ECE 385

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.

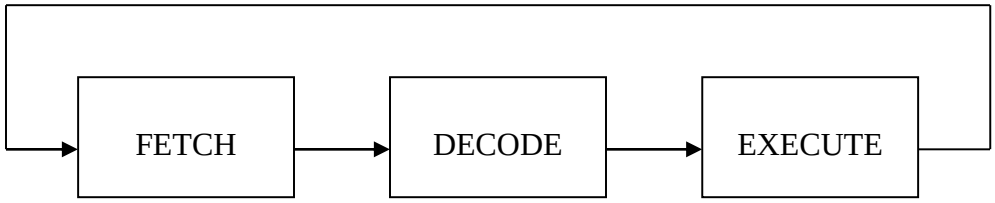


Figure 1

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.

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 Instruction[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 after 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, and Store operations you will need to set the memory signals appropriately for each state of the fetch/load/store sequence. You should incorporate the three-state Fetch, Load, and Store operations in your main Instruction Sequencing State Diagram.

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 DE2 board is equipped with one 512-KB SRAM. 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 these memory chips is as follows:

I/O	Bidirectional 16-bit data bus.
A	18-bit Address bus (shared between both chips)
CE	Chip Enable. When active, allows read/write operations. Active low.
UB	Upper Byte enable. Allows read/write operations on I/O<15:8>. Active low.
LB	Lower Byte enable. Allows read/write operations on I/O<7:0>. Active low.
OE	Output Enable. When active, RAM chips will drive output on selected banks of selected chips. Active low.
WE	Write Enable. When active, orders writes to selected banks of selected chips. Active low. Has priority over OE.

Note that I/O is declared as an **inout** port type. This means that it is a bidirectional data bus. In general, any port that attempts to both read and write to a bus needs to be declared as an inout type. A port can write to the bus through an output port. When not writing to the bus, you should assign the output a high-impedance value ("ZZZZZZZZZZZZZZZZ"). Otherwise, when reading from memory, both your circuit and the SRAM will try to drive the line at the same time, with unpredictable results (possibly including damage to one or both chips). This functionality should be included in a separate entity that you should name "tristate_buffer". This will have two inputs, the data from MDR and a select bit from the controller, and one output that you should tie to the data line. Reading from the data line requires no extra circuitry; the data line can be tied directly to the MDR input (multiplexed with MDR's other sources).

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 and the external memory (this is part of what the mem2io module does) (see figures 2 & 3)

Mem2IO

This manages all I/O with the DE2 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"**
DE2 Board Hex Display	Output	0xFFFF	Hex Display Data
DE2 Board Switches	Input	0xFFFF	Switches(15:0)

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). The memory data inout port from your

CPU should be connected to the Data_CPU inout port of the Mem2IO unit; the Data_Mem inout port of the Mem2IO unit should be connected to an inout port on your top-level entity, which should be assigned to the appropriate pin connected to physical memory. The four "HEX#" output signals should be connected to HexDriver inputs. See the partial block diagrams in figures 2 & 3.

