EXPERIMENT #6

Simple Computer SLC-3.2 in SystemVerilog

I.   OBJECTIVE

In this experiment, you will design a simple microprocessor using SystemVerilog. It will be a subset of the LC-3 ISA, a 16-bit processor with 16-bit Program Counter (PC), 16-bit instructions, and 16-bit registers. *(For LC-3, see Patt and Patel (ECE 120 textbook).*

II.   INTRODUCTION

There are three main components to the design of a processor. The central processing unit (CPU), the memory that stores instructions and data, and the input/output interface that communicates with external devices. You will be provided with the interface between the CPU and the memory (memory read and write functions). The computer will first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and then fetch again. See Figure 1.
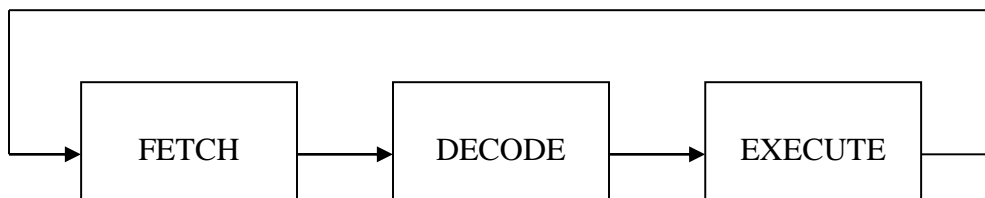
Figure 1

The CPU will contain a PC, a Instruction Register (IR), a Memory Address Register (MAR), a Memory Data Register (MDR), a Instruction Sequencer/Decoder, a status register (nzp), a 8x16 general-purpose register file, and an Arithmetic Logic Unit (ALU). All registers and instructions are 16-bits wide. The ALU will operate on 16-bit inputs. The Instruction Sequencer/Decoder will be responsible for providing proper control signals to the other components of the processor. It will contain a state machine that will provide the signals that will control the sequence of operations (fetch →decode → execute → fetch next) in the processor.

6.2

The simple computer will perform various operations based on the opcodes. An opcode specifies the operation to be performed. Specific opcodes and operations are shown in Table 1. The 4-bit opcode is specified by IR[15:12]; the remaining twelve bits contain data relevant that instruction.

In the table below, R(X) specifies a register in the register file, addressed by the three-bit address X. SEXT(X) indicates the 2's compliment sign extension of the operand X to 16 bits. nzp is the status register mentioned above. It is a three-bit value that states whether the resulting value loaded to the register file is negative, zero, or positive. This must be updated whenever an instruction performs a write to the register file (except JSR). For all instructions, PC ← PC + 1 is implicit, unless PC is stated to get some other value. In the table, right-hand-side "PC" indicates the value of the PC register <u>after</u> it was incremented immediately following fetch.

Instruction	Instruction(15 downto 0)						Operation
ADD	0001	DR	SR1	0	00	SR2	R(DR) ← R(SR1) + R(SR2)
ADDi	0001	DR	SR	1	imm5		R(DR) ← R(SR) + SEXT(imm5)
AND	0101	DR	SR1	0	00	SR2	R(DR) ← R(SR1) AND R(SR2)
ANDi	0101	DR	SR	1	imm5		R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001	DR	SR	111111			R(DR) ← NOT R(SR)
BR	0000	n z p	PCoffset9				if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100	000	BaseR	000000			PC ← R(BaseR)
JSR	0100	1	PCoffset11				R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110	DR	BaseR	offset6			R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111	SR	BaseR	offset6			M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12					LEDs ← ledVect12; Wait on Continue

Table 1: The SLC-3.2 ISA

The IR will provide the Instruction Sequencer/Decoder with the instruction to be executed. The IR will also provide the datapath with any other necessary data. As mentioned earlier, the Instruction Sequencer/Decoder will need to generate the control signals to execute the instructions in proper order. The Instruction Sequencer/Decoder will also specify the operation to

the ALU (e.g. add, etc.). Note that each operation will take multiple cycles and the Instruction Sequencer/Decoder will need to provide signals appropriately at each cycle.

