University of Illinois at Urbana-Champaign

ECE 385

LAB 5

LAB REPORT

by Shubham Gupta & Devul Nahar (sg49 & danahar2)

3/20/2022

Introduction:

A processor has three essential components: the CPU, the memory that stores instructions and data, as well as an interface that allows communication between input/output devices. In this experiment, we will construct a 16-bit CPU in system Verilog known as the SLC3 (a subset of the LC3 ISA). The CPU oversees running the machine instruction cycle, which includes fetching data and instructions from the memory, decoding it in the control unit, and then executing it in the arithmetic/logic unit.

In particular, the CPU consists of a program counter register (PC), an instruction registers (IR), memory address register (MAR), a memory data register (MDR), an Instruction decoder, a status register (NZP), an 8x16 wide register file, and an arithmetic logic unit (ALU). PC stores the address of the next instruction to be executed. IR stores the address of the current instruction that is being executed. MAR stores the address of the memory block in which we want to read/write. MDR stores or the data to be written in or data read from the memory location pointed by MAR. The instruction decoder is responsible for decoding the 16-bit instruction into several parts such as opcode and its corresponding parts. The status register is responsible for adding conditional functionality into the SLC3. Register file holds 8 registers, each 16 bits wide. The register file is used to quickly access and store data. Finally, the ALU deals with arithmetic as well as logical operations responsible for manipulating data.

We implemented the entirety of this cycle in two parts. In the first week, we implemented the FETCH phase, in which we learned about the memory system as well as how memory interacts with the CPU. In addition, we dealt with CPU entities such as the ISDU control unit as well as built a basic data path to show the working of the FETCH phase. In the second week, we implemented the DECODE and the EXECUTE phases of all the instructions in the SLC3 ISA. We had to configure all the necessary state transitions as well as the inputs/outputs of each state.

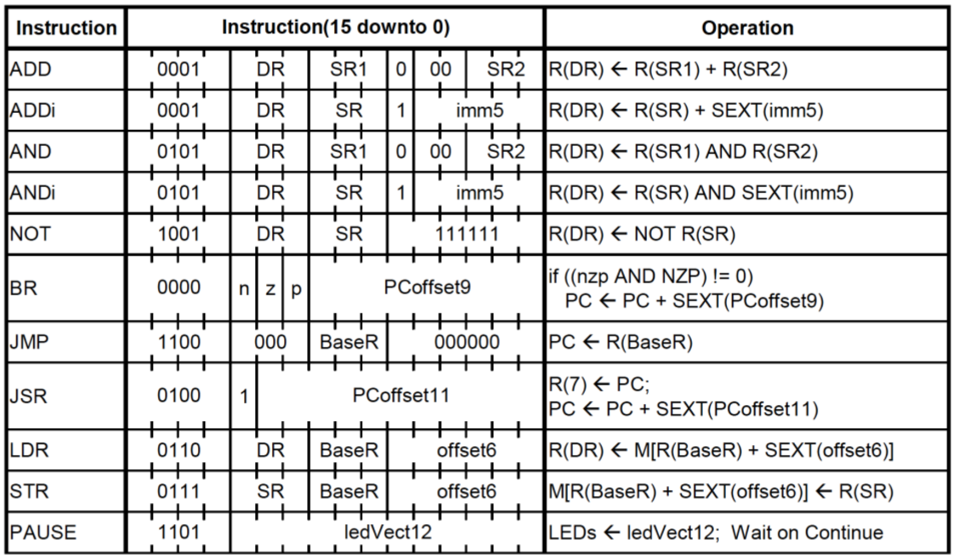
Written Description and Diagram for SLC-3:

a. **Summary of operation:** We first reset the machine by pressing CONTINUE and RUN together (note that both buttons are active low). To operate our SLC-3, we need to set the starting address of our instructions. To do this, we must set the address on the switches by hand that are present on the FPGA. Then to initialize it to the starting address we should push down the RUN key. Upon doing so we have initialized our SLC-3 and it is ready to execute the program. For example, to run I/O test 1, we set the switches to x0003. The push the RUN button. Now we can see that the program is running, whatever value we enter on the switches is displayed on the Hex Display. Similarly for programs that have a wait state, for example, I/O test 2, we must additionally press CONTINUE key to continue the execution of the process.

b. **Description of SLC-3 functions:**

For any instruction to be carried out, we need to first FETCH the instruction from the memory. The FETCH itself has 3 states that must be carried out. The states are managed by the ISDU.sv, we start from state 18 where we set control signals to set MAR to PC and PC to PC + 1. Next, we go to state 33, which basically reads data from memory at MAR and updates MDR. This is a 3-state process as we wait for the memory to finish its task of reading data. Now we go to state 35, where we set control signals to set IR to MDR. At this point we have finished the FETCH part. We DECODE our instruction in state 32 as it is present in IR. Here we set control signals such that we can update BEN. This is done by bit-slicing IR register and if statements in our code. We also get our Opcodes from IR [15:12] bits. The heart of our processor is the case statement. We use it to decide which state to enter next based on the extracted Opcode (implemented in ISDU.sv). This is the end of our DECODE. For the EXECUTE part, we have implemented the multi-state tree based on which the control signals are set to carry out various operations. In the last state of every operation, we always go back to state 18. This is how we ensure the cycle is complete.

