ECE 385

Spring 2023

Experiment 5

Simple Computer SLC-3.2 in SystemVerilog

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

This lab intends to be a project of designing a SLC-3 processor on FPGA board using System Verilog. The SLC-3 processor will first load the values on the switches to PC. Then, the processor will fetch the instruction from the address and store it in IR. Next, the processor will look at the op-code, which is the first four bits, to decode and determine the operation to execute. Last, the processor will start the execution by going to the next state for each specific instruction.

Operation of the SLC-3 Processor

The SLC-3 processor first instantiates the memory contents stored in each memory address. Once the memory is properly instantiated, the processor can then be operated. The user can opt to manually set the switches to the desired program start address, which would set the program counter. When the run signal is high (run button is pressed), the processor will fetch the content stored at the address that is equivalent to the value of the switches. The fetching state involves retrieving the contents stored at the memory address indicated by the program counter (which would begin at the program start address set on the switches). The retrieved contents from memory will then be passed into the instruction register (IR), completing the fetch state. Then, the processor will move on to the decode state, during which the contents of the instruction register will be 'decoded;' namely the opcode, or the 4 most significant bits of the IR, which indicates which instruction we need to execute. During the decode state, the BEN, or branch enable, the signal will also be set using the instruction register's 11th, 10th, and 9th most significant bits by bitwise ANDing them with the condition codes (the branch enable signal will not be used unless our opcode of the IR indicates a BR branch instruction is to be executed). As mentioned, the 4-bit opcode determines which instruction our SLC-3 will execute. In the ECE 385 implementation of the LC-3, we include 11 possible instructions: 2 variations of ADD, 2 variations of AND, NOT, LDR, STR, JSR, JMP, BR, and PSE. The first variation of ADD involves adding the contents of 2 source registers and storing the result in a destination register. The second variation of ADD involves adding the contents of one source register with the sign-extended immediate value contained in the least significant 5 bits of the IR. The first variation is taken if the 5th bit of the IR (from index 0) is 0, and vice versa. The opcode for the ADD instructions is 0001. The first variation of AND involves bitwise ANDing the contents of two source registers and storing the result in a destination register. The second variation of AND involves bitwise ANDing the contents of one source register and the sign-extended immediate value contained in the least significant 5 bits of the IR. The first variation is taken if the 5th bit of the IR (from index 0) is 0, vice versa. The opcode for the AND instructions is 0101. The NOT instruction negates the contents of the source register and stores the result in the destination register. Its opcode is 1001. The BR branch instruction will branch to the location determined by adding PC with a sign extended PC offset of 9 bits in the 9 least significant bits of the IR if any of the condition codes match the conditions stored in the status register. Its opcode is 0000. The JMP instruction copies the memory address from the source register to the PC. Its opcode is 1100. The JSR instruction jumps to a subroutine by temporarily storing current program counter in register 7 then updating the program counter by adding a sign extended PC offset of 11 bits in the 11 least significant bits of the IR. Its opcode is 0100. The LDR instruction loads the contents stored at the memory address indicated by the sum of the contents of a base register and sign

extended 6 bits from the least significant 6 bits of the IR into a destination register. Its opcode is 0110. The STR instruction stores the contents of a source register into a memory location indicated by the contents of a base register summed with the sign extended 6 bits from the least significant 6 bits of the IR. Its opcode is 0111. The PSE Pause instruction displays the 12 least significant bits of the IR into the LEDs on the FPGA. It waits in a 'halt' state until the continue button is pressed. Its opcode is 1101. Additionally, the condition codes NZP that are important to the BR branch instruction (helps determine whether to branch) are set in the ADD, AND, NOT, and LDR instructions. Once the processor performs the specific sequence of operations for the instruction indicated by the IR, we will complete the execute state and return to the fetch state of the processor for the cycle to repeat again.

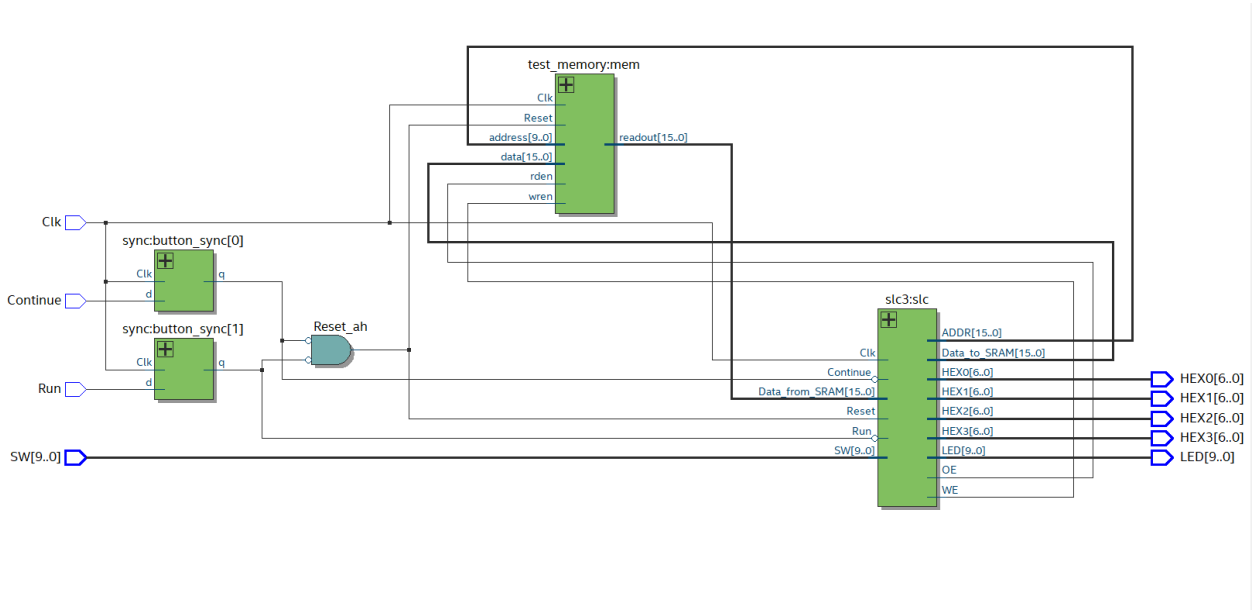


Figure 1. Block diagram of slc3.sv

ISDU is the positive-edge triggered flip-flop that has 32 states, which include a halted state, fetch state, decode state, and execute state depending on the instruction that is being executed. If the Reset signal is high, the state machine will return to the halted state. During each state, all the output signals such as LD_MDR, LD_MAR, etc. will be modified depending on the state it is at. The default value for all the outputs is set to 0. ISDU is used to control distinct components of the SLC-3 depending on the current instruction. For example, if the current instruction is ADD, then SR2MUX will be assigned to the value of IR[5] to determine whether to add SR1 with offset or SR2. Also, ALUK will be set to 0 to perform an addition of the two inputs, and GateALU will be high for the result to pass to the bus and allow CC to be loaded as LD_CC is set to high. Not to mention, SR1MUX and LD_REG will be set to high and DR_MUX to low for the result to be stored in to register file. Afterward, the state machine will take the processor back to state 18 to increment the PC and fetch new instructions. This is how ISDU controls the various components of the SLC-3.