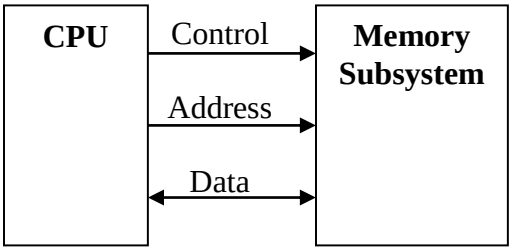


Figure 2: Conceptual Picture of Memory

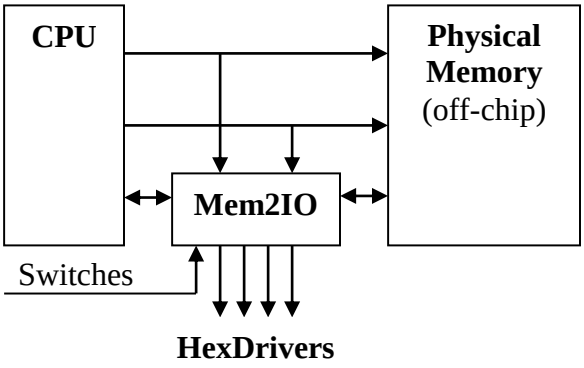


Figure 3: Memory with Mem2IO unit

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

S – logic [15:0]
Clk, Reset, Run, Continue –logic

Outputs

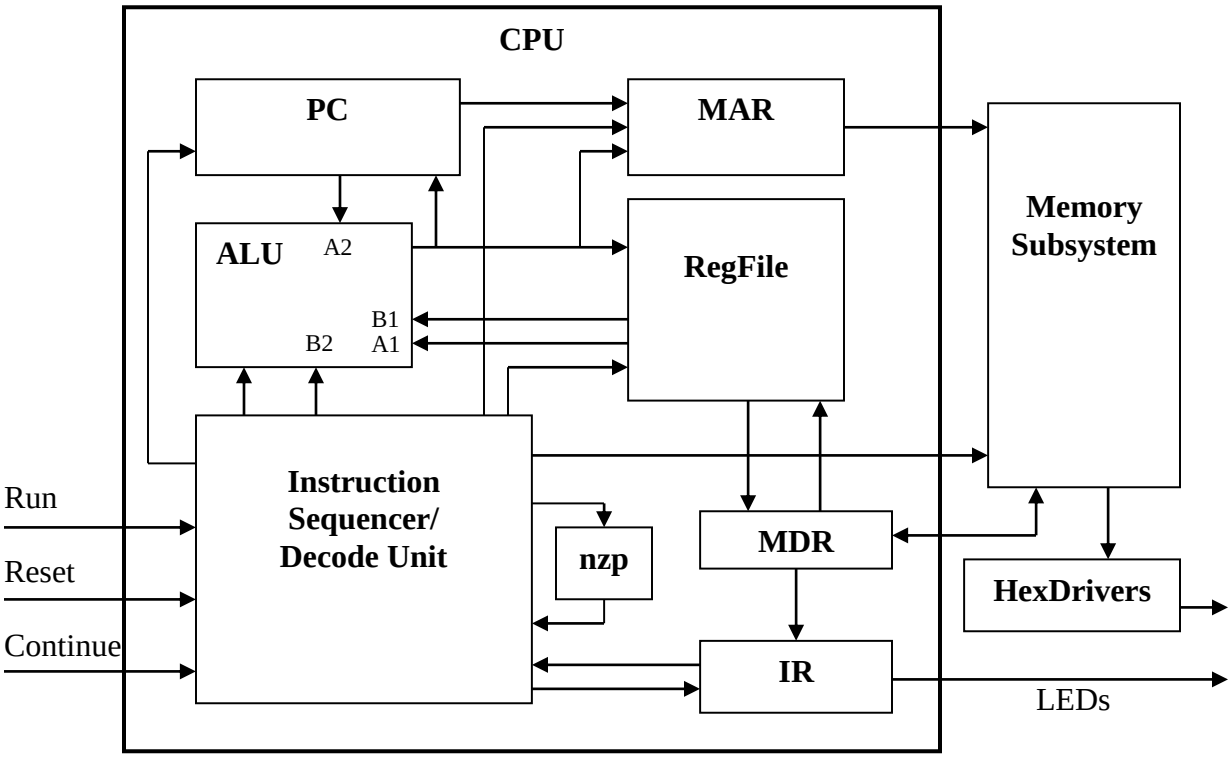
LED – logic [11:0]
HEX0, HEX1, HEX2, HEX3 - logic [6:0]
CE, UB, LB, OE, WE –logic
ADDR – logic [19:0]

Bidirectional ports (inout)

Data - logic [15:0]