On a reset, the Instruction Sequencer/Decoder should reset to the starting "halted" state, and wait for Run to go high. The PC should be reset to zero upon a reset, where it should proceed on incrementing itself when Run is pressed for fetching the instructions line by line. The first three lines of instructions will be used to load the PC with the value on the slider switches, which indicates the starting address of the instruction(s) of interest (in the form of test programs for the demo), and the program should begin executing instructions starting at the PC. Your computer must be able to return to the halted state any time a reset signal arrives.

Instruction Summary

ADD	Adds the contents of SR1 and SR2, and stores the result to DR. Sets the status register.
ADDi	Add Immediate. Adds the contents of SR to the sign-extended value imm5, and stores the result to DR. Sets the status register.
AND	ANDs the contents of SR1 with SR2, and stores the result to DR. Sets the status register.
ANDi	And Immediate. ANDs the contents of SR with the sign-extended value imm5, and stores the result to DR. Sets the status register.
NOT	Negates SR and stores the result to DR. Sets the status register.
BR	Branch. If any of the condition codes match the condition stored in the status register, takes the branch; otherwise, continues execution. (An unconditional jump can be specified by setting NZP to 111.) Branch location is determined by adding the sign-extended PCoffset9 to the PC.
JMP	Jump. Copies memory address from BaseR to PC.
JSR	Jump to Subroutine. Stores current PC to R(7), adds sign-extended PCoffset11 to PC.
LDR	Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT(offset6)). Sets the status register.
STR	Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
PAUSE	Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be "cleared" per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.

Here are the operations in more detail:

Fetch:
    MAR ← PC; MAR = memory address to read the instruction from
    MDR ← M(MAR); MDR = Instruction read from memory (note that M(MAR) specifies
                        the data at address MAR in memory).

IR ← MDR; IR = Instruction to decode
PC ← (PC + 1)

Decode:
Instruction Sequencer/Decoder ← IR

Execute:
Perform the operation based on the signals from the Instruction Sequencer/Decoder and
write the result to the destination register or memory.

Fetch, Load, and Store Operations:
For Fetch, Load (LDR), and Store (STR) operations you will need to set the memory signals
(see Memory Interface below) appropriately for each state of the fetch/load/store sequence.
Also, notice that the RAM we use does not have an R signal indicating that a read/write
operation is ready. Instead, for any states reading from or writing to RAM, we stay at those
states for several clock cycles to ensure that a memory read/write operation is complete.

FETCH:
state1: MAR ← PC
state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
state3: IR ← MDR;
     PC ← PC+1; -- "+1" inserts an incrementer/counter instead of an adder.
     Go to the next state.

LOAD:
state1: MAR ← (BaseR + SEXT(offset6)) from ALU
state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
state3: R(DR) ← MDR;

STORE:
state1: MAR ← (BaseR + SEXT(offset6)) from ALU; MDR ← R(SR)
state2: M(MAR) ← MDR; -- *assert Write Command on the RAM*

Memory Interface
The DE10-Lite board is equipped with a MAX10 FPGA chip equip with 1Mb on-chip
memory. You will need to provide a memory address in MAR, data to be written in MDR (in
the case of Store), and the Read and Write signals. The interface for the on-chip memory chip
is as follows:

q   16 bit data bus.
address   10-bit Address bus (in LC-3, the address space is only 16-bit wide, so the
             addresses take only the 10 least significant bits)
rden   Read enable. When active, allows read operations. Active high.
wren   Write enable. When active, allows write operations. Active high.LB
             Lower Byte enable. Allows read/write operations on I/O<7:0>. Active low.