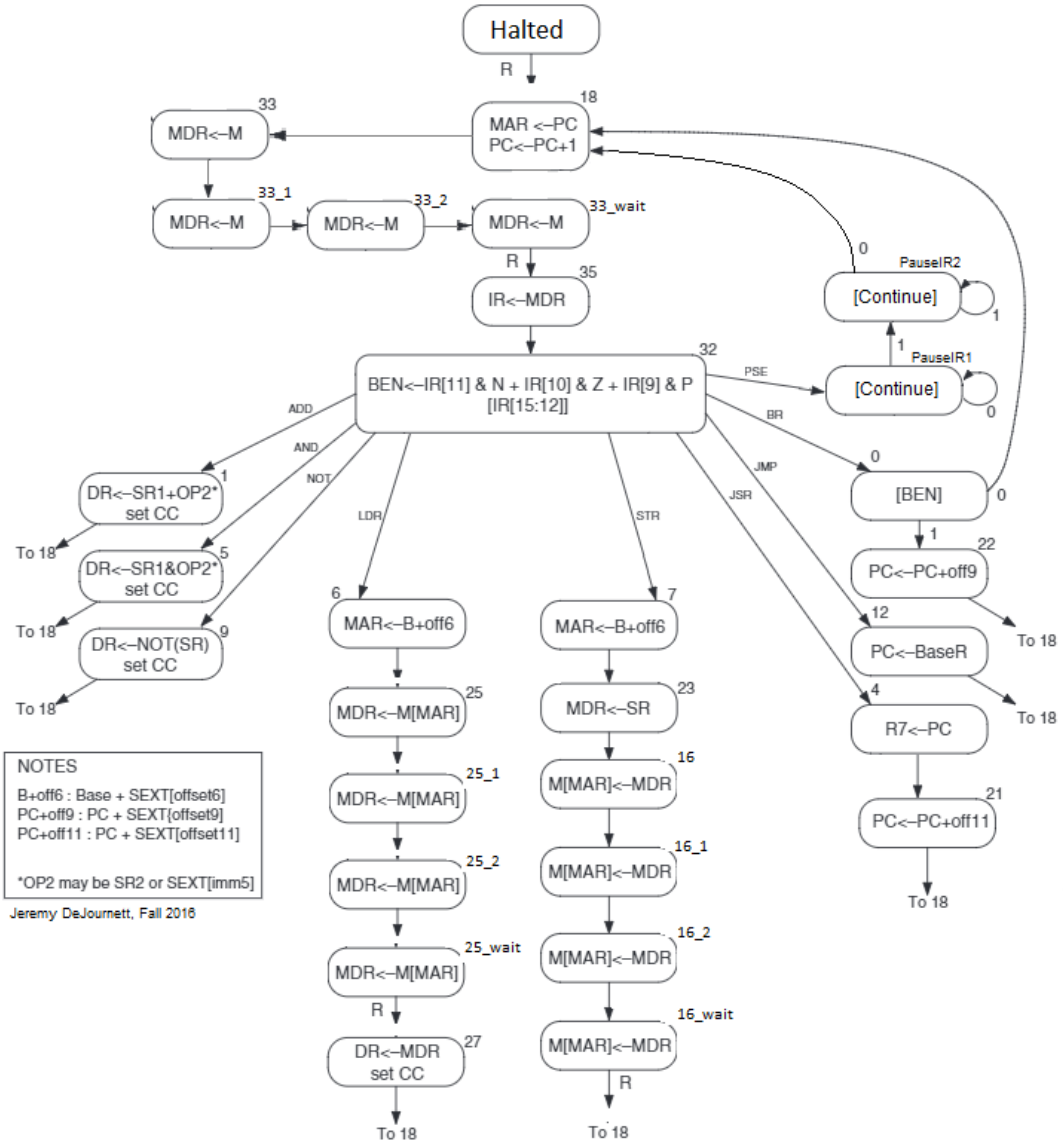


Figure 2. State Diagram of ISDU

SV Modules Summary

Module: Reg_16.sv
Inputs: [15:0] Din, Clk, Load, Reset
Outputs: [15:0] Dout
Description: This is a positive-edge triggered 16-bit register with a synchronous reset and synchronous load. When the Load is high, data is loaded from Din into the register on the positive edge of Clk. Otherwise, if the Reset is high, the register is set with a value of 0.
Purpose: This module creates the registers for MAR, MDR, IR, and PC.

Module: Reg_3.sv
Inputs: [2:0] Din, Clk, Load, Reset
Outputs: [2:0] Dout
Description: This is a positive-edge triggered 3-bit register with a synchronous reset and synchronous load. When the Load is high, data is loaded from Din into the register on the positive edge of Clk. Otherwise, if the Reset is high, the register is set with a value of 0.
Purpose: This module is used to create the register for CC.

Module: reg_unit.sv
Inputs: [15:0] Data_in, [15:0] IR, DRMUX, LD_REG, SR1MUX, Clk, Reset
Outputs: [15:0] SR1_Out, [15:0] SR2_Out
Description: This is a positive-edge triggered 16-bit register unit with a synchronous reset and synchronous load signals. The register unit acts as the register files in the SLC-3. The DRMUX, SRMUX, SR1Select, and SR2Select in the register unit will help to determine which registers from the register files to use.
Purpose: This module is used to create the register file for the SLC-3.

Module: ben.sv
Inputs: [15:0] DataPath, [15:0] IR, LoadCC, LoadBen, Clk, Reset
Outputs: ben
Description: This acts as the BEN in SLC-3. The value in DataPath will set the CC depending on whether it is zero, positive, or negative. Then, the 3-bit value nzp from the BR instruction will be compared with the CC. If they are the same, then ben is set to 1.
Purpose: This module is used as the ben in SLC-3.

Module: ALU.sv
Inputs: [15:0] sr2mux_out, [15:0] sr1mux_out, [1:0] ALUK
Outputs: [15:0] alu_out
Description: This acts as the ALU in SLC-3. The module will do distinct operations on both sr1mux_out and sr2mux_out depending on the ALUK input.
Purpose: This module is used as the ALU in SLC-3.

Module: multiplexer.sv
Inputs: [bits-1:0] in1, [bits-1:0] in2, [bits-1:0] in3, [bits-1:0] in4, select
Outputs: [bits-1:0] out
Description: This is a multiplxer unit where 2-to-1 MUX and 4-to-1 MUX are created. The module will take the parameter bits to determine how many bits the inputs for the MUX have. Then, the multiplexer will use the select bit to determine which input goes to the output.
Purpose: This module is used as the multiplexer unit to create distinct multiplexers in SLC-3.

