**ECE 385**

Spring 2023

Experiment #5

**Simple Computer SLC 3.2 in SystemVerilog**

Noah DuVal, Jason Wright

JZ / Friday 3:00 pm

Jason Zhu

**Introduction**

This week's experiment involved the creation of a SLC-3.2 microprocessor using SystemVerilog and C programming. It utilizes a 16-bit processor, 16-bit Program Counter, and 16-bit registers to implement 16-bit instructions. The microprocessor is able to complete a list of instructions that includes ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, and PAUSE. These instructions are then used to create numerous programs and test cases which cover all the instructions demonstrating their functionality.

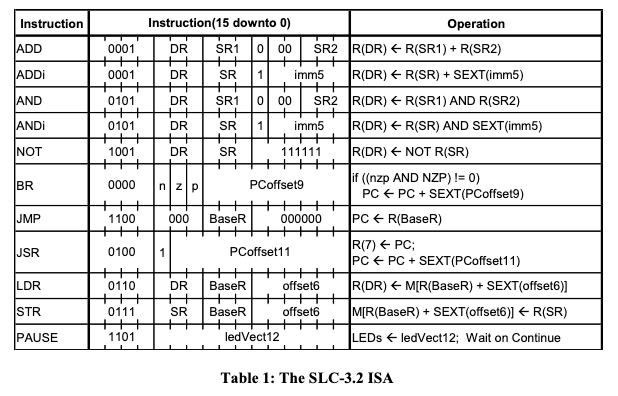
**Written Description**

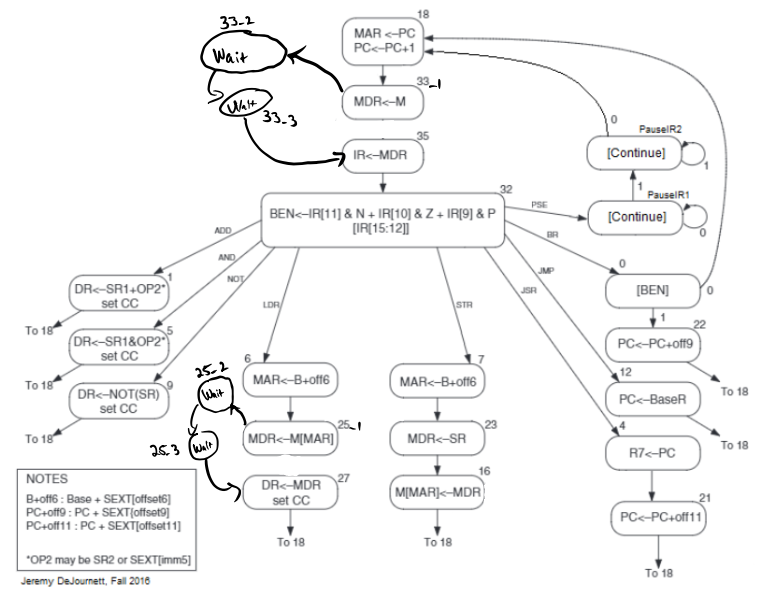
In order to use the SLC-3.2 microprocessor the desired starting instruction is selected using the switches. In the case of this experiment, the programs and test cases were already written in the memory\_contents.sv file. Once the desired test case is selected the user then presses RUN which in turn initializes the FETCH phase, the first of three. It will then move onto the DECODE phase and then to EXECUTE.

While in the FETCH phase the instruction will be sent to the BUS and loaded into the Program Counter (PC). The PC is then used to send data to the Memory Address Register (MAR) which stores the memory address of the data that is either going to be FETCHED or stored depending on the instruction. The data stored in the MAR is then sent to the Memory Data Register (MDR) which acts as a middle man storing a copy of the data at the address in the MAR. The final state in FETCH is to send the data in the MDR to the Instruction Register (IR) and increment the PC by 1. This will allow the microprocessor to move onto the next phase LOAD where it will set up the system for the final phase EXECUTE.

Next is the DECODE phase in which the data in IR is parsed and sent to their corresponding parts of the microprocessor based on the instruction. The first four bits (IR[15:12]) of the instruction are the OPCodes as it is sent to the control unit to determine which function to implement. The remaining 12-bits are used differently depending on the instruction being decoded as seen in table 1 below. They will contain a mix of what registers data is to be pulled from, where it should be stored, and PCoffsets of different sizes. Once the instruction is then broken down the processor will move on to the next phase. When in the final EXECUTE phase the desired operation is performed and the results are to be written into either the selected Destination Register (DR) or the appropriate memory location.

The way the states are controlled and performed is demonstrated in figure 1. This all takes place in the IDSU.sv module. This module contains all of the states and tells the system which states should follow the previous one. As well as what should go on in each state. It does this by using control signals to enable or disable the appropriate gates and MUXs so that the correct data is sent to and received from the BUS. It also makes sure that the correct values are sent to the MUXs. All in all it contains the full functioning control system of the microprocessor and enables the system to properly and efficiently perform the desired operations. This was programmed by manually figuring out the necessary gate and mux values needed for each state to perform its function.

