ECE 385

Spring 2019

Experiment #6

Simple Computer SLC-3.2 in System Verilog

Rob Audino and Soham Karanjikar
Section ABJ, Friday 2:00-4:50
TA: David Zhang

Introduction:

In this lab, we were tasked with creating a simplified version of the LC-3 processor that we had learned about in ECE 120 and ECE 220. This simplified “SLC-3” was capable of less instructions than LC-3 but worked in a strikingly similar way. SLC-3 is essentially a 16-bit processor that performs three operations: fetch, decode, and execute. The fetch stage involved retrieving the desired instruction, whereas the decode stage determined what should be done given the desired instruction. Lastly, the execute function simply performed all of the details of the desired operation. In order to accomplish the task of building the SLC-3, we needed 3 ‘components’: memory, a CPU, and MEM2IO to facilitate data movement between the former two.

Written Description and Diagrams of SLC-3:

As we described earlier, SLC-3 is a simplified, stripped down version of LC-3, and it has less operations. Although SLC-3 is missing the STI, LDI, LEA, JSRR, RTI, LD, TRAP, RET, and ST operations, it does add a PAUSE state.

The Fetch-Decode-Execute cycle obviously begins with Fetch, where the operation is actually retrieved. The Fetch operation itself begins with PC being loaded into MAR, after which MDR is loaded with the contents of the address that was specified by MAR. Next, the contents of MDR are stored into IR, and PC is then incremented by 1.

The next step in the cycle is decode. This step involves the ISDU selecting what the output signals become based on the contents of IR. The decode step is followed by the execute step, where the actual intentions of the instruction are carried out. Depending on the output of the decode step, different parts of the processor choose whether or not to operate and if they do operate, select what to output. The outputs of the individual active parts of the processor combine into a single output signal that is sent to the MEM2IO component. From there, MEM2IO will use inputs from the ISDU in order to select what exactly to do with the output.

The inputs of SLC come from the FPGA board, which include the reset, run, and continue buttons as well as the sixteen switches. The outputs of the SLC are the four hexadecimal displays and twelve LED’s on the FPGA board. During the operation of SLC, if Pause is pressed at any given time, the LED’s will illuminate with whatever is in IR, and will stay on until the continue button is pressed.

Written Description of all .sv Modules:

Module: ALU
Inputs: [15:0] A, [15:0] B, ALUK
Outputs: [15:0] dout
Description: Accepts two data inputs and does one of four operations on the two : AND, ADD, NOT, PASSA.
Purpose: This is a key component that is used in various instructions in the LC3. It does the basic operations required for a processor to work.

Module: tristate
Inputs: Clk, tristate_output_enable, [15:0] Data_write
Outputs: [15:0] Data_read
Description: This component takes an input and a signal that allows the output to be HI-Z or the input.
Purpose: This allows us to not have a 0 or 1 but rather a NOINPUT. This means that we can have inputs to the bus that don't matter for the current state.

Module: slc3
Inputs: [15:0] S, Clk, Reset, Run, Continue
Outputs: [11:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, CE, UB, LB, OE, WE, [19:0] ADDR
Description: This module incorporates everything that we did in the SLC3, this is the main component.
Purpose: It is instantiated by the toplevel to run the processor and use its functions.

Module: memory_parser
Inputs: none
Outputs: none
Description: Given to us.
Purpose: Parses memory accordingly.

Module: MEM2IO
Inputs: Clk, Reset, [19:0] ADDR, CE, UB, LB, OE, WE, [15:0] 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: Given to us. This makes memory from RAM available to us to work with.
Purpose: This allows us to work with memory and change its values. The creates a functionality between switches, buttons, hex displays and CPU. Just puts everything into the right place.

Module: lab6_toplevel
Inputs: [15:0] S, Clk, Reset, Run, Continue

Outputs: [11:0] LED, CE, UB, LB, OE, WE, [19:0] ADDR, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7
Description: Top level file of the project.
Purpose: This instantiates a SCL3 module and runs the processor and its functions.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [3:0] Out0
Description: Takes inputs to drive HEXLEDS.
Purpose: Drives the Hex Displays that displays the outputs of the thing we want. Without this the Hex Displays would display random stuff or just not work.

