Lab Report #5

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

October 9, 2023

Introduction

Summarize the basic functionality of the SLC-3 processor

The purpose of this week's lab was to design a microprocessor in SystemVerilog. Our microprocessor consisted of contents derived from the LC-3 ISA. It is made up of a 16-bit processor with a 16-bit Program Counter (PC), 16-bit instructions (ADD, ADDi, AND, NOT, BR, JMP, JSR, LDR, STR, and PAUSE), and lastly 16-bit registers.

Written Description and Diagrams of SLC-3

Summary of Operation

The way the microprocessor works is that one of our 16-bit registers, the Instruction Register (IR), contains the necessary operation that will need to be run. As the operation runs through its cycles, the Instruction Decoder indicates what instruction must be conducted at each cycle. The Instruction Decoder is controlled by our design's control unit. When the reset button is pressed, the cycle resets to our halt state until the user presses the run button again. After each operation, we reset our PC value to 0.

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.

As mentioned, the SLC-3 processor's functionality is described by a cycle that goes from fetch to decode to execute.

Beginning with the fetch state, we use the Memory Address Register (MAR) to store the value of our PC. Next, the memory of the MAR is then stored into the MDR. Following this, the MDR's contents are passed into the IR. Lastly, we update our PC by incrementing its value by 1.

Moving onto the decode state, this is when the IR's contents are passed into our Instruction Decoder. The decoder takes in these contents and deciphers which registers are to be used as a source and destination, the operations needed to achieve the desired result, and which tri-state buffer gate should be used to write onto our BUS.

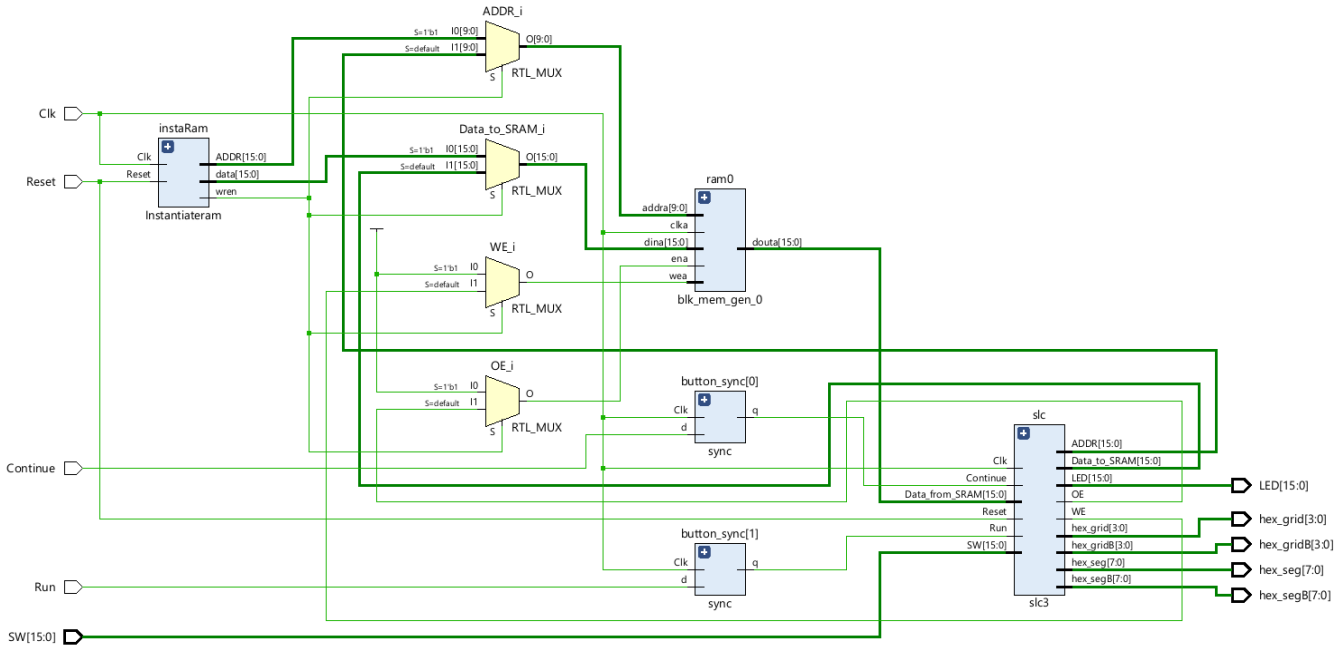
Lastly, our execute state involves running the indicated operations from the decode sequence and passing on the result into our final memory contents, which is displayed on our FPGA using our Destination Register (DR).