I/O Specifications

I/O for this CPU is memory-mapped. I/O devices are connected to the memory signals, with a special buffer inserted on the memory data bus. When a memory access occurs at an I/O device address, the I/O device detects this, and sends a signal to the buffer to deactivate the memory, and instead use the I/O device's data for the response. You will be provided with a SystemVerilog entity that encapsulates this functionality into a single module for you to insert between your processor, on-chip memory, and the external I/O (this is part of what the mem2io module does) (see figures 2 & 3)
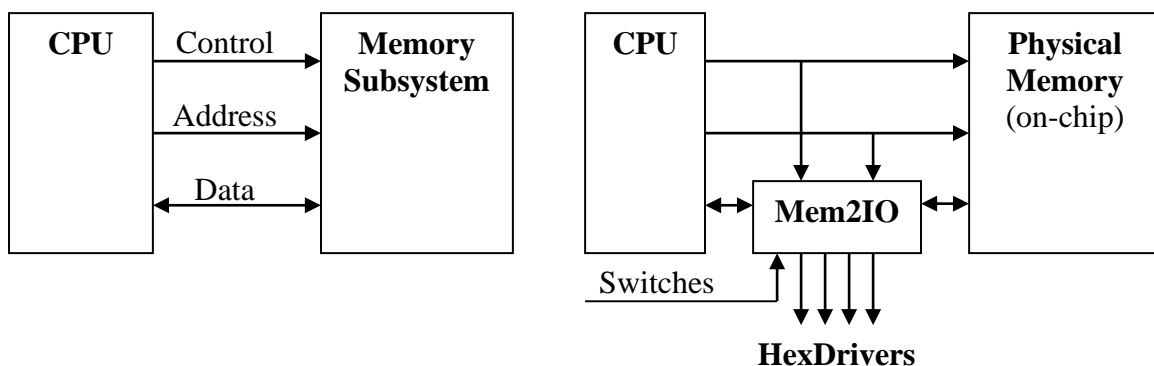
Mem2IO

This manages all I/O with the DE10-Lite physical I/O devices, namely, the switches and 7-segment displays. See Table 2. Note that the two devices share the same memory address. This is acceptable, because one of the devices (the switches) is purely input, while the other (the hex displays) is purely output.

Physical I/O Device	Type	Memory Address	"Memory Contents"
DE10 Board Hex Display	Output	0xFFFF	Hex Display Data
DE10 Board Switches	Input	0xFFFF	Switches(9:0)

Table 2: Physical I/O Device List

You will need to create a top-level port map file that includes your CPU, the Mem2IO entity, and four HexDrivers. The CPU is a high level entity that contains the majority of your modules, including the ISDU. The various memory control signal inputs and the memory address input should be connected to the corresponding outputs from the CPU (which are also output from your top-level entity to the actual memory). This lab uses on-chip memory, thus, there is no external memory connections needed. The four "HEX#" output signals should be connected to HexDriver inputs. See the partial block diagrams in figures 2 & 3.

Figure 2: Conceptual　　　　　**Figure 3: Memory with Mem2IO unit**
Picture of Memory

Usage of the Pause Instruction

The Pause instruction is to be used in conjunction with I/O. Pausing allows the user time to set the switches before an input operation, and read the output after an output operation. The top two bits of the ledVect12 field of the instruction indicate the related I/O operation as indicated in Table 3. Note that these should be considered as masks, not as mutually exclusive values. For example, a ledVect12[11:10] value of "11" would indicate that both new data is being displayed on the hex displays and the program is asking for a new switch value. The remaining ledVect12 bits should be used to output a unique identifier to communicate the location in the program to keep track of the computer's progress. NOTE: The ledVect12 vector does not mean anything to the processor; their sole purpose is to be a visual cue that the user can define (when programming) so that he/she can tell where the program is during execution. The following table reflects the convention used in the test programs.

ledVect12(11:10) Mask	Meaning (cue for the user only)
"01"	Previous operation was a write to the hex display
"10"	Next operation is a read from switches