**Table 1: The SLC-3.2 ISA**

**Figure 1: SLC-3.2 State Diagram**

**Module Descriptions**

Module: mux2

Inputs: s; [width-1:0] d0, d1

Outputs: [width-1:0] y

Description: Defining the width when calling allows the same module to be used for many multiplexers. S is the select bit which chooses which input d to put to output y.

Purpose: This module allows us to define a 2 by one mux for any data width without having to make separate modules.

Module: RegFile

Inputs: Clk, Reset, LD\_REG, [15:0] D; [2:0] sr1\_select, SR2, DR

Outputs: [15:0] SR2\_out, SR1\_out

Description: This contains a unique case setup that allows the sr1\_select and SR2 bits to choose which register contents are being output to the SR2\_out and SR1\_out elements. The DR bits choose which register’s contents will be modified when LD\_REG = 1. Additionally, this module calls reg\_16 a number of times to define each individual register.

Purpose: This module contains the necessary 8x3 muxes needed to implement the register select logic. In the same module we also define the registers R0 through R7 so they can be readily accessed.

Module: reg\_16

Inputs: Clk, Reset, Load, [15:0] D

Outputs: [15:0] Data\_Out

Description: This is a synchronous register. When load signal is 1 on a positive edge, the stored data Data\_Out is replaced by the input data D. A reset signal can also clear the Data\_Out when high.

Purpose: This module is used to define many 16 bit registers throughout the lab so that they can be accessed and loaded with data.

Module: ALU

Inputs: [1:0] ALUK; [15:0] SR2MMUX\_out, SR1\_out

Outputs: [15:0] ALU\_out

Description: This module is a 4x2 mux that is used to choose which operation (ADD, AND, NOT, PASS) is used on the two operands based on the ALUK select bits.

Purpose: This module is used to implement the operations needed by several opcodes. It takes the SR1 data as one input. The other input is determined by the SR2MUX and can either be SR2 or immediate data from the instruction, as determined by the SR2MUX outside of the module.

Module: ADDR\_Section

Inputs: ADDR1MUX, [1:0] ADDR2MUX, [15:0] ADDR1MUX\_out, ADDR2MUX\_out, SR1\_out, PC, IR

Outputs: [15:0] ADDR

Description: This file contains the various multiplexers and the adder on the left hand side of the block diagram. The one and two bit inputs are used as select variables to decide which variables get passed through the adder and onto either the PCMUX or the gateMARMUX.

Purpose: This module allows for the selection of inputs to the adder and allows the adder’s output to be accessed by other modules easily.

Module: test\_memory

Inputs: Reset, CLK, rden, wren, [15:0] data, [9:0] address

Outputs: [15:0] readout

Description: This code is used to create and define artificial memory locations for the simulation to access. The integration with outside modules is completed in the same way as on-board memory.

Purpose: This is necessary to simulate the test programs in modelsim by allowing the program to access memory locations the exact same way it would when implemented on hardware.

Module: synchronizers

Inputs: Clk, d

Outputs: q

Description: This module defines a flip flop that is synchronized with the system clock using an asynchronous input d.

Purpose: This is necessary for asynchronous signals (board buttons) to align with the clock to avoid bugs.

Module: SLC3\_2

Inputs: N/A

Outputs: N/A

Description: This file contains various function and parameter information

Purpose: This file is necessary for the memory\_contents program to function, as it contains predefined functions used to make programming within the memory contents easier by getting rid of the need to type certain lines multiple times and allowing the user to type in a traditional input format rather than needing to know the entire instruction format.

Module: slc3

Inputs: [9:0] SW; Clk, Reset, Run, Continue, [15:0] Data\_from\_SRAM

Outputs: [9:0] LED; OE, WE, [6:0] HEX0, HEX1, HEX2, HEX3; [15:0] ADDR, Data\_to\_SRAM

Description: This is the effective top level module. It contains calls to all other modules made by us and includes the connections between them all.

Purpose: This module is necessary to connect and combine all other modules in their intended manners. It also contains various logic elements between them, such as the branch enable logic that we determined did not need its own module.

Module: PCMUX\_MUX

Inputs: [1:0] PCMUX, [15:0] Bus, PC, ADDER

Outputs: [15:0] PCMUX\_out

Description: This is a 4x2 mux that uses the PCMUX bits to select what gets put to the output of the module.

Purpose: This is necessary for the PCMUX to determine whether the output is PC+1, Bus, or ADDER (Adder output)

Module: memory\_contents

Inputs: N/A

Outputs: N/A

Description: This is the file that contains the initial memory data to be loaded onto the RAM for slc3 operation. It is used in conjunction with SLC3\_2 which contains certain function definitions.

Purpose: This file is where any slc3 programming takes place, and it includes the instruction data for all test programs for this lab.

