ECE 385

Fall 2021

Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

The focus of this lab was to create an SLC-3 processor, a subset of the LC3-ISA, is a 16-bit processor with 16-bit Program Counter, 16-bit instructions, and 16-bit registers. The processor is able to perform standard LC3 instruction and thus the various programs already located in the memory_contents sv file, as well as any new programs written into the file. The updated datapath block diagram is given at the end of the lab report.

Written Description of SLC-3

The SLC-3 CPU has Von-Neumann architecture and can perform any one of nine different instructions. The implementation of a specific instruction takes the CPU through three broad phases: FETCH, DECODE, and EXECUTE

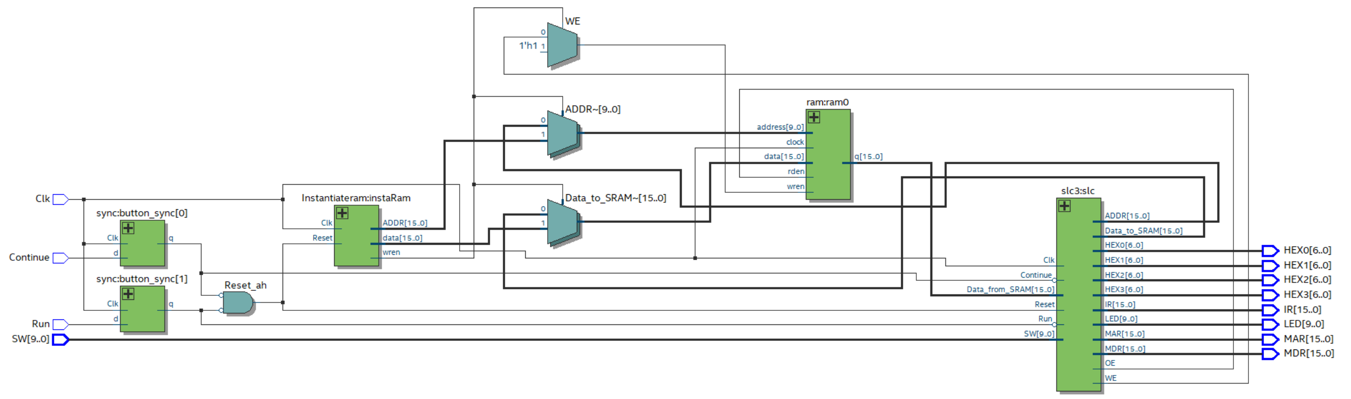
The FETCH phase has a main purpose of loading an instruction into the Instruction Register (IR). The first step of this phase is to load the Program Counter (PC) value into Memory Address Register (MAR). This is done so that the address of the next instruction can now be pointed to by MAR. Next, the memory at the location of MAR is loaded into the Memory Data Register (MDR). Finally, the contents of MDR are loaded into IR and PC is incremented by one. At this point, the instruction from memory is in the IR and the PC is ready to FETCH the next instruction after the DECODE and EXECUTE states are completed.

The DECODE stage is implemented by only one state in the ISDU state machine and its main purpose is to execute the correct instruction based on the contents of IR. The four most significant bits of the IR are used to choose the next state, which can be the first state to one of the nine different instructions. This state also sets new NZP or branch enable comparison bits by using IR[11:9]. This is relevant for the BEN instruction which is discussed below.

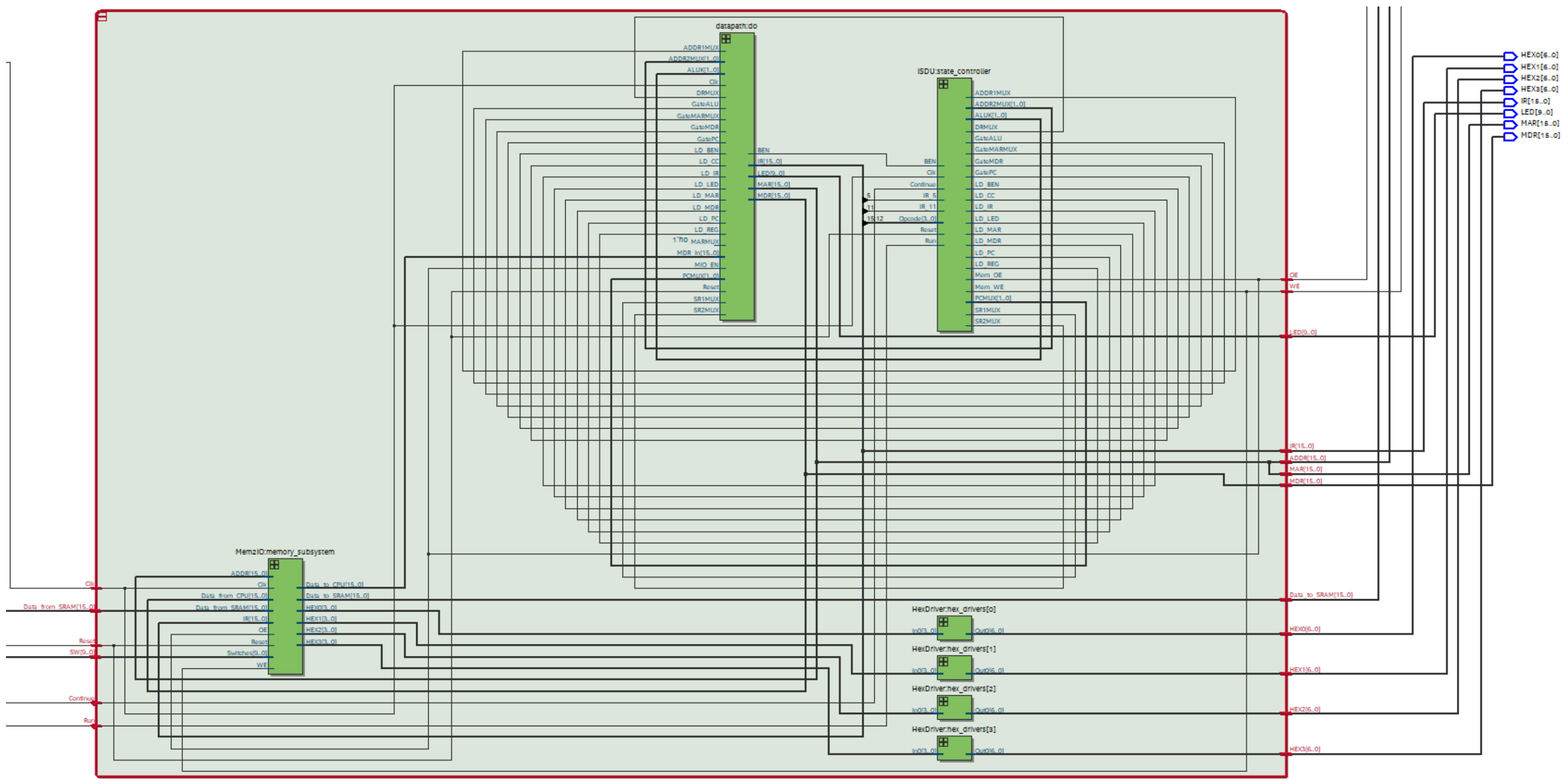
The EXECUTE phase involves the actual implantation of a given instruction. It is important to give a little more detail about the specific instructions. The ADD instruction adds two 16-bit values and stores the result in one of the 8 multi-purpose 16-bit registers. Both operands can be register contents, but it is possible for one of the operands to be the five least significant bits of the IR sign-extended up to 16 bits. The AND instruction acts very similarly to the ADD instruction, only the operation being performed is not an ADD but a bitwise AND. The NOT operation takes the contents of any of the 8 multi-purpose registers and inverts the output to be stored in any one of the aforementioned registers. The BR operation changes the PC value by either jumping it forward to skip and instruction or backwards to perform a loop. This is done by adding the sign extended 16-bit version of the nine least significant bits in IR to PC. LDR is the load instruction that reads the contents of a memory address specified by the contents of a desired register and the sign extended version of the six least significant bits of the IR. The contents of this instruction are then stored in a specified register. When the specified address is xFFFF, the data is not read from memory but instead from the on-board switches from the DE-10. STR is the same as LDR, only a specified value is now written to a chosen memory address. If the chosen address is xFFFF, the value is written to the HEX display on the DE-10. The JSR instruction is used to execute subroutines. This is done by storing the current PC value in Register 7 and then using an eleven-bit offset to take PC to the desired subroutine. JMP is used to load PC with the contents of one of the 8 multi-purpose registers. Finally, the PAUSE instruction is used to stop the CPU so that values can be manually input on switches or read on HEX display.