Module: testbench
Inputs: none
Outputs: none
Description: Testing place to debug circuit and watch how waves change.
Purpose: To debug and simulate the processor.

Module: mux_PC
Inputs: [15:0] A, [15:0] B, [15:0] C, [1:0] S
Outputs: [15:0] Out
Description: Mux that takes in 3 PC inputs from different paths to send the correct one to the bus.
Purpose: So that the correct PC value is sent into bus and there is no mismatch of what we want to do vs. what actually happens.

Module: mux_MDR
Inputs: [15:0] A, [15:0] B, S,
Outputs: [15:0] Out
Description: MUX for the MDR so we can choose what goes into the MDR.
Purpose: So that the correct value passes through and not some other input we don’t want.

Module: mux_GATES
Inputs: [15:0] A, [15:0] B, [15:0] C, [15:0] D, [3:0] S
Outputs: [15:0] Out
Description: This mux controls what goes into the BUS from one of 4 inputs.
Purpose: This makes sure that only one thing is going to bus at a time and its correct. So if we wan’t PC from the BUS, we will get that instead of some corrupted data.

Module: mux_ADDR2
Inputs: [15:0] A, [15:0] B, [15:0] C, [15:0] D, [1:0] S
Outputs: [15:0] Out

Description: Mux for ADDR2 that takes in 4 inputs that are different sign extensions.
Purpose: So that the correct value of which sign extend of IR we need is passed through instead of the wrong one.

Module: mux_ADDR1
Inputs: [15:0] A, [15:0] B, S,
Outputs: [15:0] Out
Description: This takes in two inputs, either SR1 or PC and outputs the one we need.
Purpose: So that according to the signal sent to mux, we get the correct data we need and there is no mismatch.

Module: mux_DR
Inputs: [2:0] A, S
Outputs: [2:0] Out
Description: Either takes 111 or IR[11:9] as an input and outputs correct one according to select signal.
Purpose: Same as other muxes. This allows us to use the correct value form SR2 that we need for the instruction to be carried out as necessary.

Module: mux_SR1
Inputs: [2:0] A, [2:0] B, S
Outputs: [2:0] Out
Description: MUX for SR1, takes IR[11:9] and IR[8:6] as input and outputs what we need.
Purpose: Like the mux_DR, this allows us to have the correct data picked at the correct time.

Module: mux_SR2
Inputs: [15:0] A, [15:0] B, S
Outputs: [15:0] Out
Description: Takes sign extended IR and SR2OUT as inputs and correctly outputs according to signal.
Purpose: This allows us to pick the correct data at the correct time. Though all of these muxes could be converted into 1 module and be parameterized, we chose to do this to be more modular and straight forward.

Module: NZP
Inputs: Clk, NI, ZI, PI, LD_CC
Outputs: NO, ZO, PO
Description: This does the logic of NZP from databus and outputs the correct NZP values.
Purpose: This allows us to SET_CC and correctly perform operations that require it, such as BR.

Module: reg_8
Inputs: Clk, Reset, Load, [7:0] D

Outputs: [7:0] Data_Out
Description: This is a simple 8 bit register.
Purpose: Used in different cases, but mainly to test.

Module: reg_4
Inputs: Clk, Reset, Load, [3:0] D
Outputs: [3:0] Data_Out
Description: This is a simple 4 bit register.
Purpose: Though we did not use it thoroughly in the processor, it was used for testing and allowed us to see the intermediate values we wanted.

Module: reg_16
Inputs: Clk, Reset, Load, [15:0] D
Outputs: [15:0] Data_Out
Description: This is a simple 16 bit register.
Purpose: Used in different cases, but especially to create the 8 main registers R0-R7, IR etc.

Module: eightreg_16
Inputs: Clk, Reset, Load, [15:0] D, [2:0] SR1IN, [2:0] SR2, [2:0] DRIN
Outputs: [15:0] SR1O, [15:0] SR2O
Description: This is a packed array of eight 16 bit registers
Purpose: This creates the eight registers R0-R7 into one module so they are all easily accessible in one component rather than having eight different 16 bit registers.