Table 3: ledVect12(11:10) Masks for Pause Instruction

Your top-level circuit should have **at least** the following inputs and outputs:
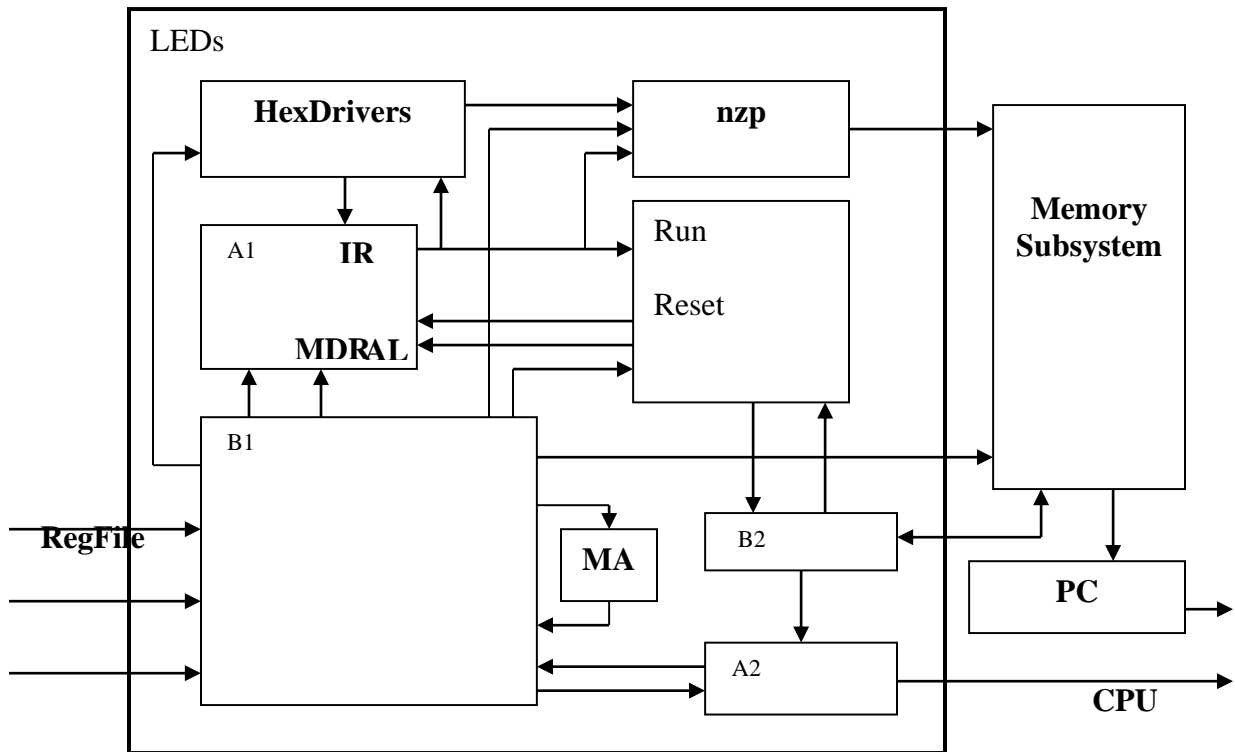 **Inputs**
    SW– logic [9:0]
    Clk, Run, Continue –logic
  Outputs
    LED – logic [9:0]
    HEX0, HEX1, HEX2, HEX3- logic [6:0]

(You may expand this list as needed for simulation and debugging.)

SIMPLIFIED SAMPLE CPU BLOCK DIAGRAM

LEDs

HexDrivers

nzp

Memory Subsystem

A1   IR

Run

Reset

MDRAL

B1

RegFile

MA

B2

PC

A2

CPU

Notes: Arrows are shown for connections between components. There may be multiple signals going from one block to the other even if there is only one connection shown between the blocks. One arrow does not mean one signal/bus in all cases. Signal multiplexing has been omitted (your diagram should show a mux in front of most registers). All registers will also have Clk as an input. Note that this block diagram does not show the mem2io module that should serve as an interface between the CPU and the memory (it is implicit in the memory subsystem block, this is just a reminder that you need to include it).