Module: ISDU.sv
Inputs: Clk, Reset, Run, Continue, [3:0] Opcode, IR_5, IR_11, BEN
Outputs: LD_MDR, LD_MAR, 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, ALUK, Mem_OE, Mem_WE

Description: This is the positive-edge triggered flip-flop that has 32 states, which includes a halted states, and fetch state, decode state, and execute states depending on the instruction that is being executed. If the Reset signal is high, the state machine will return to the halted state. During each state, all the output signals such as LD_MDR, LD_MAR, etc. will be modified depending on the state it is at. The default value for all the outputs is set to 0.
Purpose: This module is used as a state machine to help SLC-3 go through the fetch, decode, and execute cycle.

Module: test_memory.sv
Inputs: Reset, Clk, [15:0] data, [9:0] address, rden, wren
Outputs: [15:0] readout
Description: This is a simulation of on-chip memory that uses positive-edge triggered flip flop to read and write the memory at the input address depending on the rden and wren signals.
Purpose: This module is used as the on-chip memory for the simulation of the SLC-3.

Module: memory_contents.sv
Inputs:
Outputs:
Description: This module is used to store the instructions at each memory address for the SLC-3 to access during the test.
Purpose: This module is used to store the memory content for the test memory.

Module: Mem2IO.sv
Inputs: Clk, Reset, OE, WE, [15:0] ADDR, [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: This is a positive-edge triggered flip flop that resets the HEX display when Reset is high and displays the Data_from_CPU when the address is xFFFF.If OE is high, then the value from switches will be stored in Data_to_CPU. Otherwise, Data_to_CPU will store the Data_from_SRAM.
Purpose: This module is used to control the memory input and output depending on the address given in order to determine whether to use the value from switches or SRAM.

Module: Instantiateram.sv
Inputs: Clk, Reset
Outputs: [15:0] ADDR, [15:0] data, wren
Description: This is a positive-edge triggered flip flop that resets the state for on-chip memory to mem_write and address to 0 when Reset is high. It also stores memory content for each address for the SLC-3.
Purpose: This module is used to instantiate on-chip memory for the SLC-3.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: This is the Hex driver that takes in 4-bit value In0 from the register in order to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.
Purpose: This module is used to show the value in the adder in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: Synchronizers.sv
Inputs: clk, Reset, d
Outputs: q
Description: These are the synchronizers that use positive-edge-triggered flip-flops to set the output value q to the input value d at every rising edge of the clock. During the rising edge of Reset signal, the synchronizer with reset to 0 will turn q to 0 while the synchronizer with reset to 1 will turn q to 1.
Purpose: This module is used as the synchronizer for bringing asynchronous signals into the FPGA board.

Module: SLC3_2.sv
Inputs:
Outputs:
Description: This module stores all the constants, such as op_ADD, op_AND, etc, and functions, such as opCLR.
Purpose: This module is used to store all the constants and functions for the SLC-3.

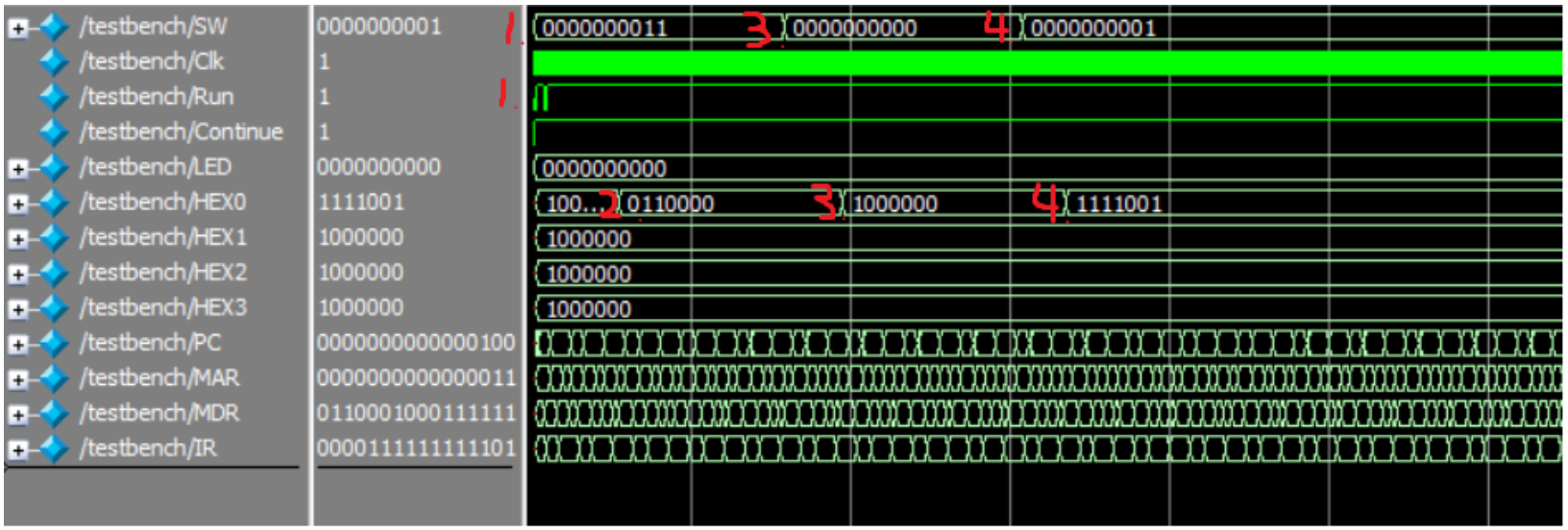
Module: slc3.sv
Inputs: [9:0] SW, Clk, Reset, Run, Continue, [15:0] Data_from_SRAM
Outputs: [9:0] LED, OE, WE, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [15:0] ADDR, [15:0] Data_to_SRAM
Description: This module instantiates all the modules that are used in the SLC-3 processor, including all the multiplexers, Mem2IO, ISDU, registers, etc, by using the input signals.
Purpose: This module is used as the organizer for the SLC-3.

Module: slc3_sramtop.sv
Inputs: [9:0] SW, Clk, Reset, Run, Continue
Outputs: [9:0] LED, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3
Description: This module uses physical RAM and synthesizes all the modules used for SLC-3.
Purpose: This module is used as the top level for the SLC-3 with physical RAM.

Module: slc3_testtop.sv
Inputs: [9:0] SW, Clk, Reset, Run, Continue
Outputs: [9:0] LED, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3
Description: This module uses test_memory and synthesizes all the modules used for SLC-3.
Purpose: This module is used as the top level for the SLC-3 with simulated RAM.

Annotated Simulation Trace

I/O Test 1



1. To test the program I/O Test 1, we set the switch value to be hex x0003, which is binary 10'b0000000011. Then we press the run button, which starts the program.