Module: flip_flop
Inputs: Clk, Reset, Load, D
Outputs: Data_Out
Description: Simple flip flop (1 bit register)
Purpose: Used to store value of Branch Enable.

Module: datapath
Inputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, MIO_EN, [1:0] PCMUX, [1:0] ADDR2MUX, [1:0] ALUK, [15:0] MDR_In
Outputs: BEN, [11:0] LED, [15:0] MAR, [15:0] MDR, [15:0] PC, [15:0] IR
Description: This module incorporates all of the muxes, registers, and buses into one so that the SLC3 does not have to instantiate all of those itself and can just use this one module.
Purpose: The main purpose of this in its most simple form is to output the correct data to the databus. However, there are multiple different components that go into this to make it possible. The most important part of this is that it picks which input to the gate_mux we pick. This allows us to get the correct data we want, modified and tweaked to suit our needs.

Module: ISDU

Inputs: Clk, Reset, Run, Continue, IR_5, IR_11, BEN, [3:0] Opcode
Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE, [1:0] PCMUX, [1:0] ADDR2MUX, [1:0] ALUK

Description: This is the state machine that controls the whole processor. It goes from one state to another depending on inputs and current state. It is a combination of the Moore and Mealy state machine. Based on the OPCODE, it goes to the correct states, and from there the processor knows what to execute. As a result, the data correctly flows through the data path to the BUS, and from there to memory.
Purpose: The list of Load signals, mux signals and enable signals, all go into the data path and the data is flows through correctly. Without this, nothing would be able to run, as the processor would not get signals to send to the data path, and either nothing would happen, or random values would be passed through the data bus into memory and we would get something we don't want. Along with datapath, this is another huge component of our SLC3.

The OPCODE, IR_5, IR_11 and BEN is how what state goes to what next state. What should happen in those states is written in the state description. That is where the signals below are written.

The outputs of this module are what control the whole processor.

GatePC, GateMDR, GateALU, GateMARMUX:
These are the signals to the Gate_MUX and it controls which of the inputs (PC, MDR, ALU, ADDR1+ADDR2) are actually sent to databus. These are the 4 signals that control the main outputs of the datapath module.

DRMUX, SR1MUX, SR2MUX, ADDR1MUX, [1:0] PCMUX, [1:0] ADDR2MUX, [1:0] ALUK:
These are the signals that control the internal components (mainly muxes) of the datapath module. These internal components are used in order to create the data that is sent to the main outs (PC, MDR, ALU, ADDR1+ADDR2).

Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE:
These signals mainly interact with the MEM2IO module. These control the writes and reads from memory. This tells the datapath whether at the current state we want to store or fetch to complete the current instruction properly.

LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED:
This picks where we want to store the correct data from the data bus. So if we want to Load MAR with current data, we keep all the other Loads off. This makes sure that no component is getting loaded with the data that isn't supposed to be in it.