Module: Mem2IO

Inputs: Clk, Reset, OE, WE, [15:0] ADDR, Data\_from\_CPU, Data\_from\_SRAM, [9:0] Switches

Outputs: [15:0] Data\_to\_CPU, Data\_to\_SRAM, [3:0] HEX0, HEX1, HEX2, HEX3

Description: This module interfaces between the CPU we designed and the onboard memory. It contains datapaths and read/write enable signals required by the RAM to access data. It also includes the hex\_data connections that allow memory contents to be displayed.

Purpose: This module allows board I/O, specifically the switches, hex displays, and LEDs to interact with the SRAM and the CPU.

Module: ISDU

Inputs: Clk, Reset, Run, Continue, IR\_5, IR\_11, BEN, [3:0] Opcode

Outputs: LD\_MAR, LD\_MDR, LD\_IR, LD\_BEN, LD\_CC, LD\_REG, LD\_PC, LD\_LED, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, Mem\_OE, Mem\_WE, [1:0] PCMUX, ADDR2MUX, ALUK

Description: This is the state machine that the entire system runs off of. A more detailed description and purpose can be found in the written description of the system earlier in this report.

Module: Instantiateram

Inputs: Reset, Clk

Outputs: wren, [15:0] ADDR, data

Description: This module contains a small state system that initially loads the ram with the memory contents. It moves into the done state before any programs can be run.

Purpose: This module is needed to initialize the memory and allow other programs to be found in and accessed within the memory.

Module: HexDriver

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This is a single unique case statement that contains the data necessary to display a 4 bit number on the hex displays.

Purpose: This allows us to avoid doing conversions ourselves by simply inputting the number we want to display and getting out the hex information needed to display it.

Module: datapath

Inputs: GatePC, GateMDR, GateALU, GateMARMUX, [15:0] MDR, PC, ALU\_out, MARMUX\_out