Block Diagram of SLC-3

The following is the top-level block diagram of our entire SLC-3 implementation. This shows how the SLC-3 CPU is connected to our synchronizers and our RAM.



The following is the block diagram for the slc3 module shown in the bottom right of the above diagram. This illustrates the connections within the CPU.



Description of all Modules

For sake of space, block diagrams are not included for all modules. However, all components are represented in larger block diagrams that are included.

Module: slc3_testtop
Inputs: [9:0] SW,
Clk, Run, Continue
Outputs: [9:0] LED
[6:0] HEX0, HEX1, HEX2, HEX3
[15:0] IR, MAR, MDR

Description: This is the top-level module used when working with the test memory in Model-Sim. Within this module, the SLC3 module is instantiated and connected to what will later be test memory through internal logic. The user inputs are all synchronized within this module.

Purpose: The purpose of the slc3_testtop module is to take user input and instantiate the CPU. The user input will be passed into the CPU to perform the desired operation.

Module: slc3_sramtop
Inputs: [9:0] SW,
Clk, Run, Continue
Outputs: [9:0] LED
[6:0] HEX0, HEX1, HEX2, HEX3
[15:0] IR, MAR, MDR

Description: This is the top-level module used when working with the on-chip memory on the DE-10. Within this module, the SLC3 module is instantiated and connected to RAM through internal logic. The user inputs are all synchronized within this module.

Purpose: The purpose of the slc3_sramtop module is to take user input and instantiate the CPU. The user input will be passed into the CPU to perform the desired operation.

Module: slc3_testmemory
Inputs: [9:0] address,
[15:0] data,
Clk, Reset, rden, wren
Outputs: [15:0] readout

Description: This module is responsible for taking the module memory_contents and parsing it into machine code. It uses the inputs address, data, rden, and wren to determine where to read or write to/from memory.

Purpose: The purpose of this module is to act as the physical on-chip memory for simulation.

Module: synchronizers
Inputs: Clk, Reset, d
Outputs: q

Description: This file contains multiple modules that all basically act as flip-flops. These flip flops take an input d and output a q value that can be equal to d or equal to a hardcoded value depending on the desired synchronizer behavior and value of Reset.

Purpose: The purpose of the synchronizers is to take a user input that is not synchronized with the Clk and synchronize that input to the clock before the data from that input is seen by the later modules in the CPU. This is important to avoid meta-stability.

Module: Slc3_2
Inputs: NONE
Outputs: NONE

Description: This file defines each instruction in terms of opINSTRUCTION(operands) rather than binary values. This is done to make reading the contents of each memory location easier to read.

Purpose: The purpose of this module is to be included in memory_contents, where opINSTRUCTION(operands) format is used rather than machine code. This is much easier to read and a great efficiency booster for the lab.

Module: slc3
Inputs: [9:0] SW,
Clk, Reset, Run, Continue,
[15:0] Data_from_SRAM,
Outputs: OE, WE,
[9:0] LED
[6:0] HEX0, HEX1, HEX2, HEX3,
[15:0] ADDR, Data_to_SRAM,
[15:0] IR, MDR, MAR

Description: This module acts as the CPU for the project. It instantiates the state machine ISDU, the Mem2IO module that interacts with the memory, and the datapath, which connects all the internal logic of the CPU. This module also directly connects the lower 10 bits of IR to the LEDs.

Purpose: The purpose of this module is to act as a top level for the CPU and instantiate all the necessary modules (except for physical memory) to implement the SLC-3 design.

Module: datapath
Inputs: Clk, Reset,
LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED,
GatePC, GateMDR, GateALU, GateMARMUX,
SR2MUX, ADDR1MUX, MARMUX,
MIO_EN, DRMUX, SR1MUX,
[1:0] PCMUX, ADDR2MUX, ALUK,
[15:0] MDR_In,
Outputs: [15:0] MAR, MDR, IR,
BEN,
[9:0] LED

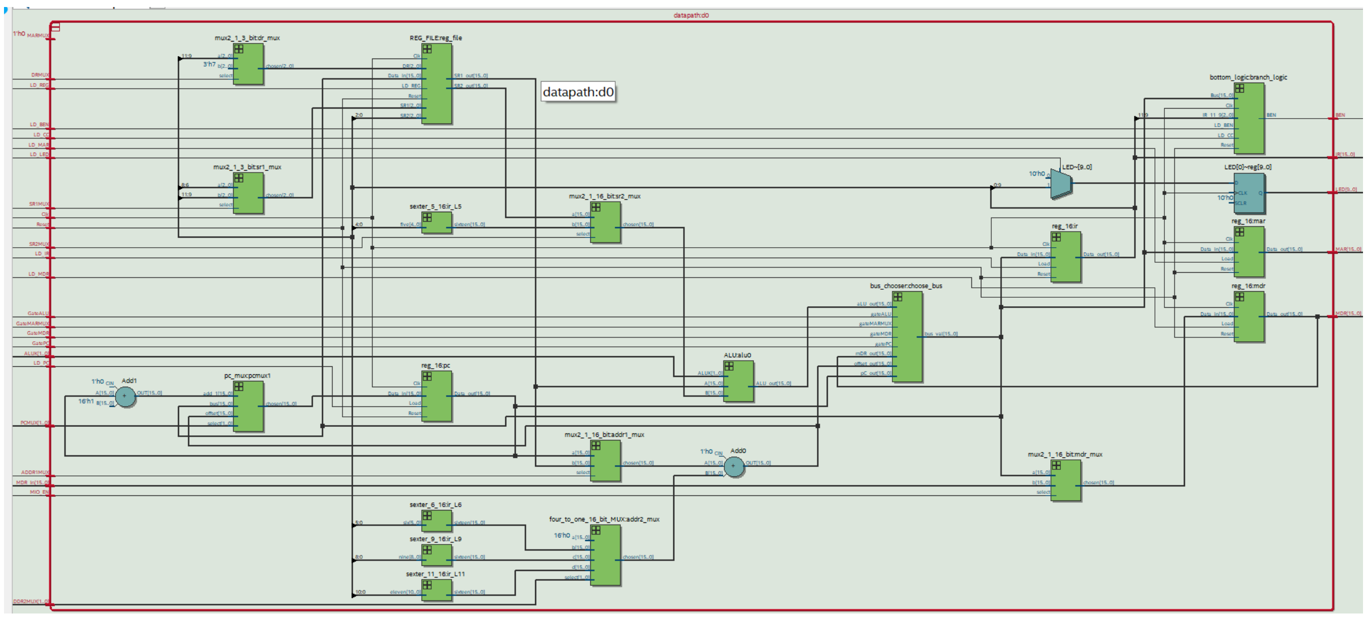
Description: This module connects all the sub-modules necessary for the CPU. Its inputs come from the ISDU control unit, and its outputs are passed back up the chain to the slc3 module.