There are 11 different main instructions which use all of these signals: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, STR, LDR, PAUSE. The details of what happens in each instructions and what states are hit can be seen in the state diagram. What happens in each state can be seen in our ISDU.sv file. It shows what signals are set in each state and what the next states are. Screenshots:

```
124
125         // Assign next state
126         unique case (State)
127            Halted :
128               if (Run)
129                  Next_state = S_18;
130            S_18 :
131               Next_state = S_33_1;
132            // Any states involving SRAM require more than one clock cycles.
133            // The exact number will be discussed in lecture.
134            S_33_1 :
135               Next_state = S_33_2;
136            S_33_2 :
137               Next_state = S_35;
138            S_35 :
139               Next_state = S_32;
140            // PauseIR1 and PauseIR2 are only for Week 1 such that TAs can see
141            // the values in IR.
142            PauseIR1 :
143               if (Continue)
144                  Next_state = PauseIR1;
145               else
146                  Next_state = PauseIR2;
147            PauseIR2 :
148               if (Continue)
149                  begin
150                     Next_state = PauseIR2;
151                  end
152               else
153                  begin
```

```
153                 begin
154                     Next_state = S_32;
155                 end
156             Pause1 :
157                 if (~Continue)
158                     Next_state = Pause1;
159                 else
160                     Next_state = Pause2;
161             Pause2 :
162                 if (Continue)
163                   begin //why do we have the begin here?
164                     Next_state = Pause2;
165                   end
166                 else
167                   begin
168                     Next_state = S_18;
169                   end
170             S_32 :
171                case (Opcode) //Different Operations/Instructions
172                     4'b0001 :
173                         Next_state = S_01; //ADD
174                     4'b0101 :
175                         Next_state = S_05; //AND
176                     4'b1001 :
177                         Next_state = S_09; //NOT
178                     4'b0000 :
179                         Next_state = S_00; //BR
180                     4'b1100 :
181                         Next_state = S_12; //JMP
182                     4'b0100 :
183                         Next_state = S_04; //JSR
```

```
180                     4'b1100 :
181                         Next_state = S_12; //JMP
182                     4'b0100 :
183                         Next_state = S_04; //JSR
184                     4'b0110 :
185                         Next_state = S_06; //LDR
186                     4'b0111 :
187                         Next_state = S_07; //STR
188                     4'b1101 :
189                         Next_state = Pause1;
190                     default :
191                         Next_state = S_18;
192                endcase
193             S_01 : Next_state = S_18;      //ADD
194             S_05 : Next_state = S_18;      //AND
195             S_09 : Next_state = S_18;      //NOT
196             S_06 : Next_state = S_25_1;    //LDR
197             S_25_1 : Next_state = S_25_2;
198             S_25_2 : Next_state = S_27;
199             S_27 : Next_state = S_18;
200             S_07 : Next_state = S_23;      //STR
201             S_23 : Next_state = S_16_1;
202             S_16_1 : Next_state = S_16_2;
203             S_16_2 : Next_state = S_18;
204             S_00 :                         //BR
205                if(BEN==1'b1)
206                   Next_state = S_22;
207                else
208                   Next_state = S_18;
209             S_22 : Next_state = S_18;
```

```
            else
                Next_state = S_18;
        S_22 : Next_state = S_18;
        S_12 : Next_state = S_18;      //JMP
        S_04 : Next_state = S_21;      //JSR
        S_21 : Next_state = S_18;
        default : ;

    endcase

    // Assign control signals based on current state
    case (State)
        Halted: ;
        S_18 :
            begin
                GatePC = 1'b1;
                LD_MAR = 1'b1;
                PCMUX = 2'b00;
                LD_PC = 1'b1;
            end
        S_33_1 :
            begin
                Mem_OE = 1'b0; //mem_we removed
            end
        S_33_2 :
            begin
                Mem_OE = 1'b0; //mem_we removed
                LD_MDR = 1'b1;
            end
        S_35 :
```

```
                LD_MDR = 1'b1;
            end
        S_35 :
            begin
                GateMDR = 1'b1;
                LD_IR = 1'b1;
            end
        PauseIR1: ;
        PauseIR2: ;
        Pause1:
            begin
                LD_LED = 1'b1;
            end
        Pause2: ;
        S_32 :
            begin
                LD_BEN = 1'b1;
            end
        S_01 :   //ADD
            begin
                SR2MUX = IR_5;
                SR1MUX = 1'b1;
                ALUK = 2'b00;
                GateALU = 1'b1;
                LD_REG = 1'b1;
                LD_CC = 1'b1;
                // incomplete...
            end
        S_05 :   //AND
            begin
```

```
261             end
262          S_05 :   //AND
263             begin
264                SR2MUX = IR_5;
265                SR1MUX = 1'b1;
266                ALUK = 2'b01;
267                GateALU = 1'b1;
268                LD_REG = 1'b1;
269                LD_CC = 1'b1;
270             end
271          S_09 :   //NOT
272             begin
273                SR2MUX = 1'b0;     //added
274                SR1MUX = 1'b1;
275                ALUK = 2'b10;
276                GateALU = 1'b1;
277                LD_REG = 1'b1;
278                LD_CC = 1'b1;
279             end
280          S_00 :
281             begin
282                Mem_WE = 1'b1; //added
283                Mem_OE = 1'b1;
284             end
285          S_22 :
286             begin
287                ADDR1MUX = 1'b0;
288                ADDR2MUX = 2'b10;
289                PCMUX = 2'b10;
290                LD_PC = 1'b1;
```

