ECE 385

Fall 2014

Experiment #5

SLC-3 CPU Implementation on FPGA with SystemVerilog

Luke Jacobs, Arya Kothari

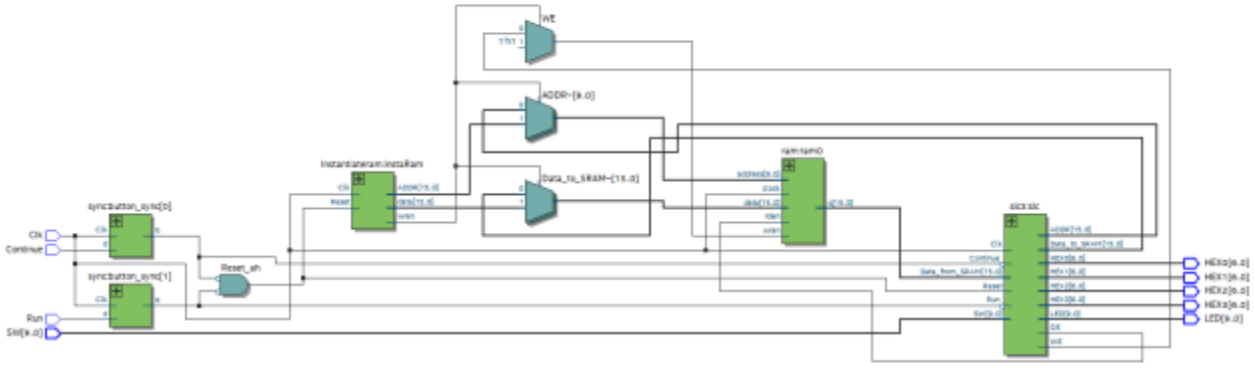
ABD

Andrew Gacek

The function of the SLC-3 processor is to execute instructions from memory. The advantage of using a processor on an FPGA instead of a full digital design solution is that a CPU is flexible; development on a CPU consists of writing relatively simple assembly instructions instead of wiring together complex logic. This way, designers can interact with sensors and outputs in a sequential way without enormous state machines. In our design, we were able to have the processor interact with the outside world by wiring in memory-mapped IO, the switches and buttons and hex display. This enabled the processor to run many different simple programs, like sorting and an XOR routine.

The SLC-3 operates by using the Fetch-Decode-Execute cycle to change memory, accept inputs, and operate on outputs. There are two main components of the SLC-3: the state machine (housed in the control unit) and the datapath (how data flows given the control signals from the state machine).

The SLC-3 has three main cycles, the Fetch, the Decode, and the Execute cycle. Each of these cycles complete important operations in order for the processor to work. In the Fetch cycle, the processor takes in the address of the instruction which is stored in PC. Then from the address that is given an instruction associated with that address is sent to the instruction register from the memory.. This leads into the decode cycle. In this cycle the instruction that is stored in the IR register is read and the output signals, in the control unit, for the appropriate operation are set. For example, if the opcode for the instruction is 0110 then the output signals to complete the LDR instruction are set as well as the next state that starts the execution. Finally in the execute cycle the actual operation is completed. Depending on the instruction many things can happen. The SLC-3 can do these operations: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, and PAUSE. Depending on the instruction, the output signals from the state machine control the path the data takes to make sure it goes through the correct operations. For example if the instruction was ADD, then the data will have to go through the register file and the ALU to compute the addition.



SLC-3 Block Diagram

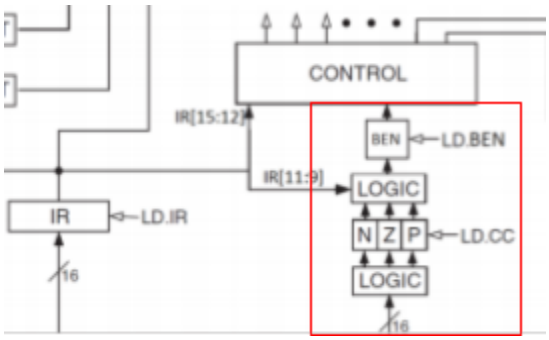
Module: branch_logic.sv

Inputs: Clk, Reset, [15:0] Bus_In, [2:0] IR_part, LOAD_CC, LOAD_BEN,

Outputs: BEN_out

Description: In this module the logic to decide whether to branch or not is held. It also holds the register for the NZP values, these are updated when LOAD_CC is high in the states that setCC. For the branch logic, the NZP values are anded with the nzp values on the bus. So if (N&n) | (Z&z) | (P&p) equates to 1 then BEN_out is 1. This means that branch is enabled, but if the value equates to 0 then BEN_out is 0, meaning that branch is not enabled.

Purpose: Computes branch logic to decide whether to branch or not as well as setting CC.