The various functions our SLC-3 can perform are: Add, And, Not, Load, Store, Jump and Branch. We can perform two versions of ADD, the first one simply adds the value of source register 1 and source register 2, and updates the destination register with the sum. The second adds the content of source register to the sign extended value of the next 5 bits (immediate 5) and stores it to the destination register. We can perform two versions of AND, the first one simply ands the value of source register 1 and source register 2, and updates the destination register with the answer. The second ands the content of source register to the sign extended value of the next 5 bits (immediate 5) and stores it to the destination register. The NOT instruction simply negates the value of the source register and updates the destination register with the answer. We can conditionally choose to branch in our code, this is implemented by the BR operation. This is implemented by checking if any of the condition codes stored in the status register match the condition then a branch is taken. To do this, we simply update the PC to PC + sign extended value of the 9-bit offset provided in the instruction. We have two versions of jump statement, JMP and JSR. JMP simply copies the value of base register to the PC to perform an unconditional jump. Meanwhile JSR is used to jump to subroutines. Therefore, before performing the jump, we store the value of PC to register 7, and then update the PC to PC + sign extended value of offset of 11 bits provided in the instruction. By Load (LDR) operation we can simply load the value of memory stored at base register + an sign extended offset of 6 bits provided in the instruction, to the destination register. Similarly, we have a store statement that stores the content of the source register at the memory index base register + sign extended offset of 6 bits provided in the instruction. The status registers are also updated upon the execution of ADD, AND, NOT, and LDR.



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

Figure 1. SLC-3 Opcodes

c. Block Diagram:

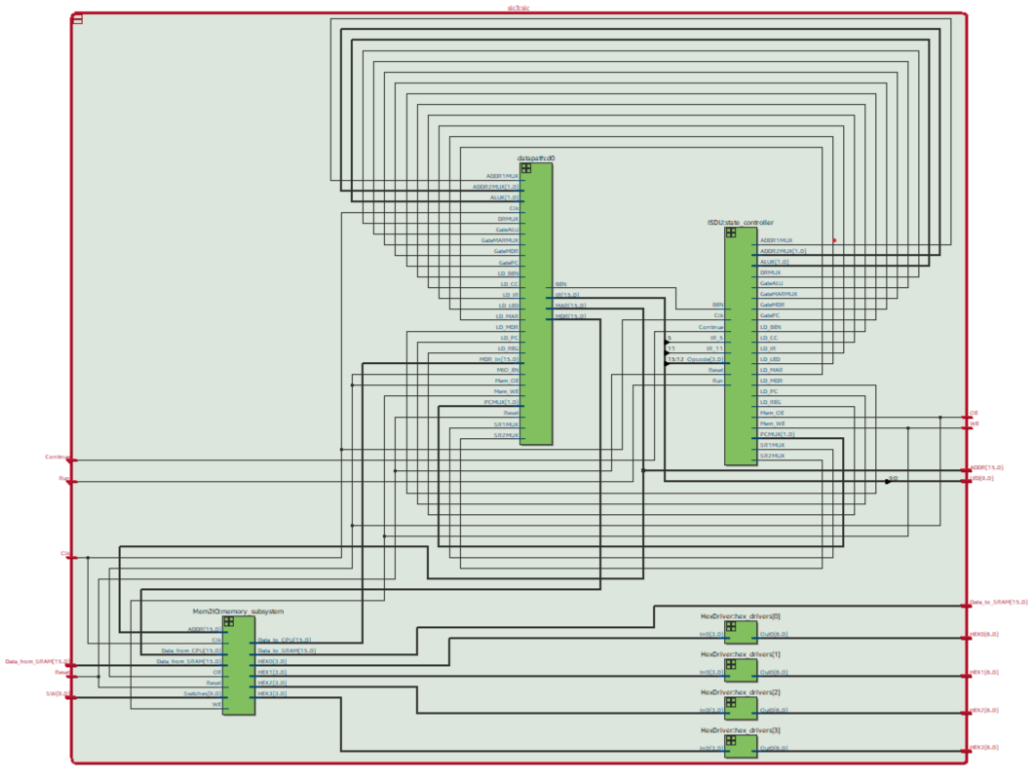


Figure 2. Block Diagram

d. Description of .SV modules:

i. <u>**Module:**</u> Reg1.sv

Inputs: Clk, Reset, Load, D

Outputs: Data_Out

Description: It is the implementation of 1-bit register using always_ff.

Purpose: It is used in various places in the data path to store 1 bit data, for example, in BEN register.

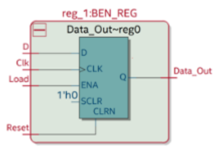


Figure 3. Reg1.sv

ii. **<u>Module:</u>** Reg3.sv

Inputs: Clk, Reset, Load, [2:0] D

Outputs: [2:0] Data_Out

Description: It is the implementation of 3-bit register using always_ff.

Purpose: It is used in various places in the data path to store 3-bit data, for example, in NZP register.

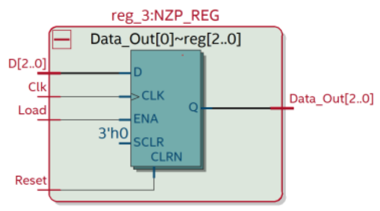


Figure 4. Reg3.sv

iii. **<u>Module:</u>** Reg16.sv

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

Outputs: [15:0] Data_Out

Description: It is the implementation of 3-bit register using always_ff.