In this section, we will go into detail about the 16-bit instructions (ADD, ADDi, AND, NOT, BR, JMP, JSR, LDR, STR, and PAUSE) that our microprocessor is able to perform. Beginning with the ADD instruction, its purpose is to add the contents of 2 different source registers (SR1 and SR2) and store the final result in the DR. It further indicates whether the result is positive, negative, or zero by setting our NZP register accordingly. A similar but slightly different instruction ADDi takes in a value from a single source register and adds it to a sign-extended 5-bit immediate value (imm5), also storing the final result in the DR and updating the NZP register accordingly. Next, the AND operation takes the logical AND of the contents of 2 source registers, storing the result into DR, and updating the NZP register based on the sign of the final value. A similar but slightly different instruction ANDi takes in a value from a single source register and ANDs it to a sign-extended imm5, also storing the final result in the DR and updating the NZP register accordingly. Next, the NOT instruction is responsible for negative values (flipping 1s to 0s and vice-versa) and storing the result in the SR as well as updating the NZP register accordingly. The branch (BR) instruction checks the condition code in the NZP register and branches (like a loop) if the code matches. On the other hand, if the NZP code has a value of '111', the branch instruction will unconditionally branch. The address the instruction

branches to is designated by a sign-extended PCoffset9, which is added to the value of the PC and its result is the branched address. The JMP instruction “jumps” to a specified address by taking the memory address from a Base Register and copying it into the PC register. The LDR instruction loads the DR with the memory contents at address calculated by: address at base register and a sign-extended 6-bit offset (offset6). The next instruction, STR, works similarly by storing the contents of the SR by the address calculated by: address at base register and a sign-extended 6-bit offset (offset6). Lastly, the PAUSE instruction stops any operations being performed until the continue button is pressed.

Block Diagram of slc3.sv

Generated from the RTL viewer, only top-level (not RTL view of every module)



Written Description of .SV Modules

Module: Instantiateram.sv

Inputs: Reset, Clk

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

Description: This module defines a state machine with transitions based on conditions and operations. Address calculations, the loading of different operations, memory writing, and program execution are all allowed by the signals defined in this module.

Purpose: This module instantiates our on-chip memory for the FPGA. A read enable, write enable, 10-bit address BUS, and a 16-bit data bus are all defined.

Module: MEM2IO.sv

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

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

Description: A 16-bit logic vector stores the data for our hex display output. Data to our CPU is generated based on our output enable and write enable signals at address x0FFFF.

The SRAM is simply copied by the data from CPU.

Purpose: This module connects memory (SRAM) and CPU data. Data can thus be read in from our switches as desired and the HEX displays on the FPGA are updated to represent our result.

Module: memory_contents.sv

Inputs:

Outputs: [15:0] mem_array

Description: Memory initializes the various instructions for each of our tests. Opcodes from the SLC3_2 file specifies the functionality of our instructions.

Purpose: This module is designed to allow us to run the different instructions of different test cases by specifying the instructions to be run when a specific hex value is inputted.

Module: slc3.sv

Submodule: register

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

Outputs: [15:0] Data_Out

Submodule: register1

Inputs: Clk, Reset, Load, D

Outputs: Data_Out

Submodule: register3

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

Outputs: [2:0] Data_Out

Submodule: Reg_Unit

Inputs: load, Clk, Reset, [15:0] BUS_in, [2:0] DR_select, [2:0] SR1_MUX_Out, [2:0] SR2_MUX_OUT

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

Submodule: MEM2IOMux

Inputs: S, [15:0] Bus, [15:0] Data_to_CPU

Outputs: [15:0] Q_Out

Submodule: PC_Mux

Inputs: [1:0] S, [15:0] PC1, [15:0] BUS, [15:0] ADDER

Outputs: [15:0] Q_Out

Submodule: ADDR2_Mux

Inputs: [1:0] S, [15:0] IR10, [15:0] IR8, [15:0] IR5, IR0

Outputs: [15:0] Q_Out

Submodule: ADDR1_Mux

Inputs: S, [15:0] PC, [15:0] SR1_Out

Outputs: [15:0] Q_Out

Submodule: ADDER

Inputs: [15:0] ADDR2, [15:0] ADDR1

Outputs: [15:0] ADDER

Submodule: SR1_Mux

Inputs: S, [2:0] IR86, [2:0] IR119

Outputs: [2:0] Q_Out

Submodule: SR2_Mux

Inputs: S, [15:0] SEXTI4, [15:0] SR2_OUT

Outputs: [15:0] Q_Out

Submodule: DR_Mux

Inputs: S, [2:0] THREEONES, [2:0] IR119

Outputs: [2:0] Q_Out

Submodule: ALU

Inputs: [15:0] SR1OUT, [15:0] SR2MUXOUT, [1:0] ALUK

Outputs: [15:0] ALUOUT

Submodule: NZP

Inputs: [15:0] BUS

Outputs: [3:0] CC_Logic

Submodule: pathdata

Inputs: GateMDR, [15:0] MDR, GateALU, [15:0] ALU, GatePC, [15:0] PC,

GateMARMUX, [15:0] MARMUX

Outputs: [15:0] BUS

Submodule: slc3

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

Outputs: [15:0] LED, OE, WE, [7:0] hex_seg, [3:0] hex_grid, [7:0] hex_segB, [3:0]

hex_gridB, [15:0] ADDR, [15:0] Data_to_SRAM

Description: This module defines numerous submodules as specified by a modified block diagram from the original LC-3 design. This includes all of our MUXs and Bus definitions.

Purpose: This module serves as our top level design. Instantiates and connects all the signals for the modules required to control the behavior of the LC-3's instruction's functionality.