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

SIMPLIFIED SAMPLE CPU BLOCK DIAGRAM



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

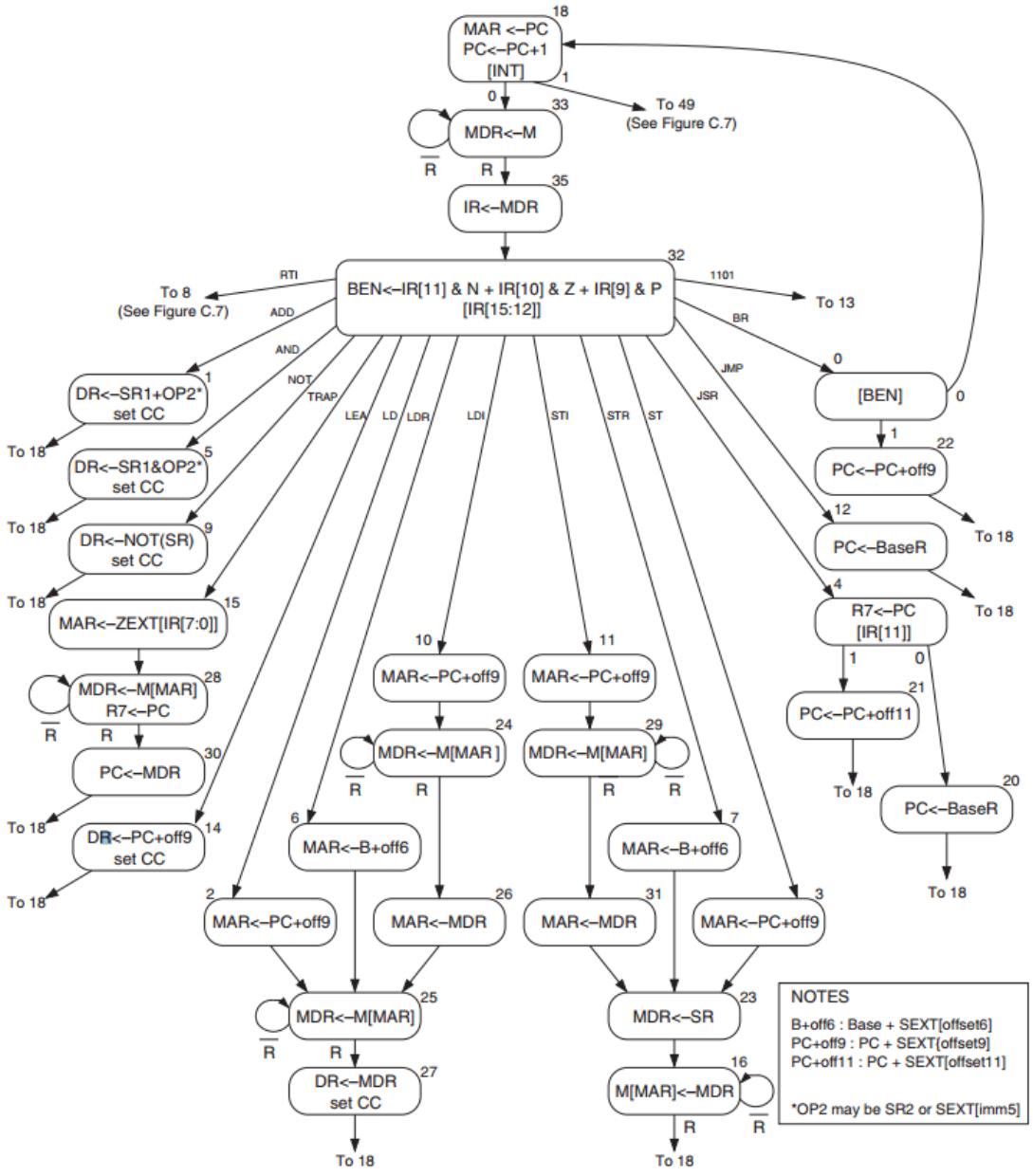


Figure C.2 A state machine for the LC-3

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 first week you do not need to use the Mem2IO, however to remove errors in week 2 there is no harm in using the entity in your design for week 1 as well. The use for Test_Memory is only to simulate your design, to replace the off chip memory. The design you upload to the FPGA should not have Test_Memory, unless it is to debug.

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, you will be required to create a higher level LC3 entity and begin implementing all the component parts of the LC3 such as the program counter, register file, and memory interface. You are free to implement the top level entity as a schematic or as raw SystemVerilog code.

Week 1 Demo Point Breakdown:

1 point: Simulate the correct value of MAR = PC and PC = PC +1; (use Test_Memory)
1 point: Simulate the correct value of MDR = M(MAR) and IR = MDR; (use Test_Memory)
3 points: Correct operation on the FPGA: Displaying correct value of IR on HEX4-7 in state PauseIR1; (Must use physical memory, i.e. do not use Test_Memory)

B. **Week 2:**

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 in order 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, in order for the ISDU to continue onto the DECODE and the EXECUTE phase instead.

Note: You should not be using Test_Memory in week 2 unless you are simulating or debugging.

