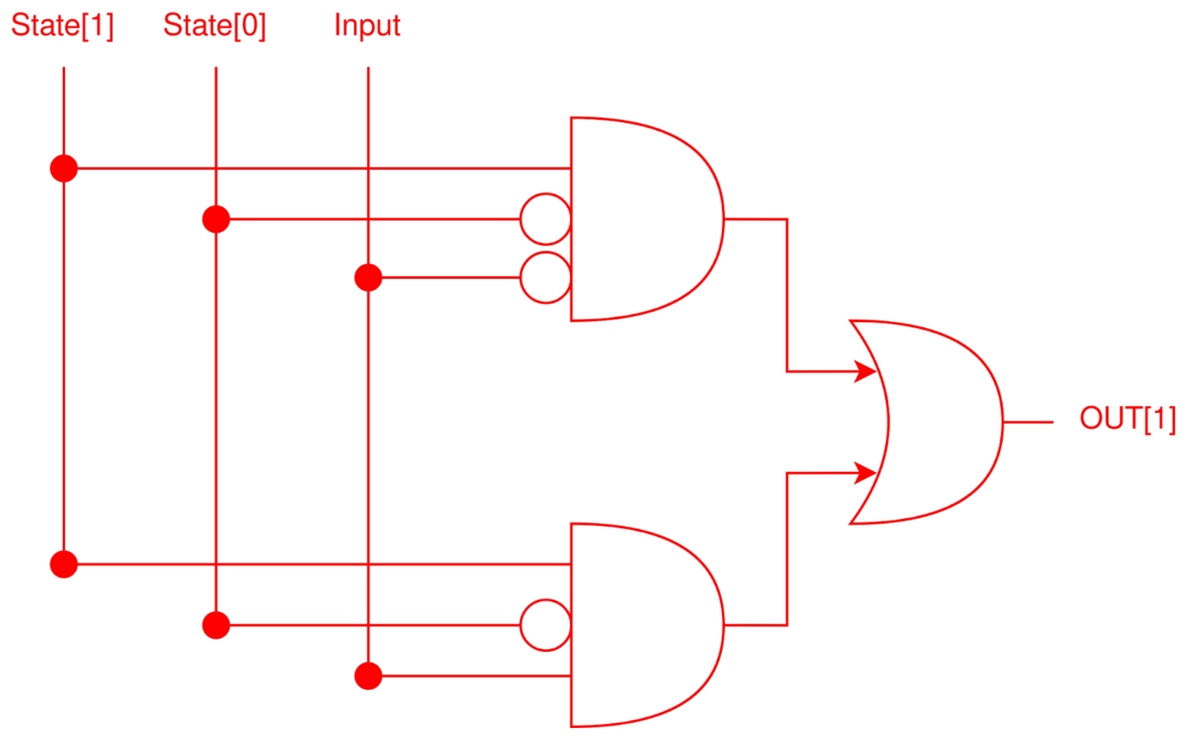


Name: ______________________________________

Part B: Complete the truth table corresponding to the state machine from part A.

State[1]	State[0]	Input	Out[1]	Out[0]	Next State[1]	Next State[0]
0	0	0	0	1	1	0
0	0	1	0	1	0	1
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	0	0	1	0	0	0
1	0	1	1	0	1	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

Part C: Draw the logic gate implementation for Out[1]. Use only AND, OR, and NOT gates.



Name: ____

Question 4 (10 Points): What value is stored in the location labeled RESULT when the following program terminates?

X0306

```
           .ORIG x3000
           AND   R1,   R1,   #0
           LEA   R0,   LABEL1
LOOP       LDR   R0,   R0,   #0
           BRz   DONE
           LDR   R2,   R0,   #1
           ADD   R1,   R1,   R2
           BRnzp LOOP
DONE       ST    R1,   RESULT
           HALT
RESULT     .BLKW  #1
LABEL1     .FILL x6000
           .END

           .ORIG       x6000
           .FILL       x5010
           .FILL       x0300
           .END
           .ORIG       x5010
           .FILL       x4020
           .FILL       xFFFF
           .END
           .ORIG       x4020
           .FILL       x0000
           .FILL       x0007
           .END
```

What kind of data structure does this program operate on?