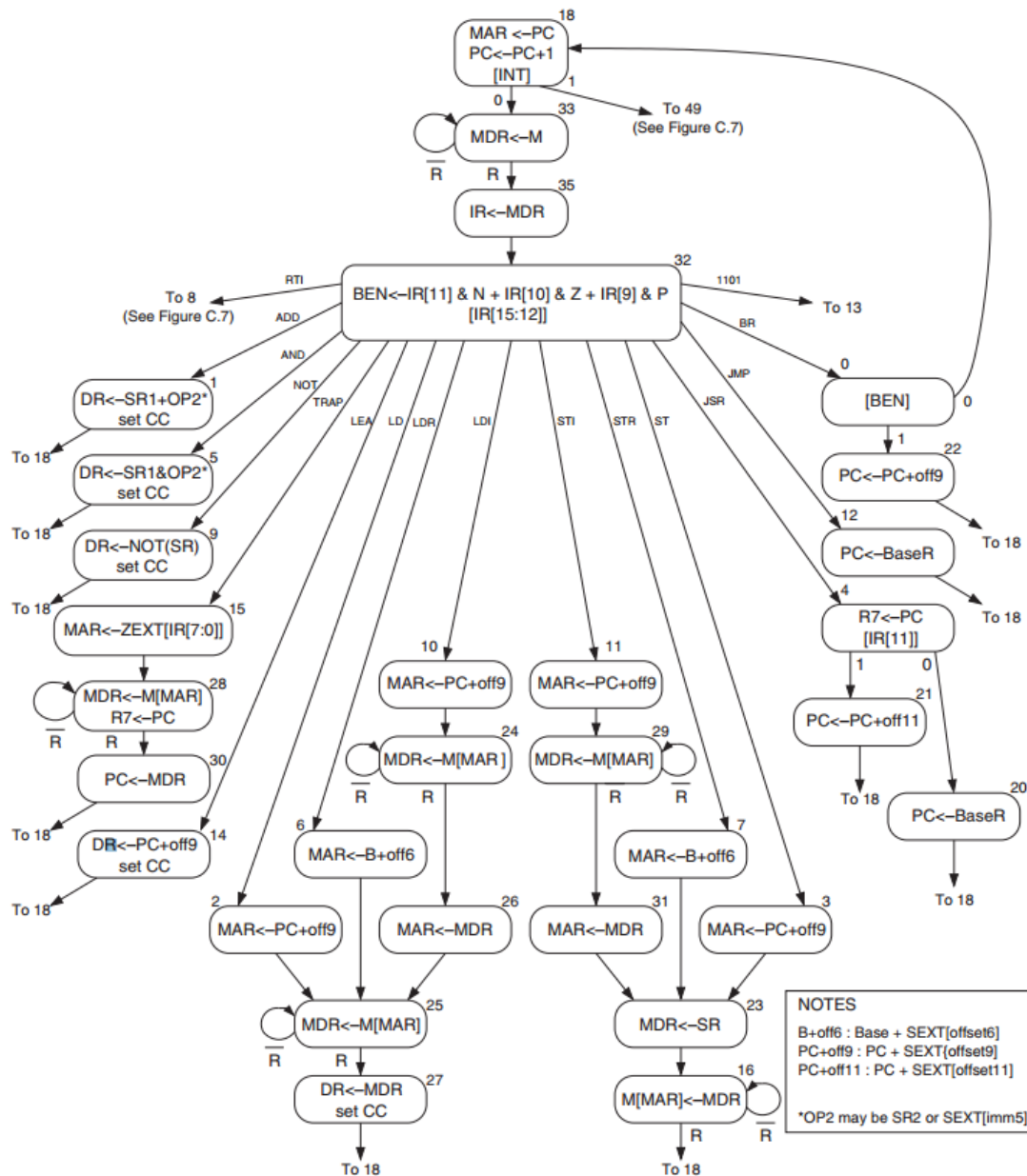# LC3 STATE DIAGRAM FROM APPENDIX C OF PATT AND PATEL

Figure C.2 A state machine for the LC-3

Notes: This is not the state diagram for the simplified LC3, it has some instructions that are not required to be implemented in this lab. Please refer to the instruction summary for details on which instructions are required to complete the lab.

III. PRE-LAB

A. **Week 1:**

Lab 6 is split up into two discrete tasks. In the first week, you will implement the FETCH phase. You will have to understand the structure of the memory system, and how the memory system interfaces with the CPU. You will also have to implement all the necessary CPU entities and ISDU controls to be able to successfully fetch the instructions line by line from the on-board memory to the CPU. Note that since you'll not be doing DECODE and EXECUTE during week 1, you don't have to pass the fetched instructions into the ISDU. But instead, you should display the content of the IR, which will be storing the fetched instructions at the end of the FETCH phase.

You are provided with the following entities on the website Mem2IO, Test_Memory, and ISDU.

