ECE 385

EXPERIMENT #5

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.

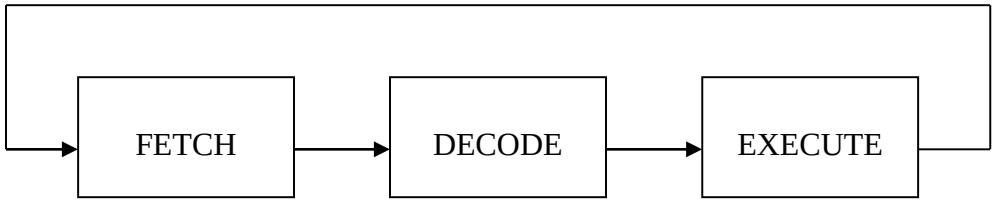


Figure 1

The CPU will contain a PC, an Instruction Register (IR), a Memory Address Register (MAR), a Memory Data Register (MDR), an Instruction Sequencer/Decoder, a status register (NZP), an 8x16 general-purpose register file, an Arithmetic Logic Unit (ALU), the CPU bus which interconnects all the modules, along with the necessary multiplexors within the data-path. All registers and instructions are 16-bits wide. The ALU will operate on 16-bit values and being that the SLC-3 is a RISC processor, the values will either come from the register file or are immediate values from the instruction register. 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 IR[15:12]; the remaining twelve bits contain data relevant to 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 slated to get some other value during execution (e.g., jump or branch). 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 data-path 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 instruction 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 and the register file should be reset to zero upon a reset, and the PC will proceed to 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, take 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 un-pause 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.

<u>Here are the operations in more detail:</u>

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 incrementor/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 Urbana board is equipped with a Spartan-7 FPGA chip with 2700Kb of on-chip memory (Block RAMs or BRAMs). 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. Consult the IPM (Instantiating IP Modules) document for information on how to instantiate the on-chip memory into your Vivado project. The interface for the on-chip memory chip is as follows:

wdata/ rdata	16-bit data bus in/out from the RAM
addr	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)
ena	Read enable. When active, allows read operations. Active high.
wren	Write enable. When active, allows write operations. Active high.
clk	Clock signal, FPGA BRAMs are synchronous.
reset	Reset memory and reload RAM contents from instantiate_ram

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 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 uses 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 cpu_to_io module does) (see figures 2 & 3)

CPU_TO_IO

This manages all I/O with the Urbana board's 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"**
Urbana Board Hex Display	Output	0xFFFF	Hex Display Data
Urbana 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 cpu_to_io entity, and one HexDriver that drives four 8-segment displays. The CPU (cpu.sv) is a high-level entity that contains most of your modules, including the control unit. The various memory control signal inputs and the memory address input should be connected to the corresponding outputs from the CPU (which then via the actual top-level entity to the memory subsystem). This lab uses on-chip memory, thus, there are 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.

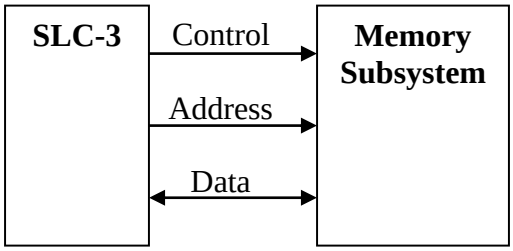


Figure 2: Organization of processor_top.sv (top-level entity)

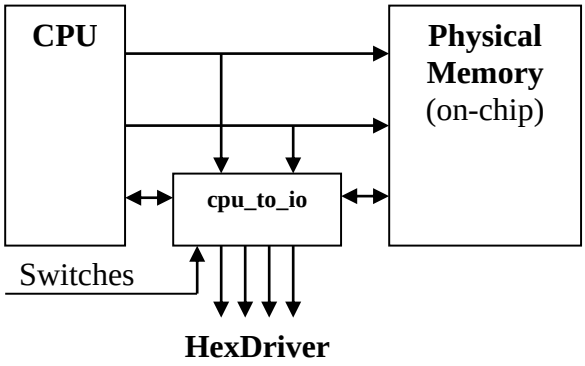


Figure 3: Organization of cpu_to_io unit

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

sw_i - logic [15:0]

