ECE 385

Spring 2016
Experiment #6

Simple Computer SLC-3.2 in Systemverilog

Rishi Thakkar and Roshan Rajan
ABO – Thursday at 3 P.M.
Zhangqi Xu and Benjamin Delay

1. Introduction

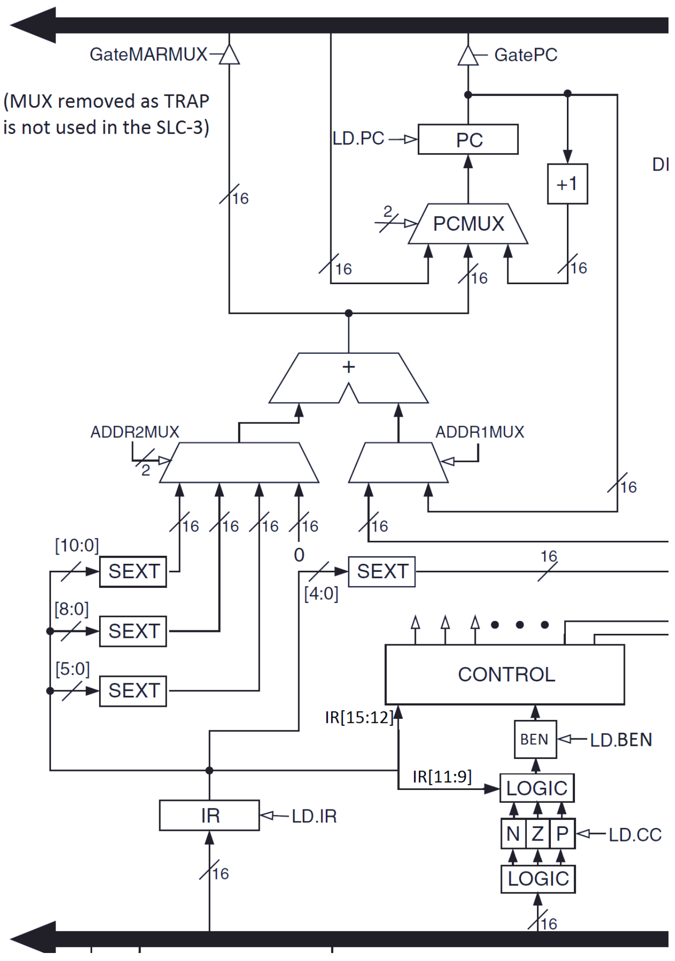
All computers contain processor that executes a given set of instructions specified by their ISAs. The microarchitectures of these computers are what enable them to implement these given instructions. The goal of this lab was to introduce us to the principles of designing a simple microprocessor using SystemVerilog. Through the use of the Von Neumann model, we had to design the microarchitecture of a simplified version of the LC3, a 16-Bit computer, to implement a subset of the LC3's instruction set.

2. Circuit Description

In this lab, we implemented a subset of the LC3 microprocessor called the SLC-3.2. Our design consisted of multiple modularly designed components and followed computer organization guidelines set forth by the Von Neumann model. The Von Neumann model consists of 5 modular units that are designed and implemented separately. These units are the control unit, processing unit, memory unit, input interface, and output interface. Our design implemented each of these through the use of several submodules which are described in the sections below. Since the Von Neumann model specifies a stored-program structure, the flow of data in our circuit was centered around the memory. During an instruction cycle, the CPU processed data that it had retrieved from memory in phases. It first went to memory and extracted the instruction during the FETCH phase. This used the PC, MAR, MDR, IR, and Memory modules. After the instruction had been placed in the instruction register, the control unit shifted to the DECODE phase. Here it checked which instruction was being executed and based on this it transitioned to the correct state. Once the instruction had been decoded, the control unit then shifted to the EXECUTE phase. In this state, it produced the control signals required to execute the given instruction. Through the use of the ALU, temporary memory, and other modules, the SLC3 performed the specified processes. We decided to use a global bus to efficiently direct data during each of these phases. The bus behaved like a central hub connecting and routing all data to its designated destinations. All data that needed to be communicated between the control unit, the memory unit, etc., was transmitted using the bus. For example, during State 18, a state that is a part of the FETCH phase, the value of the PC was placed onto the bus where it was then a acquired by MAR . During state 35, which is also part of the FETCH phase, the instruction in MDR was loaded into IR using the bus. This multipurpose feature of the bus made it possible to optimize the flow of data in our design. Through the use of the control unit, memory unit, processing unit, I/O peripherals, and the bus, we were able to implement a logically complete SLC3 microprocessor.

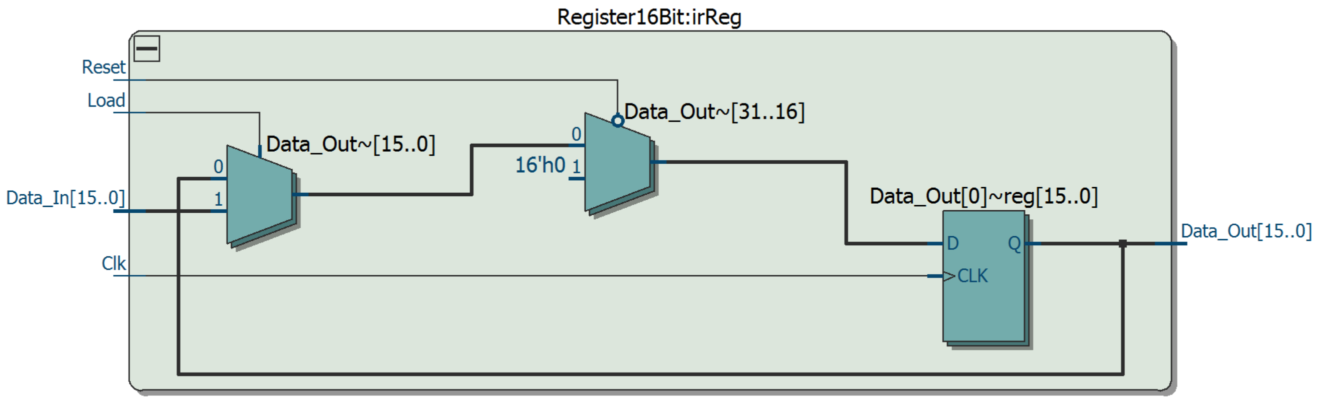
3. Control Unit

The control unit is essential for the functioning of the SLC3 microprocessor. Based on the instruction, it provides a set of control signals to the corresponding modules to perform a particular activity. It starts off with the Fetch Phase, where it stores the current address of PC into MAR, increments PC, then reads the data at that address into MDR, and finally stores that Instruction to be decoded into IR. After this Fetch Phase is complete, the control unit starts the Decode Phase, where the Instruction Sequencer/Decoder takes in the values of IR and decides which state to transition to next. Lastly, the Execute Phase is initiated. In this phase, the operation commences and the SLC3 performs the specified function. These three phases make up the core characteristics of the the control unit. The modules that make up the Control Unit are IR, ADDR Adder and Muxes, PC Mux and registers, MARMUX, the BEN and NZP, and the ISDU. The ISDU is the central module that sends all of the control signals to the rest of the control unit, processing unit, and memory unit and it also determines the current state and next state of the SLC3 machine. The different modules and components are described in more detail below. The image below shows the entirety of the control unit. The GateMARMUX and GatePC are not part of our design. The bus unit shown in section 4 is used to mimic this functionality.



3.1 IR

The IR is simply a register that holds the current instruction for a given instruction cycle. When the LD.IR value is HIGH, during the Fetch Phase, the value in IR is updated with the value currently on the bus. This is the value from MDR, which holds the instruction for that given cycle. From there it gets sent to the ADDR2Mux, the SR2 Mux and most importantly the ISDU where depending on the type of instruction that is read, a multitude of different control signals are outputted to ensure for a proper result from the SLC3. The IR is implemented through the use of the 16-bit register module. The inputs into this module are the Data_In, Load, Reset, and Clk. The outputs from this module is the 16-bit value currently stored in the register.

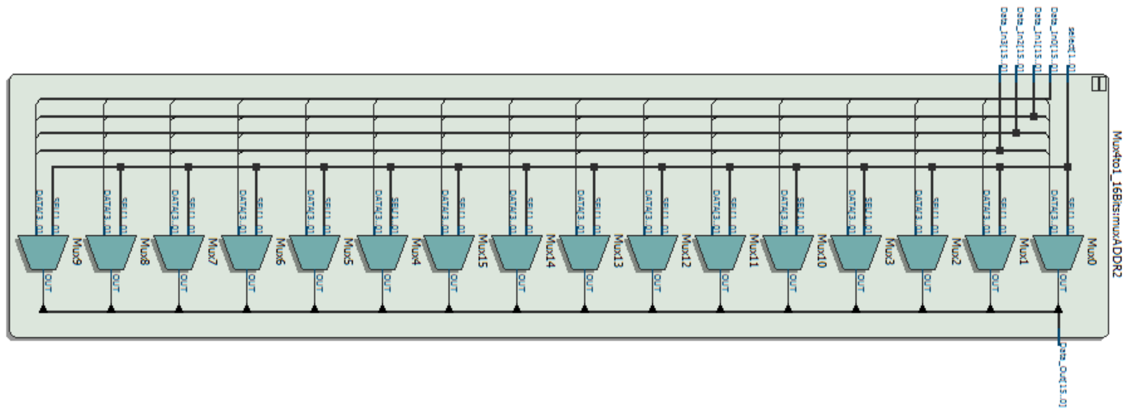


3.2 ADDR1Mux

The ADDR1MUX is used to generate the input into the ADDRAdder. The inputs are SR1 and PC. Then depending on the ADDR1MUX select bit sent from the ISDU, which depends on the current state, one of the inputs will get selected to go into the adder to then be sent out to the PC MUX and MARMUX. SR1 input will allow for any BR + offset with instructions such as LDR and the PC input is primarily used in instructions such as LD for any PC + offset component. These characteristics of the module are shown below in the section regarding ADDRAdder. The 2 to 1 16-Bit mux module is used to implement the ADDR1Mux. The schematic block diagram of this circuit is shown in section 5.3.

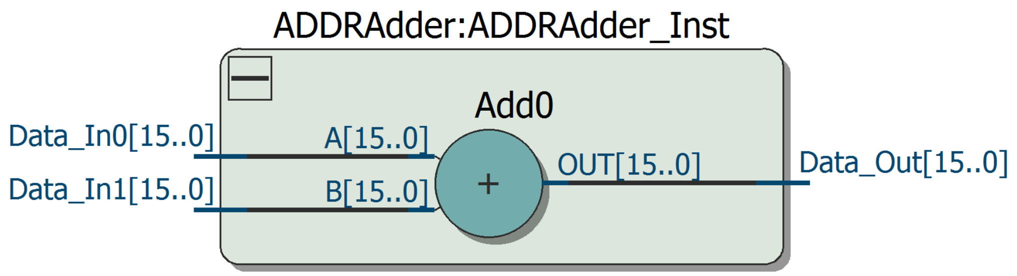
3.3 ADDR2Mux

The ADDR2Mux is another one of the inputs that goes directly into the ADDRAdder. The inputs into this Mux are zero extended or sign extended variations of the value of the instruction register. The major purpose of these bits is to allow for the addition of such sign extensions when doing PC + offset or BR + offset in instructions such as LDR or BR can be performed as per the users specifications. Once the select bit has been chosen and sent from the ISDU, then this outputs to the adder which is then sent out to the PC Mux and the MARMUX. The 4 to 1 16-Bit mux was used to implement the ADDR2Mux. The inputs into tis module were the 4 Data_In and the select bits. The output from this module was the Data_Out based on the select bits. The schematic block diagram of this module is shown in the image below.



3.4 ADDRAdder

The primary purpose of this adder is to add the outputs from the ADDR1MUX and the ADDR2MUX. As seen below, it simply adds these two inputs and continuously output this Address adder value to the Marmux and the PC Mux where it is delegated to either go directly onto the bus or change the PC value respectively depending on the instruction that was read and the current state of the machine. The image below shows the simplicity of this module.