Linked List

Name: ____________________________________

Question 5 (10 Points): A student wrote a program to multiply two positive numbers X and Y, and store the product in memory location x4002. M[x4000] contains the value X. M[x4001] contains the value Y. Unfortunately she spends too much time watching football, so there is a bug in her code.

```
           .ORIG x3000
           LDI   R1,   X
           LDI   R2,   Y
           AND   R0,   R0,   #0
LOOP       ADD   R0,   R0,   R2
           ADD   R1,   R1,   #-1
           BRz   LOOP
           STI   R0,   RESULT
           HALT
X          .FILL x4000
Y          .FILL x4001
RESULT     .FILL x4002
           .END
```

Your job: Circle the instruction causing the bug, and replace it in the box below with the correct instruction.

BRp LOOP

BRnp was also an accepted answer because the question states that the inputs were positive. However, BRzp is not a valid answer because the [z] would cause an extra iteration of the loop, resulting in the wrong answer ((R1+1)*R2 instead of R1*R2)). For instance, X = 2, Y = 3 would result in 9 instead of 6.

Name: ______________________________________

Question 6 (20 Points): You are given a program that takes 2 inputs from memory and stores the result of execution in an unknown memory location. During the execution of the program, we took 7 snapshots of the LC-3 microarchitecture. The first snapshot was taken during the first clock cycle of the program execution. Subsequent snapshots are shown in the table below in the order in which they are taken. Each snapshot contains the state number, the value on the bus during the clock cycle, and the control signals necessary to execute the state.

Part A: Fill in the blanks in the code and table. Use X if a control signal value doesn't matter.

```
           .ORIG x3000
           LD    R0,   OUTPUT
           LDR   R1,   R0    #16   ;INPUT 1
           BRz   DONE
           LDR   R2,   R0,   #18   ;INPUT 2
LOOP       JSR   SUBTRACT
           ADD   R1,   R0,   #0
           BRzp  LOOP
           ADD   R1,   R0,   R2
DONE       STI   R1,   OUTPUT
           HALT
OUTPUT     .FILL x4247
SUBTRACT   NOT R0, R2
           ADD R0, R0, #1
           ADD R0, R1, R0
           RET
           .END
```

(The first row of the table has been given to you as an example.)

State	BUS	Control Signals
18	x3000	LD.MAR = 1, GatePC = 1, LD.PC = 1, PCMUX = PC+1
6	x4257	LD.MAR = 1, ALUK = XX, SR1MUX = IR[8:6]
6	x4259	LD.MAR = 1, ALUK = XX, SR1MUX = IR[8:6]
9	xFFF5	LD.REG = 1, DR = IR[11:9], ALUK = NOT, SR1MUX = IR[8:6]
21	x3005	LD.REG = 1, DRMUX = R7, ADDR2MUX = IR[10:0]
16	-	LD.MDR = 0, MIO.EN = 1, R.W = W, LD.MAR = 0
32	-	LD.BEN = 1, J = XXXXXX, IRD = 1

Part B: Given the above table, and the fact that the RET instruction was executed 7 times, what were the inputs if R1 has value x0000 when the program terminates?

Input 1: #60 Input 2: #10

Name: ________________________________

Exact matches were not required: 8:6 or [6:8] could get the same credit as IR[8:6]. Similarly, PCoffset11 was enough to get credit for IR[10:0].

x4247

- The first instruction loads R0 with the mystery value in OUTPUT, and the second instruction puts the mystery value + offset into MAR to read from memory. Meanwhile, state 6 is the first state of LDR, where "MAR<-B+off6". To get B+off6 into the MAR, we must put it on the bus. Thus, we know that the BUS value x4257 = mystery value + #16 (#16 = x10). x4257 - x10 = **x4247**.
- **x4241**: **Common error:** Notice that the offset is given in decimal (#16) not hexadecimal (x16). x4257 - x16 = x4241.
- **x4246**: **Common error**: The bus value is unrelated to PC, but some students tried to subtract 1 from x4247 to counter the PC+1 in state 18.

ALUK = XX (b11), SR1MUX = IR[8:6] (b01)

- *A handful of students gave different answers for the first state 6 and the second state 6. However, control signals are fixed for a given state, so the two rows should have the same values.*
- **ALUK = XX**: Many students put ADD, and some put PASSA. However, GateALU is 0 in state 6, so the value of ALUK doesn't matter. The BUS is actually driven by GateMARMUX, where MARMUX selects ADDER, ADDR1MUX selects SR1, and ADDR2MUX selects SEXT[IR[5:0]].
- **SR1MUX = IR[8:6]**: The Instruction Set Handout shows that BaseR is specified by bits 8:6 of the IR. Some students put 11:9, but that gives the Destination Register instead.