clk, reset, run_i, continue_i - logic

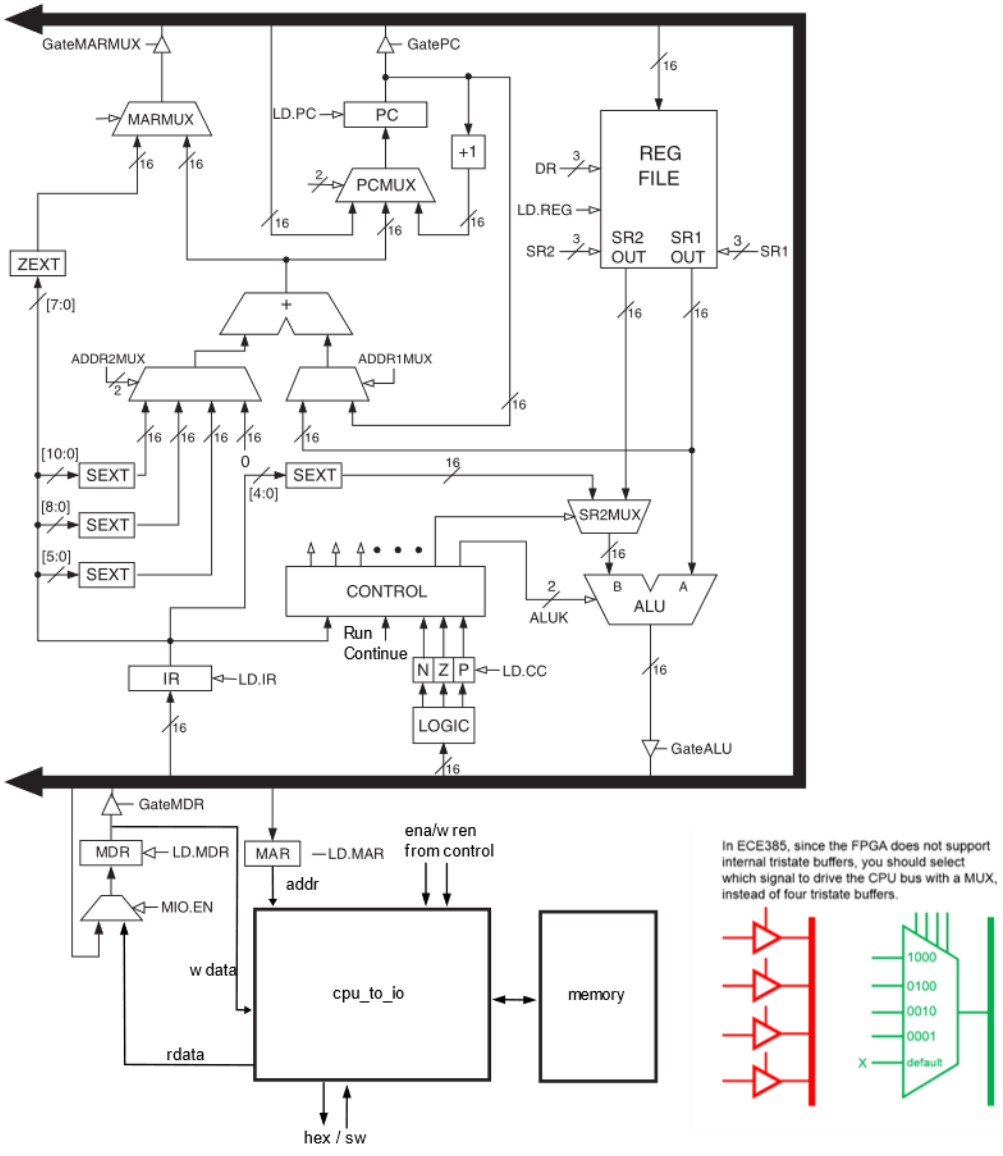
Outputs

led_o – logic [15:0]
hex_grid_left - logic [3:0]
hex_seg_left - logic [7:0]
hex_grid_right - logic [3:0]
hex_seg_right - logic [7:0]

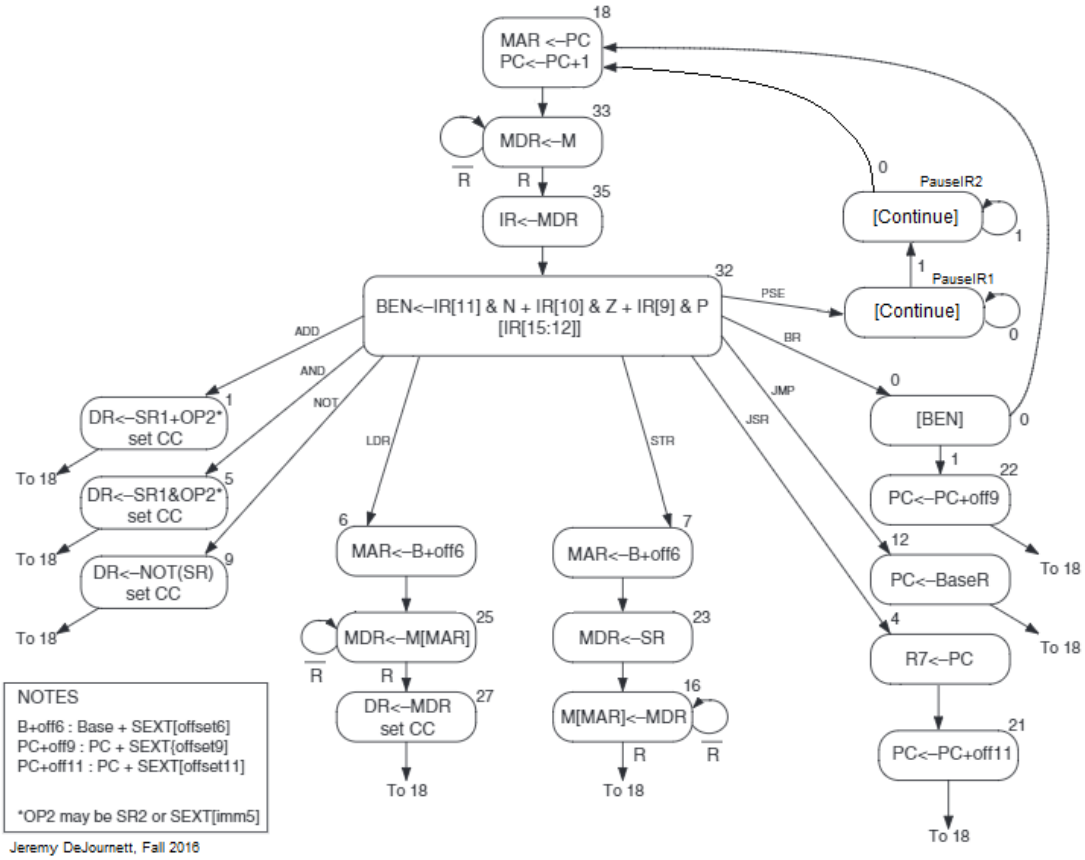
(You may expand this list as needed for simulation and debugging.)