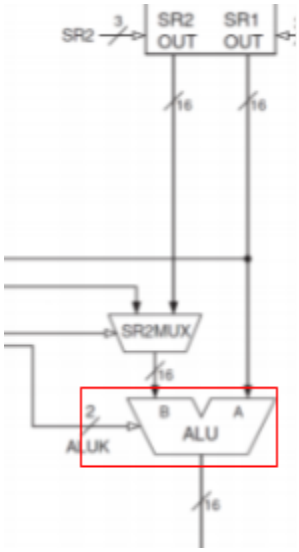
Module: ALU.sv

Inputs: [15:0] A, [15:0] B, [1:0] aluk

Outputs: [15:0] out

Description: In this module we implement the ALU that will be used in the instructions for ADD, AND, and NOT. The two inputs A and B are taken from the register file. These inputs are chosen by the register file and the control signals that were set in the decode cycle. The aluk value, set by the control unit, controls the mux that chooses which operation to compute. So, if aluk is 00 then the addition operation is chosen. The ALU can also just pass through the A input. This is used for the LDR and STR instructions.

Purpose: Compute the ADD, AND, and NOT instructions. Also used for the LDR and STR functions but no computation occurs.



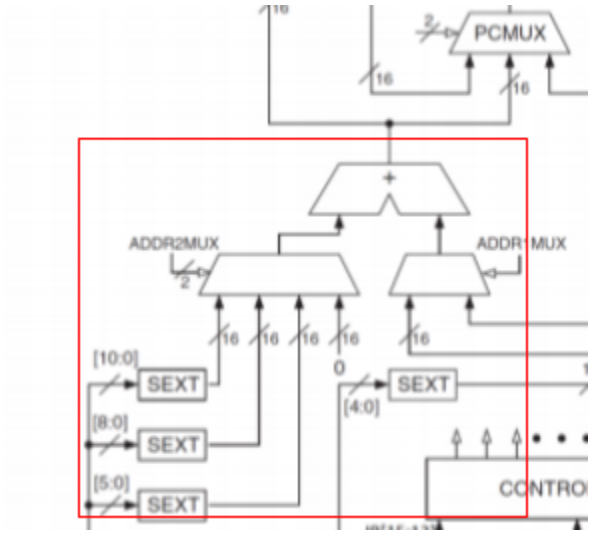
Module: address_additions.sv

Inputs: [15:0] sext_11bit, [15:0] sext_9bit, [15:0] sext_6bit, [15:0] PC, [1:0] ADDR2MUX, ADDR1MUX, [15:0] SR1_out.

Outputs: [15:0] ADDR_NEW

Description: In this module we implemented the address addition. Depending on the specific operation the ADDR2MUX will choose to either add the 11, 9, or 6 bit sext value with either the value, chosen by ADDR1MUX, of PC or SR1 (comes from reg file). For example, the JSR instruction adds the sext_11bit input with the PC input to jump to a new address. The values that choose which inputs to add are set by the control unit.

Purpose: Increment or decrement address based on the specific operation and the inputs.



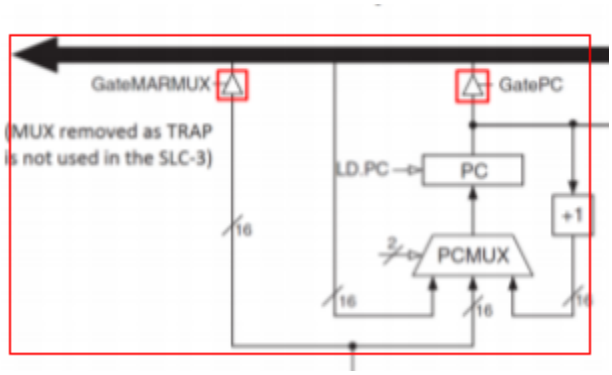
Module: PC_operations.sv

Inputs: [15:0] PC_out, [15:0] PCMUX_sig, [15:0] AddrAdder_out, [15:0] Bus_out

Outputs: [15:0] PC_in

Description: In this module we implement the PC operations. There are three ways to modify PC. Increment PC by 1, PC becomes the value of the address addition (AddrAdder_out input), or PC becomes the value on the Bus (Bus_out input). These three operations are all then put into a mux which is controlled by PCMUX_sig. This signal comes from the control unit and picks the correct output depending on the instruction. For example, in JSR, PCMUX_sig is set to 01 to tell the mux to pick the AddrAdder_out input.

Purpose: Complete operations that are done on PC in some instructions.



Module: mux_2_to_1.sv

Inputs: [width-1:0] a, [width-1:0] b, select

Outputs: [width-1:0] out

Description: This module generates a 2 to 1 mux with a signal width equal to the specified parameter “width.”

Purpose: The SLC-3 uses many 2 to 1 muxes, so we decided to make a parameterized module that could be inserted as a generic 2 to 1 mux. The parameter “width” controls the width of the input signals and the output so that multiple sizes of muxes could be generated.