Purpose: It is used in various places in the data path to store 3-bit data, for example, in PC register.

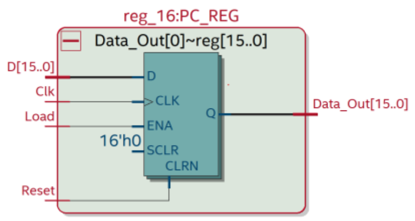


Figure 5. Reg16.sv

iv. **Module:** Mem2IO.sv

Inputs: Clk, Reset, [15:0] ADDR, OE, WE, [9:0] Switches, [15:0] Data_from_CPU, [15:0] Data_from_SRAM

Outputs: [15:0] Data_to_CPU, [15:0] Data_to_SRAM, [3:0] HEX0, [3:0] HEX1, [3:0] HEX2, [3:0] HEX3

Description: It is used to implement memory mapped IO in our design. Load data from switches when address is xffff, and from SRAM otherwise. Implemented using if statement.

Purpose: It is crucial as it maps reading and writing to xffff to the LEDs and Switches respectively.

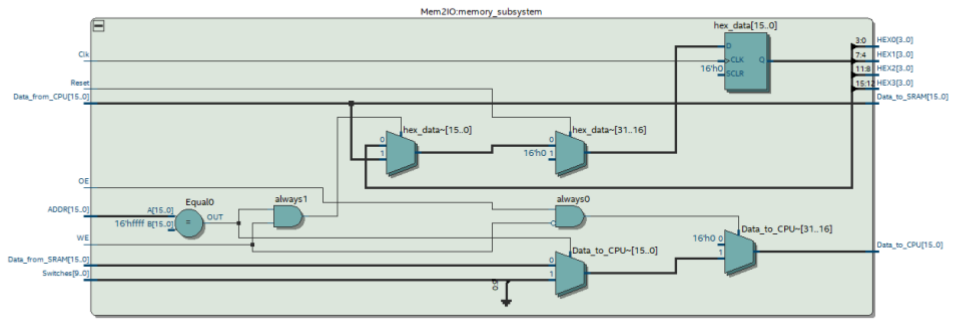


Figure 6. Mem2IO.sv

v. **Module:** HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: We use a UNIQUE switch case to map the binary values to the hexadecimal display. Given to us from previous labs.

Purpose: It maps the binary values to the hexadecimal LED display. Used for displaying values stored in registers. For user interface on the FPGA.

vi. <u>**Module:**</u> Datapath.sv

Inputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, [1:0] PCMUX, [1:0] ALUK, [1:0] DRMUX, [1:0] SR1MUX, [1:0] SR2MUX, [1:0] ADDR1MUX, [1:0] ADDR2MUX, Mem_OE, Mem_WE, MIO_EN, Reset, Clk, [15:0] MDR_In

Outputs: [15:0] IR, MAR, MDR, BEN

Description: It holds all the components of our SLC-3 and connects the inputs and outputs. It holds all gates, selection MUXes, registers, ALU, and the data bus. We have implemented gates using if/case statements. Registers are updated conditionally based on control signals by case statements.

Purpose: It is the heart of our SLC-3 implementation. It receives signals from the ISDU and connects them to all the gates and MUXes to perform operations.

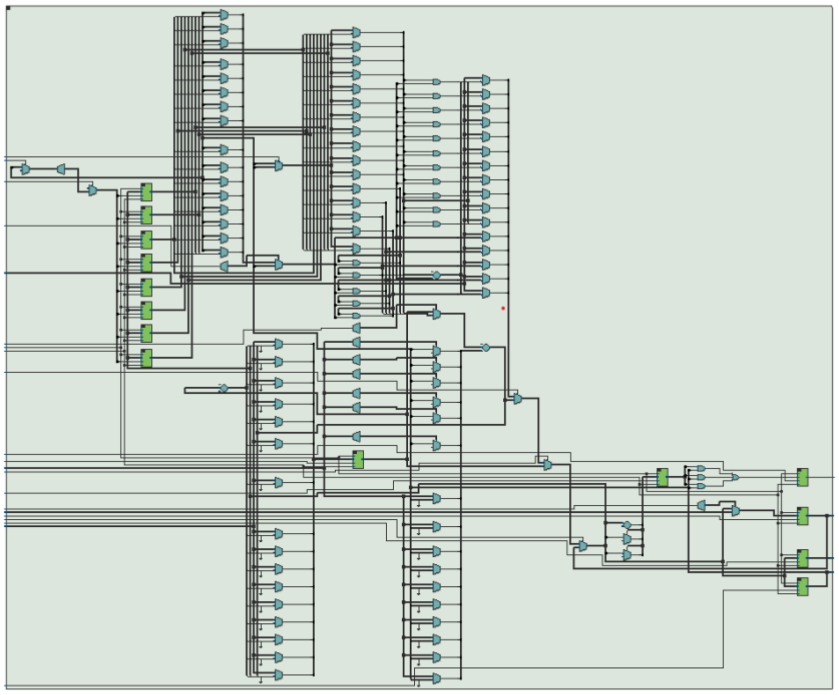


Figure 7. datapath.sv

vii. **Module:** ram.v

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

Outputs: [15:0] q

Description: It is an IP generated by Quartus, to implement the ram on board the FPGA.