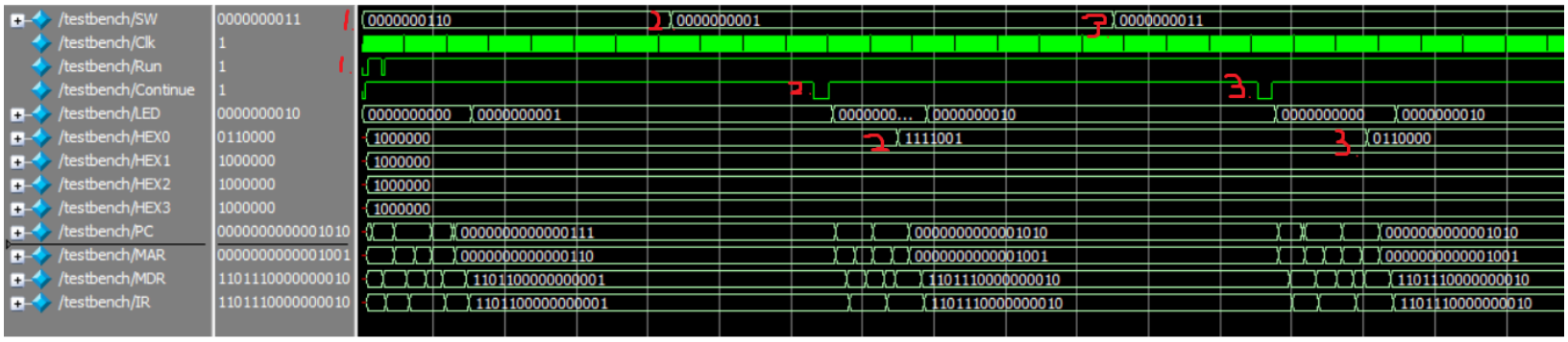
2. Once the program starts, we can see that the HEX displays display 0003 (as 7'b1000000 corresponds to '0' and 7'b0110000 corresponds to '3'). In other words, the HEX displays are beginning to always display the value of the switches, which at this point in time is still set to x0003.

3. We now set the switches to hex x0000, and we see that the HEX displays also change accordingly to display the value of the switches at all times. All of the HEX display values are now 7'b1000000, which corresponds to '0', as expected.

4. We now set the switches to hex x0001, and we see that the HEX displays also change accordingly to display the value of the switches at all times. Hence, the HEX displays display 0001 (as 7'b1111001 corresponds to '1' and 7'b1000000 corresponds to '0'.

Figure 3. Simulation Annotation for I/O Test 1

I/O Test 2



1. To test the program I/O Test 2, we set the switch value to be hex x0006, which is binary 10'b0000000110. Then we press the run button, which starts the program.

2. We can see that the HEX displays do not change (remain initialized at 0000 as 7'b1000000 corresponds to '0'), even after we set the switch to be hex x0001. Only after we press the continue button do the HEX displays update to display the value of the switches, which is now x0001 (7'b1111001 corresponds to '1').

3. We now set the switches to be hex x0003. The HEX displays, once again, do not update immediately to display the new switch value. Only once we press the continue button do the HEX displays update to display 0003 (as 7'b0110000 corresponds to '3').

