ECE 385

Spring 2020

Experiment # 6

Simple Computer SLC-3.2 in SystemVerilog

Megh Shah, Ishaan Patel
Lab Section: ABC, Tuesday 11:30 AM
TA: Gene Shiue, David Zhang

Introduction:

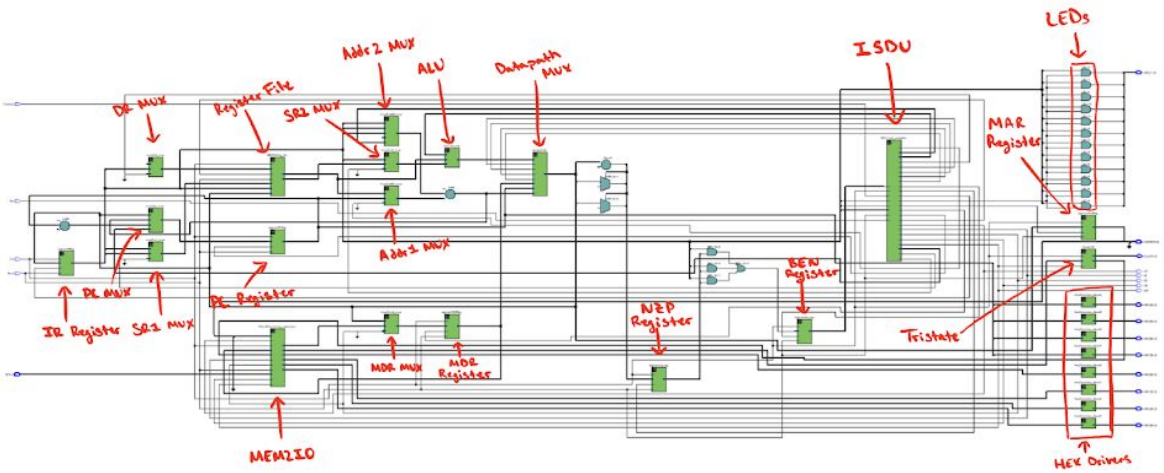
The SLC-3 is essentially a miniature and more simpler version of a computer. It possesses many logic elements, such as MUXes and adders and registers which are used coherently to perform operations on data. Some of the operations, or instructions, the processor is capable of is loading and storing values into registers, performing logical operations on values, and performing algorithms, such as sorting values in a given memory location. Overall, by incorporating these elements and devices together, we can perform many different operations and tasks regarding memory, data, and I/O. By doing this, we believed that we delved deeper into the operations and functionality of a computer and processor as we built each element one-by-one, and in the end, combined everything to create a functioning computer.

Written Description and Diagrams of SLC-3:

a. Summary of Operation - The SLC-3 processor operates by following the FETCH-DECODE-EXECUTE cycle. In this cycle, there are three different stages the processor goes through in order to complete the operation of a given task or instruction. The first stage, FETCH, essentially grabs the 16-bit instruction that the processor will execute by storing the value of the program counter, or PC, into the MAR, and then taking the value in that location as that represents the 16-bit instruction. Essentially, the processor looks at the correct memory location, reads the 16-bit instruction, and stores it into the instruction register, or IR. It also increments the program counter, PC, by one. The next stage, DECODE, takes the 16-bit instruction that was previously fetched, and begins to decode or understand the 16-bits of the instruction. The four most significant bits represent the OPCODE which is used to determine which operation the processor will execute. The bits after that can represent the source and/or destination registers, signed values to be added, or offsets to change the value of a certain register. The DECODE stage essentially takes these bits, and sends them as control signals to logic elements in the processor which will turn on or off certain paths and devices, so in the end, only the correct devices and paths and registers that needed to be accessed and used will be turned on in the processor. The final stage, EXECUTE, performs the actual operation. Since the correct devices and paths are turned on, the EXECUTE stage takes the needed values or data, from registers or memory location, and passes them through the datapath and devices which will perform the correct logic based on the previously sent control signals. After this stage is done, the program loops back to the FETCH stage again to grab the next 16-bit instruction, and perform the cycle all over again.

b. **SEE PART A.**

c. Block Diagram of slc3.sv -



d. **SEE PART C.**

e. Written Description of all .sv Modules -

i. **HexDriver.sv** - Takes in four-bit input and calculates a 7-bit output which can be displayed on the HEX displays.