Module: SLC3_2.sv

Inputs:

Outputs:

Description: This file is not a module, rather a package. It contains definitions for opcodes, Registers R0 to R7, branch conditions (NZP), and the LC-3 instructions used in our design.

Purpose: To specify the SLC-3 architecture. Operations and operands involved for our LC-3 functions are defined here. They are used for the test conditions in test_memory.sv

Module: slc3_sramtop.sv

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

Outputs: [15:0] LED, [7:0] hex_seg, [3:0] hex_grid, [7:0] hex_segB, [3:0] hex_gridB

Description: This module synchronizes our input buttons onto the FPGA's clock domain. It also defines signals that are used for memory handling and data read/write.

Purpose: To allow for synchronous RAM access and data flow in our SLC-3 architecture.

Module: slc3_testtop.sv

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

Outputs: [15:0] LED, [7:0] hex_seg, [3:0] hex_grid, [7:0] hex_segB, [3:0] hex_gridB

Description: Buttons are synchronized. Data and memory related signals are instantiated. Connections between the slc module and test_memory module as memory are defined.

Purpose: This module is a top-level file that can connect the SLC-3 processor and the test_memory Module for simulation purposes.

Module: synchronizers.sv

Inputs: Clk, d

Outputs: q

Description: The asynchronous input d is synchronized onto the FPGA's clock domain using a flip-flop triggered on the positive clock edge. It outputs the synchronized q.

Purpose: Allows multiple modules on our FPGA to utilize the same Clk signal and synchronizes it to ensure accurate timing.

Module: test_memory.sv

Inputs: Reset, Clk, data, address, ena, wren

Outputs: readout

Description: This file reads and parses memory contents from a given file. Defines read and logic for a simulation rather than real-time input/output.

Purpose: Simulates the behavior of FPGA on-chip memory, allows for memory read/write during simulation.

Description of the operation of the ISDU (Instruction Sequence Decoder Unit)

Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.

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, [1:0] ADDR2MUX, [1:0] ALUK, Mem_OE, Mem_WE

Description: State transition logic is defined and control signals based on the current states are assigned. State transitions contain case statements that move on to the next state based on certain conditions.

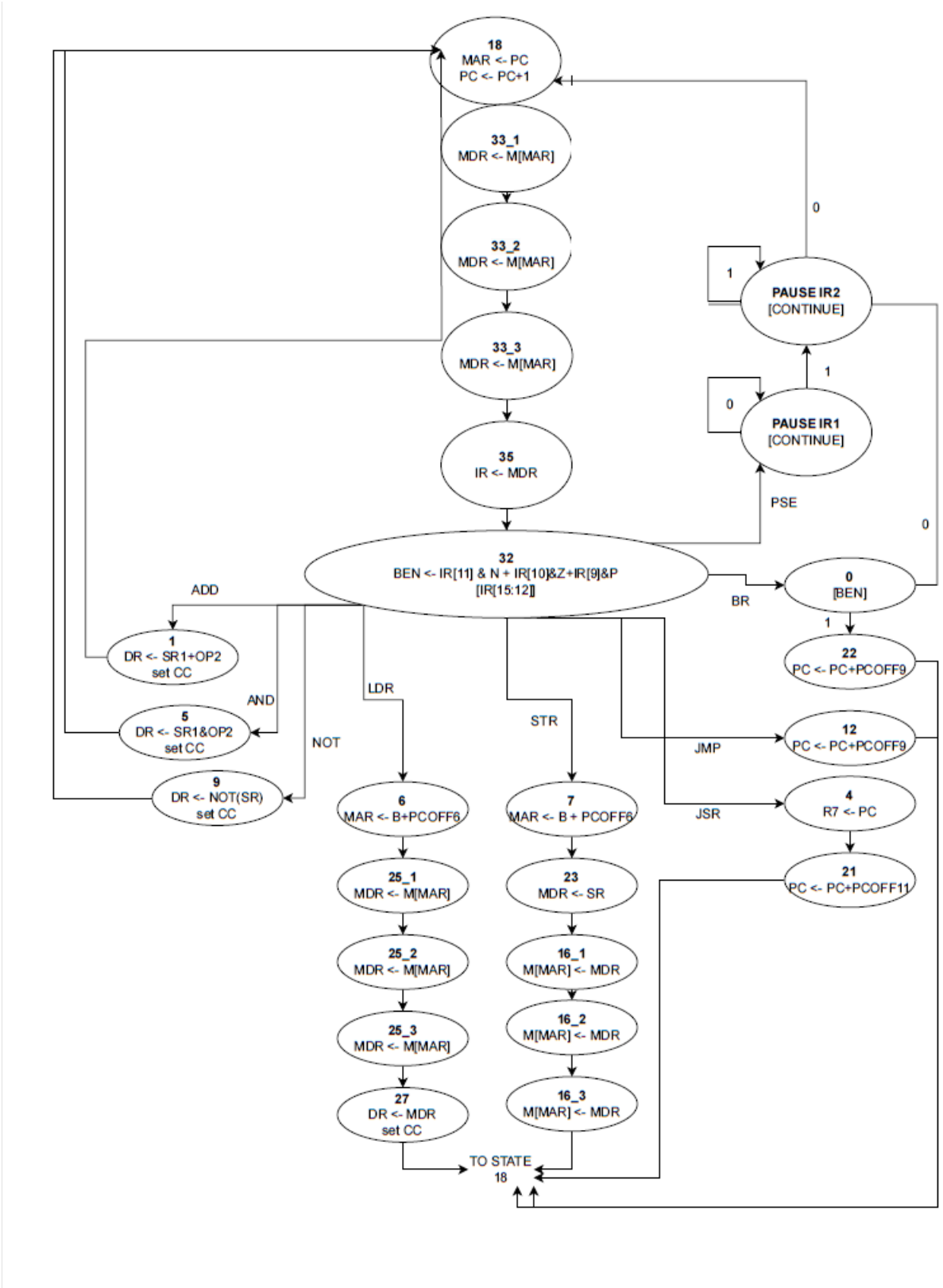
Purpose: Defines the control logic and state transitions for our sequencing and decoding process. Operations and control signals that are being performed and defined are explained here.