For the purpose of Week 1 demo, you should connect the HEX displays directly to the IR rather than connecting them to the Mem2IO (in contrast, in week 2, you will specify a special address in memory that you write to in order to show data on the HEX displays). To display the content of the IR on the FPGA, there are extra pause states at the end of the FETCH phase to be able to hold and see the content of the IR. Pressing the 'continue' button, your ISDU should loop back to perform the FETCH phase all over again, instead of continuing onto the DECODE phase.

You should not need to modify the given ISDU for week 1, a higher level LC3 entity is given and you will have to implement all other component parts of the LC3 such as the program counter, register file, and data bus.

Week 1 Demo Point Breakdown:

1 point: Simulate the correct value of MAR = PC and PC = PC +1;
1 point: Simulate the correct value of MDR = M(MAR) and IR = MDR;
1 points: Correct operation on the FPGA: Displaying correct value of IR on HEX3-0 in state PauseIR1.

B. **Week 2:**

6.10

For the second week, you will need to implement the DECODE and the EXECUTE phase. You will first extend the provided skeleton ISDU to include all the necessary state transitions and the necessary inputs/outputs in each of the states.

You will need to learn and understand the specification of the LC3 and its state diagram to figure out how to assign the various control signals in each state to produce the desired operations. At this point, you will have to take out the pause states which you have inserted after the FETCH phase during week 1, for the ISDU to continue onto the DECODE and the EXECUTE phase instead.

Week 2 Demo Point Breakdown:
1 point: Basic I/O Test 1
1 point: Basic I/O Test 2
1 point: Self Modifying Code Test
1 point: XOR Test
1 point: Multiplication Test
1 point: Sort Test
1 point: Correct "Act Once" Behavior

IV. LAB

Follow the Lab 6 demo information on the course website. Follow the *Week 2 Test Programs Documentation* in the Lab 6 information page on the course website to demonstrate the 5 tests.

Pin Assignment Table

Clk	PIN_P11	50 MHz Clock from the on-board oscillators
Run	PIN_B8	On-Board Push Button (KEY1)
Continue	PIN_A7	n-Board Push Button (KEY0)
SW[0]	PIN_C10	On-board slider switch (SW0)
SW[1]	PIN_C11	On-board slider switch (SW1)
SW[2]	PIN_D12	On-board slider switch (SW2)
SW[3]	PIN_C12	On-board slider switch (SW3)
SW[4]	PIN_A12	On-board slider switch (SW4)
SW[5]	PIN_B12	On-board slider switch (SW5)
SW[6]	PIN_A13	On-board slider switch (SW6)
SW[7]	PIN_A14	On-board slider switch (SW7)
SW[8]	PIN_B14	On-board slider switch (SW8)
SW[9]	PIN_F15	On-board slider switch (SW9)
LED[0]	PIN_A8	On-board LED
LED[1]	PIN_A9	On-board LED
LED[2]	PIN_A10	On-board LED

LED[3]	PIN_B10	On-board LED
LED[4]	PIN_D13	On-board LED
LED[5]	PIN_C13	On-board LED
LED[6]	PIN_E14	On-board LED
LED[7]	PIN_D14	On-board LED
LED[8]	PIN_A11	On-board LED
LED[9]	PIN_B11	On-board LED
HEX0[0]	PIN_C14	On-board Hex
HEX0[1]	PIN_E15	On-board Hex
HEX0[2]	PIN_C15	On-board Hex
HEX0[3]	PIN_C16	On-board Hex
HEX0[4]	PIN_E16	On-board Hex
HEX0[5]	PIN_D17	On-board Hex
HEX0[6]	PIN_C17	On-board Hex
HEX0[7]	PIN_D15	On-board Hex
HEX1[0]	PIN_C18	On-board Hex
HEX1[1]	PIN_D18	On-board Hex
HEX1[2]	PIN_E18	On-board Hex
HEX1[3]	PIN_B16	On-board Hex
HEX1[4]	PIN_A17	On-board Hex
HEX1[5]	PIN_A18	On-board Hex
HEX1[6]	PIN_B17	On-board Hex
HEX1[7]	PIN_A16	On-board Hex
HEX2[0]	PIN_B20	On-board Hex
HEX2[1]	PIN_A20	On-board Hex
HEX2[2]	PIN_B19	On-board Hex
HEX2[3]	PIN_A21	On-board Hex
HEX2[4]	PIN_B21	On-board Hex
HEX2[5]	PIN_C22	On-board Hex
HEX2[6]	PIN_B22	On-board Hex
HEX2[7]	PIN_A19	On-board Hex
HEX3[0]	PIN_F21	On-board Hex
HEX3[1]	PIN_E22	On-board Hex
HEX3[2]	PIN_E21	On-board Hex
HEX3[3]	PIN_C19	On-board Hex
HEX3[4]	PIN_C20	On-board Hex
HEX3[5]	PIN_D19	On-board Hex
HEX3[6]	PIN_E17	On-board Hex
HEX3[7]	PIN_D22	On-board Hex
HEX4[0]	PIN_F18	On-board Hex
HEX4[1]	PIN_E20	On-board Hex
HEX4[2]	PIN_E19	On-board Hex
HEX4[3]	PIN_J18	On-board Hex