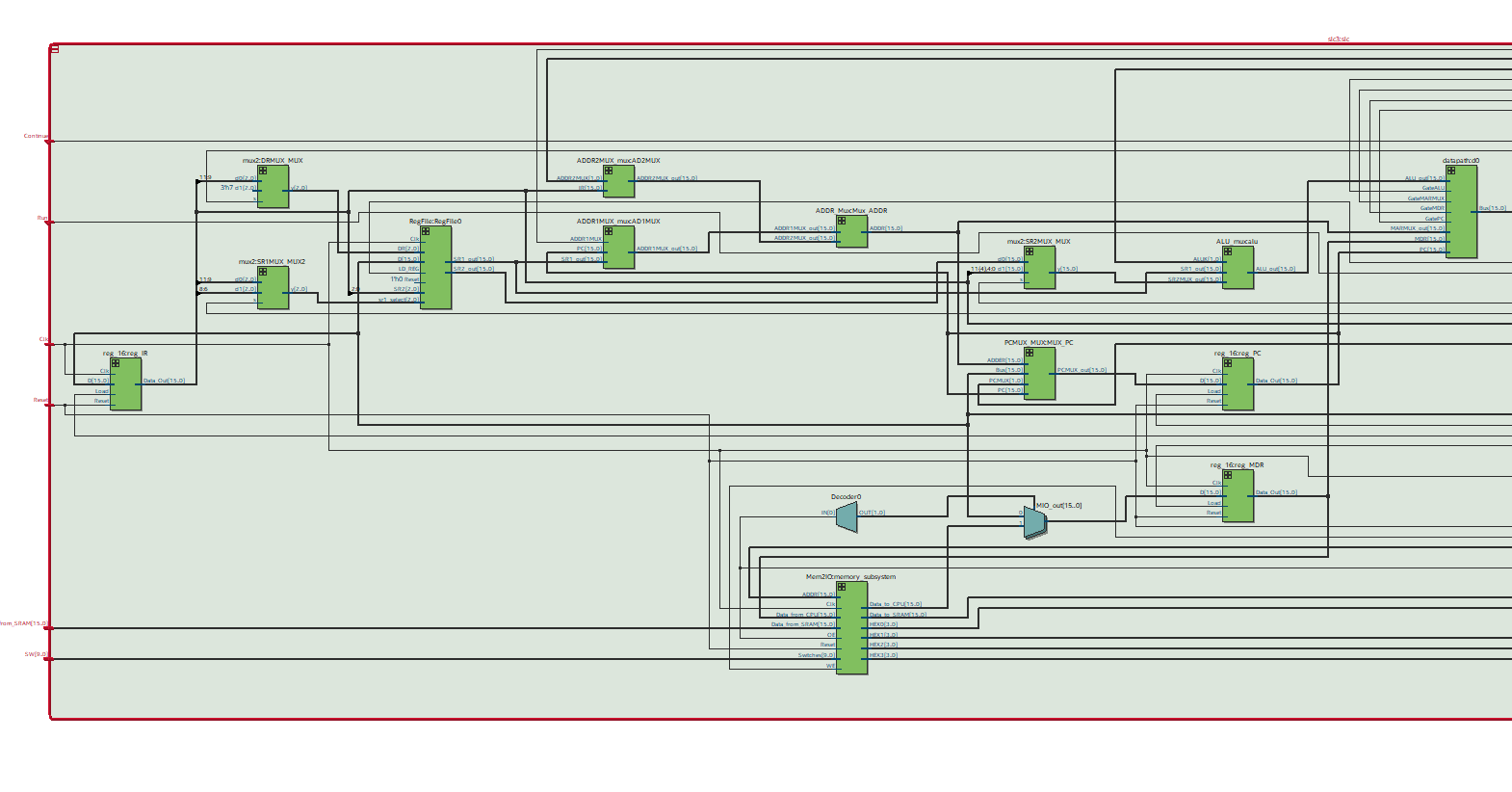
Outputs: [15:0] Bus

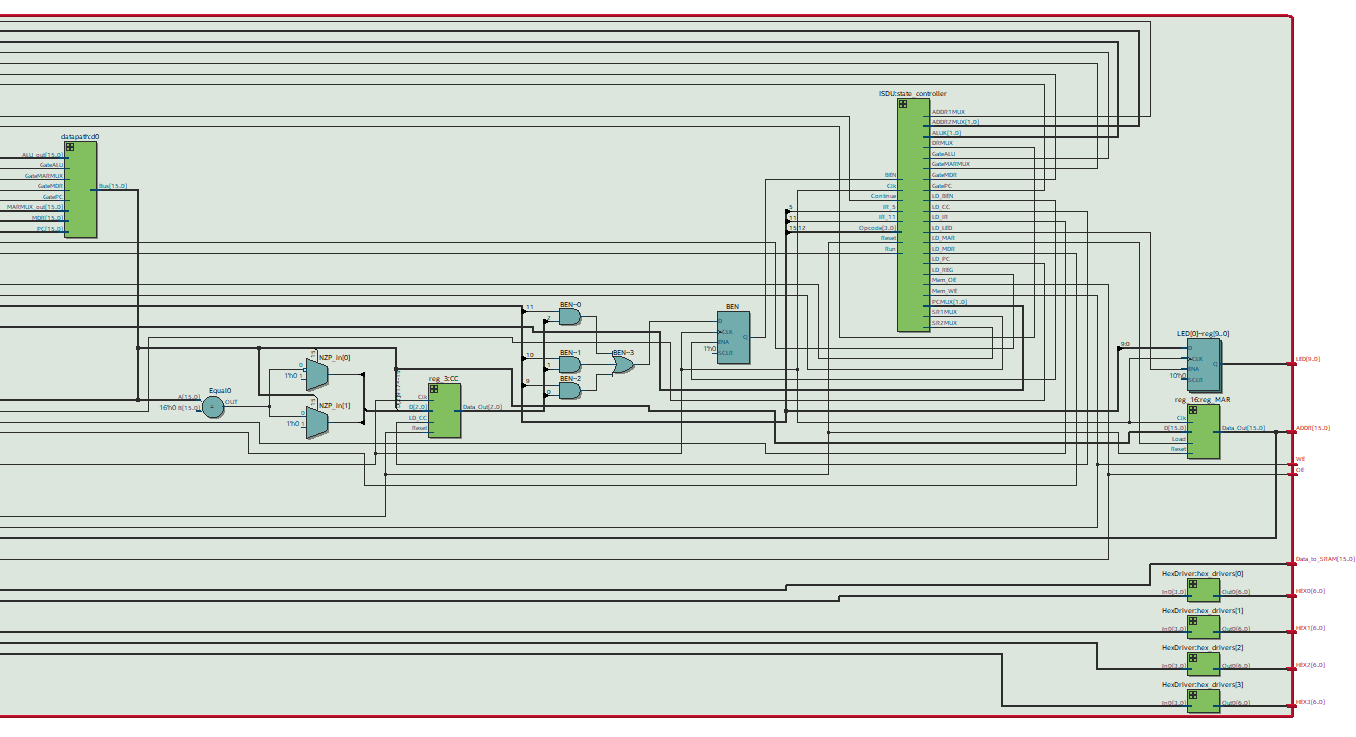
Description: This module is a unique case statement that allows only one of the input signals to reach the Bus output at a time depending on which Gate signal is active.

Purpose: This module replaces the tri-state buffers on the original LC3 by converting them into a multiplexer. The Bus output can then be accessed by many different modules to transfer the data around.