5.8

SLC3 BLOCK DIAGRAM FROM APPENDIX C OF PATT AND PATEL



SLC3 STATE DIAGRAM FROM APPENDIX C OF PATT AND PATEL



III. PRE-LAB

A. **Week 1:**

Lab 5 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 control unit controls to be able to successfully fetch the instructions line by line from the on-board memory to the CPU. Note that since you will not be doing DECODE and EXECUTE during week 1, you do not have to pass the fetched instructions into the control unit. 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 cpu_to_IO, Test_Memory, and control unit.

For the Week 1 demo, you should connect the HEX displays directly to the IR rather than connecting them to the cpu_to_IO (in contrast, in week 2, you will specify a special address in memory that you write 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 control unit should loop back to perform the FETCH phase all over again, instead of continuing onto the DECODE phase.

You should modify control unit to go directly from state 35 to PauseIR1 allows you to inspect each instruction on the hardware while waiting for Continue. On the hardware, you may also get some skipped values, due to the Continue switch not being debounced – this is acceptable for the demo and should not affect your week 2 circuit. In addition, 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 for week 2.

Week 1 Demo Point Breakdown:

See the course website for the point breakdowns.

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 control unit 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 specifications 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 control unit to continue onto the DECODE and the EXECUTE phase instead.

Week 2 Demo Point Breakdown:

See the course website for the point breakdowns.

IV. LAB

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

Pin Assignment Table

Starting with this lab, use the provided pin assignments files (.xdc) rather than manually entering the pin-assignments by hand.

V. POST-LAB

1.) Refer to the Design Resources and Statistics in IVT and complete the following design statistics table.

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

Document any problems you encountered and your solutions to them, and a short conclusion. Before you leave your lab session submit your latest project code (.sv files only) as you have with the previous labs.

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

- What is CPU_TO_IO 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 impact does this have on performance?

VI. REPORT

You do NOT need to write a report after week 1. Instead, you will write one report for the entirety of Lab 5. 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.
    b. Describe how the SLC-3 differs from the LC-3 processor you learned in ECE 120/220.
2. Written Description and Diagrams of SLC-3
    a. Summary of Operation
    b. Describe in words how the SLC-3 performs its functions. You should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.
    c. Block diagram of cpu.sv, which includes the main data-path and control unit for your CPU.
        i. provided lab materials may be used as a starting point.
        ii. This diagram should represent the placement of all your modules in the cpu.sv. Please only include the cpu.sv diagram and not the RTL view of every module (module internals can go into the module descriptions).
    d. Like we described in class, annotate graphically on the block diagram how each different execution stage operates. This will likely be multiple versions of the block diagram from 2.c with separate annotations.
    e. Block diagram of the top-level (processor_top.sv), which includes connections between the cpu, as well as the cpu_to_io, memory, and other modules.
    f. Written Description of all .sv modules
        i. A guide on how to do this was shown in the Lab 2.2 report outline.
        ii. Multiple instantiations of the same module (e.g. 16-bit register) may share a single description, if all instantiations are listed.
    g. Description of the operation of the control unit
        i. Named control.sv, this is the control unit for the SLC-3. Describe in words how the control unit 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 control.sv.
    h. State Diagram of Control Unit

   i. This should represent all states present in the control unit and their transitions.
   ii. 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. For example, how did you modify the FSM to account for the lack of an ‘R’ signal in the RAM. You will lose points if you directly copy the diagram without modifications.

3. Simulations of SLC-3 Instructions
   a. Simulate the completion of all 7 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier, Auto Count, 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.
   c. Note that for simulation traces which are long, you may truncate out the intermediate portions of the program. For example, you will likely need to do this so that your Sort test simulation is legible.
4. Post-Lab Questions
   a. Fill out the Design Resources and Statistics table.
   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.