3.5 MARMUX

While there is normally a MARMUX in the LC3, we did not have to incorporate one here. This is because we did not use any sort of Trap Instruction in the SLC3. Our entire MARMUX unit for the SLC3 is simply the GateMARMUX where the input values of this gate is the output from the ADDR adder. Therefore, whenever the GateMARMUX is initiated and goes onto the bus, this value will then be read into the MAR for the Memory Unit to decide what to do depending on the other components of the instruction. Due to how we designed our bus, we did not have a specific GateMARMUX module or another sort of gate module. Instead, we had all of the inputs go to the bus encoder so that the proper inputs including PC and other modules could be chosen during each state. This is explained in more detail in section 4.

3.6 PC

PC is simply a register that holds the current Program Counter value. This value is used to define the memory location at which the next instruction to be executed exists. The LD.PC value is used to change the value of the PC register. This value is defined by the ISDU and when it is HIGH, the value loaded into PC is the value being outputted by the PCMUX. The value that is placed into PC is dependent on the instruction and state of the machine. The module used to implement the PC was the 16-Bit Register module. Since this is the same module that was used to implement the IR, please look at section 3.1 for the schematic block diagram.

3.7 PC + 1

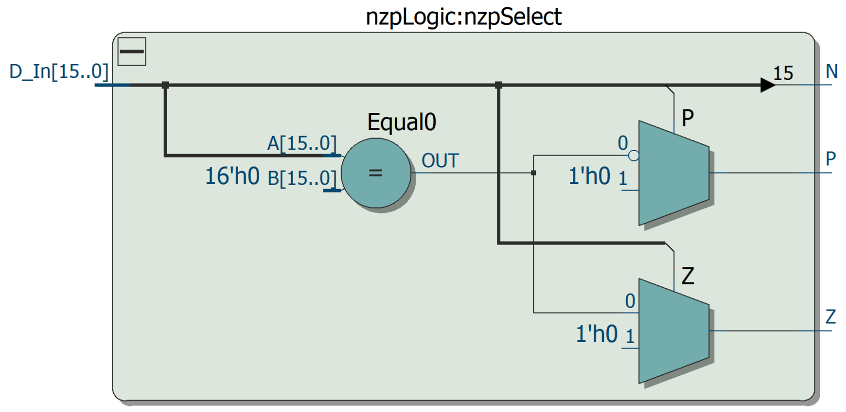
The PC + 1 module is used to increment the PC value. This is done so that the next instruction can be read in sequential order. This module simply adds 1 to the PC register whenever LD.PC is initiated and the PCMUX is set to output the value of PC+1. The module used to perform this operation is the adder module. The schematic block diagram of this module is shown in section 3.4.

3.8 PCMUX

The PCMUX is used to determine the next value of the PC. The two inputs into the PC Mux are the PC Plus 1 and the value from the ADDRAdder. The PC + 1 allows for the microprocessor to read the next sequential instruction in a program and the ADDR adder allows for instructions such as JMP, BR, and STR. Since this the PCMUX is very similar to the ADDR1Mux, we decided to use a 2 to 1 16-bit mux to implement it. The schematic block diagram for this module and more specifications are provided in section 5.3

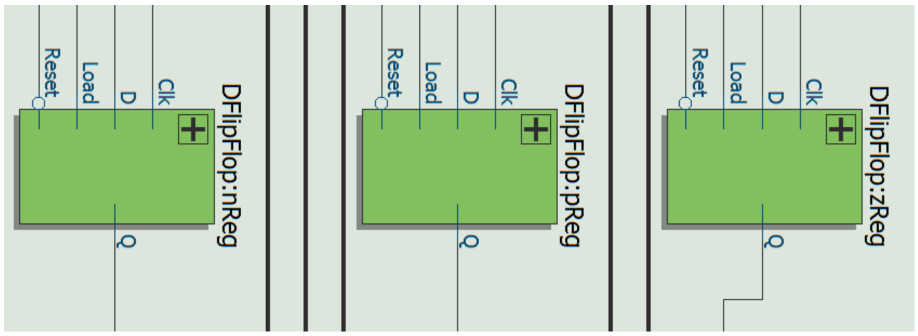
3.9 NZP Logic

The NZP logic module was used to determine the next NZP values. To do this, it took in data from the bus constantly. By simply doing a few logical checks, we were able to determine the NZP values. To see if the value is negative, we checked if the MSB of the value of the bus was 1. To determine if the value is zero, we used a comparator to see if all 16-bits are equal to 0. Lastly, to see if a value was positive, we checked whether the 16th bit is 0 and the rest of the bits are partially non-zero. Through the use of these tests, the NZP logic outputted these NZP values to the NZP module. The schematic block of this circuit is provided below.



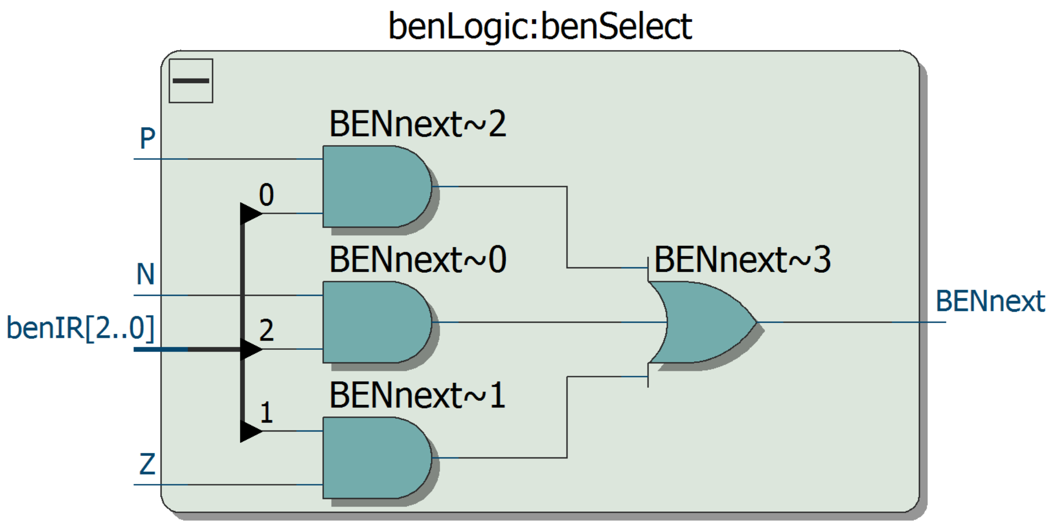
3.10 NZP

NZP values are very important for the purposes of branching. They are defined as the condition codes for the values on the bus. We decided to use 3 D Flip-Flops to hold the NZP values produced by the NZP logic modules. The inputs into each of the D Flip-Flop are the next values generated by the NZP logic and load.CC. When load.CC is HIGH, the next NZP values get loaded into the Flip-Flops. Otherwise, the Flip-Flops retain their current value. More details on how the D Flip-Flop works are provided in section 3.13. The block diagram of the 3 D Flip-Flops is provided below.



3.11 BEN Logic

BEN Logic is used to set the bit that determines if we need to branch at a given time. This module is purely combinational and takes in the current NZP values and IR[11:9], nzp, as inputs. Then it uses the combinational logic within to produce a single bit output called Next BEN. Next BEN is determined by implementing the flowing SOP: Next BEN = IR[11]&N + IR[10]&Z + IR[9]&P. What this does is it sets Next BEN as 1, if and only if there is a 1 in NZP that correspond with a 1 in IR[11:9]. By doing this we ask the question: was the result of the last operation negative, zero, or positive? Through the use of this architecture, the SLC3 allows the user to perform conditional jumps. Once the next BEN value is produced, it is then sent to a synchronizing D Flip-Flop called BEN. The image below shows the circuit described in its entirety.

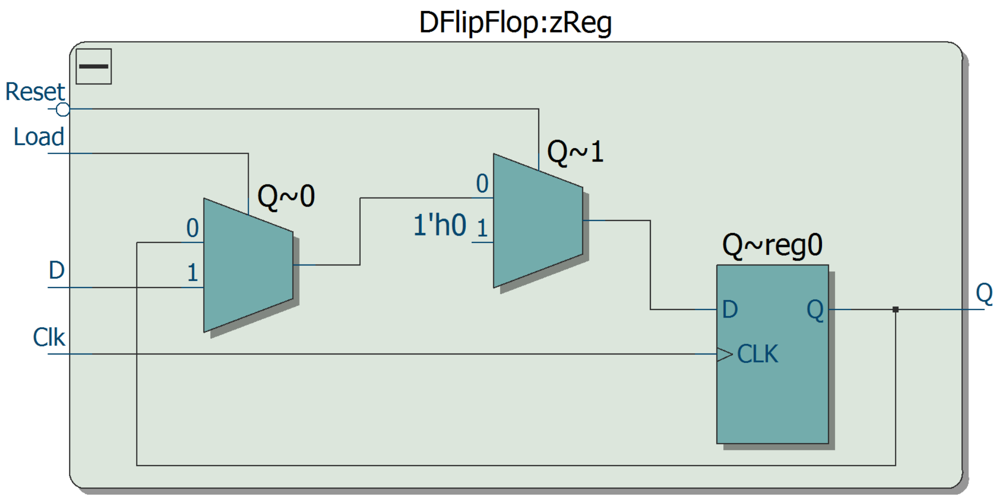


3.12 BEN

BEN is simply a 1-bit register that holds the current Branch Enable value. This value is used to determine whether or not to branch at a given time. We load this value from BEN logic at state 32 and if the opcode for a given instruction cycle is 0000, then we check if the BEN is 1. If it is 1, then the ISDU generates the signals such that the SLC3 branches to PC + PCOffset9. Otherwise, the machine does nothing and the next instruction cycle begins. To implement BEN, we decided to use a D Flip-Flip. The module implementation is described in more detail and an image of the circuit is provided in section 3.13.