BUX = x3005, DRMUX = R7 (b111 or b01), ADDR2MUX = IR[10:0] (b11)

- **BUX = x3005, DRMUX = R7**: In state 21 (JSR), the incremented PC (not the original, x3004) gets saved into R7. Setting DRMUX to IR[11:9] or IR[8:6] would save the incremented PC into a random register given by the offset from the location that is being jsr'd to instead. R7 is not specified dynamically by the instruction; it's built into one of the options of the DRMUX and appears directly in the state machine (instead of DR).
- **ADDR2MUX = IR[10:0]**: The JSR instruction specifies 11 bits of offset (shown in the instruction set and the state machine.) The offset is given by IR[10:0].

LD.MDR = 0, MIO.EN = 1, R.W = W (1), LD.MAR = 0

- Data is being written from MDR to M[MAR], meaning we are accessing memory (MIO.EN = 1), more specifically writing to memory (R.W = W). Neither MAR nor MDR are getting loaded with new values, so their loads are 0. (If X, registers/memory will be scrambled.)

32, LD.BEN = 1, J = XXXXXX

- IRD is only ever set in state **32**. Additionally, by looking at the microsequencer, we can see that IRD being set causes the next state to depend directly on the first four bits of the IR (or the opcode), so the values of the COND bits and the **J bits don't matter**.
- The first line in the state 32 bubble in the state machine: BEN<−IR[11] & N + IR[10] & Z + IR[9] & P. Thus, BEN gets a new value, so LD.BEN must be set.

Input 2 = #10 (xA)

Name: ________________________________

- In state 9 of the table, the BUS contains the NOT of R2 (Input 2). Unlike R1, R2 is never changed in the program after being loaded with Input 2, so we know that xFFF5 must be the not of Input 2, regardless of how many times we've passed that instruction. (That was also not an issue when using LDR to calculate the .FILL for OUTPUT because both occurrences of LDR were shown in the table, and because there was no loop around them in code. For the NOT, we have a loop, but it doesn't matter because the value of R2 is never overwritten, so the result of the first time the instruction is executed is the same as the result of the last time, or any time in between.) We can expand xFFF5 to b-1111-1111-1111-0101, and invert it to get b-0000-0000-0000-1010, or **x000A**.
- **#11 (xB)** : **Common error:** Some students saw that the subroutine involved getting the 2's complement of R2, so they took the 2's complement of xFFF5 instead. The key is that at state 9, right as the NOT instruction is finished, we haven't yet added 1 to R0 yet (that's the next instruction). Thus, the BUS value is only the inversion or the 1's complement of Input 2, not the 2's complement.

Input 1 = #60 (x3C)

- It was given that the output was 0 and that the RET instruction occurred seven times. Tracing the code reveals that the program serves as a modulus or remainder calculator, repeatedly subtracting Input 2 from Input 1 until the difference is negative, then adding Input 2 back, such that the result is a positive number smaller than Input 2. By looping until the difference is negative, we loop 1 more time than the integer result of Input 1 divided by Input 2. For instance, given Input 1 = 7 and Input 2 = 5, we would loop 2 times, even though 5 can only completely fit into 7 one time. After reaching -3, we would add 5 to get that the remainder is 2. Thus, Input 1 = Input 2 * (loops-1) + Output. Going back to the problem where it was given that the output was 0 and the RET was occurred 7 times, we know that Input 1 = Input 2 * (loops-1) + Output = 10 * (7-1) +0 = **#60.**
- **#70 (x46)**: **Common error:** Some students got the jist of the general algorithm, but didn't catch that the code added an extra iteration in the loop to go into the negatives. Thus, they thought the relationship was Input 1 = Input 2 * loops + Output.

Name: ____________________________________

Question 7 (20 Points): An programmer did a poor job implementing some important features in the operating system. Thankfully, he did manage to implement TRAPs. The table below shows how the memory in the system region is configured at the start of the program. Recall the exception vector for privilege mode exception is x00, for illegal opcode is x01, and for access control violation is x02.