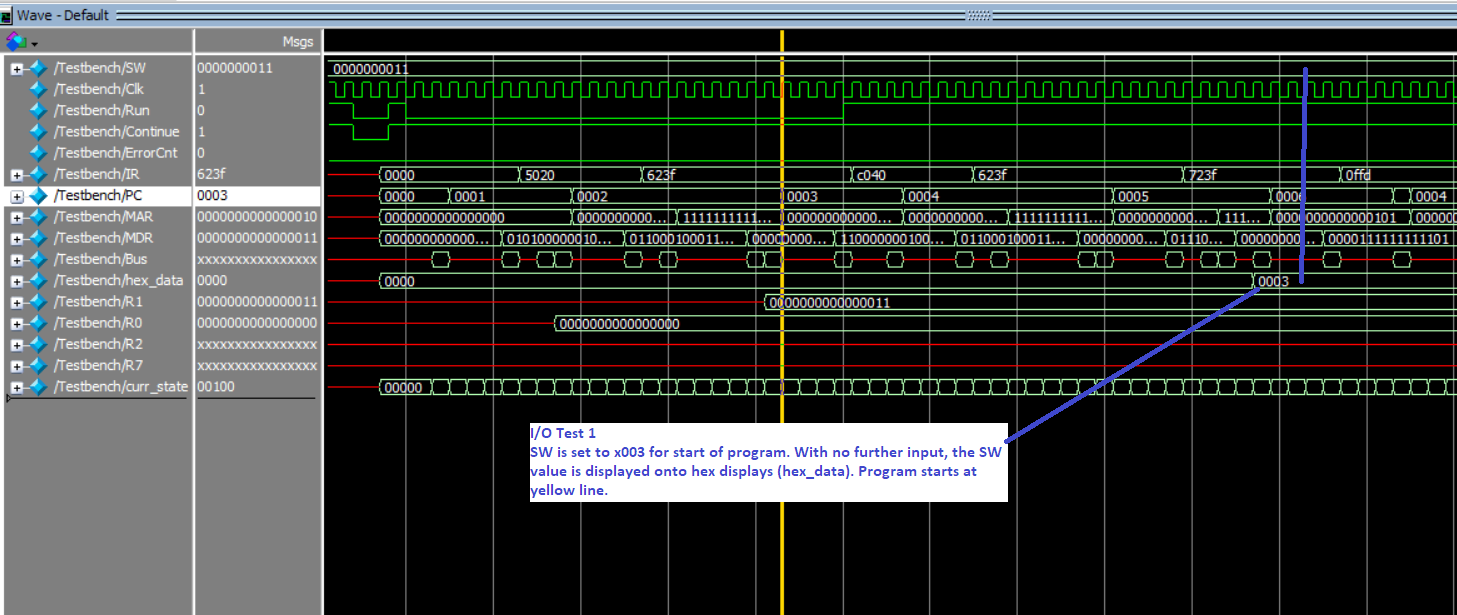
**Diagrams**

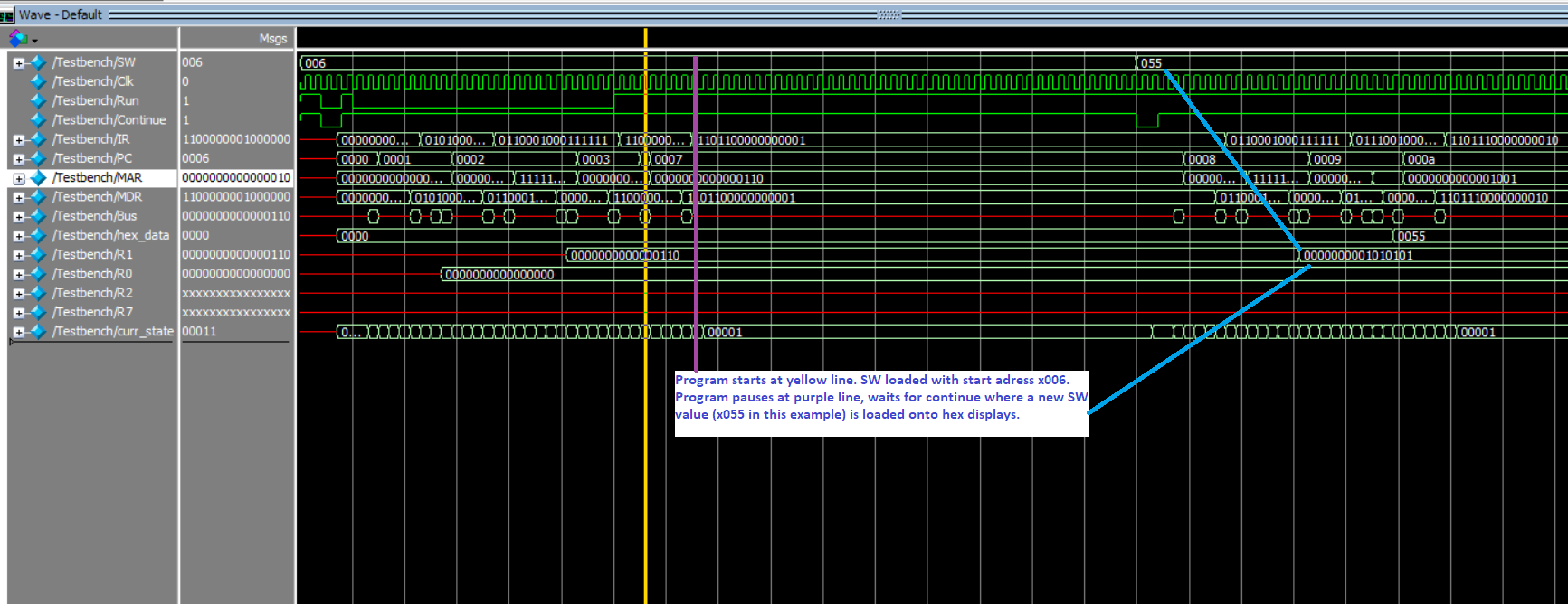
Below is the slc3 block diagram showing the placement of all modules within it. Because of how the RTL Viewer displayed the information, this is broken up into two halves. Both halves have the datapath: d0 module visible so that it can be seen how they connect together.