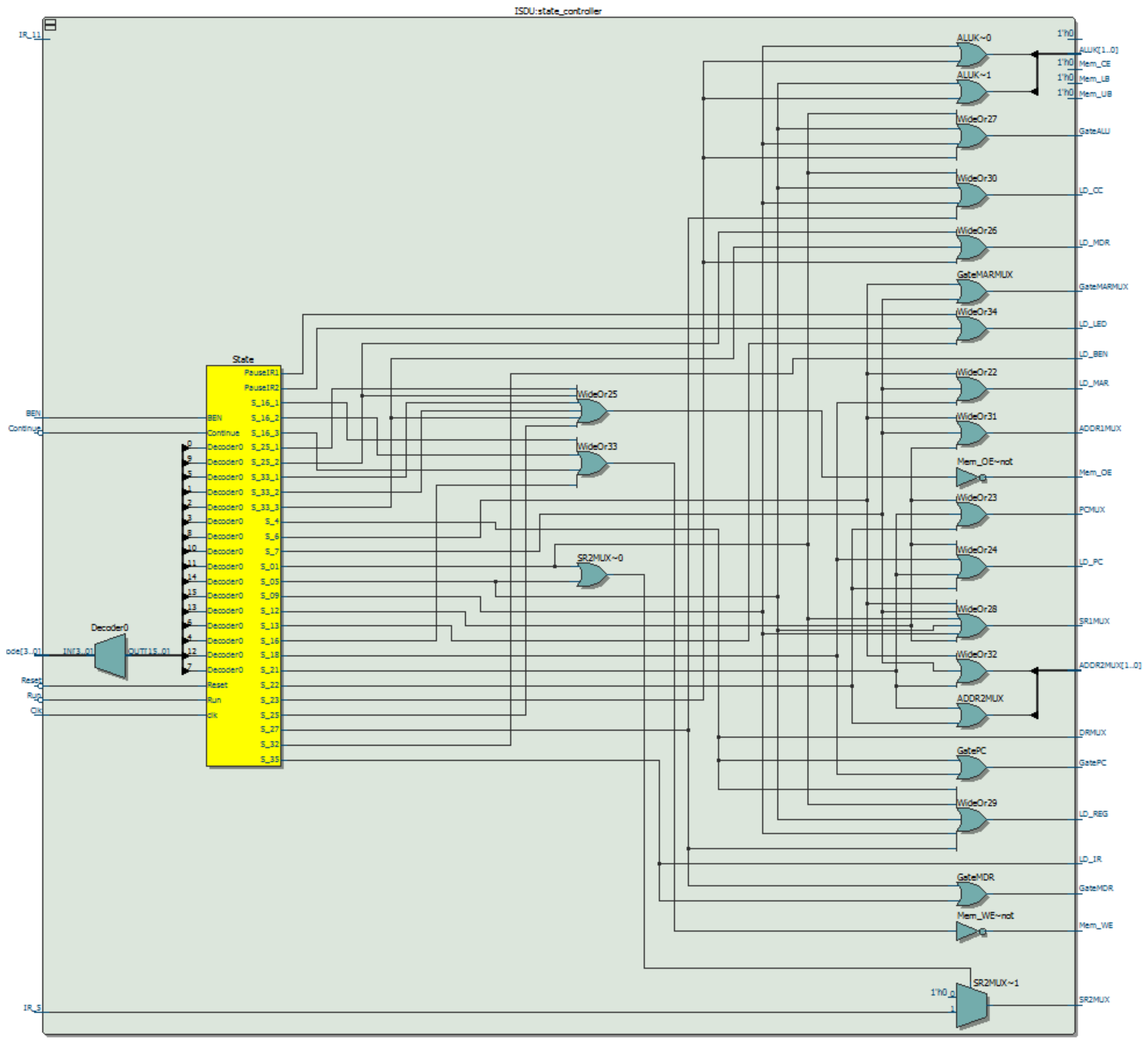
3.13 D Flip-Flop

The single bit D Flip-Flop module was simply a 1-bit register unit that was used to store the given input values. The inputs to this module were Reset, Load, D, and Clk. If Reset is high, then we clear the current value of the registers. If Load is high then, we load the input value into the register. Otherwise, we just preserve the old value of X in the register. The output from this module is Q which is the buffered input value. The schematic block diagram of this module is shown in the image below.



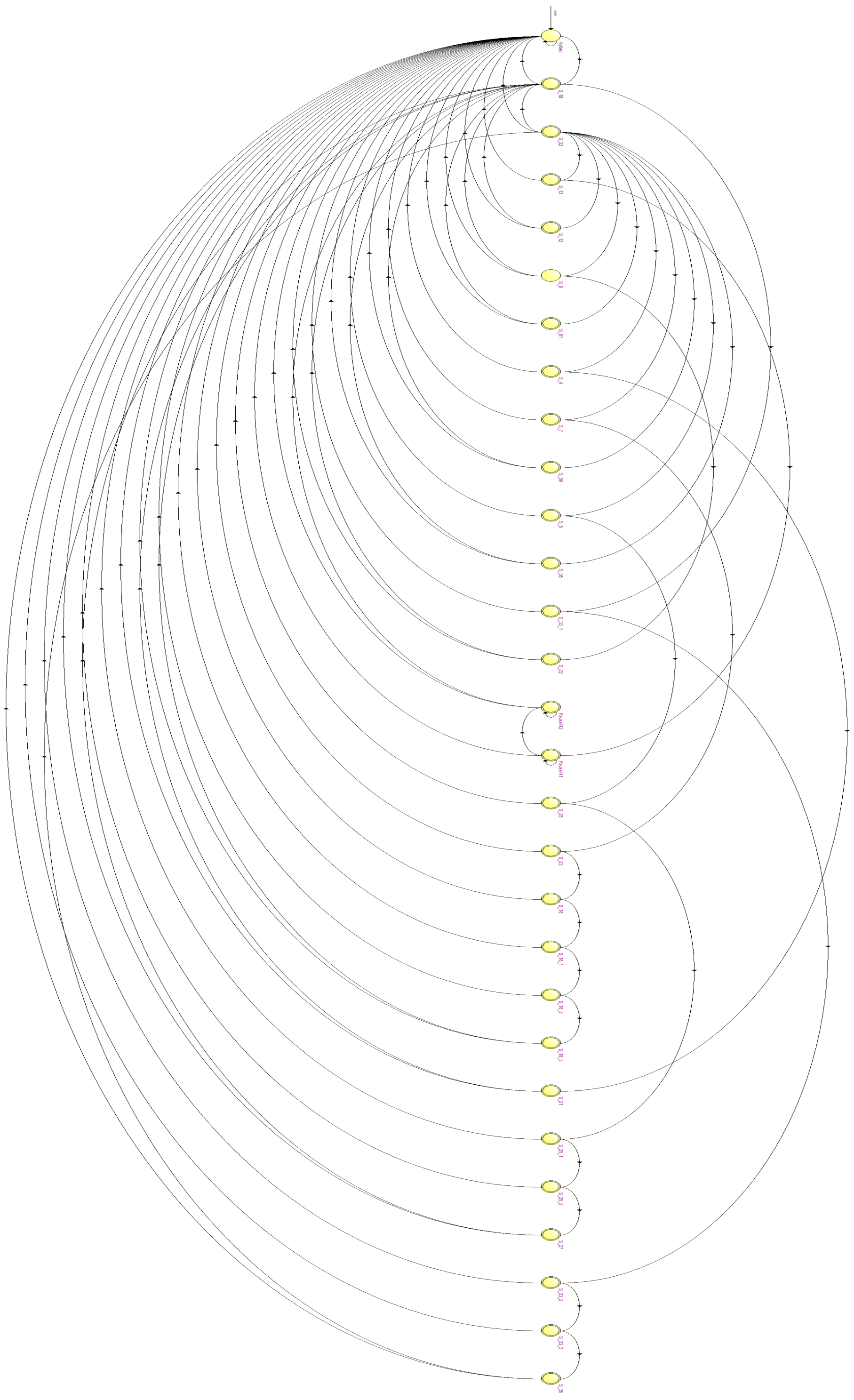
3.14 ISDU

The ISDU module is the core of the entire SLC3 microprocessor. Inputs into this module are the current instruction and Ben. Based on these inputs and the current state, the ISDU assigns the outputs that make the SLC3 perform the specific instruction. The next state diagram shown in the next section shows how the flow of transitions between states is based on the phase that the control unit is operating in. Starting at state 18, the machine is able to fetch, decode, and execute a given instruction. The three tables shown after the next state diagram explain in more detail the different phases and what occurs in each state of each phase as well as the corresponding instruction in the execute phase. The image below shows the schematic block diagram of the ISDU.



3.15 Next State Diagram

The image below shows the next state diagram of the control unit. For reference purposes, the state at the very top of the image is the HALTED state which is followed by our first operating state, state 18.



3.16 Fetch Phase

The Fetch Phase is the part of the instruction cycle where the next instruction to be executed is fetched from memory. All instruction cycles start with this phase. The states in this phase are State 18, 33, 35. More information about the specifics of each state are provided in the table below.

State	Control Signals	Description
18	GatePC = 1'b1; LD_MAR = 1'b1; PCMUX = 1'b0; LD_PC = 1'b1;	This is the start state of the SLC3. What it accomplishes is that it loads the address of the program counter to MAR and then increments the PC. The purpose of this is to get the address of the instruction to be read to be executed by the LC3 machine later.
33	Mem_OE = 1'b0; LD_MDR = 1'b1;	In this state, the data that is stored at the address in MAR is stored into MDR. We used repetitions of this state until the memory is loaded to ensure that the true data at the address is in MDR.
35	GateMDR = 1'b1; LD_IR = 1'b1;	This state sets the data in MDR that was read from MAR to be stored into IR which represents the next instruction for the LC3 microprocessor.

3.17 Decode Phase

The Decode Phase is the part of the instruction cycle where the instruction to be executed is identified. This phase consists solely of state 32. More information about this state is provided in the table below.

State	Control Signals	Description
32	LD_BEN = 1'b1;	In this state, BEN is loaded and is compared to the last piece of data on the bus in the previous instruction cycle.

3.18 Execute Phase

The Execute Phase of an instruction cycle performs the function specified by the given instruction. This phases consist of numerous states which are all presented and explained in detail in the table below.

Instruction	State	Control Signals	Description
ADD and ADDi	1	GateALU = 1'b1;	This state incorporates ADD & ADDi where it depends on the 5th bit which is then outputted into the ALU. Then it adds that output with the

		SR2MUX = IR_5; SR1MUX = 1'b1; ALUK = 2'b00; DRMUX = 1'b0; LD_REG = 1'b1; LD_CC = 1'b1;	SR1 value. AS GateALU is on, the output goes onto the bus and then back into the register file where it is stored into the DR specified in the instruction. Condition codes are also set and return to state 18.
AND	5	GateALU = 1'b1; SR2MUX = IR_5; SR1MUX = 1'b1; ALUK = 2'b01; DRMUX = 1'b0; LD_REG = 1'b1; LD_CC = 1'b1;	This state incorporates AND & ANDi where it depends on the 5th bit which is then outputted into the ALU. Then it ands that output with the SR1 value. AS GateALU is on, the output goes onto the bus and then back into the register file where it is stored into the DR specified in the instruction. Condition codes are also set and return to state 18.
NOT	9	GateALU = 1'b1; SR1MUX = 1'b1; ALUK = 2'b10; DRMUX = 1'b0; LD_REG = 1'b1; LD_CC = 1'b1;	This state simply NOTs the value in SR1. It nots every single bit in SR1 and then, as GateALU is on, the output goes onto the bus. Afterwards the output goes into the register file where it is stored into the DR specified in the instruction. Condition codes are also set and return to state 18.
BR	0	BEN	This state just checks if BEN is high which means that the data and the BR value has the same condition codes. If it does then it goes to State 22 else it goes to State 18.
	22	ADDR1MUX = 1'b0; ADDR2MUX = 2'b10; PCMUX = 1'b1; LD_PC = 1'b1;	The state takes in the PC value from ADDR1Mux and offset 9 from the ADDR2Mux. It adds these two values together and then the PC Mux chooses this adder output to load into the PC and return to state 18.
JMP	12	SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b00; PCMUX = 1'b1; LD_PC = 1'b1;	This state allows the BaseR data value that goes into ADDR1Mux and adds it with 16 bits of 0 in ADDR2Mux. The adder output goes to PCMux and then the PCMux chooses this output to load into the PC and then returns to state 18.

JSR	4	GatePC = 1'b1; DRMUX = 1'b1; LD_REG = 1'b1;	AS GatePC is HIGH. the PC value gets loaded into the bus and the register file takes in the value of the PC and stores that into Register 7 as specified with DRMux.
	21	ADDR1MUX = 1'b0; ADDR2MUX = 2'b11; PCMUX = 1'b1; LD_PC = 1'b1;	Since we do not accept instruction JSRR, our only option is to do Pc ← PC + off11. Thus in this state, ADDR1Mux takes in the PC value and ADDR2Mux takes in offset 11. It adds these two values together and then the PC Mux chooses this adder output to load into the PC and returns to state 18.
LDR	6	GateMARMUX = 1'b1; SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b01; LD_MAR = 1'b1;	This state passes the BaseR input from SR1 out to ADDR1Mux and the offset 6 to ADDR2Mux. After the adder adds the ADDR Mux then pushes this value through GateMARMUX to the bus and MAR loads in this value.
	25	Mem_OE = 1'b0; LD_MDR = 1'b1;	In this state, the data that is stored at the address in MAR is stored into MDR. We used repetitions of this state until the memory is loaded to ensure that the true data at the address is in MDR.
	27	GateMDR = 1'b1; DRMUX = 1'b0; LD_REG = 1'b1; LD_CC = 1'b1;	In this state, GateMDR is on and therefore the data that was stored at the address in MAR gets stored into the DR. When the register file gets this data, it loads into the appropriate DR and then the Condition Codes are set. Finally return to state 18.
STR	7	GateMARMUX = 1'b1; SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b01; LD_MAR = 1'b1;	When this state is initiated, we grab the Base Register Value from SR1 out through ADDR1Mux and then the offset 6 through the ADDR2Mux. Then these inputs are added with the ADDR adder and outputted through the GateMARMUX and finally get set onto the bus where MAR then loads this value.
	23	GateALU = 1'b1; SR1MUX = 1'b0; ALUK = 2'b11; LD_MDR = 1'b1;	During this state, we simply pass through the ALU the data that is in the source register. Then as GateALU is High, this data gets passed onto the bus and gets loaded into MDR.