Purpose: The purpose of this module is to instantiate the smaller modules that function within the SLC-3 design and connect these modules to the control unit signals that implement desired behavior in the correct state.

Module: datapath
Inputs: Clk, Reset,
LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED,
GatePC, GateMDR, GateALU, GateMARMUX,
SR2MUX, ADDR1MUX, MARMUX,
MIO_EN, DRMUX, SR1MUX,
[1:0] PCMUX, ADDR2MUX, ALUK,
[15:0] MDR_In,
Outputs: [15:0] MAR, MDR, IR,
BEN,
[9:0] LED

Description: This module connects all the sub-modules necessary for the CPU. Its inputs come from the ISDU control unit, and its outputs are passed back up the chain to the slc3 module.

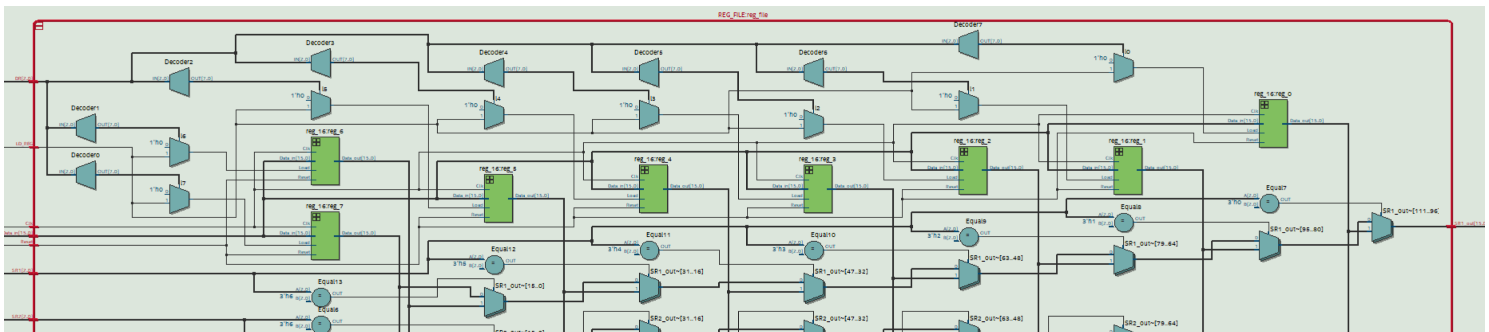
Purpose: The purpose of this module is to instantiate the smaller modules that function within the SLC-3 design and connect these modules to the control unit signals that implement desired behavior in the correct state. Block diagram for datapath is shown below.



Module: REG_FILE
Inputs: [2:0] DR, SR1, SR2,
LD_REG, Clk, Reset,
[15:0] Data_in,
Outputs: [15:0] SR1_out, SR2_out

Description: This module contains the instantiation of 8 16-bit registers. The DR, SR1, and SR2 inputs indicate the desired destination register, source register 1, and source register 2 respectively. The unit implements logic so that when LD_REG is high, only the desired DR is loaded with data and so that the output of the two source registers corresponds with the SR1 and SR2 inputs.

Purpose: The purpose of this module is to direct data to and from the correct multi-purpose registers when performing operations that require data to be loaded into registers our output from registers. The block diagram for REG_FILE is depicted below.



Module: reg_16
Inputs: Clk, Reset, Load,
    [15:0] Data_in,
Output: [15:0] Data_out

Description: The module is a basic 16-bit register that loads Data_in into Data_out if Load is high. If Reset is high, Data_out is set to x0000.

Purpose: This is the module that is instantiated for all of the registers in the SLC-3 design, including the 8 multi-purpose registers and the PC, IR, MAR, and MDR.

Module: pc_mux
Input: [15:0] bus, add_1, offset,
    [1:0] select,
Output: [15:0] chosen

Description: pc_mux is a MUX that is used to choose which value, chosen, is loaded into PC. If select is 00, chosen becomes add_1 which correlates to PC+1. If select is 01, then chosen becomes offset which correlates to a value that is calculated from the relevant offset and register contents from the instruction. If select is 10, then the chosen gets the value on the bus. Otherwise, chosen is 16'bX.

Purpose: This module controls how PC responds based on the current instruction being executed. In the FETCH phase, it will increment PC. If an instruction that requires manipulation of PC is executed, the pc_mux will make sure that PC is changed appropriately.

Module: mux2_1_16_bit
Input: [15:0] a, b,
    logic select,
Output: [15:0] chosen

Description: This is a basic 2:1 MUX that operates on 16-bit values. The output (chosen) gets a if select is 0, and b if select is 1.

Purpose: This module is used to instantiate a few different MUXes within the SLC-3 design, including the MDR MUX, the ADDR1 MUX, and the SR2 MUX.

Module: memory_contents
Input: NONE
Output: NONE

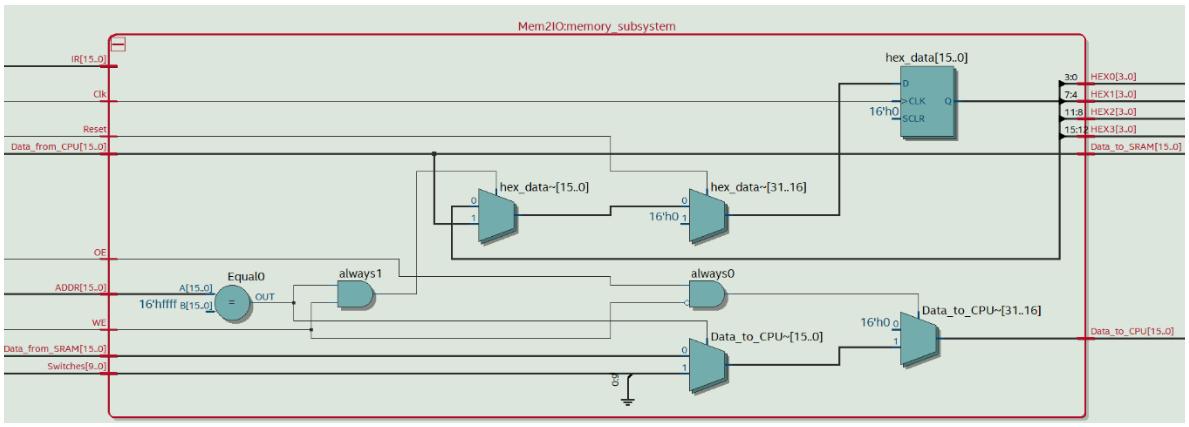
Description: This module shows the initial contents of memory using the opINSTRUCTION(operand) syntax dictated by the included SLC3_2 file.