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: 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

Port Name	Location	Comments
Clk	PIN_Y2	50 MHz Clock from the on-board oscillators
Run	PIN_R24	On-Board Push Button (KEY3)
Continue	PIN_N21	On-Board Push Button (KEY2)
Reset	PIN_M23	On-Board Push Button (KEY0)
S[0]	PIN_AB28	On-board slider switch (SW0)
S[1]	PIN_AC28	On-board slider switch (SW1)
S[2]	PIN_AC27	On-board slider switch (SW2)
S[3]	PIN_AD27	On-board slider switch (SW3)
S[4]	PIN_AB27	On-board slider switch (SW4)
S[5]	PIN_AC26	On-board slider switch (SW5)
S[6]	PIN_AD26	On-board slider switch (SW6)
S[7]	PIN_AB26	On-board slider switch (SW7)
S[8]	PIN_AC25	On-board slider switch (SW8)

S[9]	PIN_AB25	On-board slider switch (SW9)
S[10]	PIN_AC24	On-board slider switch (SW10)
S[11]	PIN_AB24	On-board slider switch (SW11)
S[12]	PIN_AB23	On-board slider switch (SW12)
S[13]	PIN_AA24	On-board slider switch (SW13)
S[14]	PIN_AA23	On-board slider switch (SW14)
S[15]	PIN_AA22	On-board slider switch (SW15)
LED[0]	PIN_G19	On-Board LED (LEDR0)
LED[1]	PIN_F19	On-Board LED (LEDR1)
LED[2]	PIN_E19	On-Board LED (LEDR2)
LED[3]	PIN_F21	On-Board LED (LEDR3)
LED[4]	PIN_F18	On-Board LED (LEDR4)
LED[5]	PIN_E18	On-Board LED (LEDR5)
LED[6]	PIN_J19	On-Board LED (LEDR6)
LED[7]	PIN_H19	On-Board LED (LEDR7)
LED[8]	PIN_J17	On-Board LED (LEDR8)
LED[9]	PIN_G17	On-Board LED (LEDR9)
LED[10]	PIN_J15	On-Board LED (LEDR10)
LED[11]	PIN_H16	On-Board LED (LEDR11)
HEX0[0]	PIN_G18	On-Board seven-segment display segment (HEX0[0])
HEX0[1]	PIN_F22	On-Board seven-segment display segment (HEX0[1])
HEX0[2]	PIN_E17	On-Board seven-segment display segment (HEX0[2])
HEX0[3]	PIN_L26	On-Board seven-segment display segment (HEX0[3])
HEX0[4]	PIN_L25	On-Board seven-segment display segment (HEX0[4])
HEX0[5]	PIN_J22	On-Board seven-segment display segment (HEX0[5])
HEX0[6]	PIN_H22	On-Board seven-segment display segment (HEX0[6])
HEX1[0]	PIN_M24	On-Board seven-segment display segment (HEX1[0])
HEX1[1]	PIN_Y22	On-Board seven-segment display segment (HEX1[1])
HEX1[2]	PIN_W21	On-Board seven-segment display segment (HEX1[2])
HEX1[3]	PIN_W22	On-Board seven-segment display segment (HEX1[3])
HEX1[4]	PIN_W25	On-Board seven-segment display segment (HEX1[4])
HEX1[5]	PIN_U23	On-Board seven-segment display segment (HEX1[5])
HEX1[6]	PIN_U24	On-Board seven-segment display segment (HEX1[6])
HEX2[0]	PIN_AA25	On-Board seven-segment display segment (HEX2[0])
HEX2[1]	PIN_AA26	On-Board seven-segment display segment (HEX2[1])
HEX2[2]	PIN_Y25	On-Board seven-segment display segment (HEX2[2])
HEX2[3]	PIN_W26	On-Board seven-segment display segment (HEX2[3])
HEX2[4]	PIN_Y26	On-Board seven-segment display segment (HEX2[4])
HEX2[5]	PIN_W27	On-Board seven-segment display segment (HEX2[5])
HEX2[6]	PIN_W28	On-Board seven-segment display segment (HEX2[6])
HEX3[0]	PIN_V21	On-Board seven-segment display segment (HEX3[0])
HEX3[1]	PIN_U21	On-Board seven-segment display segment (HEX3[1])
HEX3[2]	PIN_AB20	On-Board seven-segment display segment (HEX3[2])
HEX3[3]	PIN_AA21	On-Board seven-segment display segment (HEX3[3])
HEX3[4]	PIN_AD24	On-Board seven-segment display segment (HEX3[4])
HEX3[5]	PIN_AF23	On-Board seven-segment display segment (HEX3[5])
HEX3[6]	PIN_Y19	On-Board seven-segment display segment (HEX3[6])
CE	PIN_AF8	Chip Enable for SRAM – active low
UB	PIN_AC4	Upper Byte enable for SRAM– active low
LB	PIN_AD4	Lower Byte enable for SRAM – active low
OE	PIN_AD5	Output Enable for SRAM – active low
WE	PIN_AE8	Write Enable for SRAM – active low
Data[0]	PIN_AH3	Bidirectional Data Bus for SRAM - bit 0