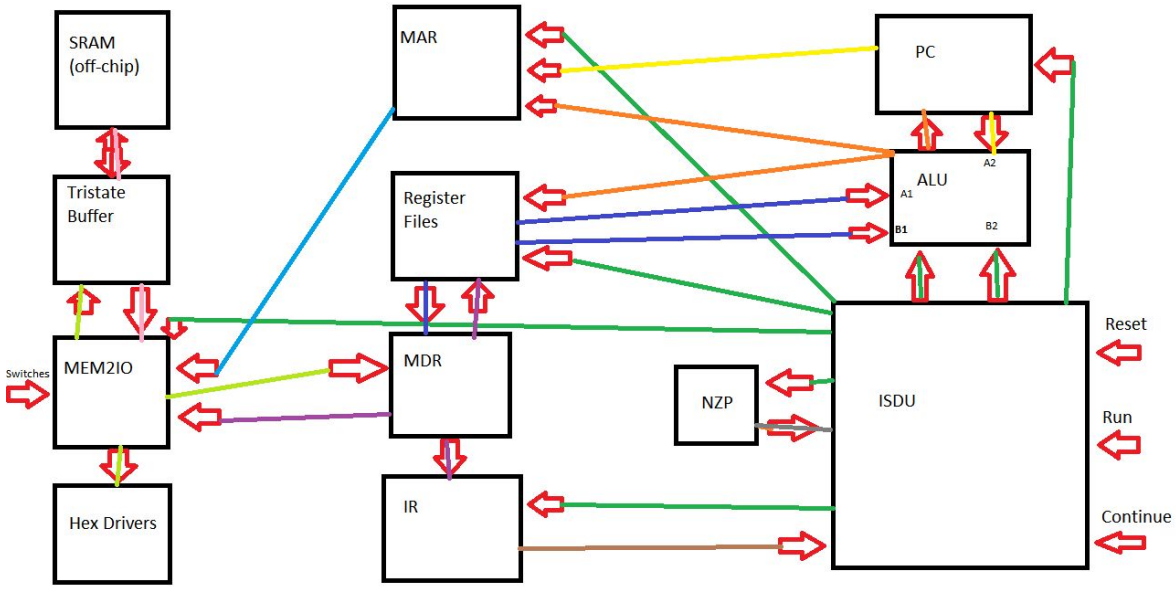
```
288                ADDR2MUX = 2'b10;
289                PCMUX = 2'b10;
290                LD_PC = 1'b1;
291             end
292          S_12 :   //JMP
293             begin
294                SR1MUX = 1'b1;
295                ADDR1MUX = 1'b1;
296                ADDR2MUX = 2'b00;
297                PCMUX = 2'b10;
298                LD_PC = 1'b1;
299             end
300          S_04 :   //JSR
301             begin
302                GatePC = 1'b1;
303                DRMUX = IR_11;
304                LD_REG = 1'b1;
305             end
306          S_21 :
307             begin
308                ADDR1MUX = 1'b0;
309                ADDR2MUX = 2'b11;
310                PCMUX = 2'b10;
311                LD_PC = 1'b1;
312             end
313          S_06 :   //LDR
314             begin
315                SR1MUX = 1'b1;
316                ADDR1MUX = 1'b1;
317                ADDR2MUX = 2'b01;
```

```
315                 SR1MUX = 1'b1;
316                 ADDR1MUX = 1'b1;
317                 ADDR2MUX = 2'b01;
318                 GateMARMUX = 1'b1;
319                 LD_MAR = 1'b1;
320              end
321           S_25_1:
322              begin
323                 Mem_WE = 1'b1;
324                 Mem_OE = 1'b0;
325              end
326           S_25_2:
327              begin
328                 Mem_OE = 1'b0;
329                 Mem_WE = 1'b1;
330                 LD_MDR = 1'b1;
331              end
332           S_27 :
333              begin
334                 GateMDR = 1'b1;
335                 LD_REG = 1'b1;
336                 LD_CC = 1'b1;
337              end
338           S_07 :      //STR
339              begin
340                 SR1MUX = 1'b1;
341                 ADDR1MUX = 1'b1;
342                 ADDR2MUX = 2'b01;
343                 GateMARMUX = 1'b1;
344                 LD_MAR = 1'b1;
```

```
333              begin
334                 GateMDR = 1'b1;
335                 LD_REG = 1'b1;
336                 LD_CC = 1'b1;
337              end
338           S_07 :      //STR
339              begin
340                 SR1MUX = 1'b1;
341                 ADDR1MUX = 1'b1;
342                 ADDR2MUX = 2'b01;
343                 GateMARMUX = 1'b1;
344                 LD_MAR = 1'b1;
345              end
346           S_23 :
347              begin
348                 SR1MUX = 1'b0;
349                 ALUK = 2'b11;
350                 GateALU = 1'b1;
351                 LD_MDR = 1'b1;
352              end
353           S_16_1 :
354              begin
355                 Mem_OE = 1'b1;
356                 Mem_WE = 1'b0;
357              end
358           S_16_2 :
359              begin
360                 Mem_OE = 1'b1;
361                 Mem_WE = 1'b0;
362              end
363           // You need to finish the rest of states
```