Module: datapath.sv

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

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

Description: The datapath wires together all the lower-level components, namely the PC operations module, the MDR register and bundled operations, the IR register, MAR, the register file, the ALU, the bus, branch logic, and combinational logic for address additions.

Purpose: The purpose of the datapath is to provide routing for individual components apart from the state machine and memory subsystem.

Module: slc3_testtop.sv

Inputs: [9:0] SW, Clk, Run, Continue

Outputs: [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3, CE, UB, LB, OE, WE, [19:0] ADDR, [15:0] Data *(data is an in/out wire)*

Description: A simulated version of slc3_sramtop.sv that can work with ModelSim. The RAM unit is simulated by test_memory.sv.

Purpose: The purpose of this module is to connect the slc3 module with the test_memory module and connect the external IO (Hex display, switches) to both of them. It is the top-level module for simulating the CPU.

Module: test_memory.sv

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

Outputs: [15:0] readout

Description: Test memory is non-synthesizable memory that simulates RAM. Accepts simple control signals and returns a 16-bit readout.

Purpose: Test memory is used for simulating the CPU.

Module: Bus.sv

Inputs: [15:0] In_PC, [15:0] In_ALU, [15:0] In_MARMUX, [15:0] In_MDR, [3:0] select_1hot

Outputs: [15:0] bus_out

Description: The bus is a 4 to 1 mux that selects its output based off of the one hot vector of {GateMDR, GateMARMUX, GateALU, GatePC}.

Purpose: The bus sends its output to essential parts of the datapath, namely IR, the branch logic, the register file, and PC. Memory operations happen through the bus.

Module: slc3_sramtop.sv

Inputs: [9:0] SW, Clk, Run, Continue

Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3

Description: Wires the SLC unit, its on-chip RAM, and an Instantiateram module which loads the RAM.

Purpose: The purpose of this module is to be the top-level module controlling the FPGA. This is like slc3_testtop except the RAM module is on-chip memory.

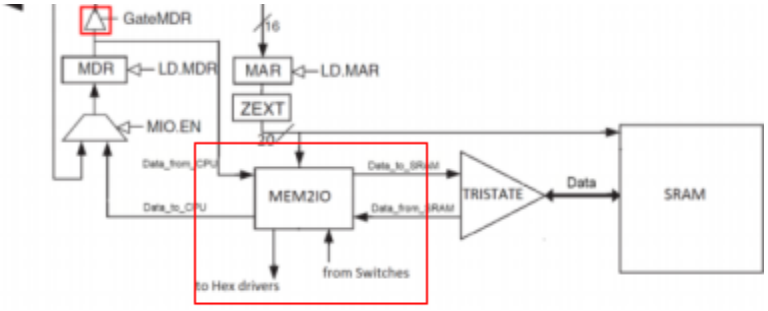
Module: Mem2IO.sv

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

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

Description: This module acts as the intermediate between the RAM module, the board IO, and the MDR register. When xFFFF is read, the switch values are returned, and when xFFFF is written, the hex registers are updated.

Purpose: The purpose of the Mem2IO module is to give an IO interface to the SLC-3 system. Without it, there would be no way for the user to interact with the machine.



Module: reg_16.sv (and reg_3.sv)

Inputs: Clk, Reset, Load

Outputs: [15:0] D, [15:0] Data_Out

Description: A 16-bit (and 3-bit) register that is made of 16 (and 3) flip-flops. Allows for reset and parallel load.

Purpose: Stores all the registers used in the SLC-3.

Module: SLC3_2.sv

Inputs: *None*

Outputs: *Functions*

Description: Contains functions that simplify the assembly-writing process. These are not synthesizable but they act as macros which allow us to write pseudoinstructions like CLR.

Purpose: Simplify writing assembly and provide us with the correct instruction set fields.

Module: Instantiateram.sv

Inputs: Reset, Clk

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

Description: This module acts as a large mux or a ROM that initializes the SRAM.

Purpose: This module writes the programs into the SRAM on the FPGA chip in this static file.

Module: ISDU.sv

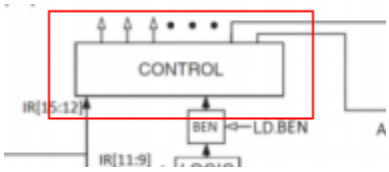
Inputs: Clk, Reset, Run, Continue, IR_5, IR_11, BEN, [3:0] Opcode

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

Description: The ISDU is the state machine for the CPU. It is coded in two parts: the next state logic and the output logic. The next state logic defines the graph for the state diagram; it shows how states transition and flow in and out of each other. The output logic defines which control signals are activated by the current state. This allows data to flow correctly in the datapath. For example, take the ADD instruction. The ADD instruction needs the following signals to be set for data to flow correctly within the system: SR1MUX=0, SR2MUX=IR[5], DRMUX=0, ALUK=01, GateALU=1, LD_REG=1, LD_CC=1. If the SR1MUX signal was, say, 1, then the register file would query the register indicated by the code at IR[11:9], which would not correspond with the

correct instruction field. In this way the ISDU implements the instruction on a field-by-field basis by setting the correct signals, some defined by IR.