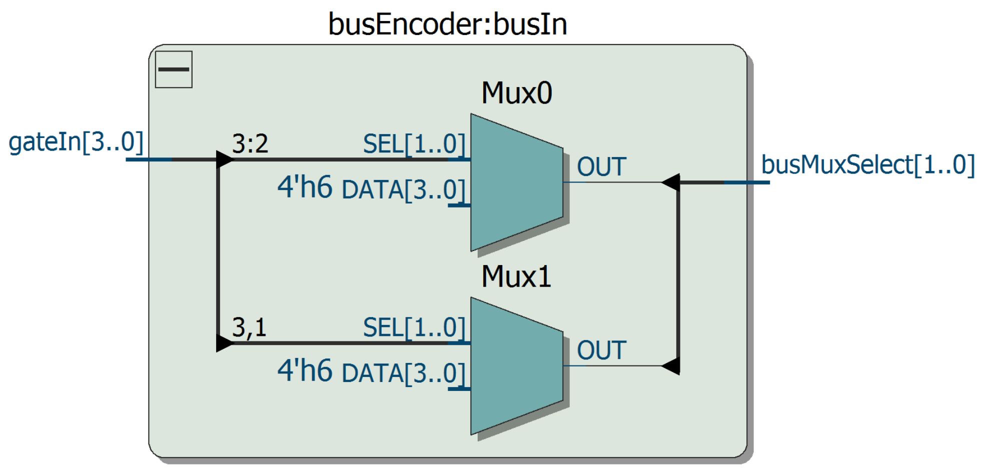
	16	Mem_WE = 1'b0;	In this state, the data that is stored in MDR is now stored into the memory of the address in MAR. We used repetitions of this state until the memory is loaded to ensure that the true data at the address is the data from MDR. Finally return to state 18.
PAUSE	13	LD_LED = 1'b1 EXECUTE KEY	In this state we simply loading the LED vector that is stored to represent whether the previous operation was a write to hex display or if the next operation is a read from the switches. This initiates the response from the SLC3 so that the user can tell where the program is during the execution. It immediately goes to PauseIR1 state.
	PauseIR1	LD_LED = 1'b1 CONTINUE KEY	This state is right after the PAUSE instruction(state 13) and simply stays in this state as long as CONTINUE KEY is still being held down. Once it is not being pressed, it moves to PauseIR2. During this state we are still loading the same LED vector.
	PauseIR2	LD_LED = 1'b1 Continue KEY	After PauseIR1 state is done, this state simply stays in this state until Continue is pressed again where it then restarts and goes to the next instruction. During this state we are still loading the same LED vector and afterwards return to state 18.

4. Bus

The bus is extremely important to the movement of data through the entire SLC3 computer. It allows the data to be sent from different locations to certain areas depending on the current state of the SLC3 machine. We see that the 4 inputs for the bus are the ALU, MDR, PC, and MARMUX as well as the Gate Enable bits for each input and the output is the value that is supposed to be on the bus. In the actual representation of the SLC3, the bus is gated using tri-state buffers. However, since tri-states are not synthesizable inside of the FPGA, we had to take a different approach. We decided to use a mux which would replicate the behavior of the tri-state buffers by only allowing one of the inputs onto the bus. The16-bit signal placed onto the bus is decided by the ISDU based on the state of the machine. In combination with the mux we used a bus encoder module to generate the select bit for the mux. More details on the encored and the mux are provided in the sections below.

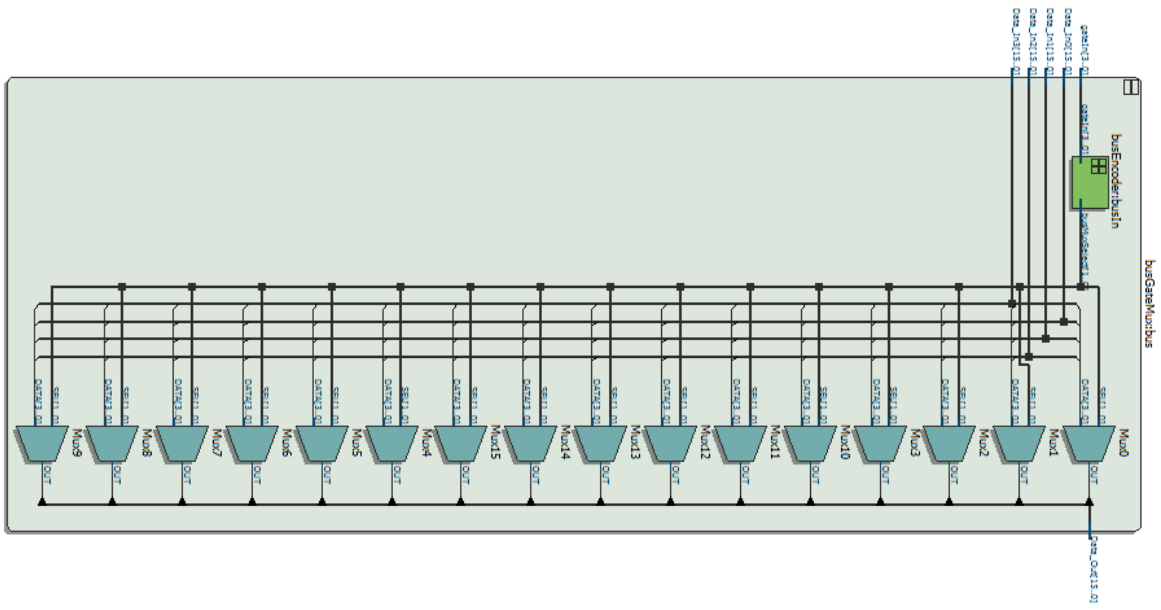
4.1 Bus Encoder Module

The bus encoder is our modified method of choosing the select bit for the value to be outputted onto the bus. It uses a simple encoder where the the 4 inputs of GatePC, GateALU, GateMARMUX, and GateMDR go into the encoder and the output result was a 2 bit select where the possibilities of the select bits were used to represent each of the possible outputs onto the bus. For example, 00 represents the PC. The select bits were then sent to the bus mux where the selection of the outputs were made. The image below shows the schematic circuit diagram of the bus encoder module.



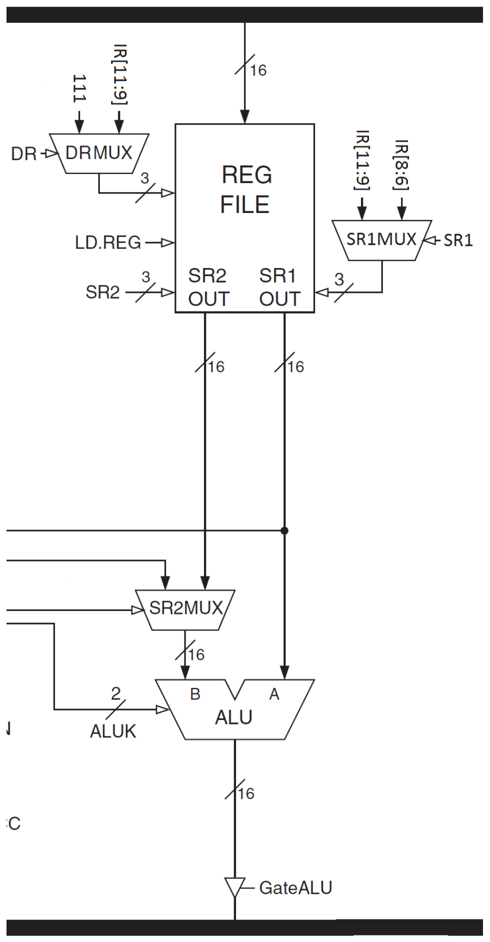
4.2 Bus Gate Mux Module

After the submodule called bus encoder has outputted the select bits, the purpose of the bus mux was to select each of the 16 bits to be outputted onto the bus. This was done using muxes. As seen in the image below, each of the bits from the data in values and the select bits from the encoder go to each mux to select the correct data out value to go to the bus. Within the modules, the inputs are Data_In0 (PC), Data_In1 (MDR), Data_In2 (MARMUX), Data_In3 (ALU) and these data inputs were chosen to be outputted onto the bus depending on the requirements of the current instruction and state of the SLC3. The image below shows the entire circuit block diagram of the Bus Gate Mux module.



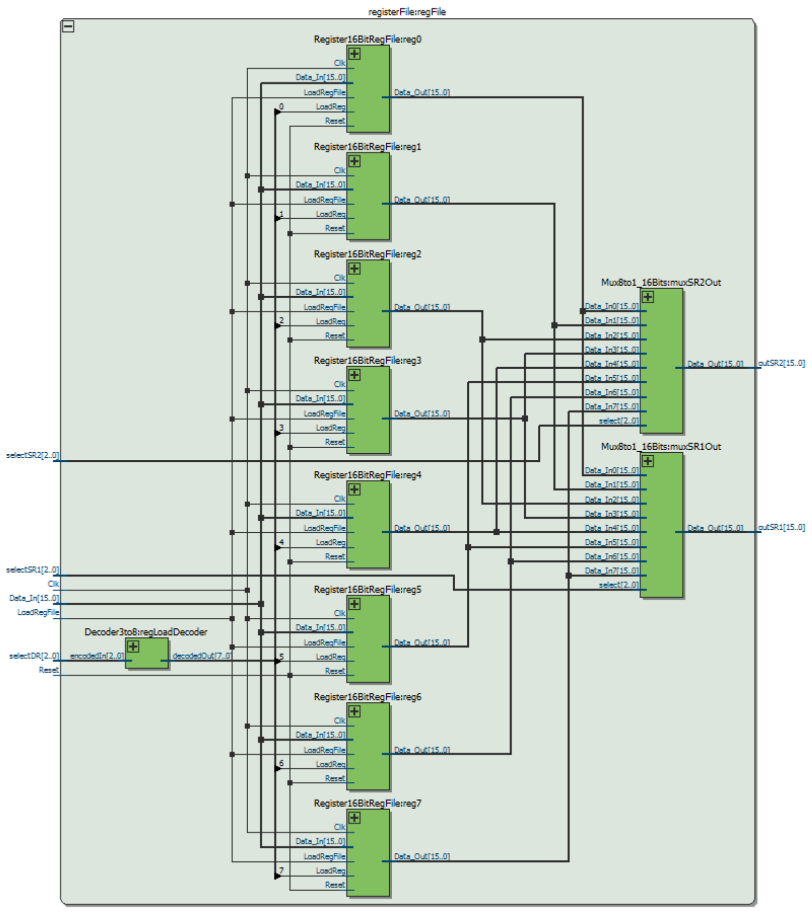
5. Processing Unit

The processing unit played an important role in the manipulation of data. It allowed us to perform arithmetic and logical operations on specific pieces of data. Since all computer programs require this functionality, the processing unit is a necessary part of any computer architecture. In our design, the processing unit had one data input from the bus, one data output to the bus, and multiple control signals that were generated by the control unit. We decided to use five main submodules to implement the entire processing unit. These modules were the register file, the ALU, DRMUX, SR1MUX, and the SR2MUX. The image below shows the schematic block diagram of the processing unit. The thick, black line is the bus and all of the other modules have been labeled. Our design did not have a GateALU. The connections with the bus have been explained in section 4. For more information on the three submodules, look at the sections below.