The ISDU, or our control unit, defines a state machine to represent data flow across different input and output signals. Functionality such as register loading, necessary operations, mux selections are all defined here. Opcodes from the IR are utilized to identify which state executes when. We also include 3 holding states each for some states to compensate for the lack of an ‘R’ bit. Tracing the beginning of our program, all our logic is set to low. The ISDU then begins in a HALT state, then transitions to our fetch state, then decoding state, and finally to an execute state. Based on the opcode, a certain instruction is carried out. If the ISDU is in its PAUSE state, the ISDU waits for input from the switches before performing other operations that require input values. The BEN value determines the next state. The reset button clears all the signals to low again and brings us back to our HALT state.

State Diagram of ISDU

This should represent all states present in the ISDU and their transitions. 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.

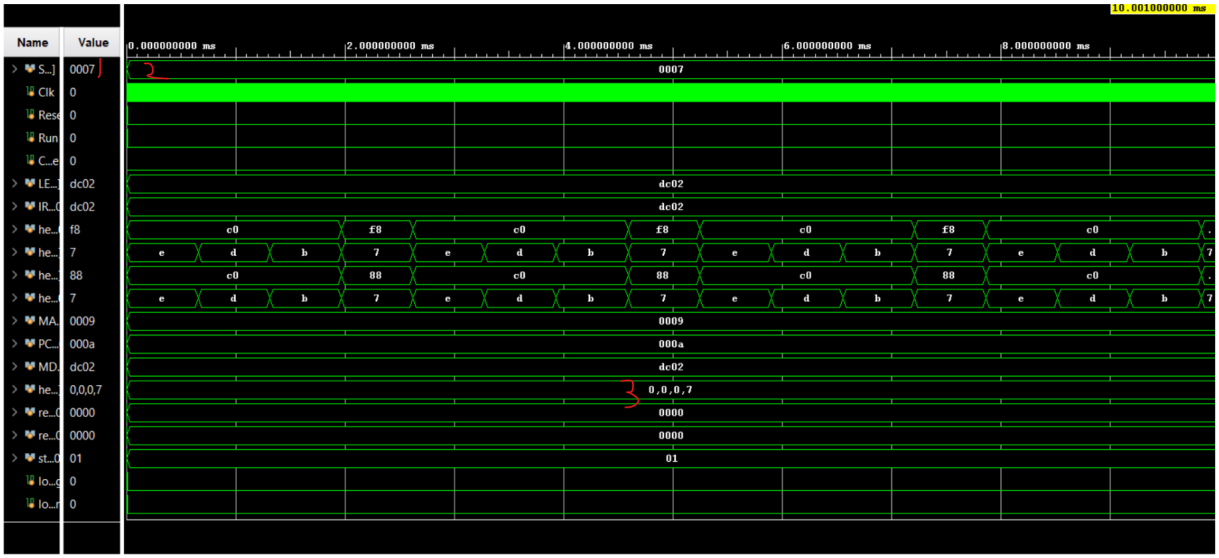
On the next page:


18
MAR <- PC
PC <- PC+1
33_1
MDR <- M[MAR]
33_2
MDR <- M[MAR]
33_3
MDR <- M[MAR]
35
IR <- MDR
32
BEN <- IR[11] & N + IR[10]&Z+IR[9]&P
[IR[15:12]]
PAUSE IR2
[CONTINUE]
PAUSE IR1
[CONTINUE]
PSE
0
1
ADD
AND
NOT
LDR
STR
BR
JMP
JSR
1
DR <- SR1+OP2
set CC
5
DR <- SR1&OP2
set CC
9
DR <- NOT(SR)
set CC
6
MAR <- B+PCOFF6
25_1
MDR <- M[MAR]
25_2
MDR <- M[MAR]
25_3
MDR <- M[MAR]
27
DR <- MDR
set CC
7
MAR <- B + PCOFF6
23
MDR <- SR
16_1
M[MAR] <- MDR
16_2
M[MAR] <- MDR
16_3
M[MAR] <- MDR
0
[BEN]
22
PC <- PC+PCOFF9
12
PC <- PC+PCOFF9
4
R7 <- PC
21
PC <- PC+PCOFF11
TO STATE
18