6.12

HEX4[4]	PIN_H19	On-board Hex
HEX4[5]	PIN_F19	On-board Hex
HEX4[6]	PIN_F20	On-board Hex
HEX4[7]	PIN_F17	On-board Hex
HEX5[0]	PIN_J20	On-board Hex
HEX5[1]	PIN_K20	On-board Hex
HEX5[2]	PIN_L18	On-board Hex
HEX5[3]	PIN_N18	On-board Hex
HEX5[4]	PIN_M20	On-board Hex
HEX5[5]	PIN_N19	On-board Hex
HEX5[6]	PIN_N20	On-board Hex
HEX5[7]	PIN_L19	On-board Hex

V.   POST-LAB

1.)   Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table.

LUT	
DSP	
Memory (BRAM)	
Flip-Flop	
Frequency	
Static Power	
Dynamic Power	
Total Power	

Document any problems you encountered and your solutions to them, and a short conclusion. Before you leave from your lab session submit your latest project code including the .sv files to your TA on his/her USB drive. TAs are under no obligation to accept late code, code that doesn't compile (unless you got 0 demo points) or code files that are intermixed with other project files.

2.) Answer at least the following questions in the lab report

- What is MEM2IO used for, i.e. what is its main function?

- What is the difference between BR and JMP instructions?

- What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of the signal in our design? What implications does this have for synchronization?

VI. <u>REPORT</u>

You do NOT need to write a report after week 1. Instead you will write one report for the entirety of Lab 6. Write a report, you may follow the provided outline below, or make sure your own report outline includes at least the items enumerated below.

1. Introduction
   a. Summarize the basic functionality of the SLC-3 processor
2. Written Description and Diagrams of SLC-3
   a. Summary of Operation
   b. Describe in words how the SLC-3 performs its functions. In particular, you should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.
   c. Block Diagram of slc3.sv
   d. This diagram should represent the placement of all your modules in the slc3.sv. Please only include the slc3.sv diagram and not the RTL view of every module (this can go into the individual module descriptions).
   e. Written Description of all .sv modules
      i. A guide on how to do this was shown in the Lab 5 report outline.
   f. Description of the operation of the ISDU (Instruction Sequence Decoder Unit)
      i. Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.
      ii. If you prefer to, you can lump this section into the module description section under ISDU.sv.
   g. State Diagram of ISDU
      i. This should represent all states present in the ISDU and their transitions. The diagram from Patt & Patel Appendix C can be used as a starting point but would need to be modified to be representative of the ECE385 implementation of the LC-3. You will lose points if you just copy the diagram.
3. Simulations of SLC-3 Instructions

6.14

      a. Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier and Sort.

      b. Annotations for the above simulations, should include at a minimum, start of the test program, any user input (for example, entering the numbers in the multiplier test), and reading the expected result.

4. Post-Lab Questions

      a. Fill out the Design Resources and Statistics table from Post-Lab question one

      b. Answer all the post-lab questions. As usual, they may be in their own section or dispersed into the appropriate sections in the rest of the report.

5. Conclusion

      a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it

      b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.