Memory Address	Data	Comments
…		
x0100	x1500	
x0101	x1510	
x0102	x1520	
…		
x1500	x8000	
…		
x1510	x8000	
…		
x1520	xD025	
…		

Table below shows the user program to be executed. The user program starts executing at x3000 with priority 0.

Memory Address	Data	Assembly	Comments
x3000	x5020	`AND R0, R1, #0`	; Start of User Program
x3001	x103F	`ADD R0, R0, #-1`	
x3002	xA003	`C LDI R0, A`	
x3003	x07FE	`BRzp C`	
x3004	xA002	`LDI R0, B`	
x3005	xF025	`TRAP x25`	
x3006	x0E00	`A .FILL x0E00`	
x3007	x0E01	`B .FILL x0E01`	

Name: ________________________________

Part A: During the first 300 cycles of executing this program, what exception(s) occur if any? Assume that each memory access takes 5 cycles. Also, use the state diagram and the datapath that handles interrupts and exceptions.

ACV, illegal opcode exception

Part B: What is in memory locations x2FFC~x2FFF after 300 cycles of execution?

Memory Address	Content
x2FFC	x1520
x2FFD	x0004
x2FFE	x3002
x2FFF	x8004

Part C: Suppose the operating system engineering changes the access control violation service routine as follows.

```
           ST    R0,   SAVE_R0
           LDR   R0,   R6,   #0
           ADD   R0,   R0,   #1
           STR   R0,   R6,   #0
           LDR   R0,   R0,   #-1
           ST    R0,   LABEL
           LD    R0,   SAVE_R0
LABEL      .BLKW #1
           RTI
SAVE_R0    .BLKW #1
```

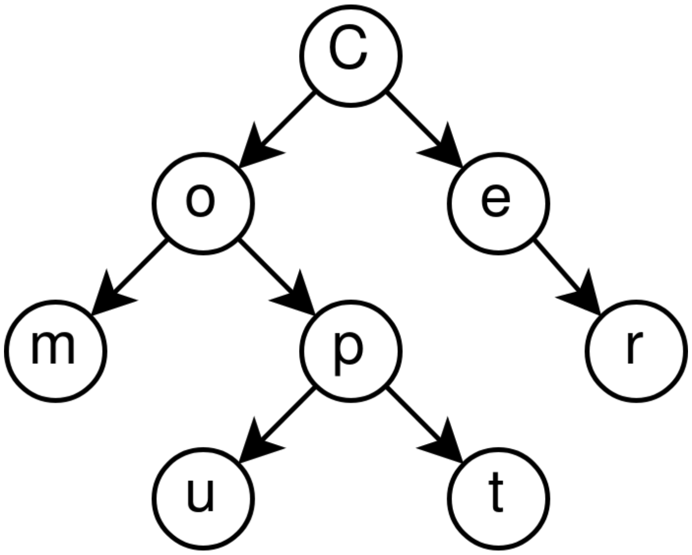
He hopes to fix the exception by executing the faulting instruction in privileged mode. Unfortunately, the code above does not do what he wishes for the user program on the previous page. Why?

The code above tries to execute the instruction causing the ACV in privilege mode by copying the instruction from user space to the memory location LABEL. This would have worked if it was a LDR or STR instruction. However, because the instruction uses PC-relative addressing modes, the location it would have loaded from would not be what the user intended to load from.

Note that if the DR in the instruction is R6, this would not have worked either as R6 is the system stack pointer. Playing with the system stack pointer would cause RTI to not work.

Name: ___________________________________

Question 8 (20 Points): A binary tree is a common data structure, used to represent (among other things) family trees and organization charts. A binary tree consists of nodes, connected by links as shown in the example below.

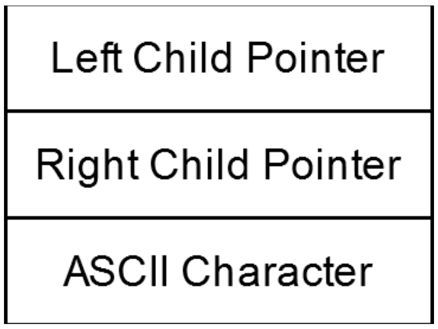


If this tree were an organization chart, C would be the CEO and o and e employees who report directly to him. If the tree were a family tree, C would be the patriarch.