Simulations of SLC-3 Instructions

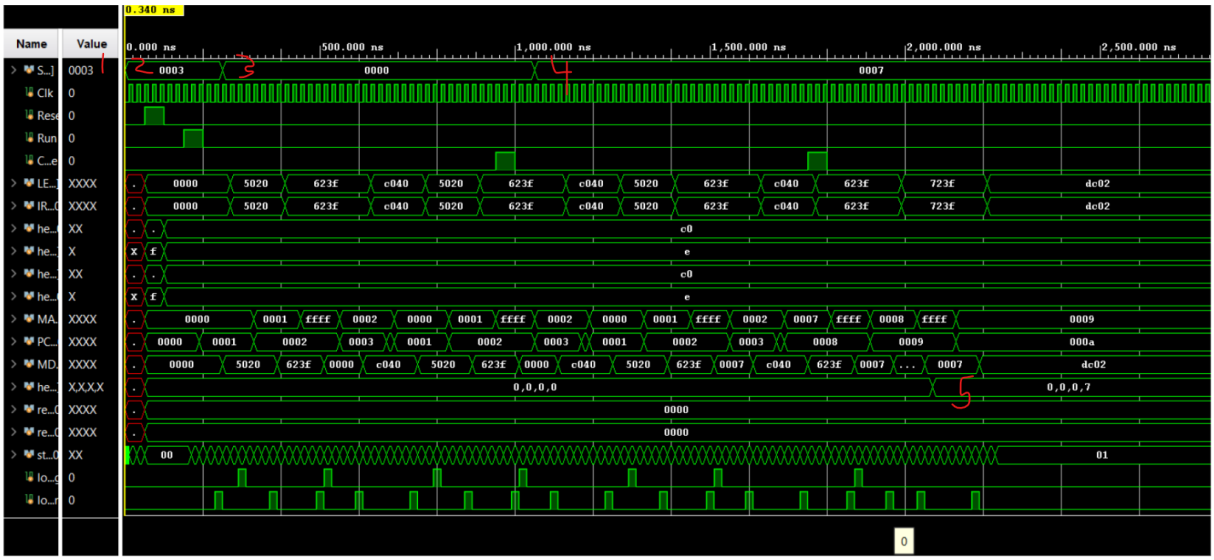
Annotations for the below simulations should include at a minimum: start of the test program, any user input, and reading the expected result. For long simulation traces, you may truncate out the intermediate portions of the program.

Simulate the completion of I/O Test 1



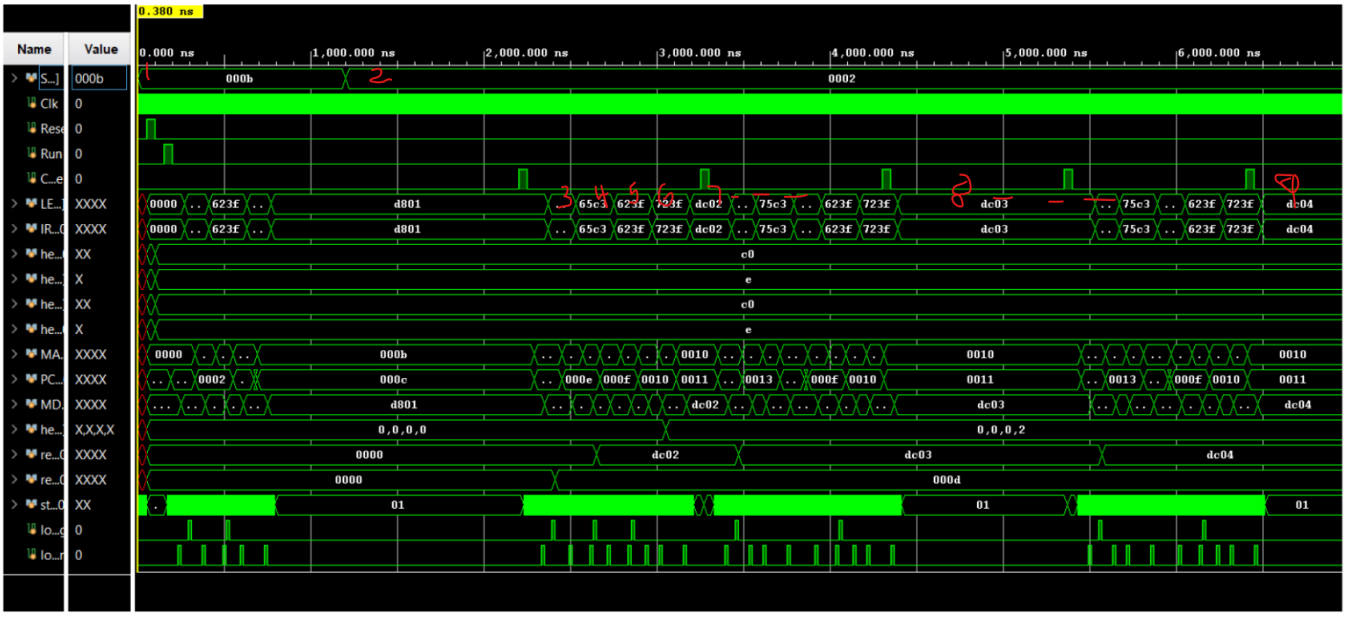
1) **x0003 is inputted to signify I/O Test 1**
2) **Value x0007 is inputted into the switches**
3) **Hex display outputs 0,0,0,7 as specified by the switches**