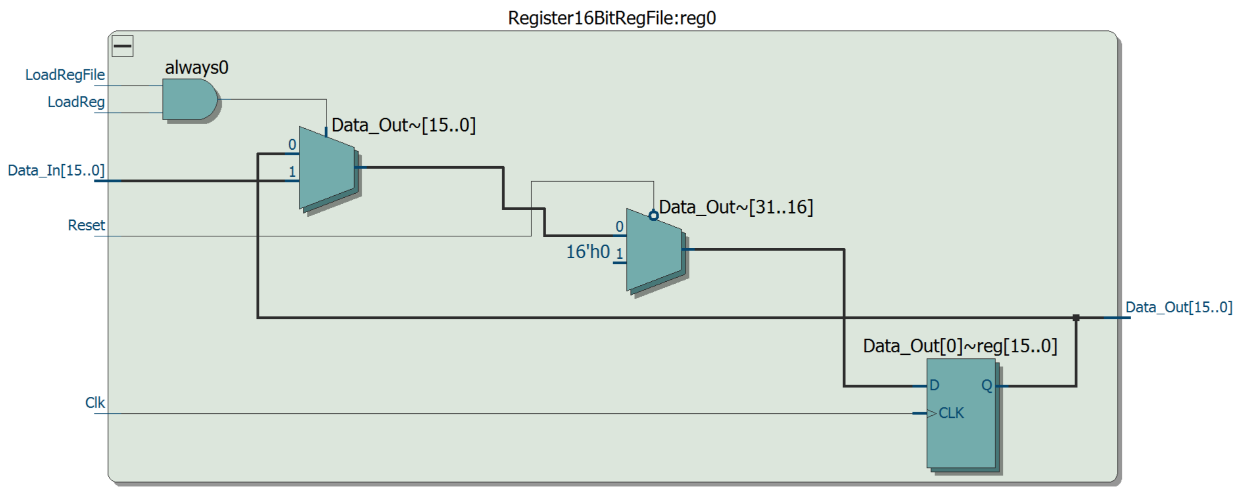
5.1 Register File Module

The register file was a source of temporary storage within the processing unit to hold the values on which we are performing the arithmetic operations. The purpose of this temporary storage was so that the computer doesn't have to keep going to the memory to grab values that it needs. Having to go to and from memory requires a lot more time and so it's very inefficient. By storing the values, we needed in this register file we had more instant access to them. The inputs into our register file were the select signals for DR, SR1, and SR2, the load signal, the reset signal, the Clk, and the Data_In. The outputs from our register file were the chosen values of SR1 and SR2. We decided to use 8 16-Bit registers to build the register file. Our design also consisted of a 3 to 8-Bit decoder and 2 8 to 1 16-Bit muxes. The sections below provide more details on these submodules. The image below shows the schematic block diagram of the register file.



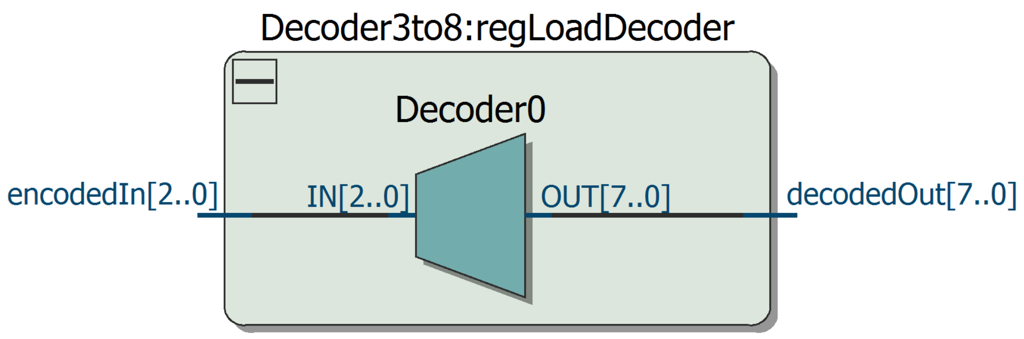
5.1.i 16-Bit Register for Register File

The 16-bit register module played an important role in the register file. It was used as a general purpose register module to store the value that was on the bus. All 8 registers were connected directly to the bus and based on the load signals that were provided to them, specific registers would retain the value that was on the bus at a given time. Our design allowed us to parallel load all of the values and output all of the values in parallel. The inputs into the register module were 2 load signals, Data_In, reset, and Clk. If reset was low, then we cleared the value in the register and stored 0. We needed 2 load signals because one of the load signals was used an enable for the entire register file while the other load signal was used to select one of the 8 registers specifically. This allowed us to select only one register at time to choose as the destination register. The outputs from the register were current bits stored in the register. The image below shows the schematic block diagram of the 16-Bit Register used in the Register file. This module is very similar to the one seen in section 3.1. The only difference is that this module accounts for the extra load signal mentioned above.



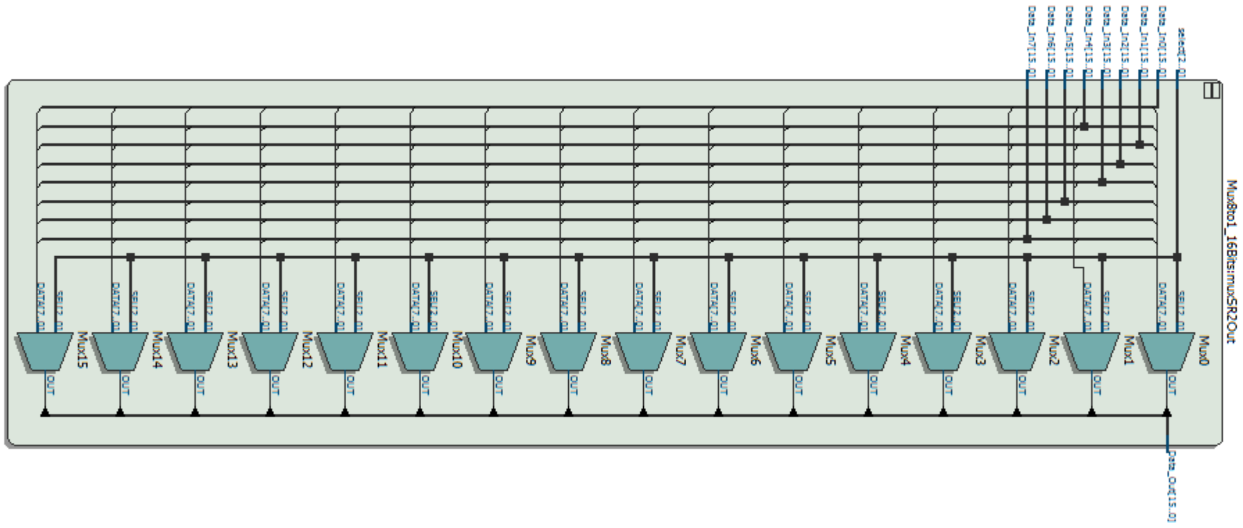
5.1.ii 3 to 8-Bit Decoder

The DR(destination register) select signal, the input into this module, was a 3-Bit wide signal and thus was able to map to each of the 8 registers. The 3 to 8-Bit Decoder module was used to decode the select signal and and provide an 8-Bit output signal that would pick the correct register. For example, if the DR select signal was 111, then the 8-Bit decoded signal would set the load signal for register 7 high and all other load signals as low. This would thus allow us to pick all of the registers correctly when we needed to. The schematic block diagram of this module is provided below.



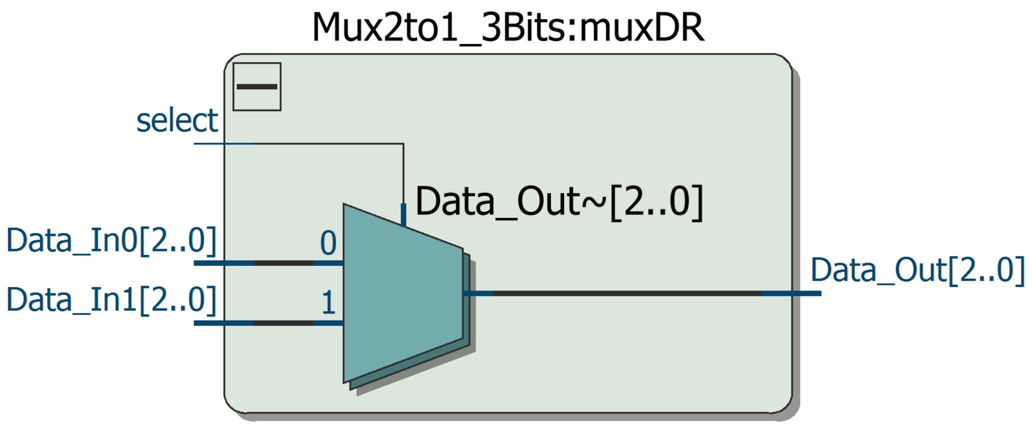
5.1.iii 8 to 1 16-Bit Muxes

2 instances of this 8 to 1 16-Bit muxes module were used to select the outputs for the register file. The inputs to the muxes where each of the 16-Bit values of each of the 8 registers. Based on the SR1 and SR2 select signals, the muxes chose the corresponding register and output its value from the register file to be used by the processing unit. The schematic block diagram of mux is shown in the image below.



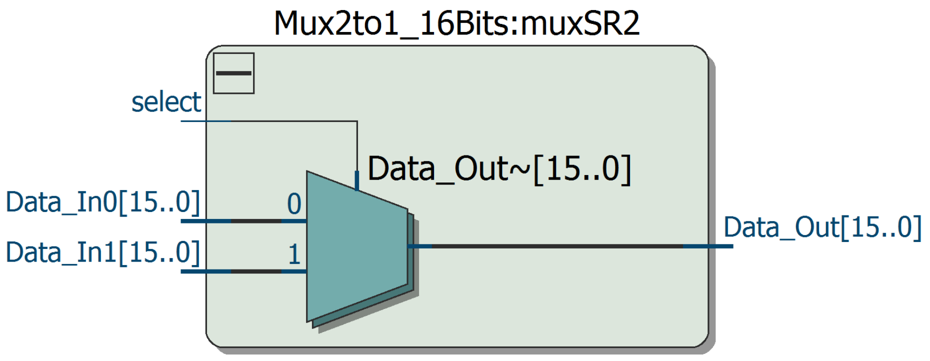
5.2 DRMUX and SR1MUX Modules

The DR mux was used to relay the correct DR select signal to the register file. The SR1 mux was is also doing the same thing except it is used to relay the correct select signals to pick the correct SR1 output values. Both of the muxes are 2 to 1 3-Bit muxes. The input to the muxes were a 2 3-bit values taken from the IR and the select signals which were values produced by the control unit. The outputs are the select signals for SR1 and DR. The schematic block diagram for this 2 to 1 3-Bit mux is shown below.



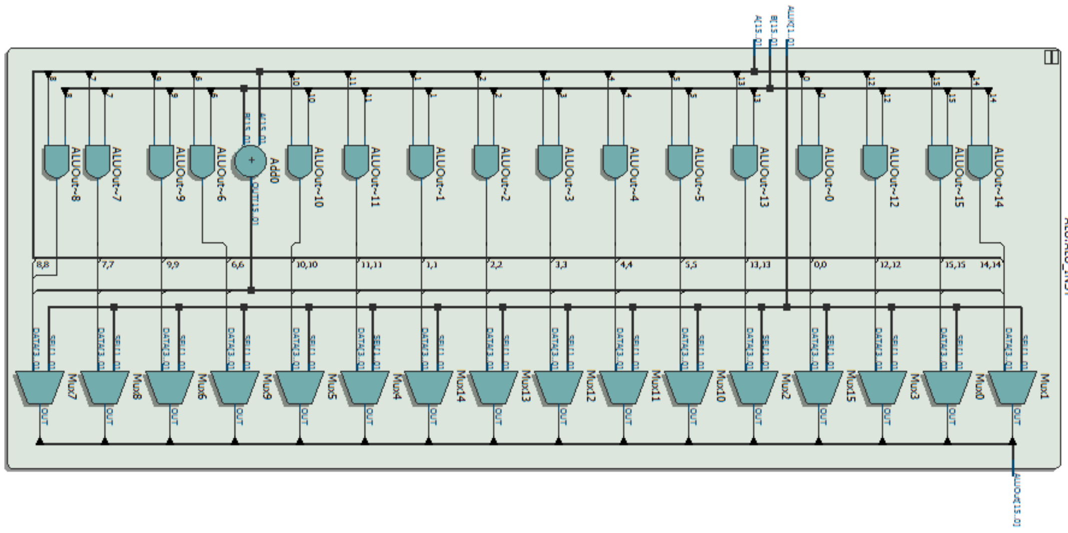
5.3 SR2MUX Module

The SR2 Mux takes in the input from the register file and a 16-Bit sign extension of IR[5:0] then selects between the two values. For normal operations, the output value is the value of the register file. However, when we want to perform either an immediate ADD or an immediate AND, we pass pass through the sign extension of IR[5:0]. The select bit to select between these two is generated by the control unit. The schematic block diagram of this 2 to 1 16-Bit mux module is shown in the image below. It is very similar to the schematic block diagram shown in section 5.2.