Figure 4. Simulation Annotation for I/O Test 2

Self-Modifying Code

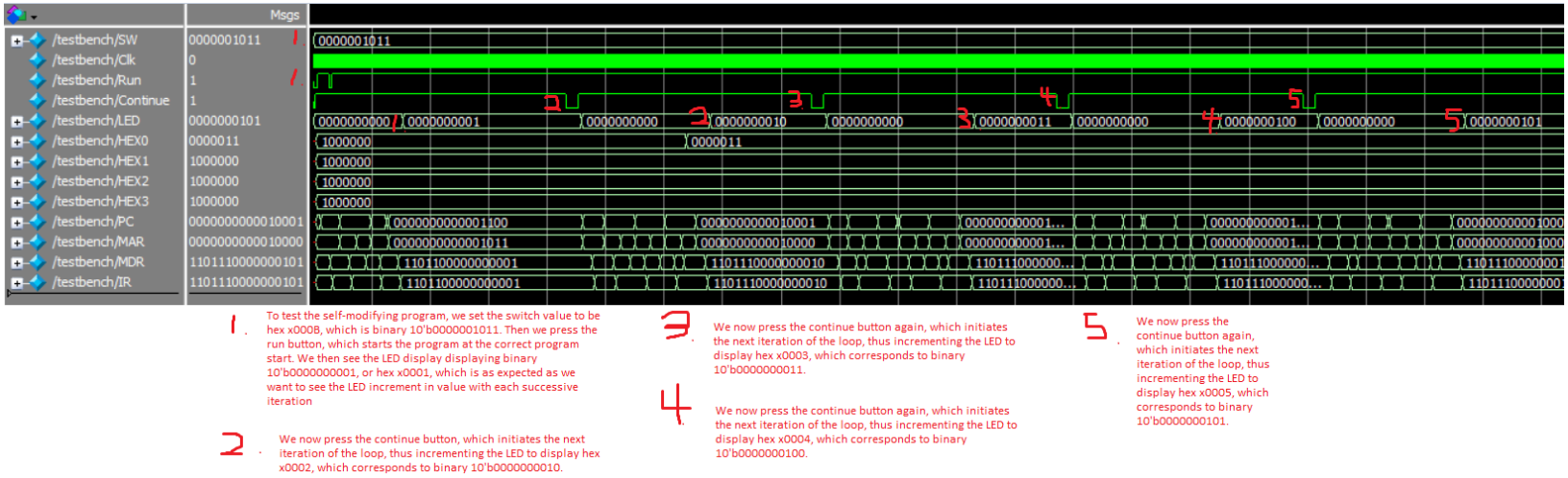


Figure 5. Simulation Annotation for Self-Modifying Code

XOR

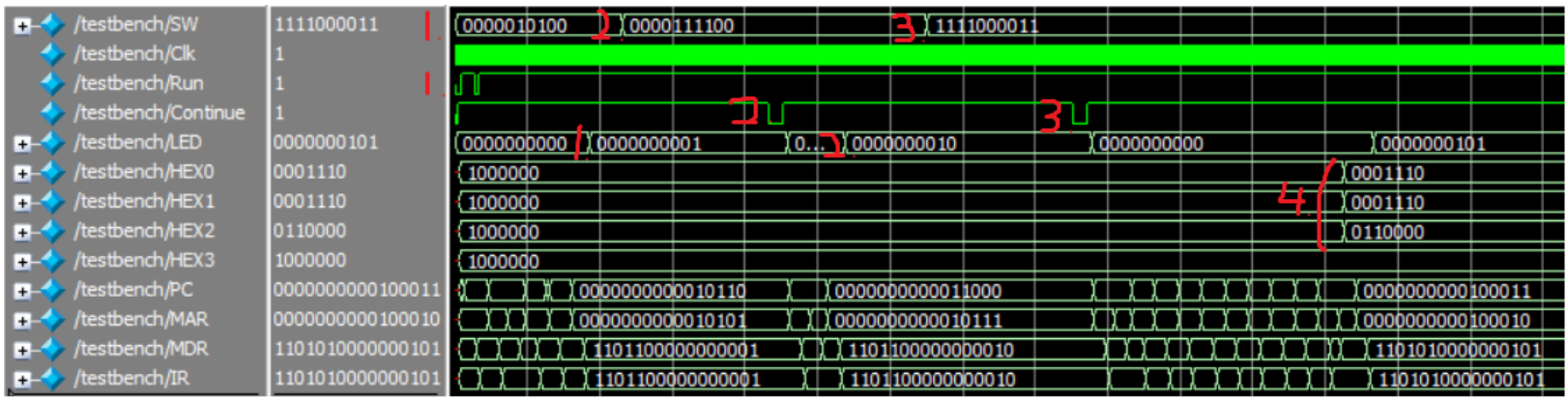
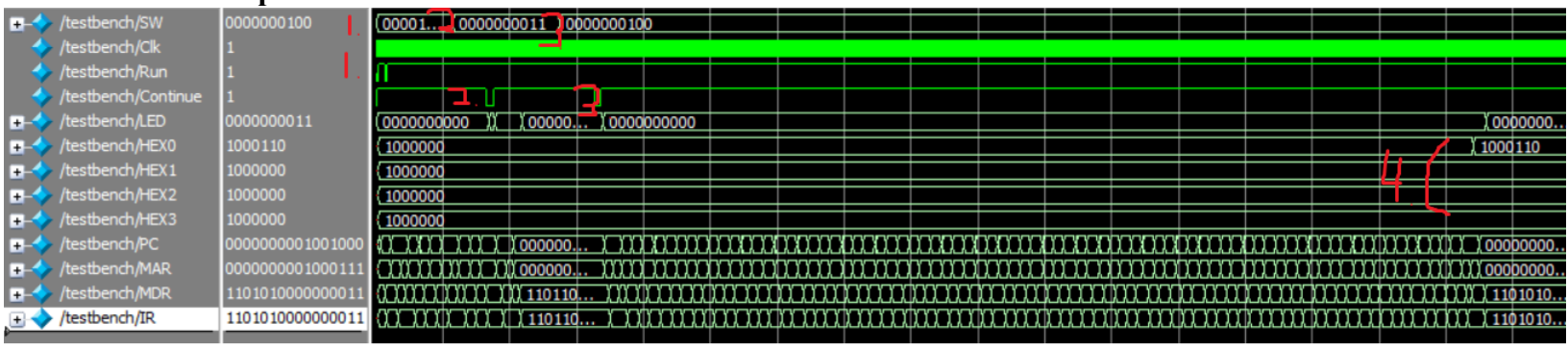


Figure 6. Simulation Annotation for XOR

Multiplier



1. To test the multiplication program, we set the switch value to be HEX x0031, which is binary 10'b0000110001. Then we press the run button, which starts the multiplication test program.

2. Once we press the run button to initiate the program, we set the switch value to be our first input to the multiplication program. We set the switch in this case to 10'b0000000011, or hex x0003. We then press the continue button to load this input into the program.

3. Once we load the first input into the program, we set the switch value to be our second input. We set the switch in this case to 10'b0000000100, or hex x0004. We then press the continue button to load this input into the program and perform the computation.

4. After setting the second input and pressing the continue button, the program then computes the result, which is then displayed on the HEX displays. The HEX displays now display 000C (7'b1000000 corresponds to '0' and 7'b1000110 corresponds to 'C'), which is as expected because hex x0003 multiplied with hex x0004 produces output hex x000C.

Figure 7. Simulation Annotation for Multiplier

Sort

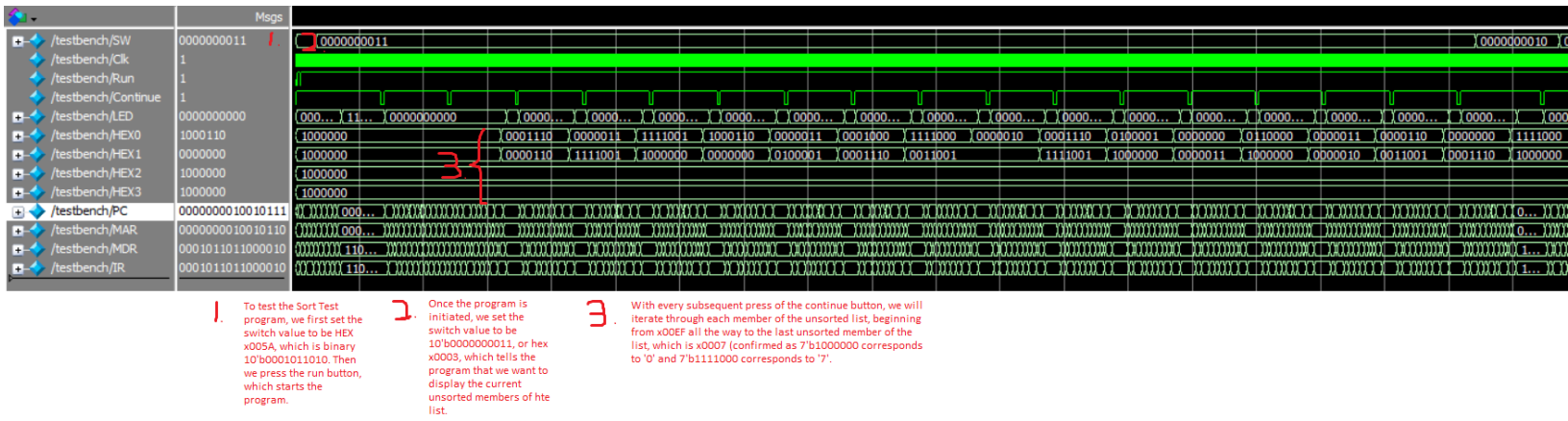


Figure 8. Simulation Annotation for Sort (Display unsorted)