Data[1]	PIN_AF4	Bidirectional Data Bus for SRAM - bit 1
Data[2]	PIN_AG4	Bidirectional Data Bus for SRAM - bit 2
Data[3]	PIN_AH4	Bidirectional Data Bus for SRAM - bit 3
Data[4]	PIN_AF6	Bidirectional Data Bus for SRAM - bit 4
Data[5]	PIN_AG6	Bidirectional Data Bus for SRAM - bit 5
Data[6]	PIN_AH6	Bidirectional Data Bus for SRAM - bit 6
Data[7]	PIN_AF7	Bidirectional Data Bus for SRAM - bit 7
Data[8]	PIN_AD1	Bidirectional Data Bus for SRAM - bit 8
Data[9]	PIN_AD2	Bidirectional Data Bus for SRAM - bit 9
Data[10]	PIN_AE2	Bidirectional Data Bus for SRAM - bit 10
Data[11]	PIN_AE1	Bidirectional Data Bus for SRAM - bit 11
Data[12]	PIN_AE3	Bidirectional Data Bus for SRAM - bit 12
Data[13]	PIN_AE4	Bidirectional Data Bus for SRAM - bit 13
Data[14]	PIN_AF3	Bidirectional Data Bus for SRAM - bit 14
Data[15]	PIN_AG3	Bidirectional Data Bus for SRAM - bit 15
ADDR[0]	PIN_AB7	SRAM address bit 0
ADDR[1]	PIN_AD7	SRAM address bit 1
ADDR[2]	PIN_AE7	SRAM address bit 2
ADDR[3]	PIN_AC7	SRAM address bit 3
ADDR[4]	PIN_AB6	SRAM address bit 4
ADDR[5]	PIN_AE6	SRAM address bit 5
ADDR[6]	PIN_AB5	SRAM address bit 6
ADDR[7]	PIN_AC5	SRAM address bit 7
ADDR[8]	PIN_AF5	SRAM address bit 8
ADDR[9]	PIN_T7	SRAM address bit 9
ADDR[10]	PIN_AF2	SRAM address bit 10
ADDR[11]	PIN_AD3	SRAM address bit 11
ADDR[12]	PIN_AB4	SRAM address bit 12
ADDR[13]	PIN_AC3	SRAM address bit 13
ADDR[14]	PIN_AA4	SRAM address bit 14
ADDR[15]	PIN_AB11	SRAM address bit 15
ADDR[16]	PIN_AC11	SRAM address bit 16
ADDR[17]	PIN_AB9	SRAM address bit 17
ADDR[18]	PIN_AB8	SRAM address bit 18
ADDR[19]	PIN_T8	SRAM address bit 19

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IQT.30-32 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 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?

VI. REPORT

You do NOT need to write a report after week 1. Instead you will write one report for the entirety of Lab 6.

In your lab report, should hand in the following:

- An introduction;
- Written description of the operation of your circuit;
- Written purpose and operation of each module, including the inputs/outputs of the modules;
- State diagram for your controller;
- Description of how the instruction sequencer/decoder works. Describe how the control signals from the instruction sequencer/decoder are used to perform proper operations in each state.
- Schematic block diagram with components, ports, and interconnections labeled;
- Annotated pre-lab simulation waveforms. The simulation waveforms should contain at least the complete simulations for one instruction from each of the three instruction categories (ADD/ADDi/AND/ANDi/NOT; MR/JMP/JSR; LDR/STR);
- Design stats ("Total Logic Elements", "Total Registers" and "Max operating frequency")
- Answers to post-lab questions;
- A conclusion regarding what worked and what didn't, with explanations of any possible causes and the potential remedies.