Purpose: This module implements the RAM functionality on the SRAM present on the FPGA.

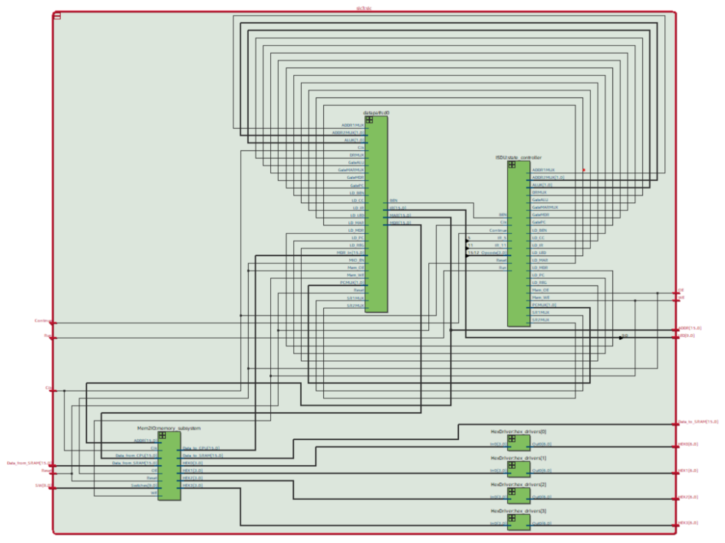
viii. **Module:** slc3.sv

Inputs: [9:0] SW, Clk, Reset, Run, Continue, [9:0] LED, [15:0] Data_from_SRAM

Outputs: OE, WE, [6:0] HEX0, HEX1, HEX2, HEX3, [15:0] ADDR, [15:0] Data_to_SRAM

Description: We initialize the instances of Mem2IO, data path and ISDU. We also create instances of hex drivers for the hex display here.

Purpose: It is the top-level entity of our on-board design. It connects the inputs and outputs of data path, ISDU and Mem2IO. Also connects the hex displays.



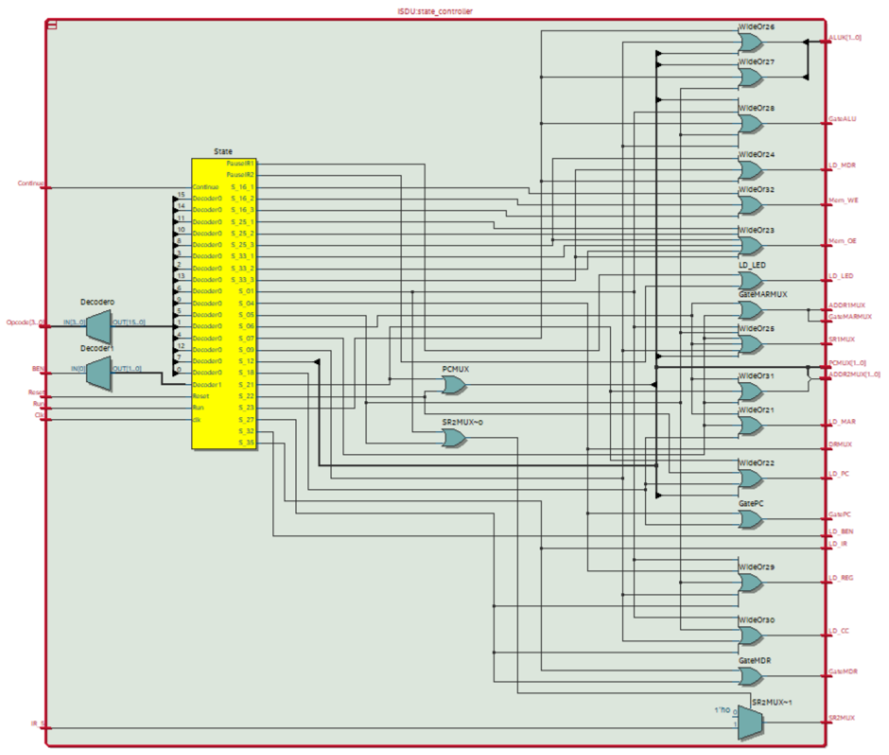
ix. **Module:** ISDU.sv

Inputs: Clk, Reset, Run, Continue, [3:0] Opcode, IR_5, IR_11, BEN

Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, [1:0] PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, ALUK, Mem_OE, Mem_WE

Description: This module implements the FSM for our SLC-3. We have implemented the FSM using case statements. Based on our CURRENT_STATE, we use a CASE, then send out the control signals respectively. After which we set the NEXT_STATE variable to the following state. This updates every cycle.