Simulate the completion of I/O Test 2



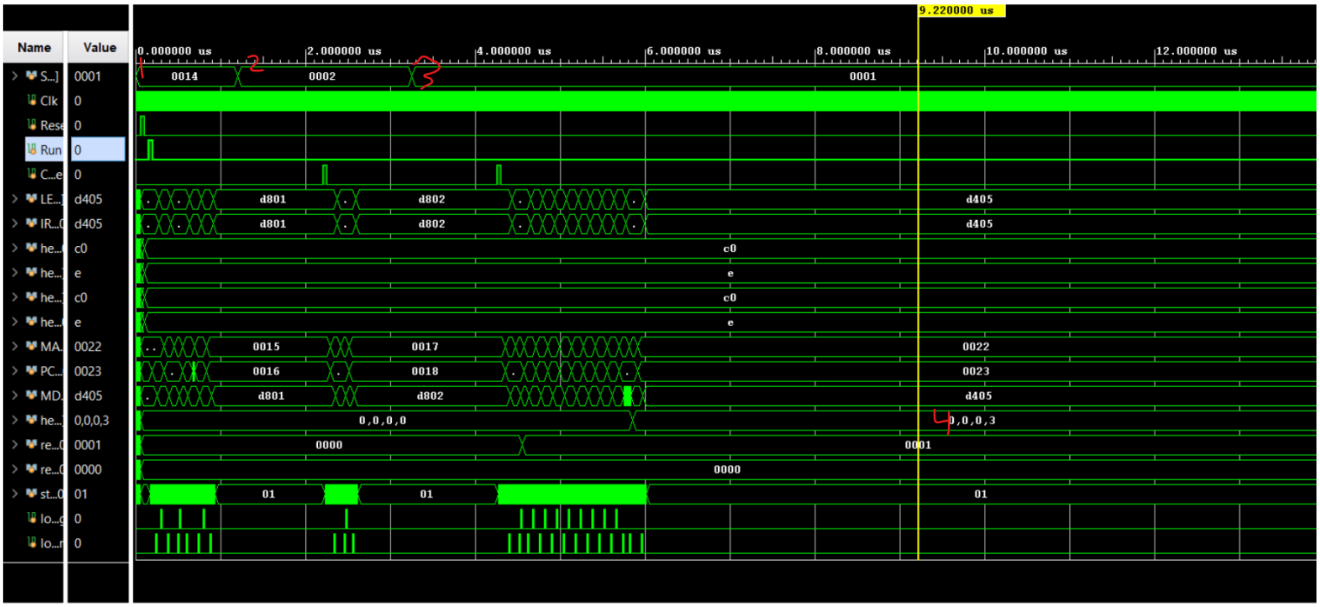
1) **x0006 is inputted to specify IO Test 2**
2) **x0003 is inputted but continue never pressed so does not display on hex switches**
3) **Reset to x0000**
4) **x0007 is inputted and continue is pressed**
5) **0,0,0,7 is displayed on the hex driver accordingly**

Simulate the completion of Self-Modifying Code



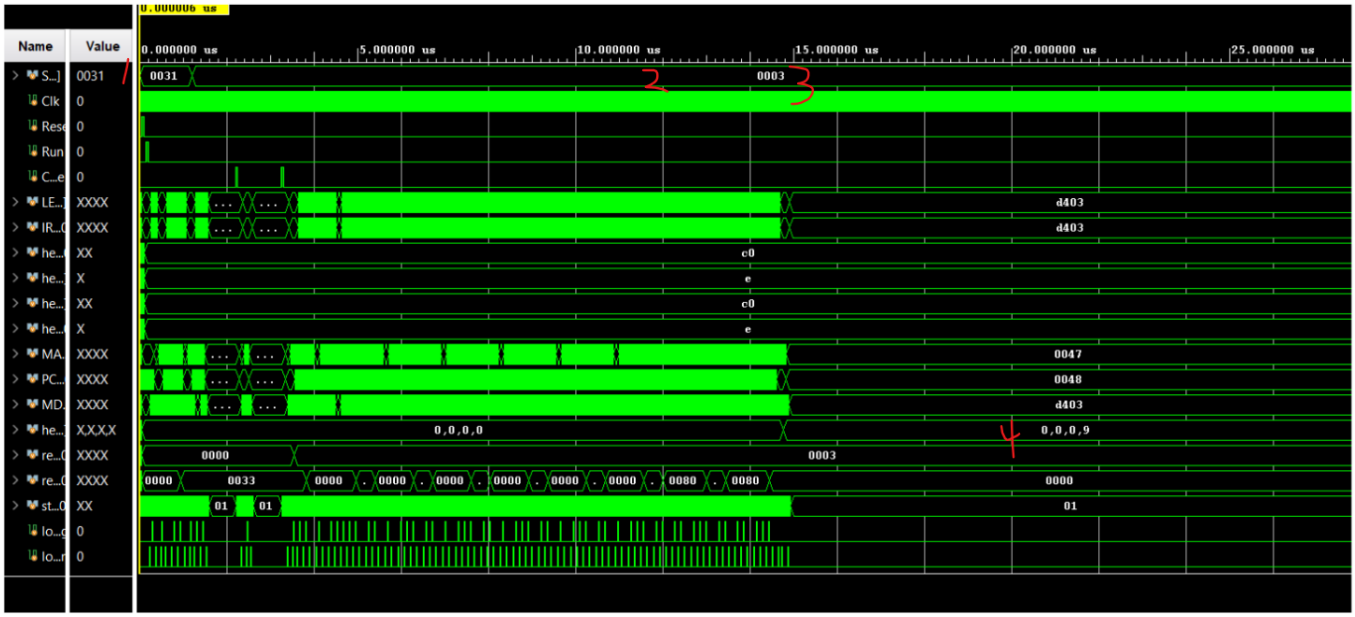
1) **x000b is inputted to signify self-modifying code test**
2) **Random value inputted into the switches (doesn't affect LED output)**
3) **From 3-9 it can be seen that the LEDs are incrementing by one**