ii. **Tristate.sv** - Connection between the SRAM and MEM2IO to ensure that there is only one set of data being transmitted and in only one direction at one time.

iii. **test_memory.sv** - Used to mimic an SRAM IC chip. Uses the memory contents in memory_contents.sv to create an array of memory locations filled with 16-bit instructions which can be used to represent the same function as an SRAM chip.

iv. **SLC3_2.sv** - Contains constants and functions which can be used for convenience as we do not have to specify specific bits, but can specify the names of the stages or instructions. It is needed for memory_contents.sv as it uses these constants and functions.

v. **slc3.sv** - The core of the processor as it contains registers, datapaths, gates, MUXes, adders, and other logical elements. This file contains most of the logical elements needed for the processor to use when executing the FETCH-DECODE-EXECUTE cycle.

vi. **memory_contents.sv** - Contains different 16-bit instructions which all represent different operations. Used to mimic the RAM file.

vii. **mem2IO.sv** - Bidirectional as it can send and take in data from the CPU. The main purpose is to take data or information provided by the CPU or the I/O on the FPGA board and send it to the other one.

viii. **lab6_toplevel.sv** - Serves as the top-level entity as it connects the processor and all other necessary files and I/O together to be compiled.

ix. **ISDU.sv** - Contains the state machine of the SLC-3 processor. This file contains all the various states for the FETCH-DECODE-EXECUTE cycle, and specifies exactly which logic elements are being set to on or off or are getting certain control signals. Certain operations have multiple states as there are multiple tasks that are needed to be done while others have dummy states which are needed to stall enough time for memory reading or writing to be completed. This file controls the various logic elements as it determines what exact operation is being performed by looking at the 16-bit instruction, and setting the correct logic devices to on or off .

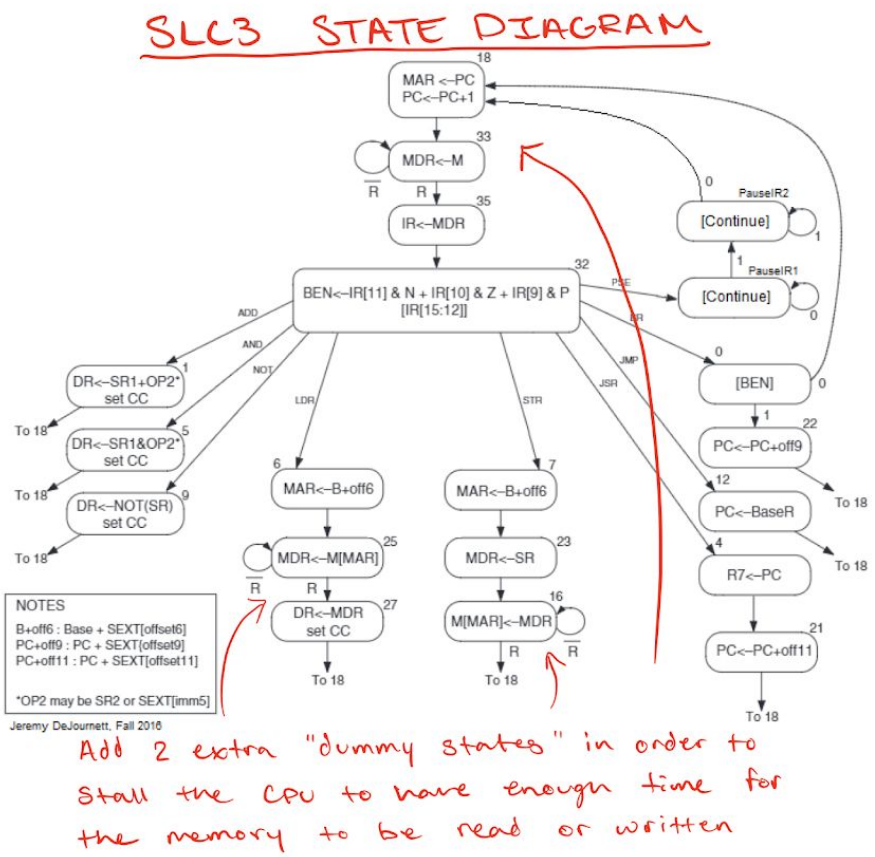
x. **datapath.sv** - Mimics a tristate buffer by using a 4-to-1 MUX. Determines which of the four gates needs to be opened so the datapath can take in the 16-bits of that specific path.

xi. **register.sv** - Creates the register file whose purpose is to hold eight registers which can be used to store data and instructions. These registers can be accessed using MUXes.