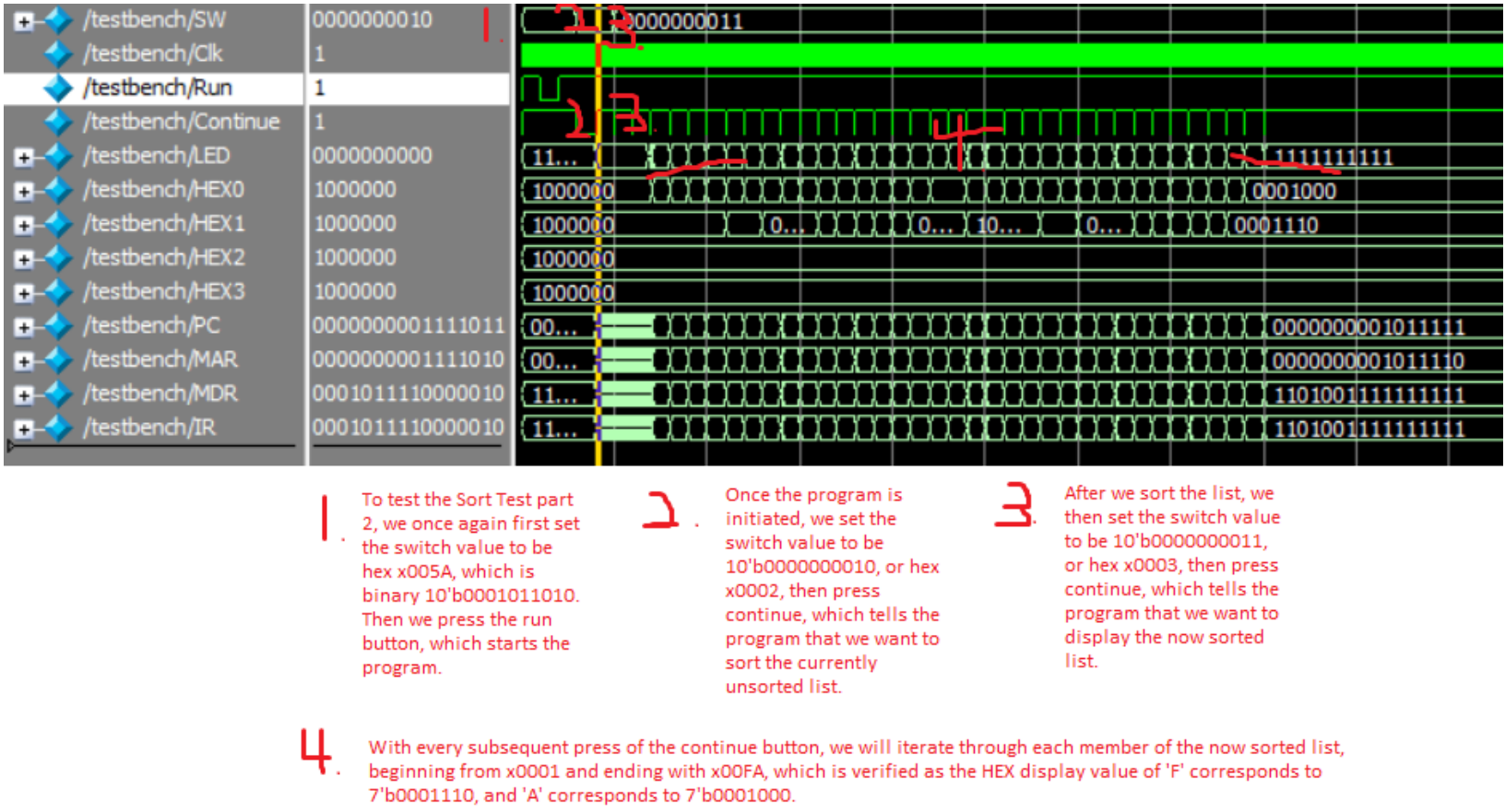


Figure 9. Simulation Annotation for Sort (Display sorted)

Post-Lab Questions

1. Fill in the table your design's statistics.

	Multiplier
LUTs	904
DSP	0
Memory (BRAM)	18,432 / 1,677,312
Flip-Flop	261
Max Frequency	65.53 MHz
Static Power	90.00 mW
Dynamic Power	9.63 mW
Total Power	110.48 mW

Document any problems you encountered and your solutions to them.

We did not run into any particular issues during week one of the lab. However, when implementing the decode and execute phases, we encountered a few problems. First, we noticed in our modelsim that we kept looping in the fetch phase, meaning we kept updating the instruction register but did not actually decode or execute any of the instructions. We were able to solve this by simply going back over our state transitions and making sure that we properly entered the "decode" state, which we were originally not doing.

Another major problem that we encountered was our branch instruction not executing properly. By using modelsim, we were able to isolate the problem to be due to our condition codes NZP not updating properly. We were using the instruction register instead of the datapath signal to update NZP, which was incorrect. To solve this issue, we simply replaced the IR with the datapath signal in our logic to compute the condition codes.

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

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

Mem2IO serves as the interface between the memory unit, input and output (switches and HEX displays), as well as the central processing unit. Essentially, it is the interface that controls the flow of data to and from the memory and the CPU and the input/outputs of the processor. The input signals OE and WE tell it when to read or write the memory. More importantly, the input ADDR tells it when to store the value of the switches to Data_to_CPU.

What is the difference between BR and JMP instructions?

Both BR and JMP instructions modify the PC; however, there are some differences between these two instructions. The first difference between BR and JMP is that BR has a condition code, which only allows it to change PC if it fits the conditions (i.e. if any condition code matches the condition stored in the status/condition register). On the other hand, JMP simply just jumps to the assigned PC. Furthermore, BR can only go to the PC within a certain range of the current PC as it can only modify the current PC by adding a sign-extended 9-bit PC offset. However, JMP can jump to any PC since it uses the 16-bit value stored in the states source 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?

R signal is used as the ready signal in Patt and Patel to signal that the memory is ready to be accessed/modified. It is important because it may take several clock cycles for the memory to be ready. Accessing the memory before it is ready may lead to corrupted or incorrect data being accessed. To compensate for this signal, we simply add 3 additional waiting states to those states that need the R signal (i.e. involve the accessing or writing of data from/to memory. Thus, instead of waiting for the R signal, we simply go through additional states in order to guarantee that the memory is ready to be accessed. However, this might cause the state machine to run slower with the additional states. Not to mention, if the input to the memory were to change during the three waiting states where the data is written to the memory, then there might be errors.

Conclusions

Through this lab, we successfully implemented a modified version of the LC-3 ISA. The processor incorporates three components, the processing unit, memory unit, as well as the input/output interface. Together, the three components work together to control the sequence of operations fetch, decode, and execute. The instructions implemented in our modified processor include ADD (2 variations), AND (2 variations), NOT, BR. JMP, JSR, LDR, STR, and PSE/PAUSE.

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

Overall, our design works well and passes all the given tests. As mentioned above, we did originally run into errors with our transition from the fetch to the decode and execute states, which was easily solved through the analysis of the modelsim and noting that we were stuck in a loop in the fetch state. The fix was simple as we noticed the next state transition of our final state of the fetch phase going back to the first state of the fetch phase instead of the decode phase. Making the appropriate next state change fixed our issue. On the other hand, we also originally ran into issues involving the BR branch instruction. Analysis of the modelsim showed that our condition codes were not updating properly. Looking at our branch enable module, we noticed that we were using the IR instead of the datapath signal to compute the condition codes. A simple switch of the signals enabled us to fix our issue.

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 does not get changed.

No, there was nothing particularly difficult to understand in the lab manual. The provided control signals summary was particularly helpful in writing our modules for the SLC-3 program.