Block Diagram of slc3.sv:

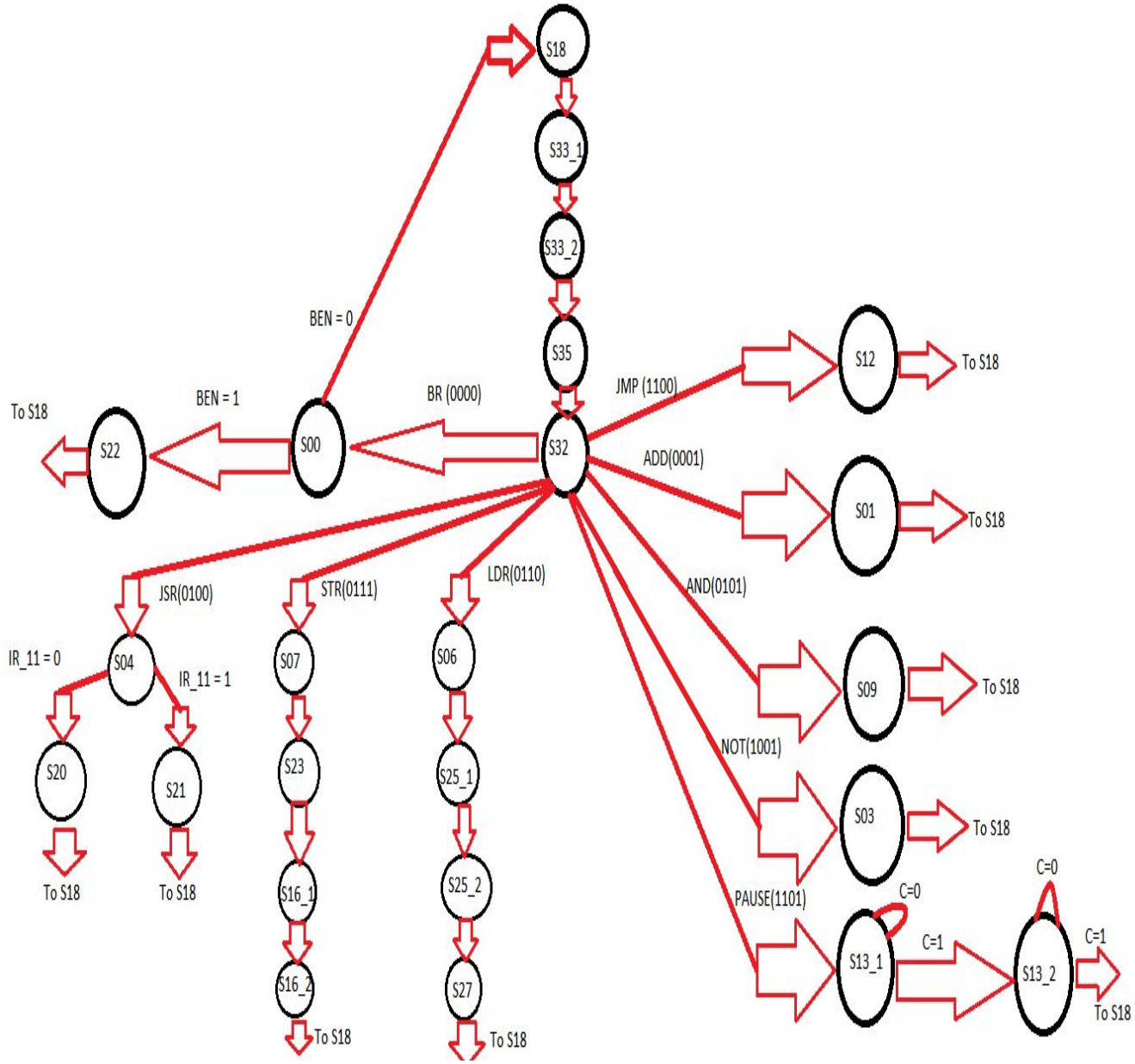
❏ This diagram should represent the placement of all your modules in the slc3.sv. Please only include the slc3.sv diagram and not the RTL view of every module.