Purpose: The memory contents are initialized so that the provided tests can be performed on the SLC-3 design.

Module: Mem_2_IO
Input: Clk, Reset,
[15:0] ADDR, IR,
OE, WE,
[9:0] Switches,
[15:0] Data_from_CPU, Data_from_SRAM
Output: [15:0] Data_to_CPU, Data_to_SRAM,
[3:0] HEX0, HEX1, HEX2, HEX3

Description: The Mem_2_IO module is instantiated by slc3 and is used to connect the memory to the CPU. This is done conditionally based on the OE and WE signals. The module also intercepts the address xFFFF to instead direct the memory write/read to the I/O switches or HEX display.

Purpose: The purpose of Mem_2_IO is to implement the bidirectional connection between memory and the CPU in a way that does not create a tri-state buffer, since a tri-state buffer cannot be implemented on the DE-10.



Module: Instantiateram
Input: Reset, Clk
Output: [15:0] ADDR, data
wren

Description: Instantiateram loads the memory_contents into the ram on the DE-10. This is done through use of a small state machine with three states called idle, mem_write, and done.

Purpose: This module is necessary to initialize the contents of the ram so that tests can be performed on the SLC-3.

Module: HexDriver
Input: [3:0] In0
Output: [6:0] Out0

Description: The HexDriver module takes a four-bit input and uses a case statement to transform it into the correct HEX digit on the HEX display.

Purpose: This module is necessary when performing any I/O so that the output can be seen on the HEX displays and the user can get feedback.

Module: bus_chooser
Input: gatePC, gateMDR, gateALU, gateMARMUX,
[15:0] aLU_out, offset_out, mDR_out, pC_out
Output: [15:0] bus_val

Description: bus_chooser implements a one-hot 4:1 MUX using a case statement on a concatenation of all the gate enable values. Only one of the gates should be open at any given time, so whichever gate is high outputs its value to the bus. If no gate is high, the bus gets a don't care.

Purpose: This module chooses what drive the bus. It is implemented so only one of PC, MDR, ALU, or MARMUX can output its value to the bus.

Module: four_to_one_16_bit_MUX
Input: [15:0] a, b, c ,d,
[1:0] select,
Output: [15:0] chosen

Description: This is a 4:1 16-bit MUX, it chooses between four inputs based on the two select bits: if select = 00, chosen=a, if select=01, chosen = b, if select = 10, select = c, if select = 11, chosen = d.

Purpose: This module is used to implement the ADDR2 MUX in the SLC-3 design.

Module: sexter_5_16, sexter_6_16, sexter_9_16, sexter_11_16
Input: [4,5,10,12:0] five, six, nine, eleven
Output: [15:0] sixteen

Description: This description includes four modules that all serve the same purpose. All sign-extend a small number of bits into a sixteen-bit value to be used into a CPU. The value is specified by the module name and in the number of bits for the input

Purpose: These modules are used to perform offset calculations based on different segments of the IR by sign extending them first. This is necessary, since SLC-3 is a 16-bit CPU.

Module: mux2_1_3_bit
Input: [2:0] a, b,
select,
Output: [2:0] chosen

Description: This module implements a 2:1 3-bit MUX. If the select bit is 0, chosen = a, else chosen = b.

Purpose: This module is instantiated to implement the SR1 MUX and the DR MUX.

Module: ALU
Input: [15:0] A, B,
[1:0] ALUK,
Output: [15:0] ALU_out

Description: This module uses the two-bit input ALUK to select an operation to perform on its 16-bit operands A and B. ALU_out gets the result of the chosen operation. If ALUK=00, the ALU adds A and B. If ALUK=01, the ALU performs a bitwise AND on A and B. If ALUK=10, then A is inverted and passed through the ALU. Finally, If ALUK = 11 then an unmodified A is passed through the ALU.

Purpose: This module is used to perform the requested operations in the AND, ADD and NOT instruction, as well as access the unmodified value of a multi-purpose register.

Module: NZP_reg
Input: [15:0] Bus,

Load, Clk, Reset,
Output: [2:0] nzp

Description: This module is used to set new n, z, and p values based on the contents of the bus, It is a 3-bit register that can store 100 if the value on the bus is negative (i.e. Bus[15]=1), 010 if the value on the bus is 0, and 001 otherwise.

Purpose: The NZP_reg stores the new nzp values to compare to the NZP values set in state 32. This is done to determine if a BR instruction should branch or not.

Module: bottom_logic
Input: Clk, Reset, LD_CC, LD_BEN,
[15:0] Bus,
[2:0] IR_11_9,
Output: BEN

Description: This module takes the output of the NZP_reg and compares it to bits [11:9] of the IR. For a BR instruction, these bits contain the nzp bits where a branch should be taken. If LD_BEN is high, the branch enable signal is updated to the bitwise and of the NZP_reg output and IR bits.

Purpose: This logic is necessary to update BEN to the correct value based on the current NZP value and determine whether to branch or not.