Purpose: The purpose of the ISDU is to manage all the other components of the CPU and to feed the right inputs and outputs out of and into the correct registers.



Module: slc3.sv

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

Outputs: OE, WE, [6:0] HEX0-3, [15:0] ADDR, [15:0] Data_to SRAM

Description: This module is the one where the three big parts of the circuit are initialized and wired together. In this the MEM2IO module, the datapath module, and the IDSU module all get connected together using local variables made in the file. This module also assigns the LEDS to the IR register value.

Purpose: Connects main parts of circuit

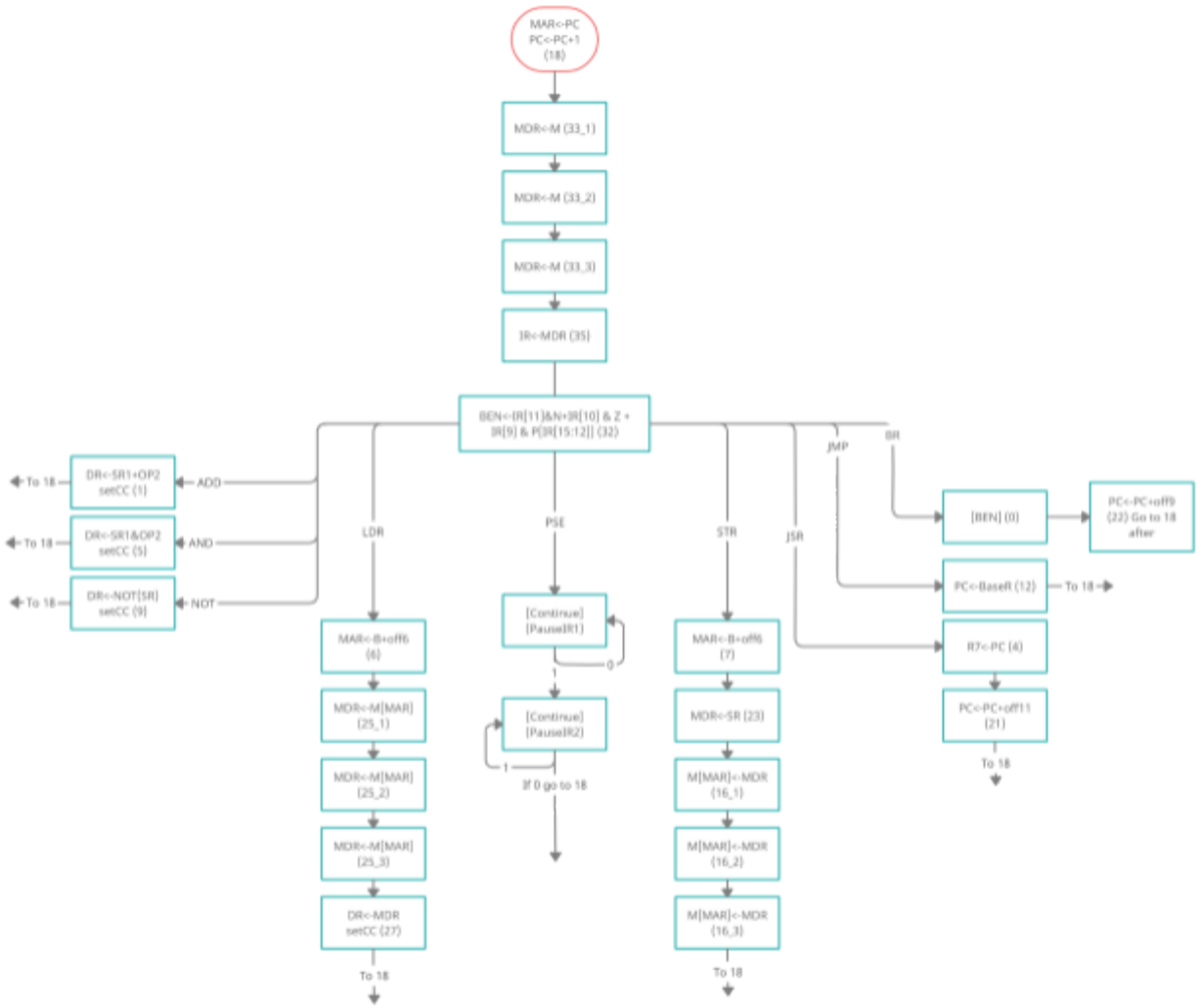
Module: memory_contents.sv

Inputs: None

Outputs: None

Description: This module populates the memory with the code needed for the test programs. This is done using codes that correspond to certain instructions in order to compute them.