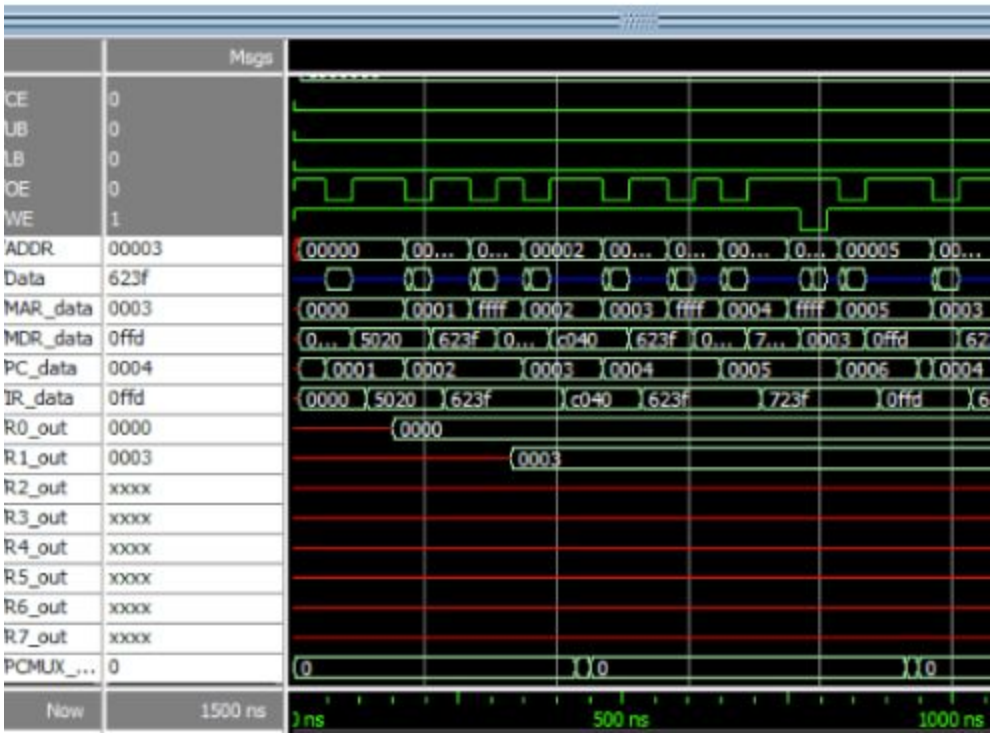
Description of the operation of the ISDU (Instruction Sequence Decoder Unit):

Combined with description/purpose

State Diagram of ISDU:



Simulations of SLC-3 Instructions:



The three instructions visible in the image above are ANDi, LDR and JMP. Further the changing of MDR scrolling through memory is also visible.