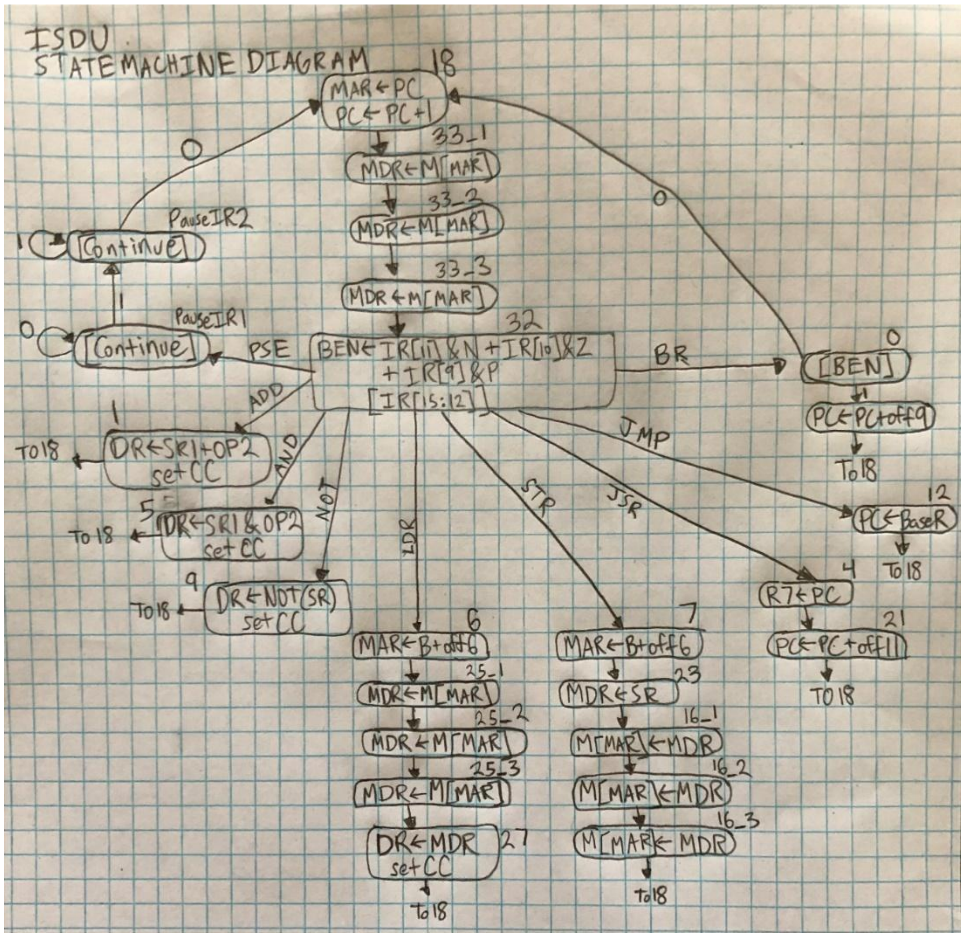
Module: ISDU (control unit)
Input: Clk, Reset, Run, Continue, IR_5, IR_11, BEN
[3:0] Opcode,
Output: LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED
GatePC, GateMDR, GateALU, GateMARMUX,
DRMUX, SR1MUX, SR2MUX, ADDR1MUX, Mem_OE, Mem_WE
[1:0] PCMUX, ADDR2MUX, ALUK

Description: The ISDU is the control center for the whole SLC-3 design. It is implemented by a state machine that outputs the necessary signals at each state. The full state diagram illustrates how the state transitions work and is illustrated below. The default value for all the output signals listed above is 0 or 00 depending on if the signal is a 1 or 2-bit signal. The state machine starts in a Halted state. In this state, all signals are default. If Run is pressed, the state machine then transitions to state S_18, the first state of the instruction cycle. It will continue to go through the states of the FETCH cycle, all the while changing its output values to reflect the desired CPU behavior at that state. Once the ISDU has reached state 32, it has now reached the DECODE phase. At this point, the state machine uses the Opcode input to decode what the instruction is and branch to the correct next state. For example, of the Opcode is 0001, the ISDU will branch to S_01 because the opcode

0001 corresponds to an ADD instruction and S_01 is the first state of the ADD instruction set. This is implemented by a case statement on the Opcode. Then, the ISDU will transition through all the states for the correct instruction (EXECUTE) and cycle back to state S_18 to start the FETCH process again.

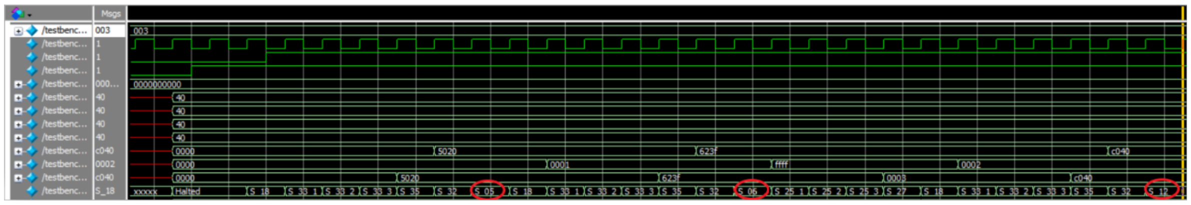
Purpose: The ISDU controls the whole CPU through the MUX-select, Load, Gate-enable, and memory read/write signals that it sends out.

Below is the state diagram of the ISDU. And important thing to notice is that the states 25, 16, and 33 have three identical states. This is because these states either read from or write to memory. This can take longer than one clock cycle to complete and there is no ready signal in the chosen implementation of SLC-3. This can be worked around by extending each state to three clock cycles, which is the implementation that was chosen.



\

Simulations of SLC-3 Instructions



```
mem_array[  0 ] =    opCLR(R0)              ;      // Clear the register so it can be used as a base
mem_array[  1 ] =    opLDR(R1, R0, inSW)    ;      // Load switches
mem_array[  2 ] =    opJMP(R1)              ;      // Jump to the start of a program
```

Every test begins with a reset and the same 3 instructions, an ‘and’ to clear, a ‘load’, and a ‘jmp’ to get to the program that correlates to the memory contents file and the value held in the switches. The end of these first three instructions will be shown by the location of the yellow bar.

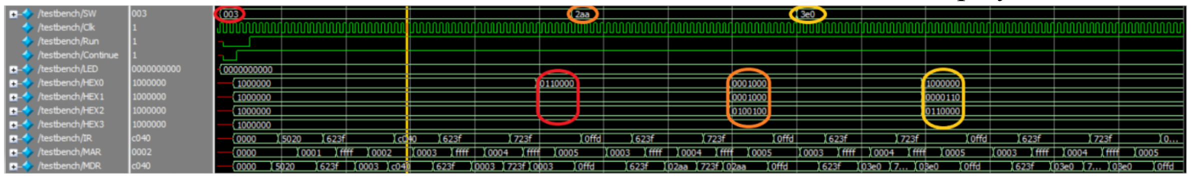
Hex Values

To interpret the values held in the hex register, each out0 correlates with the hex value commented out to the right in the picture below. So if the hex register outputs a 1000000, it represents a hex value of 0.

```
7'b1000000; // '0'
7'b1111001; // '1'
7'b0100100; // '2'
7'b0110000; // '3'
7'b0011001; // '4'
7'b0010010; // '5'
7'b0000010; // '6'
7'b1111000; // '7'
7'b0000000; // '8'
7'b0010000; // '9'
7'b0001000; // 'A'
7'b0000011; // 'B'
7'b1000110; // 'C'
7'b0100001; // 'D'
7'b0000110; // 'E'
7'b0001110; // 'F'
```

I/O Test 1

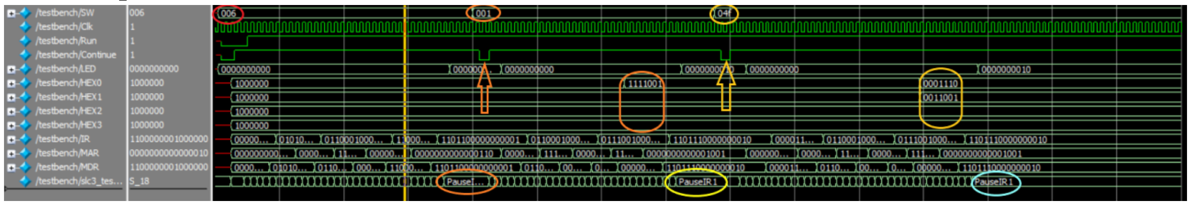
I/O Test 1 reads data from the switches and writes the values of the hex displays.