Extra Credits

XOR: PC = x1A (Decimal = 26)

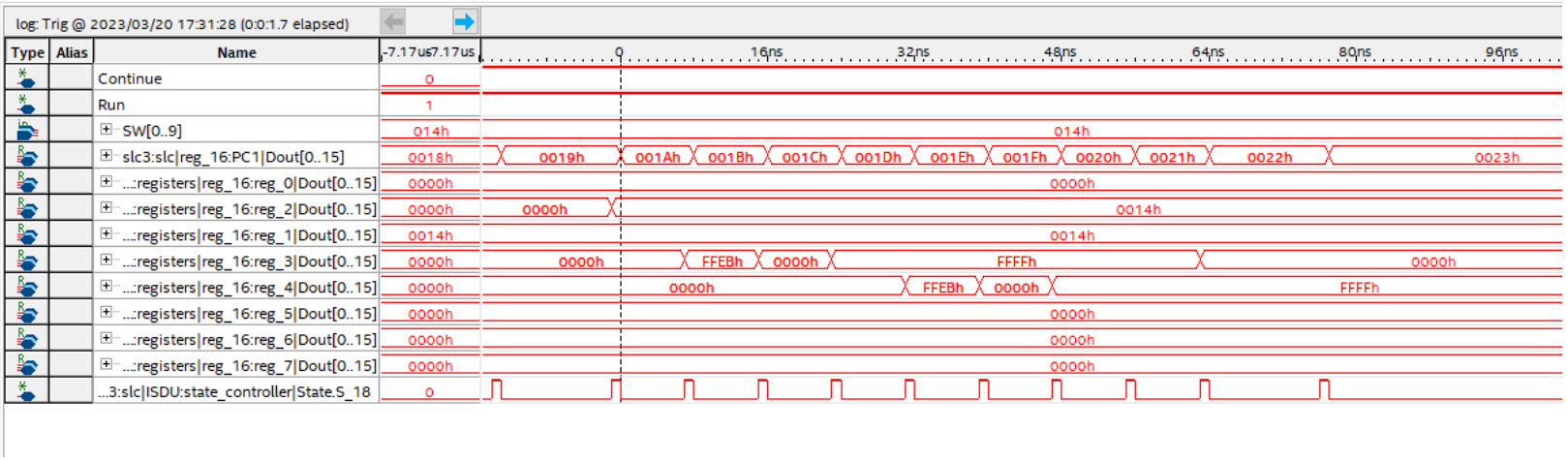


Figure 10. Signal Tap for XOR test

When PC is at x22 (Decimal = 34), the output is ready in register 3 as shown. Thus, subtracting x1A from x22, we get 8 instructions. 8 instructions / 64 cycles * 50,000,000 cycles per second = 6.25MIPS.

Multiplier: PC = x3A (Decimal = 58)

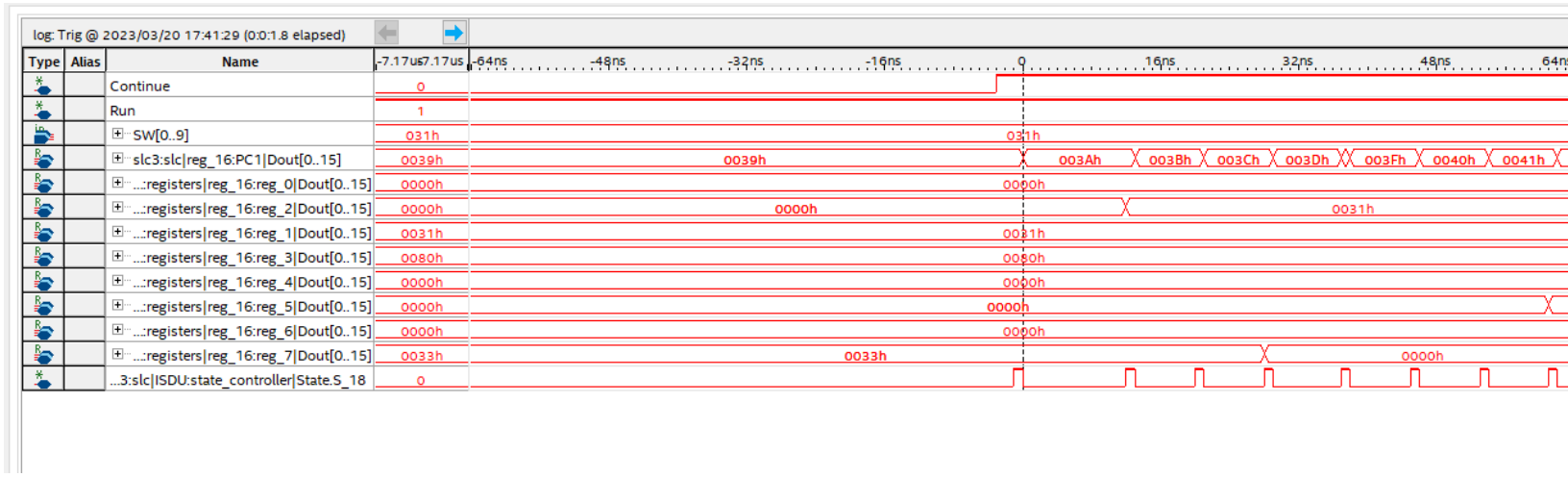


Figure 11. Signal Tap for the start of the Multiplier test

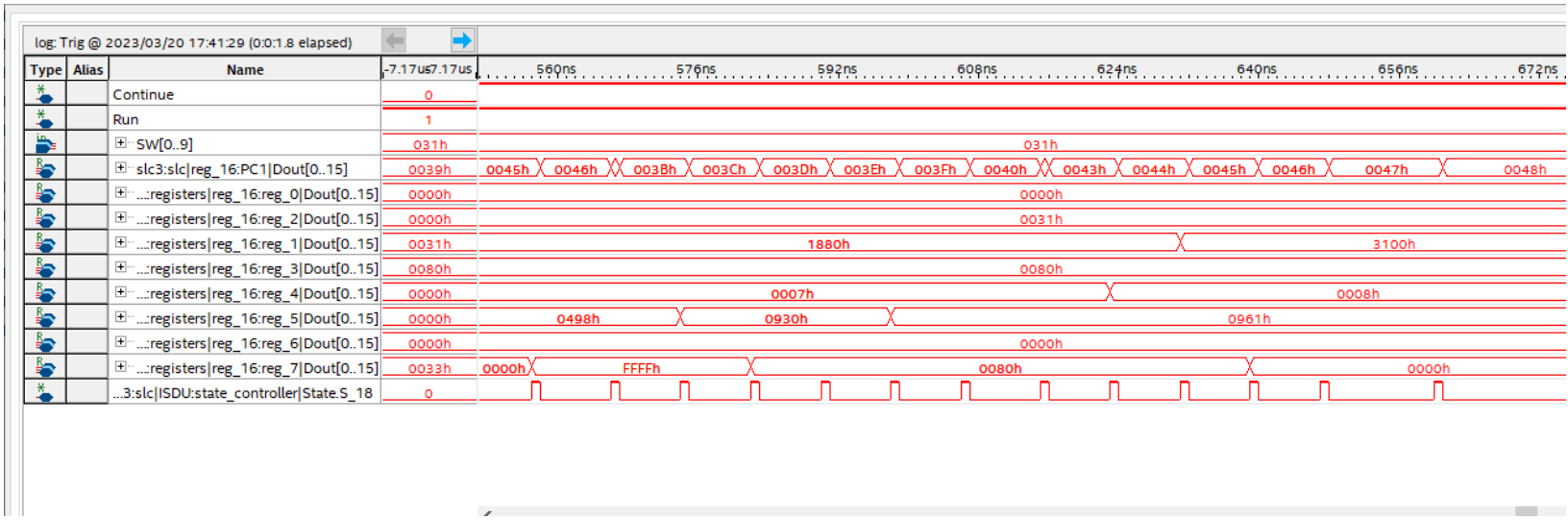


Figure 12. Signal Tap for the end of the Multiplier test

When PC is at x47 (Decimal = 71), the output is ready in register 5 as shown. Thus, subtracting x3A from x47, we get 13 instructions. However, accounting for the iteration loop, there are actually 79 instructions after counting. There are 648 cycles. 79 instructions / 648 cycles * 50,000,000 cycles per second = 6.095MIPS.