Purpose: It is the heart of our design; all the control signals are generated via this module based on the current state. This module controls the FSM of the entire SLC-3, manages the next state and current state.



e. State Diagram of ISDU:

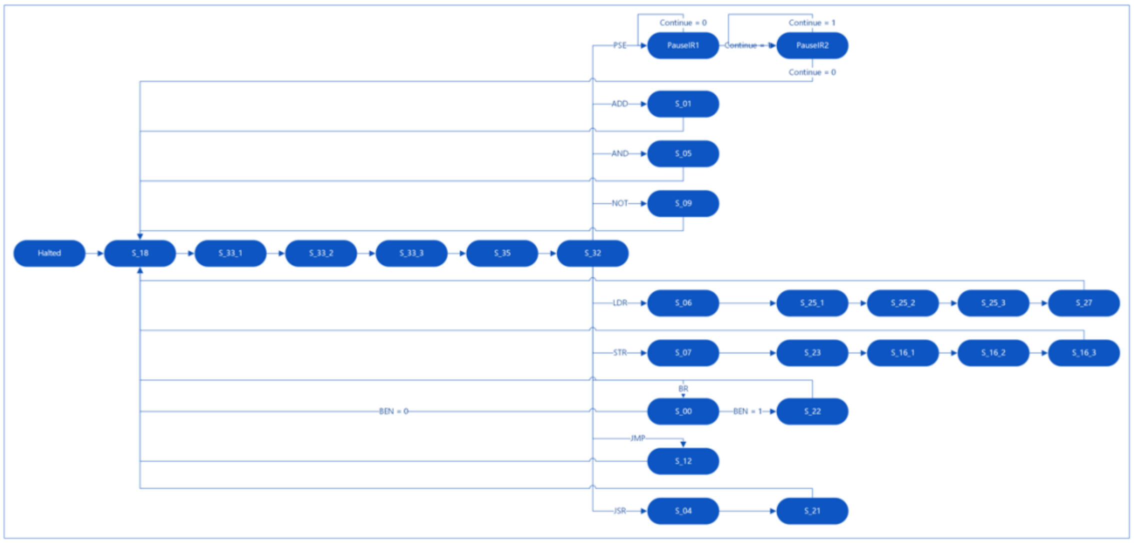


Figure 8. State Diagram of ISDU

<u>Simulations of SLC-3 Instructions:</u>

To decode what is on the HEX displays use the following table:

Hex Display Binary Code	Hex Equivalent
1000000	0
1111001	1
0100100	2
0110000	3
0011001	4
0010010	5
0000010	6
1111000	7
0000000	8
0010000	9
0001000	A
0000011	B
1000110	C
0100001	D
0000110	E
0001110	F

IO Test 1:

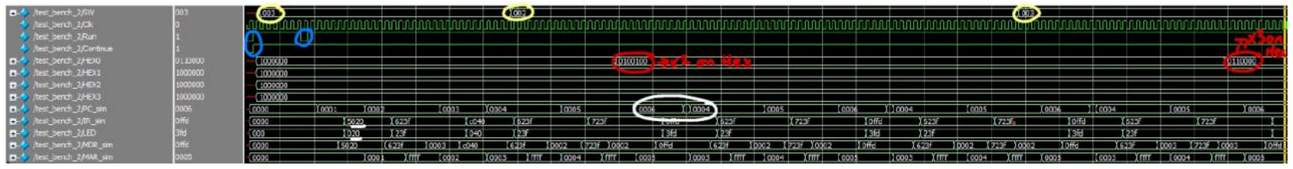

/test_bench_2/SW 003
/test_bench_2/Clk 0
/test_bench_2/Run 1
/test_bench_2/Continue 1
/test_bench_2/HEX0 0110000
/test_bench_2/HEX1 1000000
/test_bench_2/HEX2 1000000
/test_bench_2/HEX3 1000000
/test_bench_2/PC_sim 0006
/test_bench_2/IR_sim 0ffd
/test_bench_2/LED 3fd
/test_bench_2/MDR_sim 0ffc
/test_bench_2/MAR_sim 0005
x2 on Hex
x3 on Hex

Figure 9. IO Test 1

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the first test resides at the memory location x3, so x3 is put in the switches and run is pressed so that PC starts executing instructions from that memory address. Notice that the four instructions ANDi, LDR, STR, and BR are executed as shown by the IR register. Once these four instructions are done executing, x2 is loaded into the switches, and we can see that after a certain execution cycle, x2 also shows up on the Hex displays. Similarly, after we put x3 in the switches we get x3 on the Hex displays. Notice the loop-like behavior, where we always loop back to instruction x3, which is required to do this test.

IO Test 2:

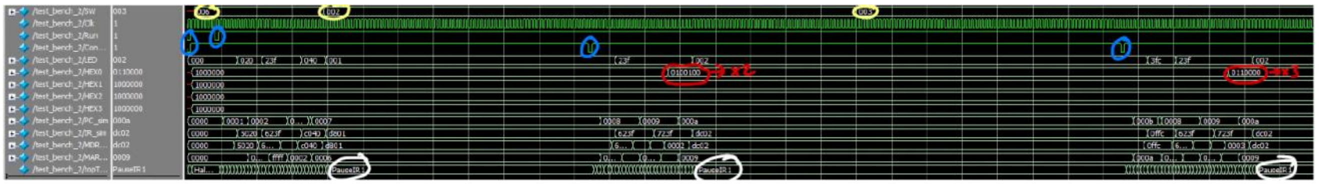

/test_bench_2/SW 003
/test_bench_2/Clk 1
/test_bench_2/Run 1
/test_bench_2/Con... 1
/test_bench_2/LED 002
/test_bench_2/HEX0 0110000
/test_bench_2/HEX1 1000000
/test_bench_2/HEX2 1000000
/test_bench_2/HEX3 1000000
/test_bench_2/PC_sim 000a
/test_bench_2/IR_sim dc02
/test_bench_2/MDR... dc02
/test_bench_2/MAR... 0009
/test_bench_2/topT... PauseIR1
x2
x3