Simulate the completion of XOR



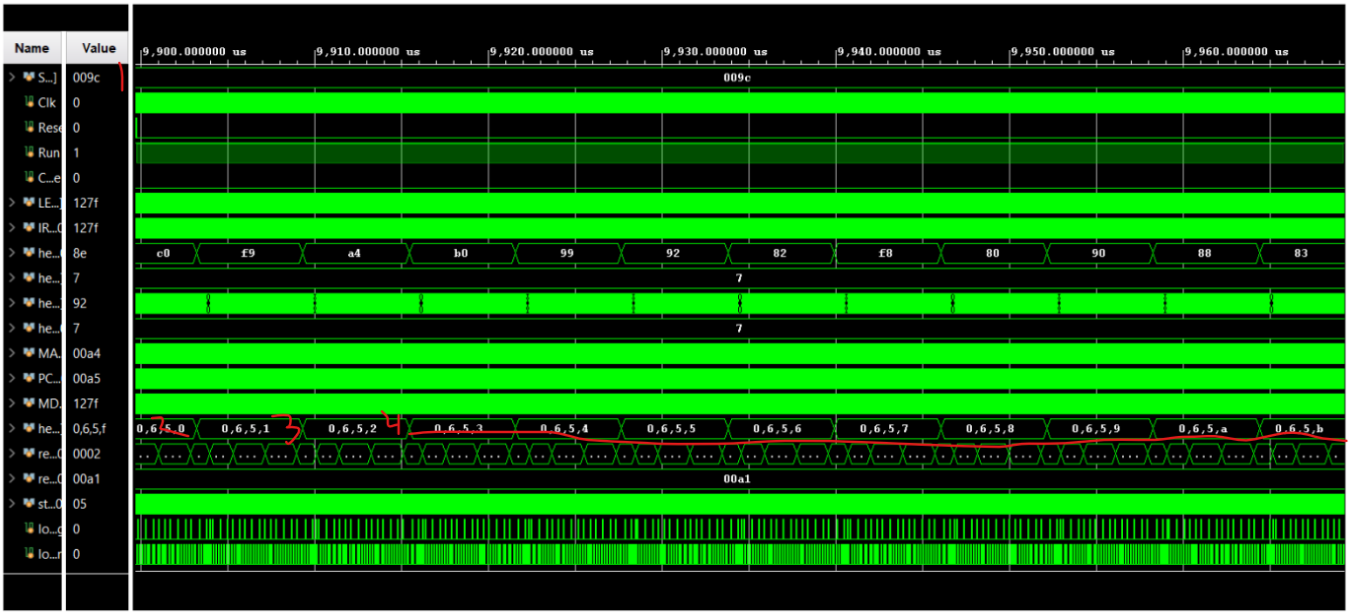
1) **x0014 inputted to specify XOR**
2) **Input x0002 using the switches**
3) **Input x0001 using the switches**
4) **XOR result x0003 displayed on the hex driver as 0,0,0,3**