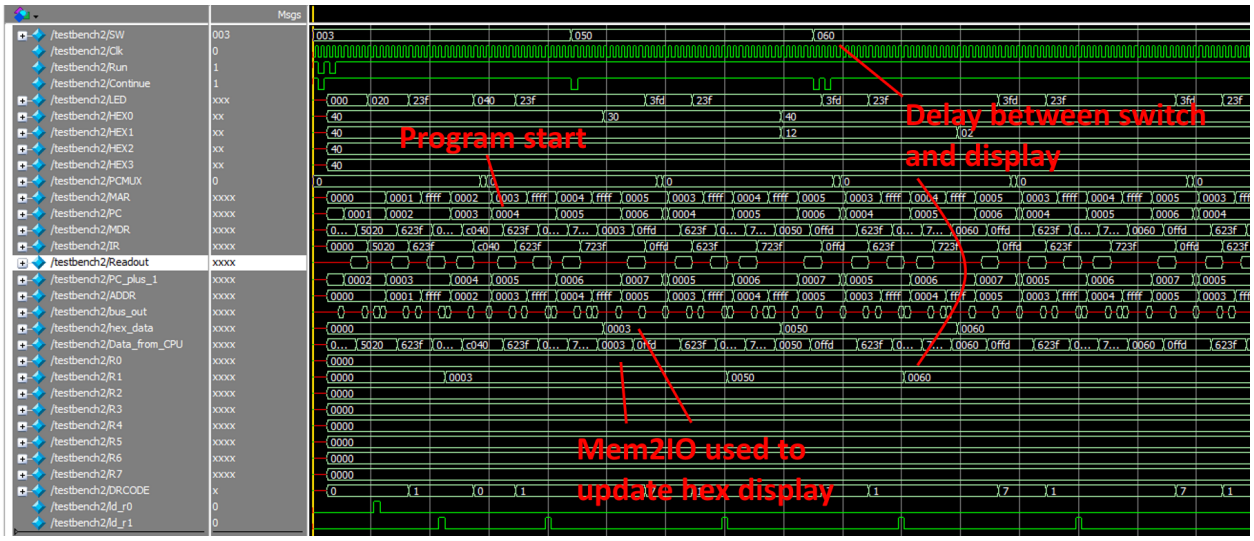
Purpose: Populates memory



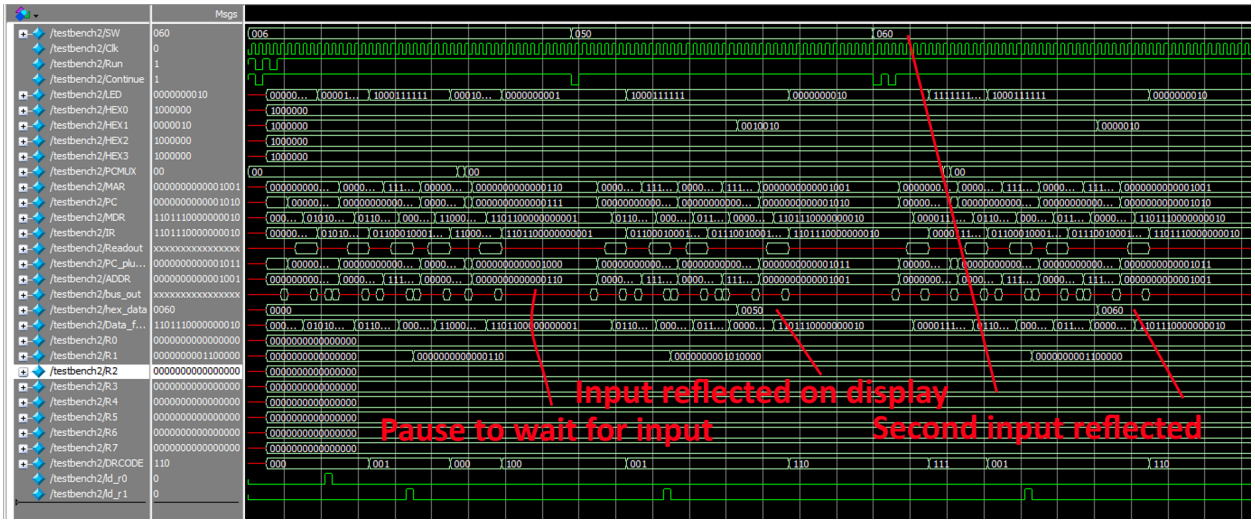
ISDU State Diagram

Simulation Waveforms:

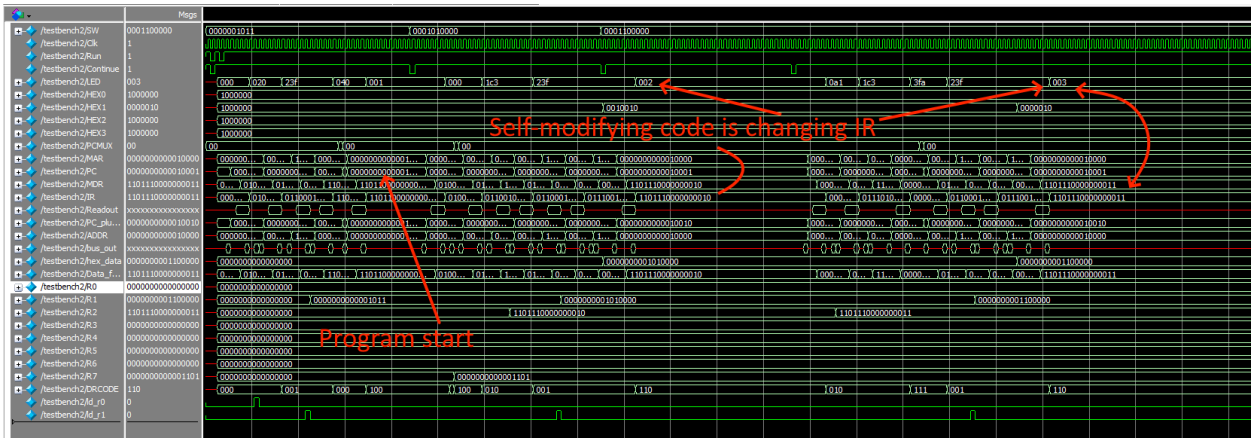
IO Test 1



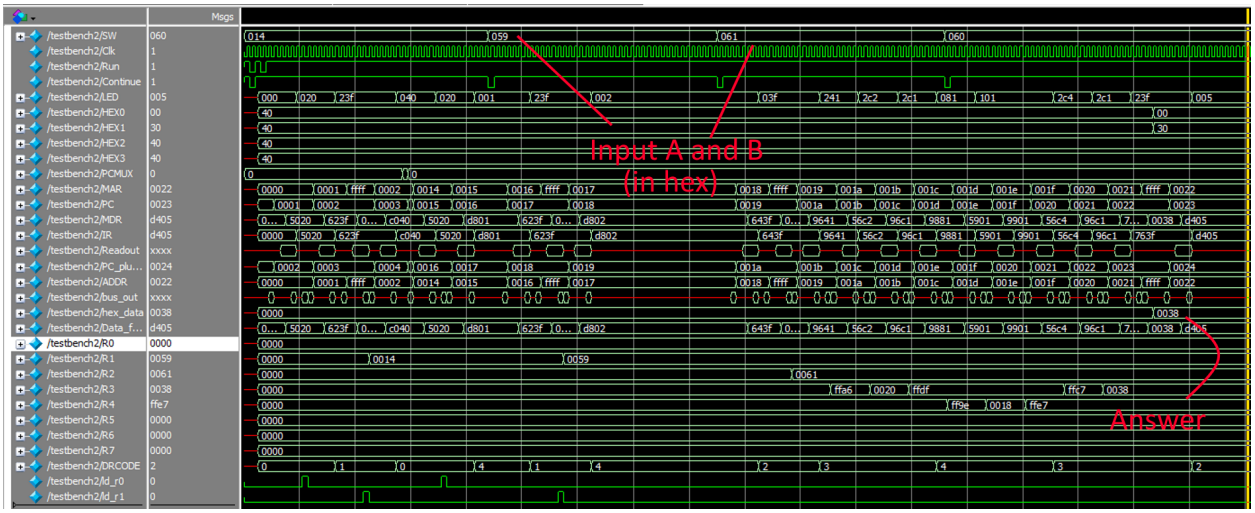
IO Test 2



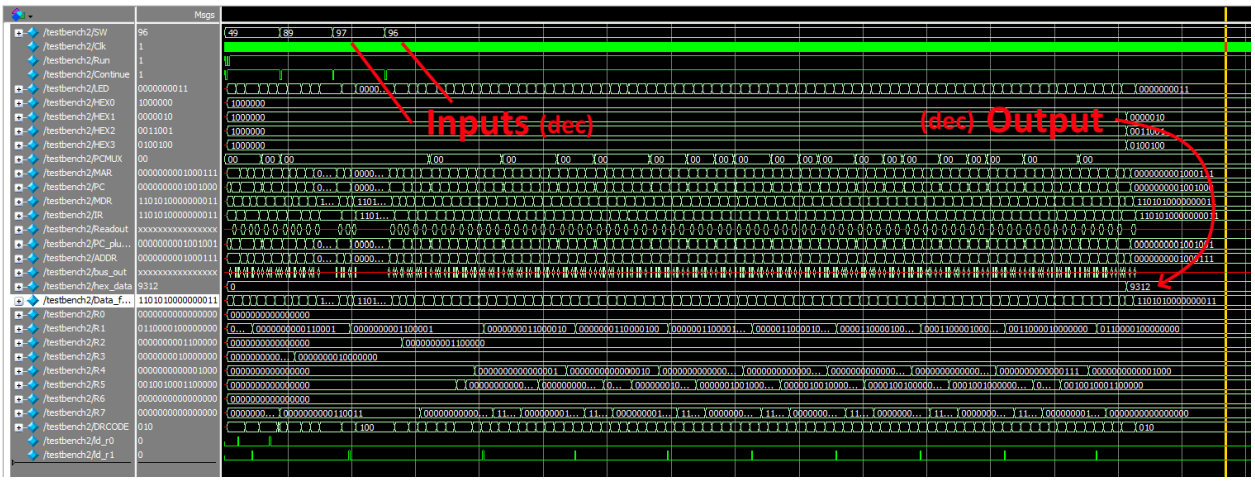
IO Test 3 (Self-modifying code)