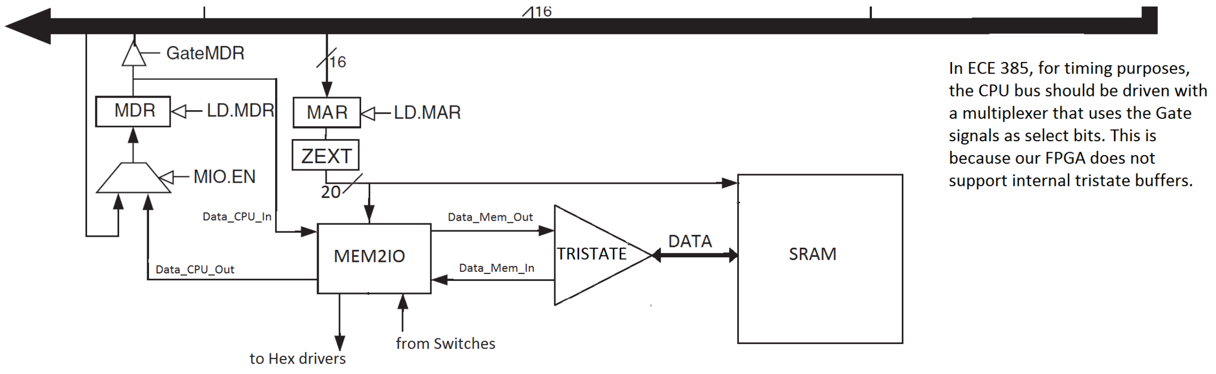
5.4 ALU Module

The ALU, also known as the arithmetic logic unit, was the part of the processing unit that performed the actual arithmetic and logical operations. The inputs into the ALU were the selected SR1 value as A, the selected SR2 value as B, and the control signals as ALUK. Based on the control signals provided, the ALU would perform either the ADD, AND, NOT A, or PASS A operations. The schematic block diagram of the ALU is shown below.



6. Memory Unit

Since our microprocessor design follows the stored-program structure, the memory unit plays a very important role in the functionality of our SLC3. The memory unit consists of six main modules. These modules are the MDR, MDR Mux, MAR, MEM2IO, hex driver, Tristate, and Memory. These modules are described with more details in the sections below. The image below shows the schematic block diagram of the memory unit. Our design did not have a GateMDR. The connections with the bus have been explained in section 4.


16
GateMDR
16
MDR
LD.MDR
MAR
LD.MAR
ZEXT
MIO.EN
20
Data_CPU_In
Data_Mem_Out
MEM2IO
TRISTATE
DATA
SRAM
Data_CPU_Out
Data_Mem_In
to Hex drivers
from Switches
In ECE 385, for timing purposes, the CPU bus should be driven with a multiplexer that uses the Gate signals as select bits. This is because our FPGA does not support internal tristate buffers.

6.1 MDR and MAR

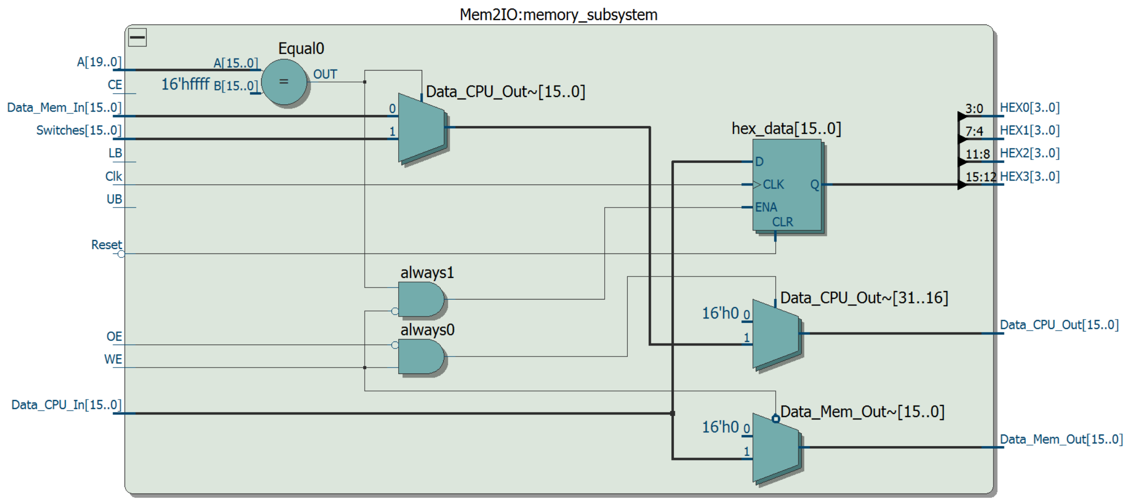
The MAR(Memory Address Register) and MDR(Memory Data Register) were used to interface with the memory. The MDR contained that data that the system was working with and MAR contained the address that the system was working with. For example, when trying to store x4747 at address x0002, the MAR contained x0002 and MDR contained x4747. Since MAR and MDR are both 16-Bit registers, the input into them is a 16-Bit data in, load, reset, and Clk. The output from them is the parallel out of each of the 16-Bits. The schematic block diagram of the 16-Bit register module used for the MAR and MDR is shown in section 3.1.

6.2 MDR Mux

The MDR mux selects the the value that will will be loaded into the MDR. It takes in two inputs and a select signal. The inputs are 16-Bit values directly from the bus and MEM2IO. The output is a 16-Bit value that goes to the MDR. This allows the MDR to either be loaded with a value from memory or from the bus to be stored in memory. Since the MDR mux is just a 2 to 1 16-Bit mux, the schematic block diagram of the MDR mux is equivalent to that of the circuit in section 5.3.

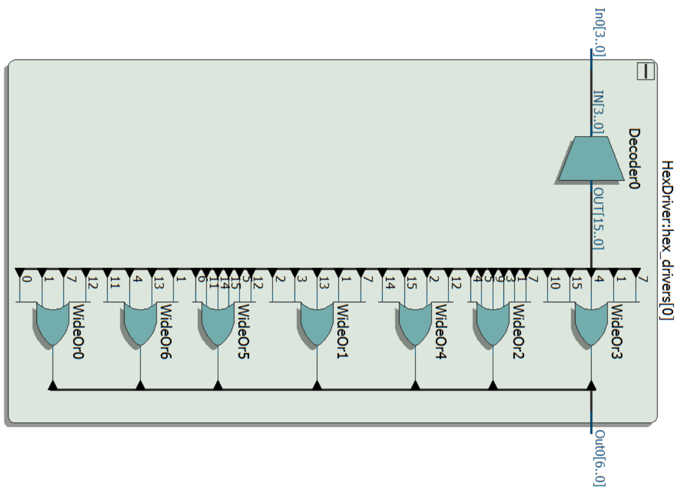
6.3 MEM2IO Module

MEM2IO is the part of the SLC3 that behaves as the input/output interface for the user. This module allows the user to use the switches on the FPGA to input values and communicates with the user by outputting values to the hex displays. This input/output occurs in special circumstances. As specified by the design, the user input is taken from the switches when the address being READ from is xFFFF. Similarly, the output to the hex display occurs when the address being WRITTEN to is xFFFF. The inputs into the MEM2IO are Data_CPU_In, Data_Mem_IN, and Address. The outputs are Data_CPU_Out and Data_Mem_Out. The schematic block diagram of the circuit is shown in the image below.



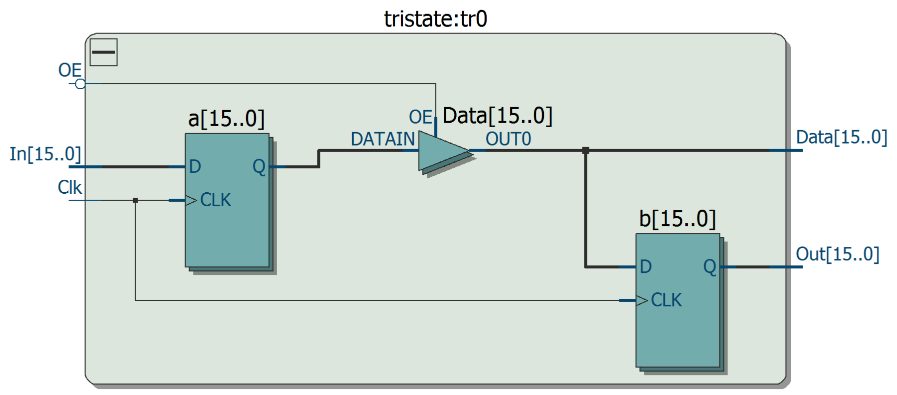
6.4 Hex Driver Module

The hex driver module simply provided us with a way to show the user the current values of the register A and B by using of the hex displays on the development board. The driver took in a 4 bit input and used a decoder to define the meaning of each of the 16 patterns that might arise. The schematic of the hex diver module is shown in the image below. The inputs to each of the hex drivers come from registers from MEM2IO and then outputs connect directly to the hex displays.



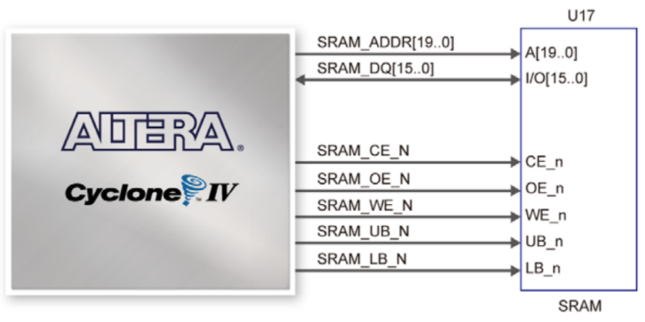
6.5 Tristate Module

The tristate module is the part of the memory unit which communicates directly with the the external memory, the SRAM. It has an inputs of Clk, OE, and In and an output called Out. There is also an inout wire called Data which is connected to the external memory. If OE is high, meaning that we are writing to memory, the value on the inout Data wire is set have the value of In. This allows us to write whatever we want to memory. When OE is low, we are reading from memory and thus we need to let memory have control of the Data wire. To do this we set the Data wire to have 16-Bits of high Z. This allows the external memory to put whatever value it wants on the data wire. Once this value has been assigned by the external memory, it is then relayed to Out on the positive edge of the clock. By doing this, we are able to use a single inout wire to communicate with any external memory unit. The schematic block diagram of the tristate module is shown in the image below.