Simulate the completion of Multiplier



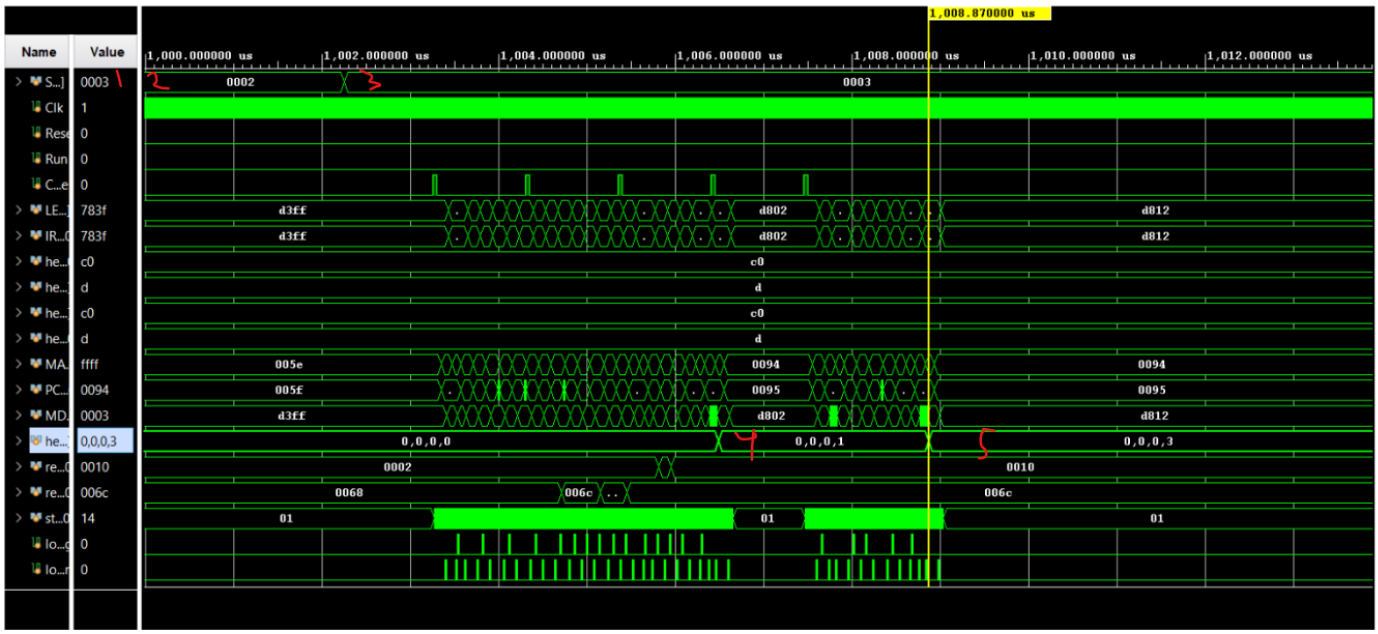
1) Input x0031 to signify multiplication test
2) Input x0003 and then press continue
3) Input x0003 again and press continue
4) Outputs 3*3 = 9 on the hex driver as 0,0,0,9

Simulate the completion of Auto Count



1) **Input x009C to indicate auto count**
2) **Program starts counting up from 2-4 and so on, in this case I made it so that it starts at value 0,6,5,0 and waveform counts up by 1 until 0,6,5,b**

Simulate the completion of Sort



1) Input x009C to indicate sort test
2) Input x0002 and press continue to sort
3) Input x0003 to display and press continue
4) Value 0,0,0,1 shows
5) Continue pressed again and value 0,0,0,3 shows (in accordance with input values 1,3,7,d…etc)

Post-Lab Questions

Fill out the Design Resources and Statistics table from Post-Lab question one

LUT	444
DSP	0
Memory (BRAM)	0.5
Flip-Flop	264
Latches	0
Frequency	103.167234 MHz
Static Power	0.071 W
Dynamic Power	0.018 W
Total Power	0.089 W

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

The provided file MEM2IO handles data transfer between the CPU unit, switches, and SRAM based on address values. It defines our BUS I/O and transfers memory contents to the Urbana FPGA. For the purposes of this lab, the MEM2IO file reads the contents at the address (OE and WE values) 0xFFFF, which specifies if the value of the switches should be loaded into our registers. If instead it detects the store operation, it stores the result of the function and emits it

onto our HEX displays. The Data from our CPU is assigned to the value we want to pass in to our Data to SRAM. In essence, we are storing our result from each operation into a register and displaying it onto our switches instead of on the SRAM.

What is the difference between BR and JMP instructions?

The branch instruction is primarily used for conditional branching. Based on the NZP codes, negative, zero, and positive, the program will branch to a different address value. This address is calculated by adding the offset to our original PC value. It is important to note if we have '111' as our NZP, it can unconditionally branch. The JMP instruction, on the other hand, is used as an unconditional jump. Jump means that the program transfers control to a different address without any conditions, such as the NZP code. This jump will always occur as long as a JMP instruction is specified. The address that is jumped to is calculated by the value of the Base Register.

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 corresponds to a ready signal. Making this R signal high signified that the process of reading from memory has been completed. The need for a signal originates from the fact that states 16, 25, and 33 require multiple cycles for the memory to be accessed. In our design, we lack this signal. Instead, to compensate, we created 3 hold states each for the aforementioned states in our FSM (as specified in our control unit). This allows us to be able to wait the necessary amount of time for the memory to be accessed. In terms of synchronization, it means that the hold states must be long enough so that an adequate time and frequency is reserved so that all the data can be transferred over.

Conclusion

Discuss functionality of design, what didn't work and what could have been done to fix it

Since the lab materials explained the process thoroughly, there weren't many issues outside of general debugging that required a redesign. The biggest obstacle encountered was the transition from lab 5.1 to 5.2. During our lab 5.1, it was simple to create new register and mux modules and simply instantiate them onto our slc3.sv file. When it came time to transition to lab 5.2, looking at the number of modules based on our block diagram seemed daunting enough to modify our design so that all of them were located within the slc3.sv itself. While this may have increased some of the complexity of our design, it was rather helpful when keeping track of our signals and connecting them together in the end.

Ambiguity or difficulty in lab materials for next semester

N/A, the lab materials were sufficient to carry out this lab to completion.