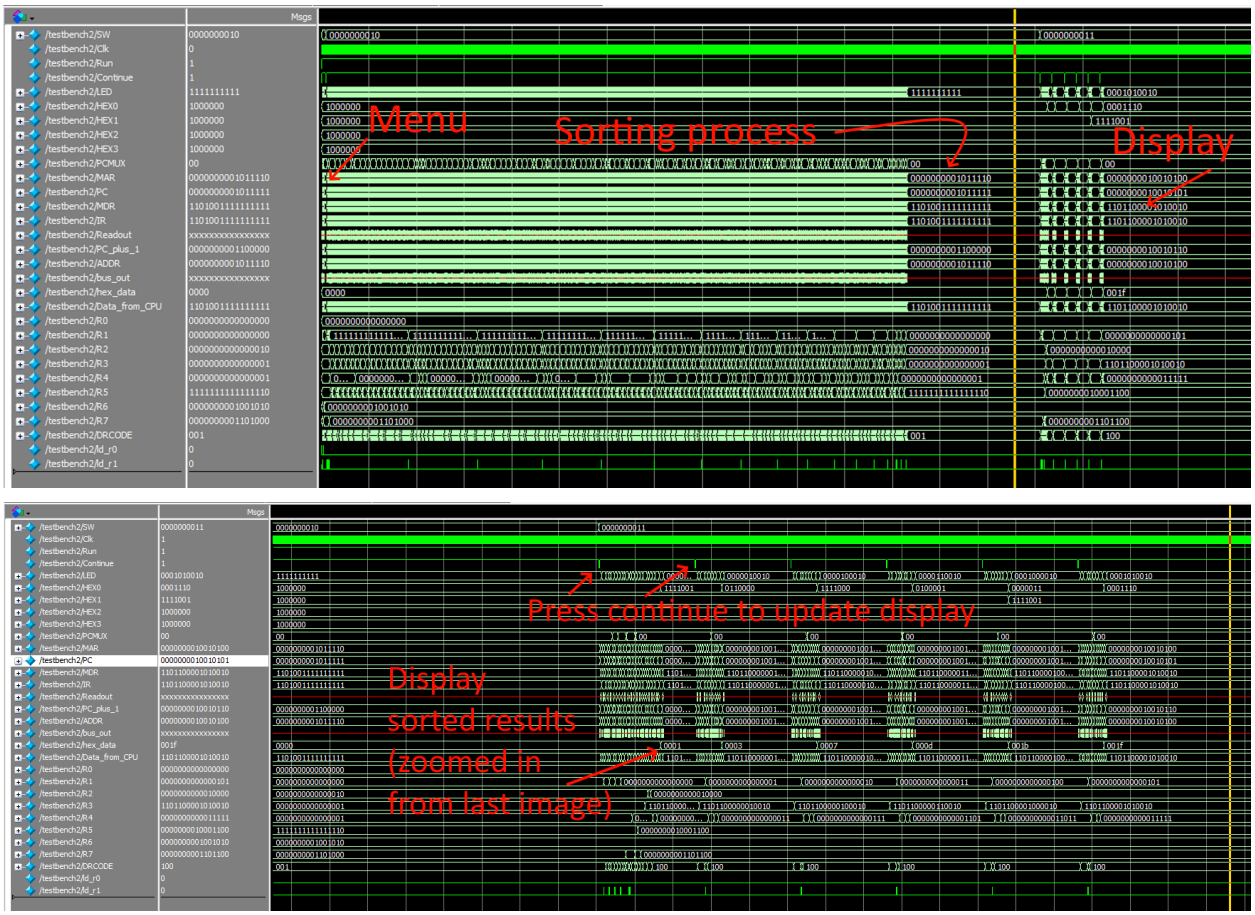
XOR Test



Multiplication Test



Sorting Algorithm



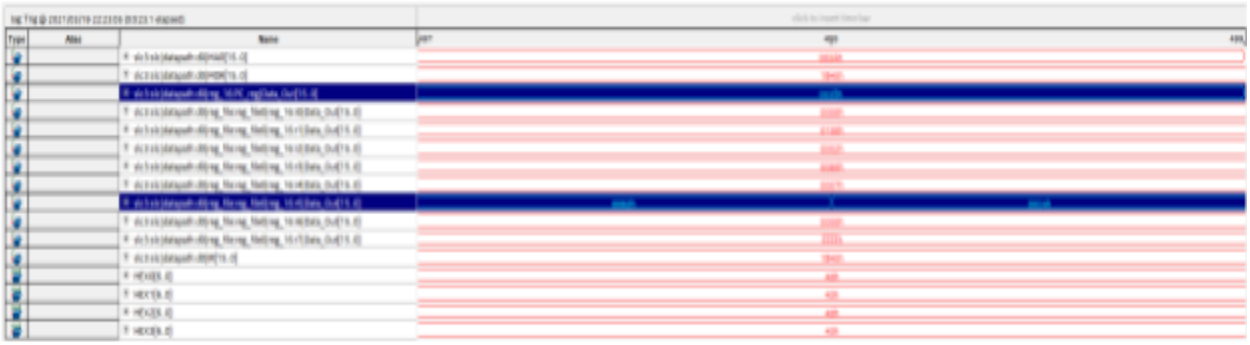
Act Once

Extra Credit: (Higher Quality Images in Zip File)

XOR Test

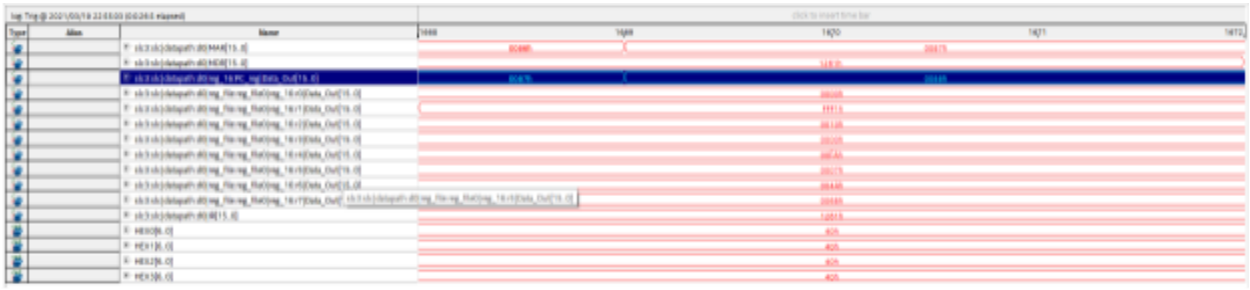
XOR Test
SignalTap

As you can see in the picture the output is available in R3 whenever PC is at instruction 0x21 at cycle 55 this means that there were 7 instructions until the output was available. This means that our circuit runs at (7/55)*50,000,000 = 6.36 MIPS



Multiplier Test SignalTap

As you can see in the picture the output is available in R5 whenever PC is at instruction 0x3B This means that the multiplier went through 104 instructions because it loops through 13 instructions 8 times. This output was available on cycle 498. So (104/498)*50,000,000 = 10.4 MIPS



Sort Test SignalTap

As you can see in the picture the output is available whenever PC is at instruction 0x88. This means that the test went through 1605 instructions. We calculated this by counting how many times the program looped and multiplied by the amount of instructions in each loop. You can see that instruction 0x88 occurs at cycle 1669. So (1605/1669)*50,000,000 = 48.1 MIPS

Average MIPS = 21.62

Post-Lab:

LUT	1934
DSP	0
Memory (BRAM)	119,808
Flip-Flop	2967

Frequency	90.24MHz
Static Power	90.07 mW
Dynamic Power	24.59 mW
Total Power	126.14 mW