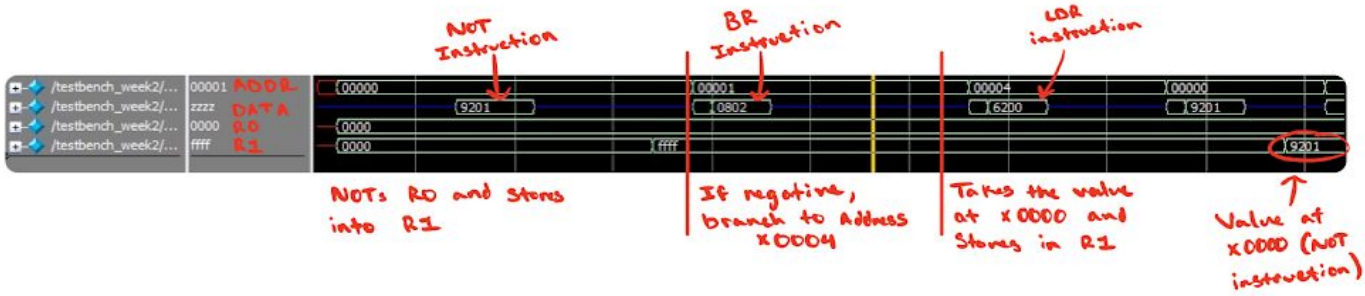
xii. **testbench.sv** - A test file whose values can be changed to mimic an actual processor with actual values. Creates a simulation waveform which can be decoded to see exact values in registers and memory locations.

f. **SEE PART E.ix.**

g. State Diagram of ISDU -



Simulation of SLC-3 Instructions:



For the simulation, we used the NOT, BRn and LDR instructions. First, we NOT R0 and store it in R1(which is now xFFFF). Then we branch to line memory address x0004 if R1 is negative (which it is). The instruction in x0004 is LDR which stores M[R0] into R1. M[R0] is the NOT instruction (x9201). As you can see R1 has x9201 thus showing the three instructions working.

Post-Lab Questions:

LUT	700
DSP	0
Memory (BRAM)	0
Flip-Flop	268
Frequency	69.34 Mhz
Static Power	98.65 mW
Dynamic Power	7.58 mW
Total Power	175.80 mW

The only problem that we had was with our sort function. We for some reason couldn't get the branch to work when we called the LDR opcode. Upon further analysis, we didn't set CC in one of the LDR states. Once we added that command our sort function worked.

2.

- What is MEM2IO used for, i.e. what is its main function?
    - MEM2IO is used for when we want to write to the hex display or take switch input. The way it does it is by checking if the address is xFFFF and if it is then we don't use SRAM and use either the HEX display or switches.
- What is the difference between BR and JMP instructions?
    - Branch will do PC<-PC+offset9 depending on the value of the BEN register. This is different from the JMP instruction which will make pc point to any 16 bit memory address regardless of setCC.
- 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?
    - The R signal is used to check when either a read or store to memory is done. The memory module would change the R bit to 1 when either operation has been completed, this would move the PC to the next state. We compensate for the lack of this signal by extending a memory read or store state. We know that it would take two states for the SRAM to finish thus we can just double the memory states. Not waiting for a R signal makes our design fully synchronous.

Conclusion:

The functionality of our design was very straightforward. We did not have many issues or errors since we prepared well beforehand. One issue that we did run into was when testing the SORT function on the FPGA board, our board was not able to go back to the menu after choosing a submenu. We discovered that this was because in the state machine for our LDR operation, we forgot to set LD.CC equal to 1. After fixing this, we were able to test our SORT function and discover that it worked as intended. Regarding the lab manual, there was nothing that was very unclear or ambiguous. Even though we had to read the lab manual a few times, we were eventually able to figure out everything and understand how we were going to approach and complete this lab. Overall, we believe that by completing this lab, we learnt a lot more and gain a deeper understanding regarding the operation and creation of computers and processors.