ANDi is the first instruction that executes at 150ns. SR=R0, DR=R0, IMM5 = #0. This puts 0000 into R0.

LDR is the second instruction that executes. This occurs at 275ns. DR = R1, BaseR = R0, Offset = #-1. This puts the value stored in Memory address [R0 + -1] into R1.

Finally, the third instruction that occurs is JMP at 350ns. BaseR = R1. This puts whatever is in R1 into PC. So PC becomes 0003.

Post-Lab Questions:

1. Q: Fill in the table shown in 5.6 with your design's statistics
A:

LUT	593
DSP	0
BRAM	0
Flip-Flop	280
Frequency	94.47 MHz
Static Power	98.47 mW
Dynamic Power	0
Total Power	179.29 mW

2. Q: What is the function of the MEM2IO.sv module?
A: MEM2IO essentially functions as a moderator for the connections between memory, the CPU, the hex displays, and the switches. If the write enable signal is high, MEM2IO will write the received data into the hex drivers. However, if the read enable signal is high, and MEM2IO receives the address of an I/O device, then it will read the data input from the switches directly.

3. Q: What is the difference between the BR and JMP instructions?
A: The first major difference between the BR and JMP instructions is that JMP is an unconditional jump, whereas BR will only jump if the stated condition codes are matching. Secondly, the jump for BR is specified in the offset inside of the 16 bit instruction and starts from the current PC, whereas the jump for JMP is to whatever address is inside of BaseR.

4. Q: 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?
A: The R signal lets us know if a fetch from memory has been completed properly: if so, then we proceed to the next state. If not, it simply loops. The way we did it is as follows: we knew that it would not take an infinite number of cycles to fetch, so we just set it to a finite number. To do so, we account for it by the number of states we have for instructions which requires a fetch operation to be completed from memory. We just add an extra state whenever we need to read enable.

Conclusion:

Overall, Lab 6 was far and away the most difficult, time consuming, and tedious lab we've worked on so far. The fact that it was a two week lab was sort of unassuming: week one didn't require much work, but it needed to be spent understanding the datapath and the isdu so that we could get part two working the following week. While many students created the entire datapath week one, we decided to do the bare minimum and just create the sections required for week one: namely, the ones required for the fetch operation. We received all of the points week one and went into week two not understanding how much work was necessary. The majority of the time we spent debugging concerned two issues: the ISDU and which inputs and outputs to include for each of the modules. After doing some experimenting, we eventually decided what inputs were necessary for each of the modules, and we moved on to the ISDU.

We began to go through the test programs for the lab. Thankfully, the test programs were created so that you could figure out which operations worked and which ones didn't. Our program passed the two Basic I/O tests and the XOR tests, but kept on failing the Self-Modifying code test. We scanned desperately through our ISDU file and eventually figured out that we had misplaced the Write Enable signal into states 33_1 and 33_2, and forgotten to include it in state 0 (the branch state). After correcting this, we were able to successfully perform both the Self-Modifying and Multiplication tests, but we were still having issues with the Bubble Sort test. Thankfully, we found out from another student that we were actually performing the test incorrectly, and once he did the test the correct way, our Bubble Sort actually worked.

In retrospect, this was a very challenging, but rewarding lab. We had to work so much out by ourselves and decipher the code that was given to us. The lab instructions weren't as clear as we would have liked, so we had to figure a lot out for ourselves, especially when it came to MEM2IO and the ISDU, so I think it would be helpful in upcoming semesters for the lab instructions to be a bit more thorough. If we could change something about the lab, we would have the workload split up more evenly between the two weeks; however, with there only being three phases in the fetch-decode-execute cycle, I can see why the course staff has organized it the way they have.