Figure 10. IO Test 2

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the second test resides at the memory location x6, so x6 is put in the switches and run is pressed so that PC starts executing instructions from that memory address. The main difference between IO Test 1 and IO Test 2, is that in IO Test 2, instead of changing the HEX displays automatically, the system waits for the user in the

PauseIR state to press continue before displaying the contents of the switches on the Hex Display. We can see exactly this behavior represented in the simulation. Though x2 is put on switches, the value of the switches doesn't change until the continue button is pressed. Similar behavior occurs when x3 is put on switches.

Self-Modifying Test:

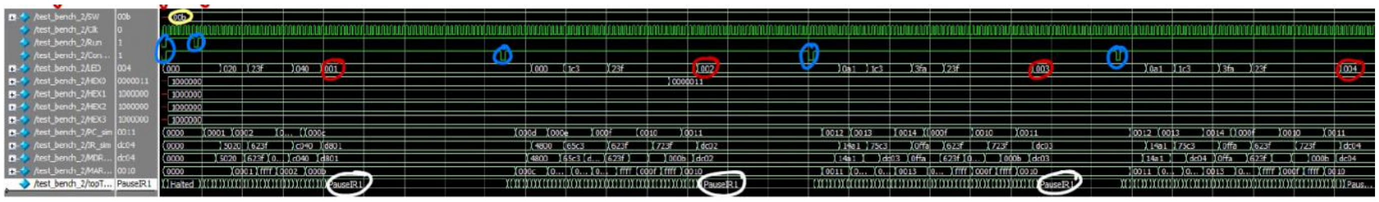


Figure 11. Self-Modifying Test

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the third test resides at the memory location xB, so xB is put in the switches and run is pressed so that PC starts executing instructions from that memory address. This test uses a similar loop as the last program, however, the data for each pause instruction is incremented by one each time the person presses continue. As you can see from the simulation, the machine remains in the PauseIR state until continue is pressed, and when continue has pressed the value of LED is incremented by 1 (shown in red). This happens each time continue is pressed and the machine is in the PauseIR state.

XOR Test:

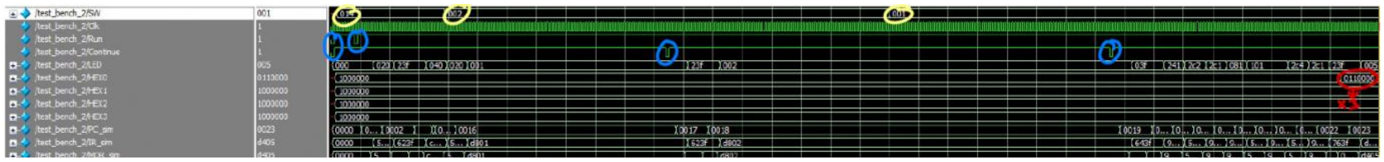


Figure 12. XOR Test

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the fourth test resides at the memory location x14, so x14 is put in the switches and run is pressed so that PC starts executing instructions from that memory address. In this test, I am going to be XORing x2 and x1 together to get an output of x3. To do this, I first push x2 into the switches and then press continue (while the machine is in the PauseIR state). Similarly, I put x1 into switches and press continue. Finally, I get output of x3 in the HEX displays as I was expecting.

Multiplier Test:

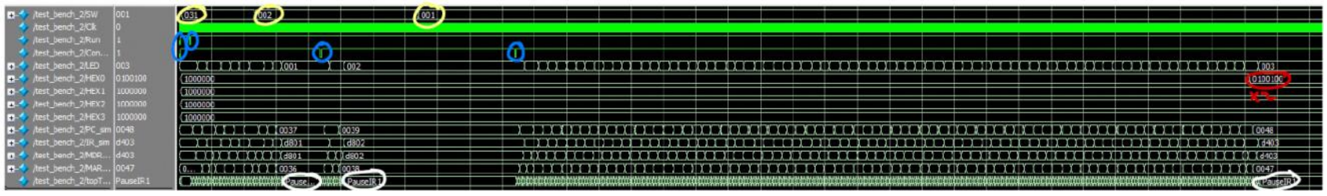

/test_bench_2/SW 001
/test_bench_2/Clk 0
/test_bench_2/Run 1
/test_bench_2/Con... 1
/test_bench_2/LED 003
/test_bench_2/HEX0 0100100
/test_bench_2/HEX1 1000000
/test_bench_2/HEX2 1000000
/test_bench_2/HEX3 1000000
/test_bench_2/PC_sim 0048
/test_bench_2/IR_sim d403
/test_bench_2/MDR... d403
/test_bench_2/MAR... 0047
/test_bench_2/topT... PauseIR1
031
002
001

Figure 13. Multiplier Test

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the fifth test resides at the memory location x31, so x31 is put in the switches and run is pressed so that PC starts executing instructions from that memory address. In this test, I am going to be multiplying x2 and x1 together to get an output of x2. To do this, I first push x2 into the switches and then press continue (while the machine is in the PauseIR state). Similarly, I put x1 into switches and press continue. Finally, I get output of x2 in the HEX displays as I was expecting.