The switches are set to 0x03 in the red circle at the top left. So, after the yellow bar, I/O test 1 is running. After progressing through the instructions in I/O test 1, the HEX0 value is changed to 7’b0110000, which correlates to hex value 3, while the HEX1, HEX2, and HEX3, remain the same, 7’b1000000, which represents hex value 0. When the switches are flipped again, shown in the orange and yellow circles at the top, the hex values also change to match the values of the switches, shown by the larger orange and yellow circles in the middle of the simulation. The orange switch values are 0x2aa and HEX0 and HEX1 are 7’b0001000, HEX2 is 7’b0100100, and HEX3 remains as 7’b1000000. Together, the HEX values make 0x02aa, which is as expected. A similar comparison can be made on the yellow values.

I/O Test 2

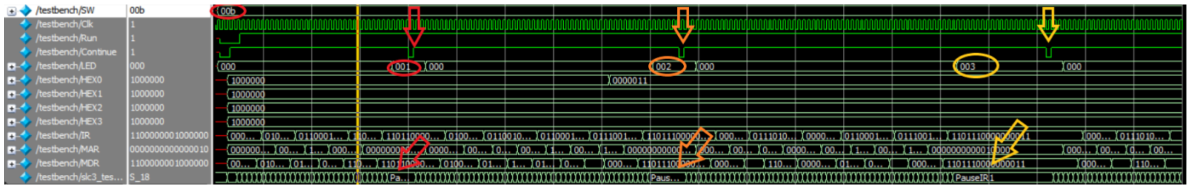
I/O test 2 is very similar to I/O test 1 because both tests display the value held in the switches onto the hex displays. The only difference is that test 2 requires that the continue button is pressed before the values are loaded.



The switches are set to 0x06 in the top left red circle, so after the standard initialization, I/O test 2 is running after the yellow bar. The program enters a pause state shown by the bottom orange circle which provides time for the user to input values into the switches and clicking the continue button. After the continue button is pressed, the program leaves the pause state and loads the values onto the HEX drivers. Because we loaded 0x01 in the top orange circle, the HEX0 driver changes to 7'b1111001 while the other HEX drivers remain as 7'b1000000 to represent the hex value 0x0001. A similar process is shown by the yellow circles. At the end of this simulation, we can see that the program enters a pause state and none of the values change because the continue button has not been pressed.

Self-Modifying Code

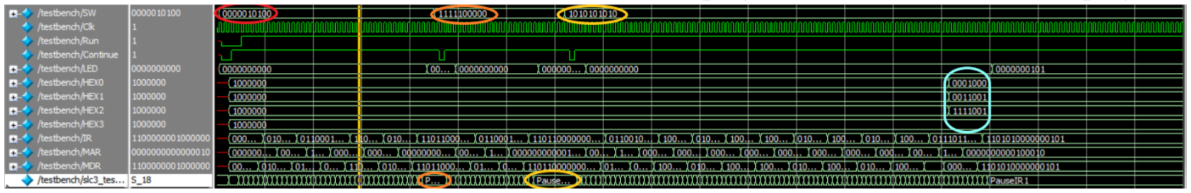
The self-modifying code builds upon I/O test 2 but adds an LED counter to signify what number load this is.



The switches are set to 0x0b in the top left red circle, so after the standard initialization, the self-modifying code program is running after the yellow bar. The program enters a pause state and since this is the first pause state, the LED holds a value of 1. After 'continue' is pressed, the LED is turned off until the next pause state, shown in orange. Since this is the second pause state and will be the second load, the LED holds the value of 2. Once 'continue' is pressed again, the LED turns off. In the third pause state, shown in yellow, we see the LED value incremented once again.

Xor

The Xor program allows users to input two values and, through simple 'And' and 'Not' instructions, calculates the XOR function and displays the result on the hex display.

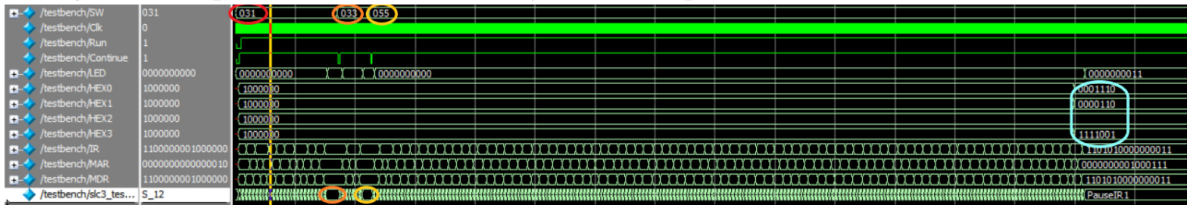


The switches are set to 0x0014 in the top left red circle, so after the standard initialization, the XOR program will be running after the yellow bar. The program has two pause states, the first one shown in orange, and the second shown in yellow. In the first pause state, the binary value 10'b1111100000 is loaded and in the second pause state, the binary value

10’b1010101010 is loaded. Once the two values are loaded, after the instructions, the result is shown in the hex display, shown in the blue circle, which represents 0x014a or 10’b0101001010, which is the XOR value of the switch's values loaded earlier. Finally, the program reaches another pause state in which case, if the continue button is pressed, the program will loop allowing the user to input two new values to be XORd together.

Multiplier

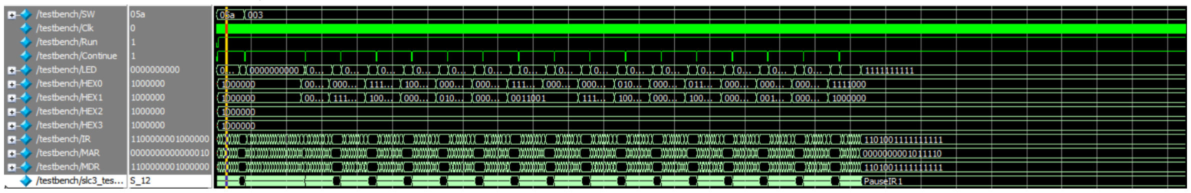
Similar to the XOR program, the multiplier program allows users to input two values and performs multiplication using a shift-and-add algorithm. It should be noted that the algorithm implemented in this lab only works for positive numbers.



The switches are set to 0x031 in the top left red circle, so after the standard initialization, the multiplication program will be running after the yellow bar. The program has two pause states, the first one shown in orange, and the second one shown in yellow. In the first pause state, the hex value 0x33 is loaded and in the second pause state, the hex value 0x55, is loaded. Once the two values are loaded, after the instructions, the result is shown in the hex display, shown in the blue circle, which represents 10EF, which is the product of the switch’s values loaded earlier. Finally the program reaches another pause state in which case, if the button is pressed, the program will loop allowing the users to input two new values to be multiplied together.