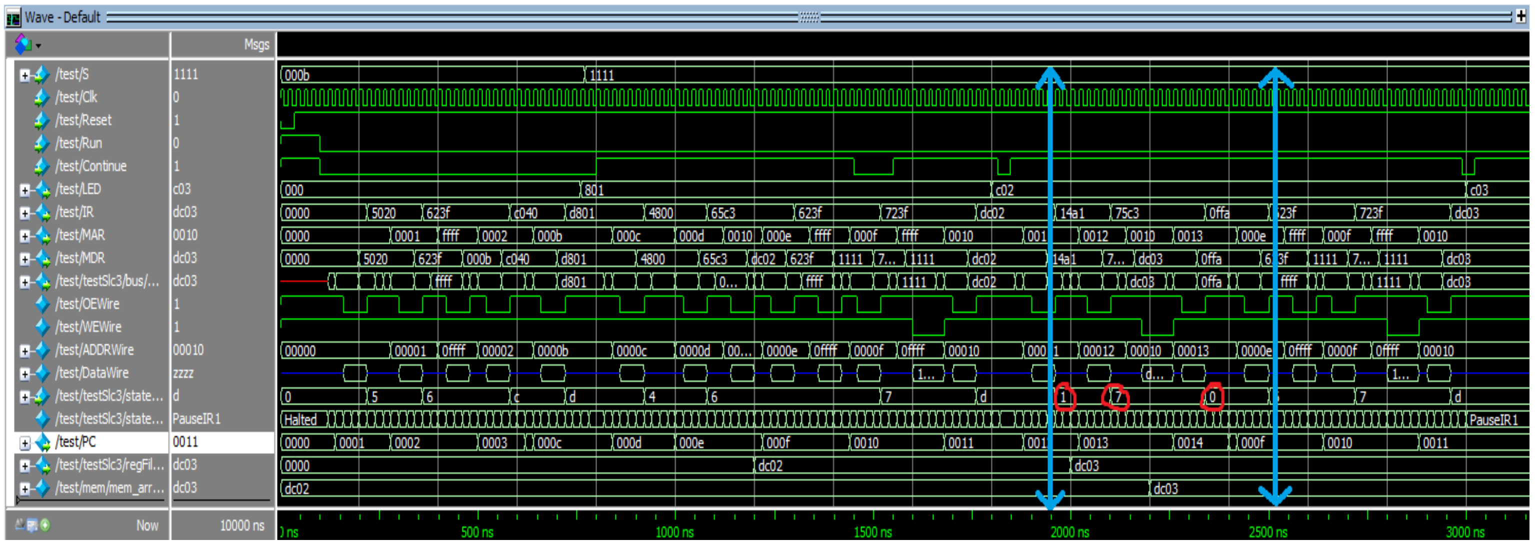
6.6 External Memory

The DE2-115 board has 2MB of onboard memory with a width of 16-bits. This makes it perfect for use with our design of the SLC3. The inputs into this memory were a 20-bit Address, a 16-bit inout Data wire, a chip enable, an output enable, a write enable, an upper byte enable, and a lower byte enable. The CE, OE, WE, UB, and LB were all active low and thus were assigned appropriately by the control unit. The connection between the SLC3 and the SRAM are shown in the image below.



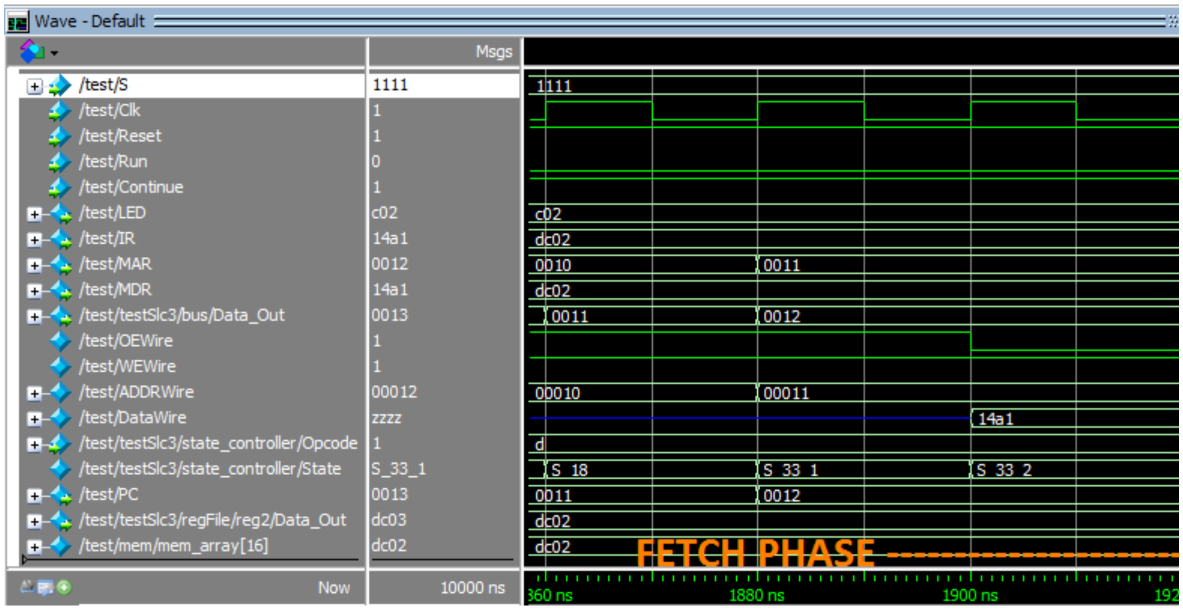
7. Pre-Lab Simulation Waveforms

The image below shows the waveform of our SLC3 performing the Basic I/O test 3. This program takes in an input from the switches and displays it on the hex displays. At the same time, it increments the value at memory address 16 by 1. This tests I/O and self modifying capabilities of our SLC3. We see that at the start of the waveform reset is set to low. This resets all of the memory units and then sets the current state of the machine to be in the HALTED state. The run signal is then set to low for a short period and we this starts up the SLC3. During the first three instruction cycles we see the SLC3 clears R0, loads value of switches + R0 into R1, and then jump to address in R1. Once the jump has occurred, the PC is pointing to the Basic I/O test 3. As the program executes, we can see that it uses the ANDi, ADDi, LDR, STR, BR, JSR, and PSE instructions. The last three instructions in the sequence of this loop are ADDi, STR, and BR. They are shown in the image below between the 2 blue markers. We will explore these three instructions in detail in the sections below.



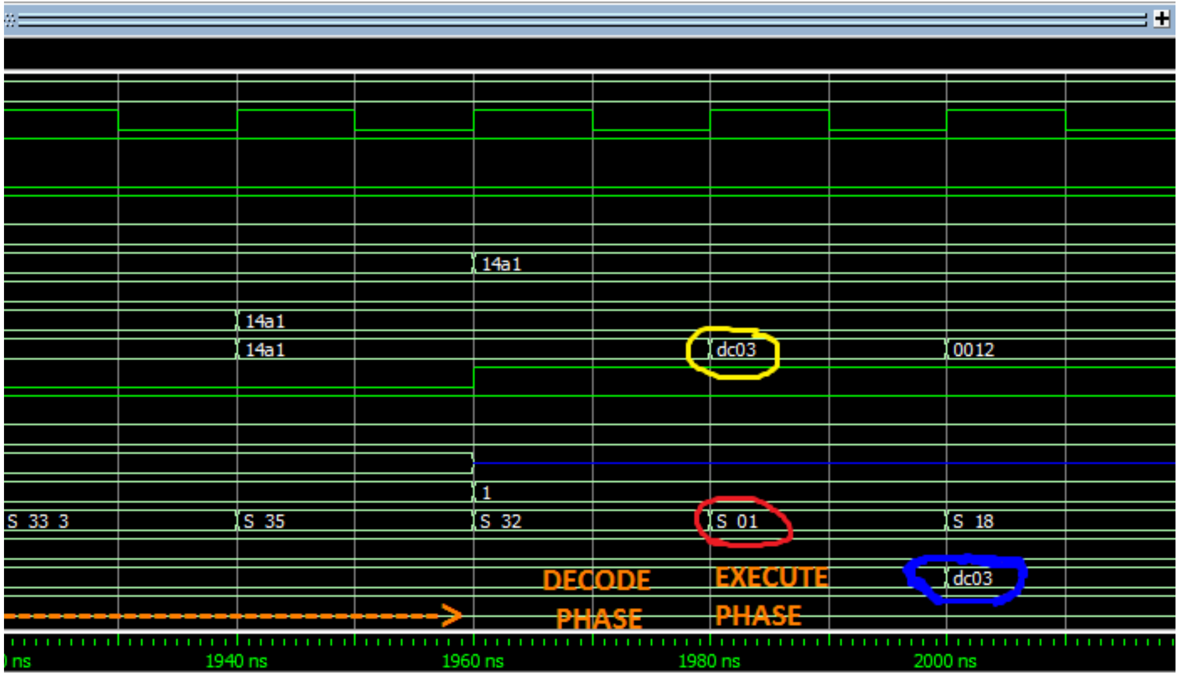
7.1 ADDi Simulation

The image below shows the simulation waveform of the ADDi instruction cycle. After the instruction has been fetched and decoded, the state machine transitions to state 1. This transition has been marked in red on the image. In this state, the control unit provides the signals to the ALU to add the value outputted from the register file by the immediate value provided by the IR. This value is seen in the waveform on the bus. This has been marked in yellow. On the next clock cycle we see that the value is taken from the bus and placed into register 2. This is seen in blue at the bottom right corner. This sequence of events completes the ADDi instruction. We had to break the image into two parts to make the values in the waveform readable.



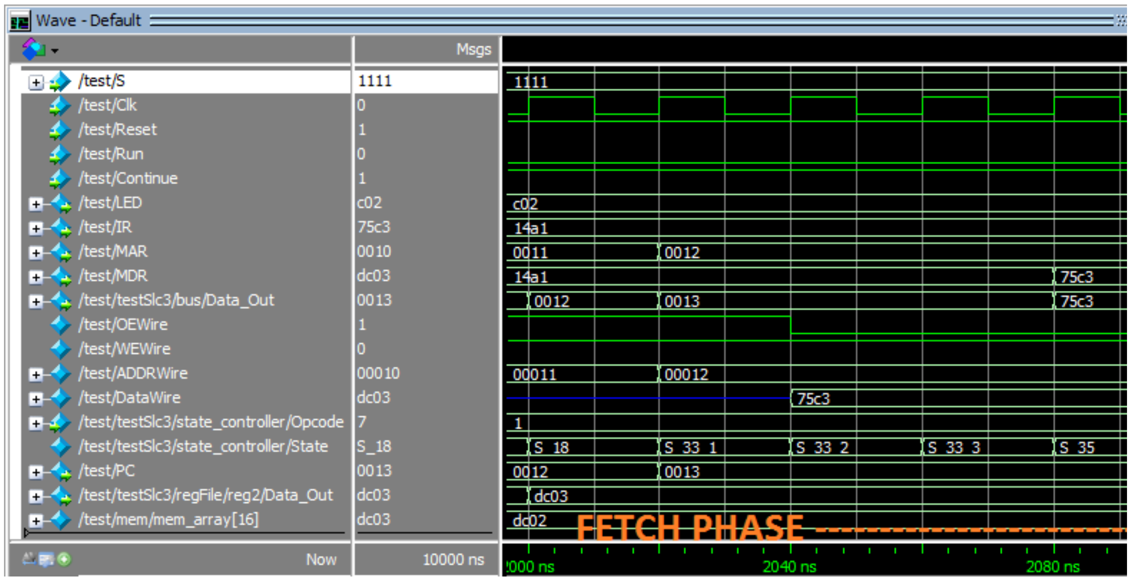
Wave - Default

	Msgs
/test/S	1111
/test/Clk	1
/test/Reset	1
/test/Run	0
/test/Continue	1
/test/LED	c02
/test/IR	14a1
/test/MAR	0012
/test/MDR	14a1
/test/testSlc3/bus/Data_Out	0013
/test/OEWire	1
/test/WEWire	1
/test/ADDRWire	00012
/test/DataWire	zzzz
/test/testSlc3/state_controller/Opcode	1
/test/testSlc3/state_controller/State	S_33_1
/test/PC	0013
/test/testSlc3/regFile/reg2/Data_Out	dc03
/test/mem/mem_array[16]	dc02
Now	10000 ns