Sort Test:

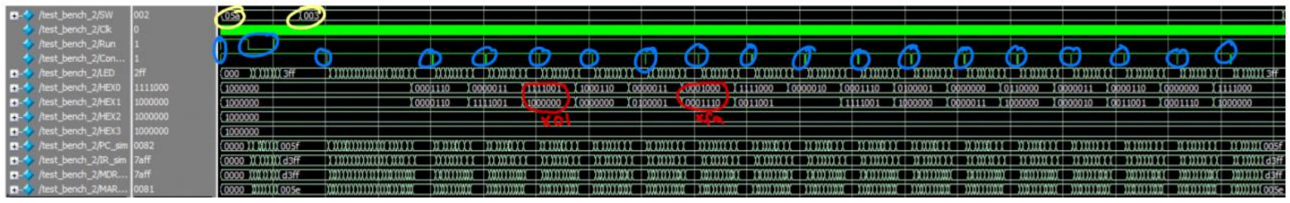


Figure 15. Sort Test part 1

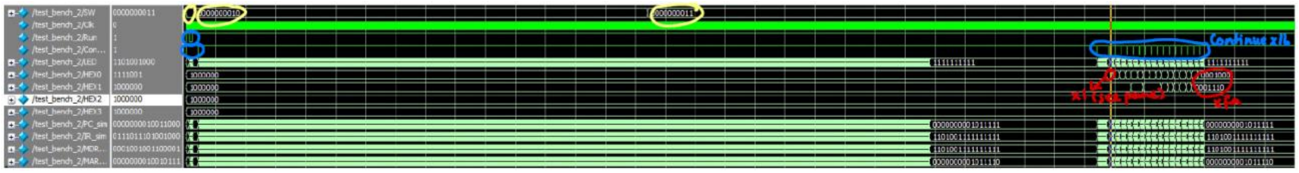


Figure 14. Sort Test part 2

Before we execute the test, we first reset the machine by pressing continue and run together (note that both buttons are active low). Now, the sixth test resides at the memory location x5a, so x5a is put in the switches and run is pressed so that PC starts executing instructions from that memory address. Once this is done, the user must decide between 1 of 3 modes: 1) Entering x0001 will call the “data entry” function (which will allow the user to put in any values the user wants inside the memory), entering x0002 will call the “sort” function (sorts whatever data is present in the memory contents), and entering x0003 will call the “display” function (will display all the values inside the memory). We first want to see, the contents of memory and therefore we put x3 on switches and press continue. After some clock cycles, we see the first element of memory appear on the HEX display. Now, every time we press continue, we see the next element on memory. Note that x01 (the smallest element) is in the third position while xfa (the largest element) is in the fourth position. We can sort the memory contents by putting x2 on the switches and then pressing continue. To see sorted memory elements, we again put x1 on switches and press continue. Once we do that, we can see that x1 ends up as the first element of memory, while xfa becomes the last element on the list.

<u>Post-Lab Questions:</u>

LUT	821
DSP	0
Memory (BRAM)	18432
Flip-Flop	258
Frequency	64.62 MHz
Static Power	90 mW
Dynamic Power	10.81 mW
Total Power	112.15 mW

Our design for both weeks was fully functional. However, we had two major problems while we were trying to debug week 2 of the lab. Firstly, we noticed that in quite a few places we had forgotten to set LD.REG for register file to high whenever we were trying to set a particular register to some value. The other mistake we did was using one always comb block for the entire data path. This was quite difficult to debug as in theory it should have worked as everything in the always_comb is supposed to have sequentially.

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

Mem2IO provides us the functionality of memory mapped I/O in our SLC-3 design. The basic functionality is to provide a memory address to our CPU that are physical peripherals to our design like LEDs, switches etc. So, whenever CPU reads or writes to the special memory addresses, Mem2IO makes sure that the data is being read or written from switches or LEDs respectively.

What is the difference between BR and JMP instructions?

In a BR statement, first we check BEN, if it is 1 then we set PC to PC + offset9 (PC is updated with an offset).

In a JMP statement, we unconditionally set PC to whatever value is stored in the given base register. No offset.

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?

R is the ready signal sent by the memory in Patt and Patel. It is 1 whenever memory is done reading or writing, i.e., finished its process and is ready to perform a new task. We do not have this signal in our design because we are using a SRAM. We compensate by having 3 wait states for our SRAM to finish its read/write. This is because we know it takes 3 cycles for the memory to done processing. Therefore, by having 3 wait states, we know for a fact that the data would be available to us in the third state. Hence, we avoid any synchronization error with our clock as the processing time for memory is fixed.