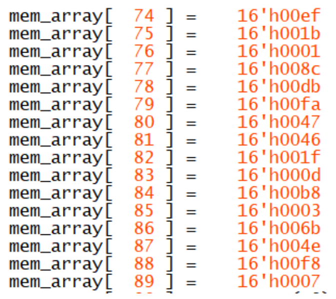
Bubble Sort

The sort had three functions. The first function writes 16 values into memory. The second function sorts the 16 values into ascending order. The third function displays what’s stored in the memory



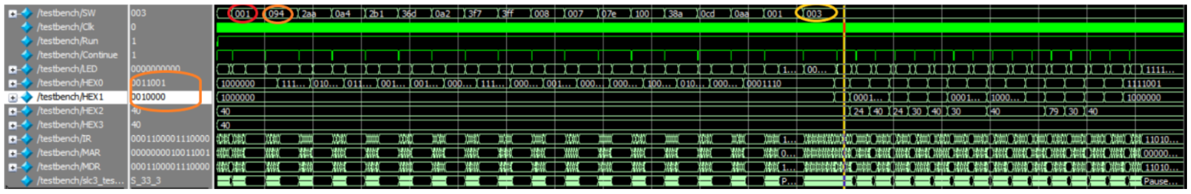
The switches are set to 0x5A in the top left, so after the standard initialization, the sort function will be running after the yellow bar. In this specific wave, the third function is on display, every time ‘continue’ is pressed, another value is displayed on the hex drivers. The values shown are the same values stored in these sixteen specific memory locations

.

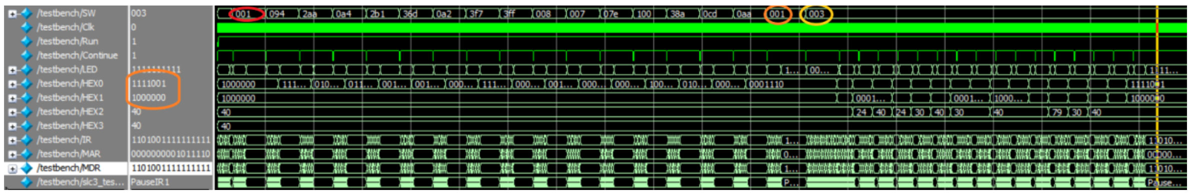


```
mem_array[  74 ] =    16'h00ef
mem_array[  75 ] =    16'h001b
mem_array[  76 ] =    16'h0001
mem_array[  77 ] =    16'h008c
mem_array[  78 ] =    16'h00db
mem_array[  79 ] =    16'h00fa
mem_array[  80 ] =    16'h0047
mem_array[  81 ] =    16'h0046
mem_array[  82 ] =    16'h001f
mem_array[  83 ] =    16'h000d
mem_array[  84 ] =    16'h00b8
mem_array[  85 ] =    16'h0003
mem_array[  86 ] =    16'h006b
mem_array[  87 ] =    16'h004e
mem_array[  88 ] =    16'h00f8
mem_array[  89 ] =    16'h0007
```

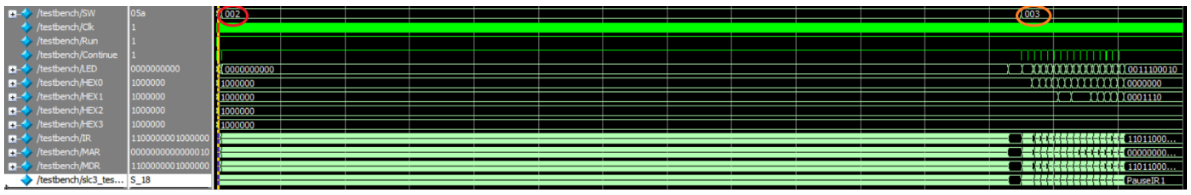
Figure: memory



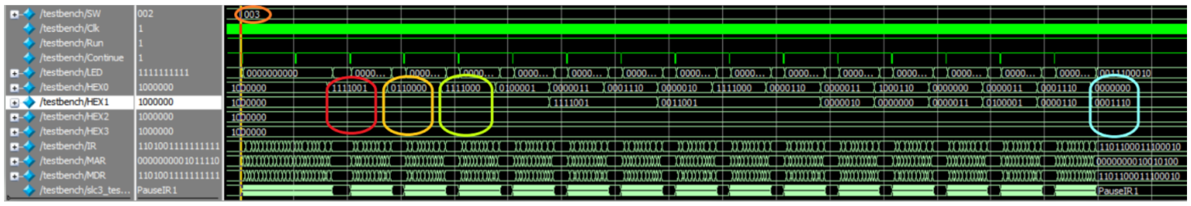
Here we see first the insert function signified by the red circle, and then after the sixteen inserts, the display function to show what is stored in the dedicated memory locations. It should be noted here that the yellow bar does not represent the end of the standard initialization and instead, is placed at the first displayed element to see what is stored in mem_array[74] more clearly. The orange circle at the top of the screen points out the first insert in the simulation, 0x94. The first display shows the corresponding hex driver binary values to represent 0x94, as seen by the orange circle to the left.



In this simulation, the orange circle at the top of the screen points out the sixteenth insert in the simulation, 0x01. Again, if the hex driver values are converted to hex, the sixteenth value inserted is the sixteenth element displayed. This similar analysis can be done on all sixteen values stored as the original values will have been replaced. It should be noted that the list has not yet been sorted.



To sort the element, the second function must be called via the switches. Because this program takes a much longer time compared to the other programs, it is hard to see the individual states so a close up simulation waveform will be provided below. However, looking at the whole program, it can be seen that function two is called in the red circle. Because we didn't insert new values in this simulation, the original values in Figure: memory will be sorted. After the values are sorted function three is called, circled in orange, which displays the newly sorted values.



```
/testbench/SW        002
/testbench/Clk       1
/testbench/Run       1
/testbench/Continue  1
/testbench/LED       1111111111
/testbench/HEX0      1000000
/testbench/HEX1      1000000
/testbench/HEX2      1000000
/testbench/HEX3      1000000
/testbench/IR        1101001111111111
/testbench/MAR       0000000001011110
/testbench/MDR       1101001111111111
/testbench/slc3_tes... PauseIR1
```

A close up of the values displayed is shown above. This simulation begins where the original orange circle on the previous waveform was located, the beginning of the display function. The red circle shows the new first element which, if converted to hex, is 0x01. The yellow and green represent the second and third values respectively, which, if also converted to hex, is 0x03 and 0x07. The final value, circled in blue, is 0xfa if converted to hex. These values, as well as the remaining values in between, can be compared to the table below to see that they are indeed sorted.