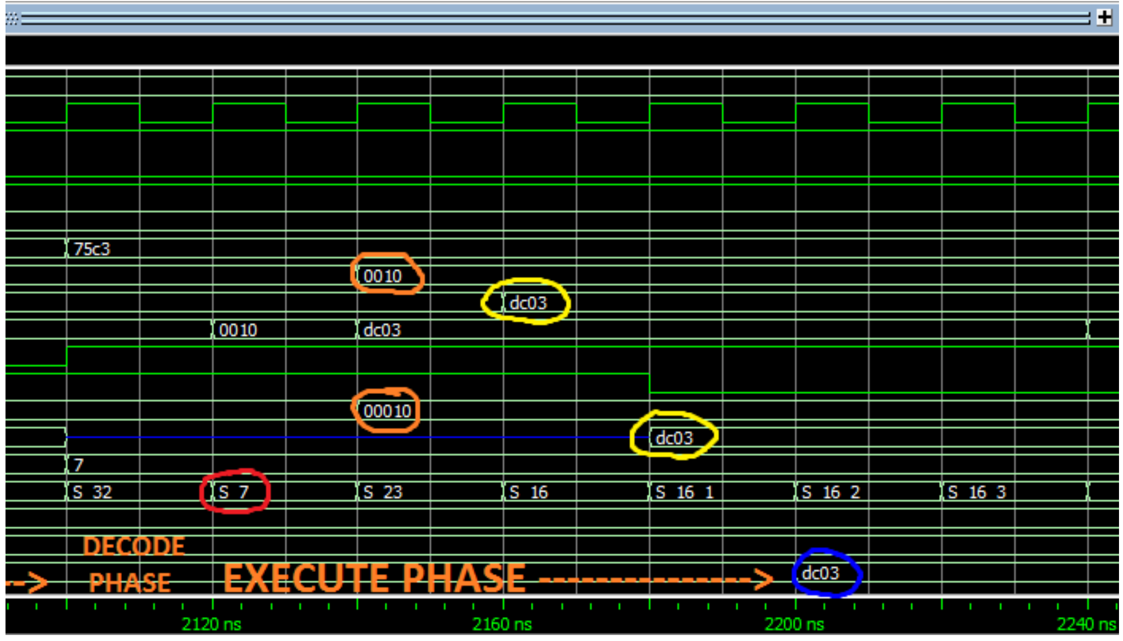


7.2 STR Simulation

The image below shows the simulation of the STR instruction cycle. After the instruction has been fetched and decoded, the state machine transitions to state 7. This transition has been marked in red on the image. In this state, the address at which we are storing is taken from the register file and loaded into the MAR. This has been marked in orange in the image below. At the next clock cycle, the state machine transitions to state 23. Here it provides the control signals such that the value that needs to be stored is taken from register 2 and loaded into the MDR. This has been marked in yellow below. At this point, the state machine transitions to state 16. In this state the value in MDR is stored in memory at the address specified in MAR. The completion of the store instruction is shown in the bottom right hand corner in blue. This sequence of events completes the STR instruction cycle. We had to break the image into two parts to make the values in the waveform readable.

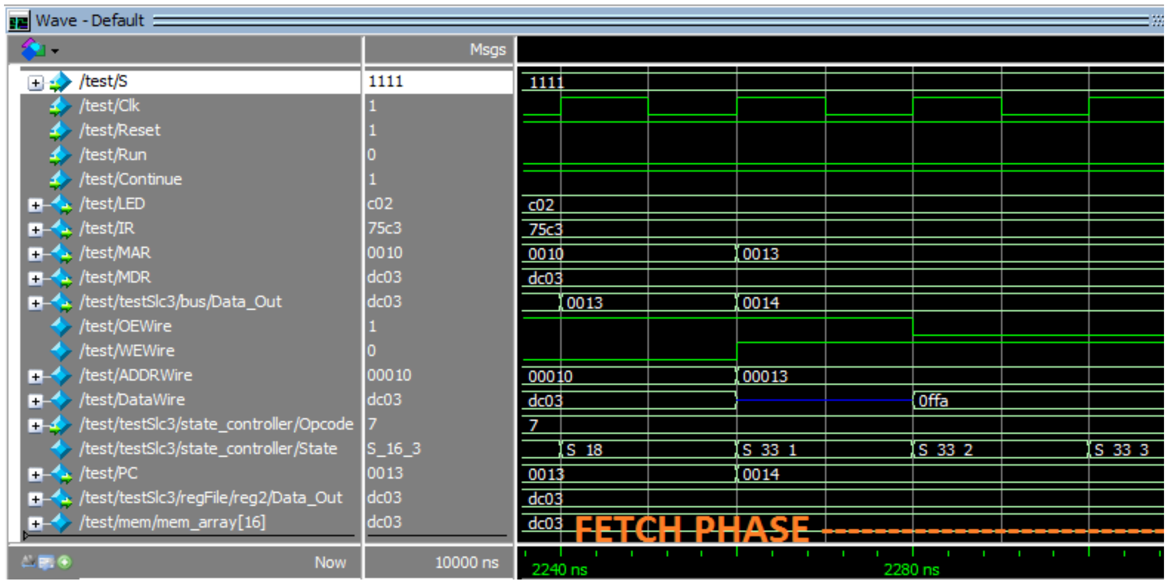


	Msgs
/test/S	1111
/test/Clk	0
/test/Reset	1
/test/Run	0
/test/Continue	1
/test/LED	c02
/test/IR	75c3
/test/MAR	0010
/test/MDR	dc03
/test/testSlc3/bus/Data_Out	0013
/test/OEWire	1
/test/WEWire	0
/test/ADDRWire	00010
/test/DataWire	dc03
/test/testSlc3/state_controller/Opcode	7
/test/testSlc3/state_controller/State	S_18
/test/PC	0013
/test/testSlc3/regFile/reg2/Data_Out	dc03
/test/mem/mem_array[16]	dc03



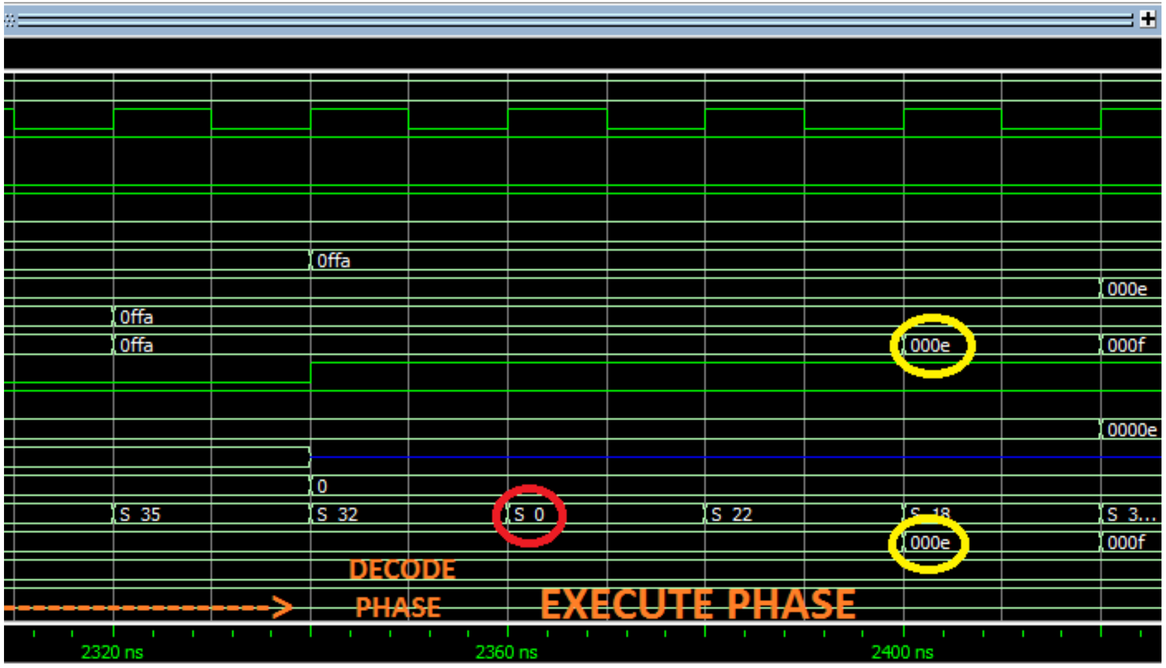
7.3 BR simulation

The image below shows the simulation of the BR instruction cycle. Once the instruction has been fetched and decoded, the state machine transitions to state 0. This has been marked in red below. At this state, the state machine checks if the branch enable bit is high. If BEN is high, then the machine transitions to state 22. Once in state 22, the PC is loaded with the value of PC plus PCOffset9. This offset value is specified by the IR. We see this value is then loaded into the PC on the next clock cycle. This is marked in yellow below. This sequence of states completes the BR instruction cycle. We had to break the image into two parts to make the values in the waveform readable.



Wave - Default

	Msgs
/test/S	1111
/test/Clk	1
/test/Reset	1
/test/Run	0
/test/Continue	1
/test/LED	c02
/test/IR	75c3
/test/MAR	0010
/test/MDR	dc03
/test/testSlc3/bus/Data_Out	dc03
/test/OEWire	1
/test/WEWire	0
/test/ADDRWire	00010
/test/DataWire	dc03
/test/testSlc3/state_controller/Opcode	7
/test/testSlc3/state_controller/State	S_16_3
/test/PC	0013
/test/testSlc3/regFile/reg2/Data_Out	dc03
/test/mem/mem_array[16]	dc03
Now	10000 ns



8. Post-Lab Questions

8.1 Design Resources and Statistics

The table below shows the statistics of our design of the SLC3. The values are within the expected range. Our frequency max is slightly lower than it should be, however, it is high enough such that our design will work correctly on the FPGA.

Category	**Data**
LUT	509
DSP	0
Memory (BRAM)	0
Flip-Flop	286
Frequency (MHz)	67.27
Static Power (mW)	98.61
Dynamic Power (mW)	7.05
Total Power (mW)	164.29

8.2 Question 1: What is MEM2IO?

MEM2IO interfaces with the user and controls all inputs and outputs for the user. This component allows the user to input values through the 16 switches and outputs values to the user through the hex display. This occurs based on on the instruction and state of the SLC3. Because of how it was designed, we see that the MEM2IO connects directly to the CPU through the MDR and IR. It also connects to the Physical Memory through the Tristate Buffer module. Through this connection we can directly allow the switches input to go the MDR and output the value in MDR directly to the hex display. This input/output occurs in special circumstances. The user input will be take taken from the switches when the address being READ from is xFFFF. Similarly, the output to the hex display occurs when the address being WRITTEN to is xFFFF. This module thus allowed use to complete our SLC3 and meet the requirements of being a Von Neumann architecture.

8.3 Question2: What is the Difference Between BR and JMP?

The Branch instruction sets the PC to the address specified in the instruction. This target address is specified as the offset from the current PC as PC + SEXT(IR[8:0]). This is one of the limitation to the Branch Instruction as it can only go within PC ± 128. This is also dependent on the NZP values of the instruction and only when the condition codes match, does the branch occur. The JMP instruction is an unconditional branch instruction where PC is set to the value of the base register and thereby allows any branch target to be specified. The difference is that the BR allows for the checking of certain conditions whenever it is called upon but limits the possibilities for a target address while JMP removes that limitation but unconditionally moves to the target address whenever called.

9. Conclusion

This lab taught us how to design and implement a complex state machine that was able to perform different instructions required specified by the SLC3 ISA. While implementing the core objectives of this lab, we came across many difficulties. One of the major issue in our implementation of the SLC3 was that there were conflicts with the provided resources. What this means is that the way we treated signals did not match with the way they wanted signals to be treated. We had treated signals related to test memory as all active low. However, the reset to memory was active high. Due to this difference and misunderstandings in how the given modules were developed, much of the debugging time was spent in trying to understand the error when there was no real error. For future labs, we realize that it is better to completely understand what is given and what is expected from us during the lab to reduce this type of debugging. One positive element that we learned from previous labs, that greatly helped us through this lab, was modularly testing each of the modules. Doing this allowed for us to be sure that whenever the proper signals were given to the module, the expected result was the same as the output result. This greatly reduced some of our debugging time and allowed us to better focus on the issue described before. Another minor issue we faced was with regards to timing. We noticed that, while many of our modules were similar to other groups, we had an extremely low fmax when compared to other groups. While it still passed the minimum requirement of 50 MHz expected of us, it was disappointing to see such a large discrepancy. For future labs, we would like to address the issue by understanding the longest data path and seeing how our modules interacted with each other. While this lab did provide and teach us about multiple issues, its importance lies in the understanding of the concepts of microprocessor design and proper engineering techniques for implementation of future designs.