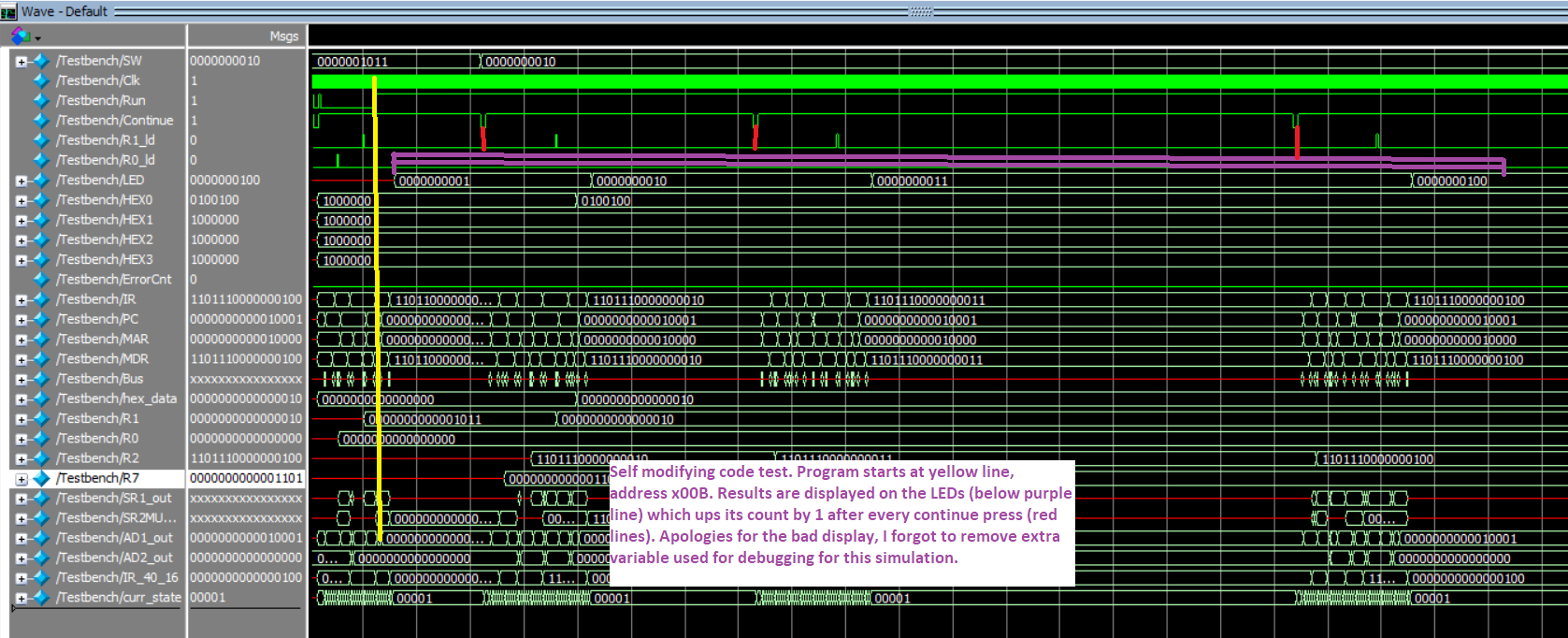
**Figure 2**

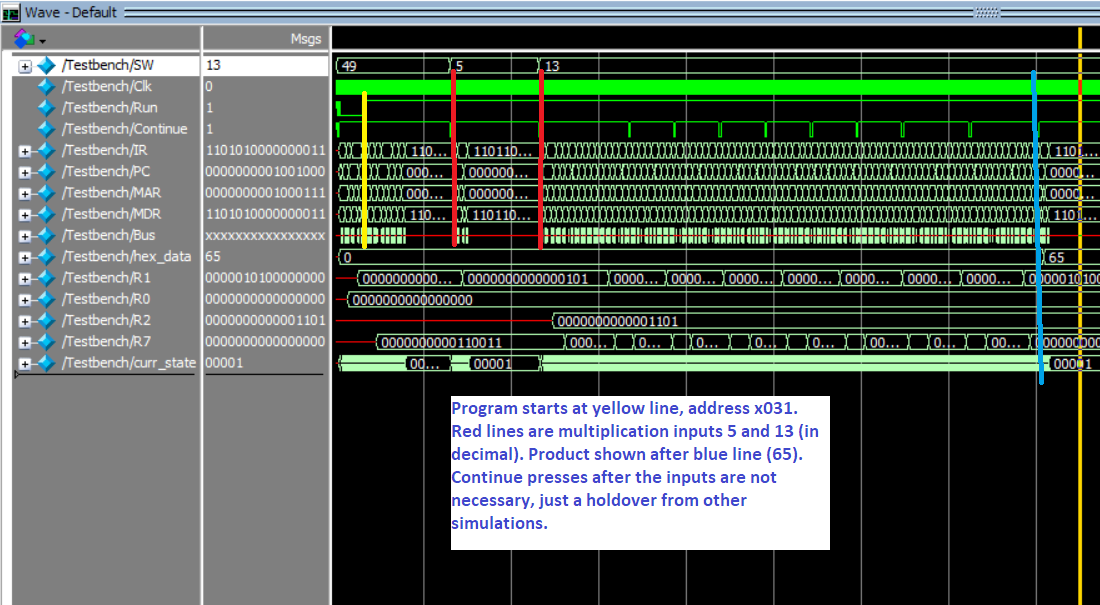
**Figure 3**

**Simulations**

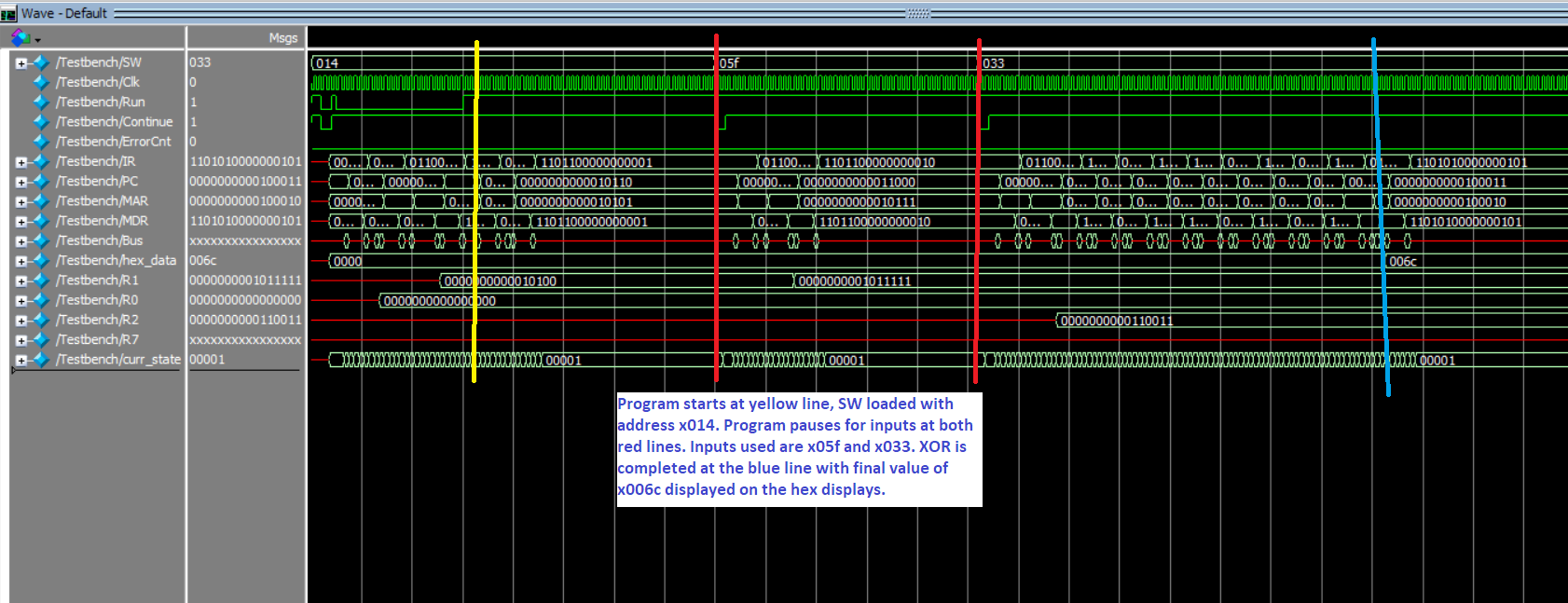
**Figure 4: I/O Test 1**

**Figure 5: I/O Test 2**

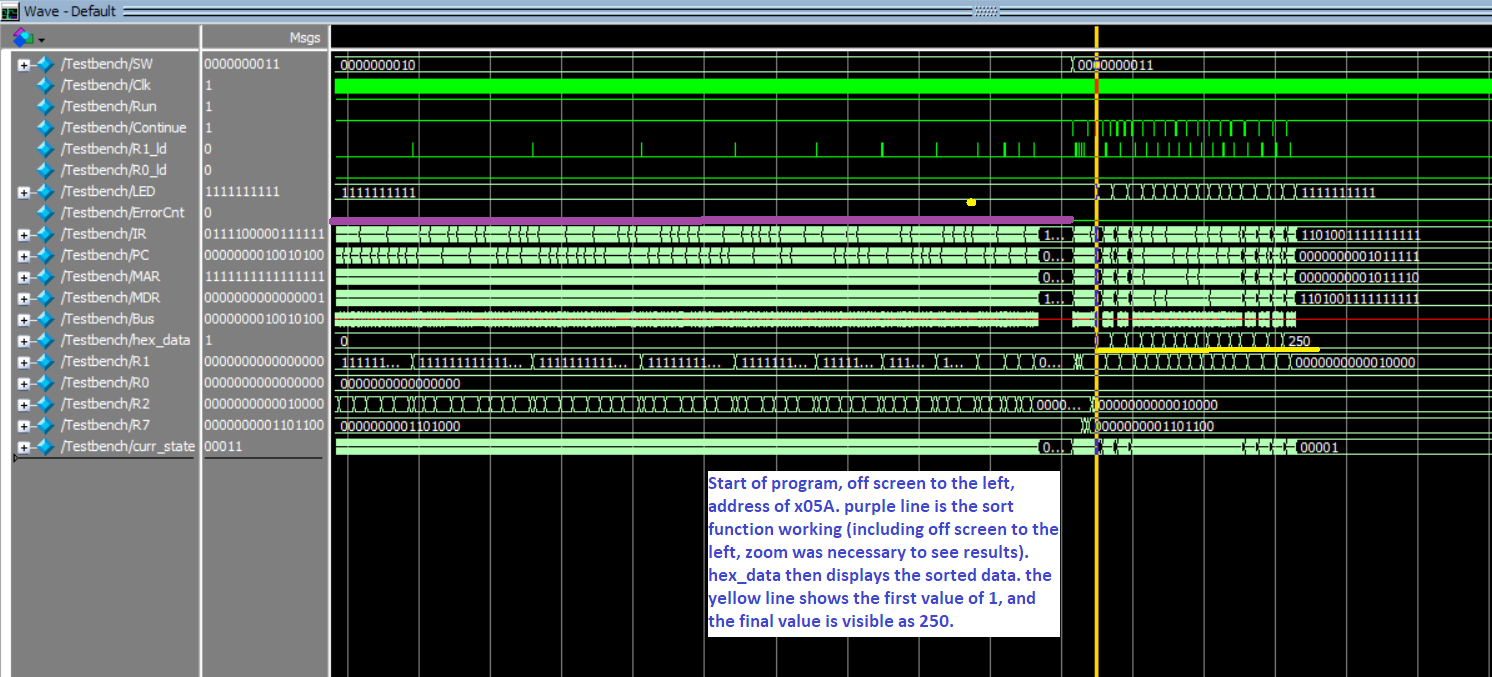
**Figure 6: Self Modifying Code Test**



**Figure 7: Multiplication Test**



**Figure 8: XOR Test**



**Figure 9: Sort Test**

**Post-Lab Questions**

| **LUT** | 911 |
| --- | --- |
| **DSP** | 0 |
| **Memory (BRAM)** | 16,384 bits |
| **Flip-Flop** | 267 |
| **Frequency** | 65 MHz |
| **Static Power** | 90.00 mW |
| **Dynamic Power** | 9.41 mW |
| **Total Power** | 110.74 mW |

**Table 2: Design Statistics**

Table 2: Design Statistics

Note on table 1: numbers were found using sramtop rather than testtop. Some numbers change between the two, for example, the LUT is only 526 for testtop.

MEM2IO is the interface between the memory unit and the CPU that we programmed. It makes the connections necessary for our CPU to access the on unit memory for reading and writing. Additionally, it allows the I/O, particularly the switches, to be used by accessing address xFFFF, which allows us to choose the program to run from the pre-written memory.