Index	Before Sort	After Sort
0	x"00ef"	x"0001"
1	x"001b"	x"0003"
2	x"0001"	x"0007"
3	x"008c"	x"000d"
4	x"00db"	x"001b"
5	x"00fa"	x"001f"
6	x"0047"	x"0046"
7	x"0046"	x"0047"
8	x"001f"	x"004e"
9	x"000d"	x"006b"
A	x"00b8"	x"008c"
B	x"0003"	x"00b8"
C	x"006b"	x"00db"
D	x"004e"	x"00ef"
E	x"00f8"	x"00f8"
F	x"0007"	x"00fa"

Design Resources and Statistics

LUT	861
DSP	0
Memory (BRAM)	18,432
Flip-Flop	268
Frequency	66.96 Mhz
Static-Power	90.21 mW
Dynamic Power	8.22 mW
Total Power	157.32 mW

Post-Lab Questions

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

The Mem2IO is used to interface with memory. As elaborated on in the module description, the main function is to connect the FPGA to the on-chip RAM, it implements a tri-state buffer to allow for the same data to either read-from or written into. Another important function of the Mem2IO is intercept the xFFFF address and interpret that as either an input from the switches or an output to the HEX display depending on whether the instruction is an LDR or STR.

What is the difference between BR and JMP instructions?

The main difference between JMP and BR is that JMP unconditionally alters the value of PC whereas BR only alters PC if the NZP conditions are satisfied. However, it is possible to get an unconditional branch from the BR instruction. Then, the main difference becomes the "distance" that PC is able to jump from each instruction. In BR, only 9 offset bits can be added to PC (after being sign extended of course). The JMP instruction loads PC with the contents of one of the multi-purpose registers, which is sixteen bits. Because of this, for large jumps JMP should be used rather than PC so that the span of the large jump can be achieved.

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?

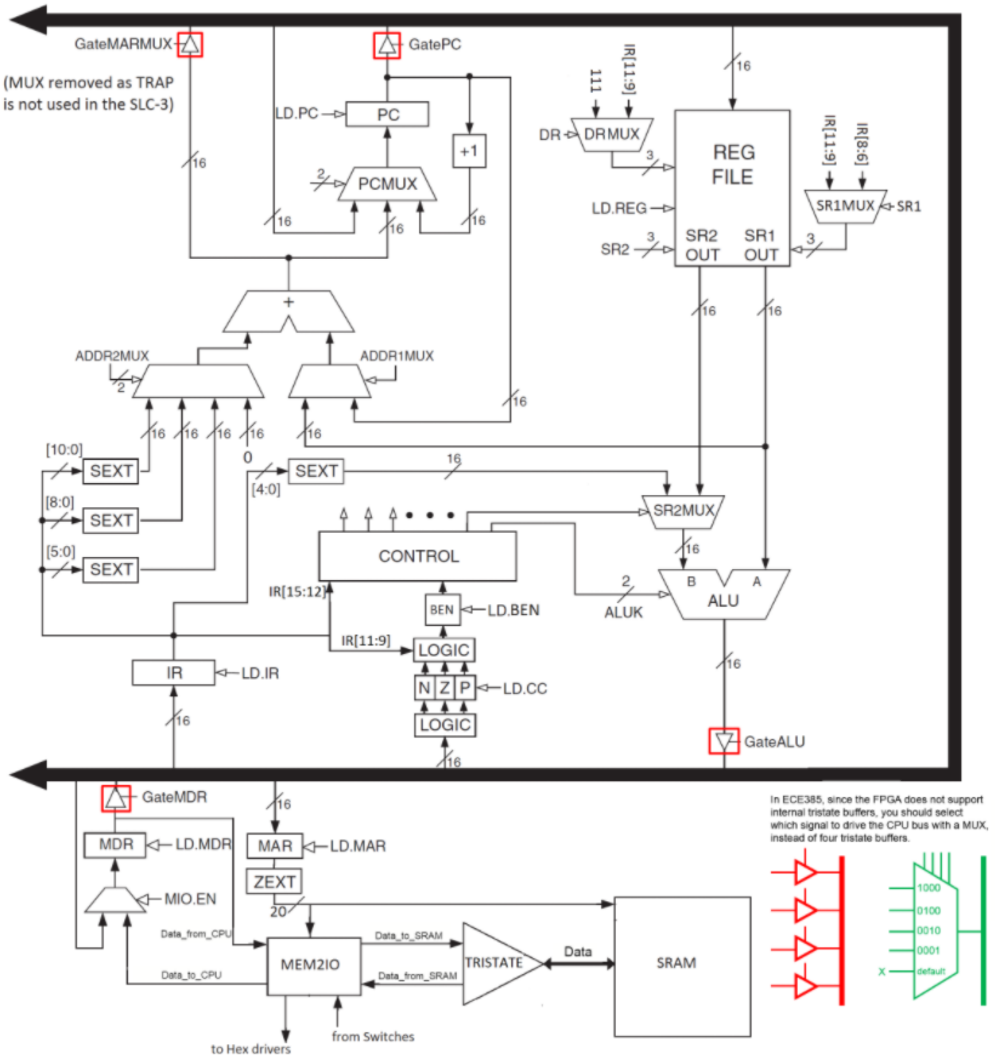
The R signal is used to signify when either a read or write has been completed in memory. It's the way for the on-chip memory to communicate to the CPU that it has finished its appropriate memory operation and the controller can proceed to the next state. We compensate for this in our design by adding multiple states for every state that interacts with memory. This is to ensure that the memory interaction is complete before we change states. If we were to use the R signal, it would have to be synchronized like all the other inputs to our ISDU. Since this is not the case, we did not have to worry about synchronizing the signal.

Conclusion

By the end of this lab, we had created a fully functioning SLC-3 CPU. All the tests gave the correct behavior, indicating correct operation of the whole design. The problems that had to be debugged when testing the SLC-3 design were minor (flipping a 1 to a 0 in System Verilog code) and only required a little digging in Model-Sim to find the error.

There was nothing ambiguous about this lab. It was very clearly laid out in the lab documentation and provided files. One thing that could have been improved on might be explicitly stating that OE is meant for reading for memory and WE is meant for writing to it. This is something we were able to guess easily by tracing signals but might be useful to define more explicitly. Otherwise, all of the lab documentation was extremely helpful.

Updated Datapath for ECE 385


GateMARMUX
(MUX removed as TRAP is not used in the SLC-3)
GatePC
LD.PC
PC
+1
PCMUX
111
IR[11:9]
DR
DRMUX
REG FILE
LD.REG
SR2
SR2 OUT
SR1 OUT
IR[11:9]
IR[8:6]
SR1MUX
SR1
ADDR2MUX
ADDR1MUX
[10:0]
[8:0]
[5:0]
SEXT
[4:0]
CONTROL
SR2MUX
ALUK
ALU
IR[15:12]
BEN
LD.BEN
IR[11:9]
LOGIC
N
Z
P
LD.CC
IR
LD.IR
GateALU
GateMDR
MDR
LD.MDR
MIO.EN
MAR
LD.MAR
ZEXT
Data_from_CPU
Data_to_CPU
MEM2IO
Data_to_SRAM
Data_from_SRAM
TRISTATE
Data
SRAM
to Hex drivers
from Switches
In ECE385, since the FPGA does not support internal tristate buffers, you should select which signal to drive the CPU bus with a MUX, instead of four tristate buffers.
1000
0100
0010
0001
default