MEM2IO is used for intercepting memory addresses to allow for memory-mapped IO. MEM2IO is the bridge between MDR and SRAM. If any MAR value other than 0xFFFF wants to be accessed, MEM2IO passes through the data received from SRAM to MDR. However, if the processor reads 0xFFFF, it returns the switch vector, and if the processor writes to 0xFFFF, it updates the hex drivers to display MDR on the hex display.

The difference between the two instructions is that BR sets PC to its current value plus an offset when its conditions are satisfied, but JMP sets PC to the value of a base register. This means that BR is fit for small jumps, but it is more limited in range than JMP. JMP is unconditional, whereas BR is conditional.

The purpose of the R signal is to indicate to the state machine whether memory is finished with the given operation (i.e. ready to move on). We compensate for the lack of this signal by adding more states to the machine, thus delaying execution so that the SRAM can respond. For read and write states we extended the operation to cover 3 sub-states, giving the memory 3 clock cycles to respond. The implication of this design is that the state machine does not ever stop - if for some reason the memory has not responded with a new read value by the time the 3rd read cycle ends, the processor is going to take the last (old) read value. With the R signal design, the state machine was partially asynchronous.

Conclusion:

Our design was perfectly functional. Some of the bugs we encountered were related to not specifying the correct control flow signals, but we were able to fix those by looking at ModelSim. This lab felt like the most constructive lab, and nothing was ambiguous.