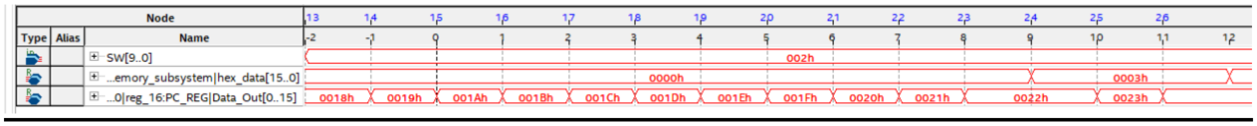
Conclusion:

Our design for both weeks was fully functional. However, we had two major problems while we were trying to debug week 2 of the lab. Firstly, we noticed that in quite a few places we had forgotten to set LD.REG for register file to high whenever we were trying to set a particular register to some value. The other mistake we did was using one always comb block for the entire data path. This was quite difficult to debug as in theory it should have worked as everything in the always_comb is supposed to have sequentially. However, in practice, things don't exactly work that way, and thus we learned we should always try to use multiple always comb blocks for different parts of the data path whenever possible.

There was nothing particularly ambiguous about this lab other than writing the test bench. I feel like writing good test benches is something that has not been taught to us and is extremely important to learn to be successful in this course. So, something that could be improved is giving better resources teaching us how to write test benches. Something that the lab did well was the level of description and resources provided to help complete the lab (such as the lab manual, the control signal, etc).

Extra Credit:

XOR Test:



As we can see, the clock is 0 at PC = x1A. We have XOR'ed the values x01 and x02 which should give us x03. As we can see from the screenshot, it takes 9 clock cycles to compute the final answer. Hence, we get the answer finally at PC x23.

Cycles: 11

Instructions: 9

Instructions per seconds: (9*50,000,000)/11 = 40.909 MIPS.

Multiplier Test:

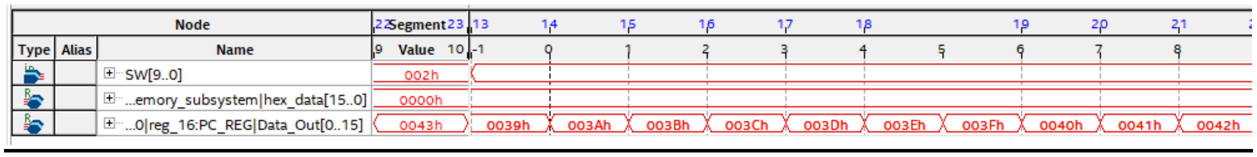


Figure 16. Begin of Multiply

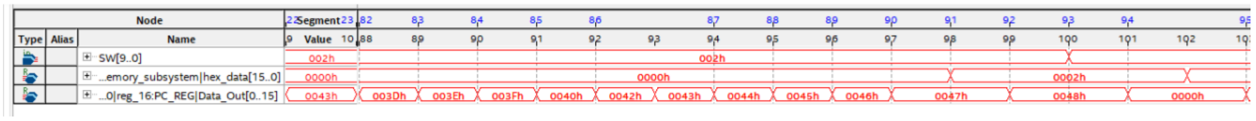


Figure 17. End of Multiply

As we can see, the clock is 0 at PC = x3A. We have multiplied the values x01 and x02 which should give us x03. As we can see from the screenshot, it takes 99 clock cycles to compute the final answer. Hence, we get the answer finally at PC x48.

Cycles: 99

Instructions: 98

Instructions per seconds: (98*50,000,000)/99 = 49.494 MIPS.

<u>Sort Test:</u>

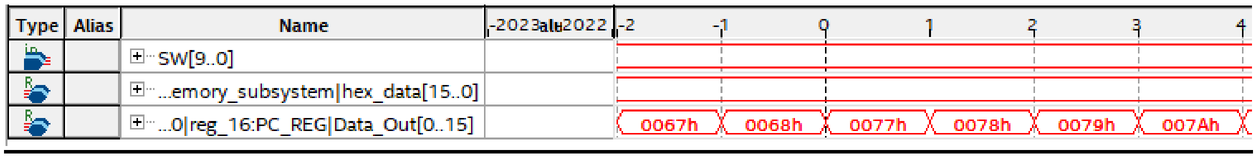

Type
Alias
Name
SW[9..0]
...emory_subsystem|hex_data[15..0]
...0|reg_16:PC_REG|Data_Out[0..15]
0067h
0068h
0077h
0078h
0079h
007Ah

Figure 18. Begin of Sort

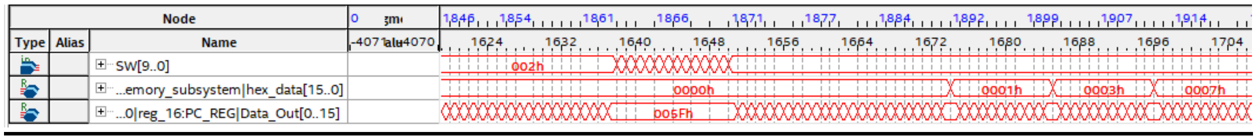

Node
Type
Alias
Name
SW[9..0]
...emory_subsystem|hex_data[15..0]
...0|reg_16:PC_REG|Data_Out[0..15]
002h
0000h
0001h
0003h
0007h
005Fh

Figure 19. End of Sorted

As we can see, the clock is 0 at PC = x77. We have sorted the values which were already present in the memory. As we can see from the screenshot, it takes 1651 clock cycles to compute the sorted answer. Hence, we get the answer finally at PC x5F.

Cycles: 1651

Instructions: 1638

Instructions per seconds: (1638*50,000,000)/1651 = 49.606 MIPS.