Each link connects a **"parent" node** to one of its possible **"child" nodes**. For example, node o has two children nodes, m and p. Node r has no children. Every node except one has a single parent. The node that does not have a parent is called the "**root**." Each node can have up to two children. The children are the roots of subtrees. For example, the entire tree has a root (node C), a left subtree (consisting of nodes o, m, p, u, t) and a right subtree (consisting of node e and r). Node o is the root of a subtree having a left subtree (consisting of node m) and a right subtree (consisting of nodes p, u, and t).

Unlike a linked list which has only one order in which all the nodes of the link list can be visited, the nodes of a tree can be visited in several different orders. One of the orders is called "pre-order traversal," in which the root is visited first, then all the nodes of its left subtree, then all the nodes of its right subtree. In the tree above the pre-order traversal visits C then o, m, p, u, t, e, and r in that order.

If we identify the data of each node as the ASCII code of a typed character, each node in the tree above occupies three words of LC-3 memory as shown below: a link to the left child, a link to the right child, and the data (ascii code). If the node does not have a left (or right) subtree, the corresponding link is x0000.



Name: ______________________________________

Below is a program which prints all the characters in the tree in the order described above. **For each node it visits, it keeps track of the right child (if there is one), and then follows the left child. When it visits a node with no left child, the program goes through the right children in reverse order.** Assume the tree is already in memory with the root at address x4000.

Your Job: Fill in the missing instructions**.**

```
           .ORIG x3000
           LD R6, STACK
           LD R1, TREE_ROOT
LOOP       LDR R0, R1, #2
           TRAP x21             ; OUT
           LDR R0, R1, #1
           BRz SKIP
           ADD R6, R6, #-1
           STR R0, R6, #0
SKIP       LDR R1, R1, #0
           BRnp LOOP
           LD R2, DONE_VAL
           ADD R0, R2, R6
           BRz DONE
           LDR R1, R6, #0
           ADD R6, R6, #1
           BRnzp LOOP
DONE       HALT

TREE_ROOT  .FILL    x4000
DONE_VAL   .FILL    x0200 ;xFE00+x0200=x0000
STACK      .FILL    xFE00
           .END
```

Name: ________________________________

Question 9 (20 Points): In this question, **we modify how JSR/JSRR and RET works in LC3.** JSR/JSRR will use the stack as the linkage instead of R7 by pushing the return PC onto the stack. Similarly, RET will load the PC with the value popped from the top of the stack.

In memory, there are 10 subroutines to print the 10 digits. For example, the subroutine to print the number “5” consists of the following instructions:

```
Print5      LD R0 Label5
            OUT
            RET
Label5      .FILL x35
```

Since each subroutine requires 4 memory locations, the starting address of each subroutine is always 4 greater than the starting address of the previous subroutine. The starting address of the subroutine Print0 is x5000. Therefore, the starting address of the subroutine Print1 is x5004, the starting address of the subroutine Print2 is x5008, and so on.

A student wishes to surprise Dr. Patt by printing our course number “306” with a program that does not use any JSR/JSRR instructions nor I/O trap service routines in the main program.
Your Job: Fill in the blanks in the main program, so that executing this program will result in printing the digits “306” on the console, and **then halt the machine**.

Hint: What happens when a RET is executed without a JSR being executed beforehand?

```
         .ORIG x3000
         LD   R6,      SP_INIT
         LD   R0,      LABELH
         ADD  R6       R6       #-1
         STR  R0,      R6,      #0
         LD   R0,      LABEL6
         ADD  R6       R6       #-1
         STR  R0,      R6,      #0
         LD   R0,      LABEL0
         ADD  R6       R6       #-1
         STR  R0,      R6,      #0
         LD   R0,      LABEL3
         ADD  R6       R6       #-1
         STR  R0,      R6,      #0
         RET
LABELH   .FILL    x3012
LABEL6   .FILL    x5018
LABEL0   .FILL    x5000
LABEL3   .FILL    x500C
         HALT
SP_INIT  .FILL    xFE00
         .END
```

Name: ____________________________________

Explanation for Q9: This is actually a real world security attack called Return-Oriented-Programming, invented by one of UT CS professors, **Hovav Shacham (utexas.edu)**

The premise of this attack/question is that how to execute some instructions that are never part of your main program. The idea is to chain together a new sequence of instructions together through return instructions. Because RET instructions would pop the top of the stack, all the user have to do is to find instructions followed by a RET instruction and put that address on the stack. The system will then execute the new sequence of instructions. This attack is powerful as this the program can look benign at first glance.