Sort: PC = x77 (Decimal = 119)

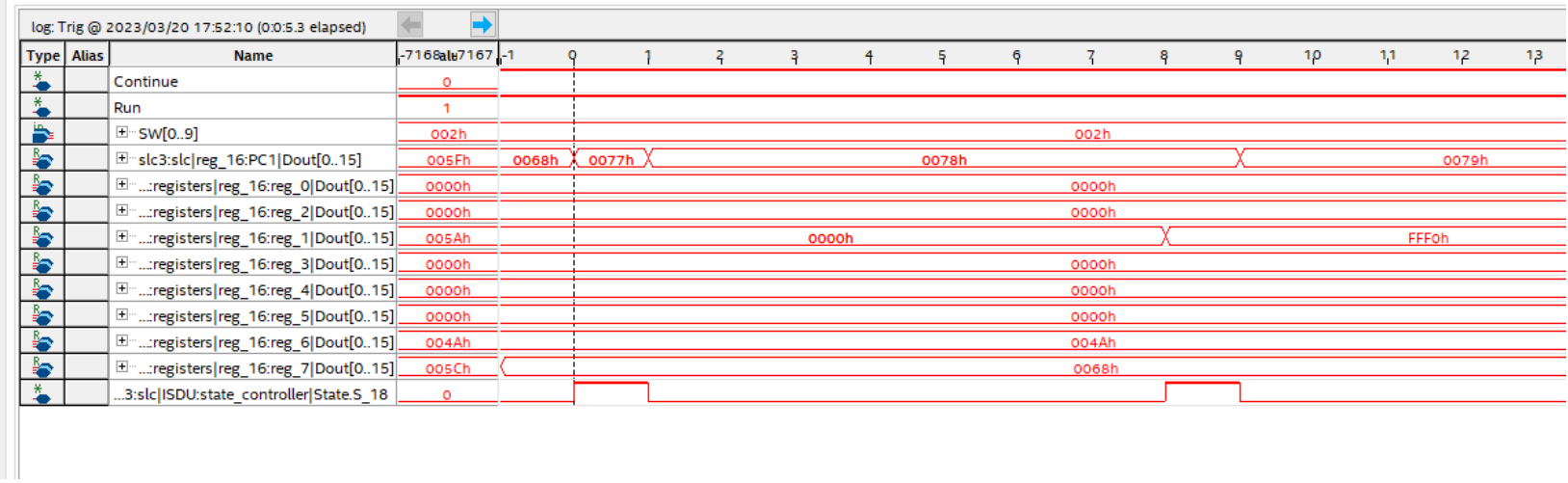


Figure 13. Signal Tap for the start of the Sort test

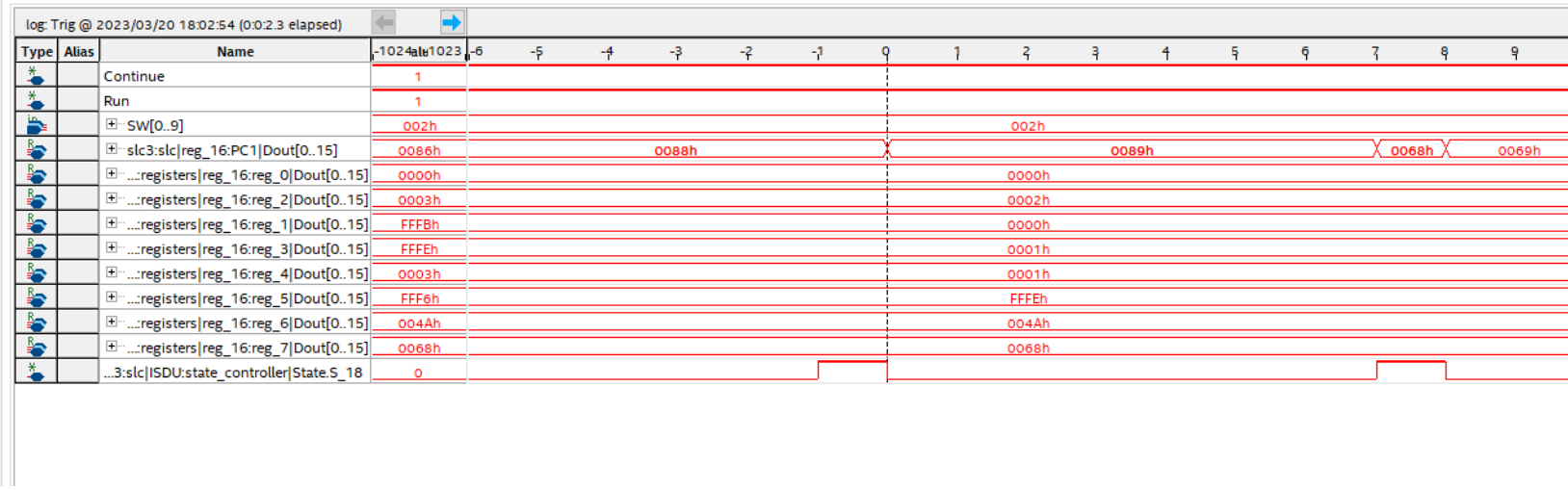


Figure 14. Signal Tap for the end of the Sort test

When PC is at x89 (Decimal = 137), the output is ready. Thus, subtracting x77 from x89, we get 18 instructions. Minus the four instructions that do not have to do with the loop, we get 14 instructions. Thus, we need 14 instructions to do a bubble sort once and have to do this 16 times as there are 16 numbers. Thus, accounting for the iteration loop, there are actually 14*16+4 = instructions after counting. As each loop costs about 115 cycles, there are 115*16 +4 = 1844 cycles. 228 instructions / 1844 cycles * 50,000,000 cycles per second = 6.18MIPS.

Thus, the average will be (6.25 + 6.095 + 6.18) / 3 = 6.175 MIPS. To estimate the instructions for each test, I first count the changes in PC for XOR. For the other two tests with loops, I count the PC changes in a period and then times the number of loops it runs. For the clock cycle, I did the same thing by looking at how many cycles have passed for XOR or in a loop for the other tests. Then, estimate it by timing the number of loops it runs.