BR and JMP are different primarily for one reason: the return instruction. BR changes PC so that the next instruction can come from a different place in memory. It also can be used conditionally depending on the current NZP. JMP stores the current PC into register seven so that it can be returned to when required. This storage is useful when implementing custom functions in SLC-3.

Patt and Patel uses an R signal to signify when memory has successfully been read or written into. This signal is necessary so that the other states that rely on the memory contents do not activate prematurely and cause error. Our memory does not have a ready signal, so we get around this by adding artificial delay between states when reading. By delaying the next state by three clock cycles, we ensure that the memory contents have time to be read and used. This is not necessary when writing to memory because those memory contents will never need to be accessed within the three cycles it takes for them to be finalized.

**Conclusion**

Upon completion of this experiment the SLC-3.2 microprocessor was able to obtain full functionality. With that being said everything went fairly smooth in the construction of the microprocessor. However, in the instructions there were portions that got a little confusing as well as in the provided code. In other words there were a few parts in the instructions that made it somewhat difficult to decipher what was week 1 and what was week 2 in terms of what needed to be implemented. While that was not exactly an issue it was more of just an inconvenience as it added avoidable confusion. Another thing that was minor, but also worth looking into is the naming of variables in the provided code. As there were multiple instances where the naming of variables got redundant as they were not specific enough. This was mostly with the MUX’s as there were points where it was difficult to keep track of what was the MUX input, output, and the actual MUX itself. For example, most of the inputs to the MUX’s were labeled as the name of the MUX and not the MUX input. Thus, adding more confusion in the debugging stage and creating unneeded confusion.