ECE 385
Fall 2019
Experiment #6

Simple Computer SLC-3.2 in SystemVerilog

Pouya Akbarzadeh and Patrick Stach
Section: ABC
Lab TA: Vikram Anjur, Yuhong Li

Index

6.1 Introduction

6.1.1 Overall Goal of the Lab

- Basic functionality of the SLC-3 processor

In this experiment, we designed a simple microprocessor using the beloved SystemVerilog. It is a subset of the LC-3 ISA, a 16-bit processor with 16-bit Program Counter (PC), 16-bit instructions, and 16-bit registers.

6.1.2 Week 1

- What was the goal of week 1
  In week 1 our goal was to implement the FETCH instruction both in simulation and on the FPGA. This included loading PC into MAR and loading MDR into IR. Although week 1 seemed to have much less in terms to do in terms of designing the SLC-3, a good portion of it was used setting up and learning how to program the SRAM and familiarize ourselves with the code we were given.

6.1.3 Week 2

- What was the goal of week 2
  In week 2, we implemented the rest of the SLC3. This includes the DECODE instruction, which will check for the BEN bit and move to other parts of the state machine based on the opcode, in addition to the EXECUTE instruction: ADD/ADDi, AND/ANDi, NOT, BR, JMP, JSR, LDR, STR or PAUSE.

6.2 SLC-3 Explained

6.2.1 Summary of Operation

Little Computer 3, or LC-3, is an assembly language, which is a type of low-level programming language created by the dynamic duo Yale N Patel and Sanjay J Patel. The SLC-3 that we created in this lab differs from the original LC-3 because it is a bit more simple and differs. The design still uses FETCH to get instructions, decodes them, and executes. Which is explained in section 6.2.2 of this lab. The difference is that the SLC-3 lacks some instructions such as TRAP, STI, and there is no use of R signal. We overcame the lack of our R, ready signal, by adding more states to wait until the correct time based on the worst possible case. The reason for the lack of the R signal is due to the fact that the SRAM in use did not support it.

6.2.2 Performance of Function

- Describe in words how the SLC-3 performs its functions. In particular, you should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.

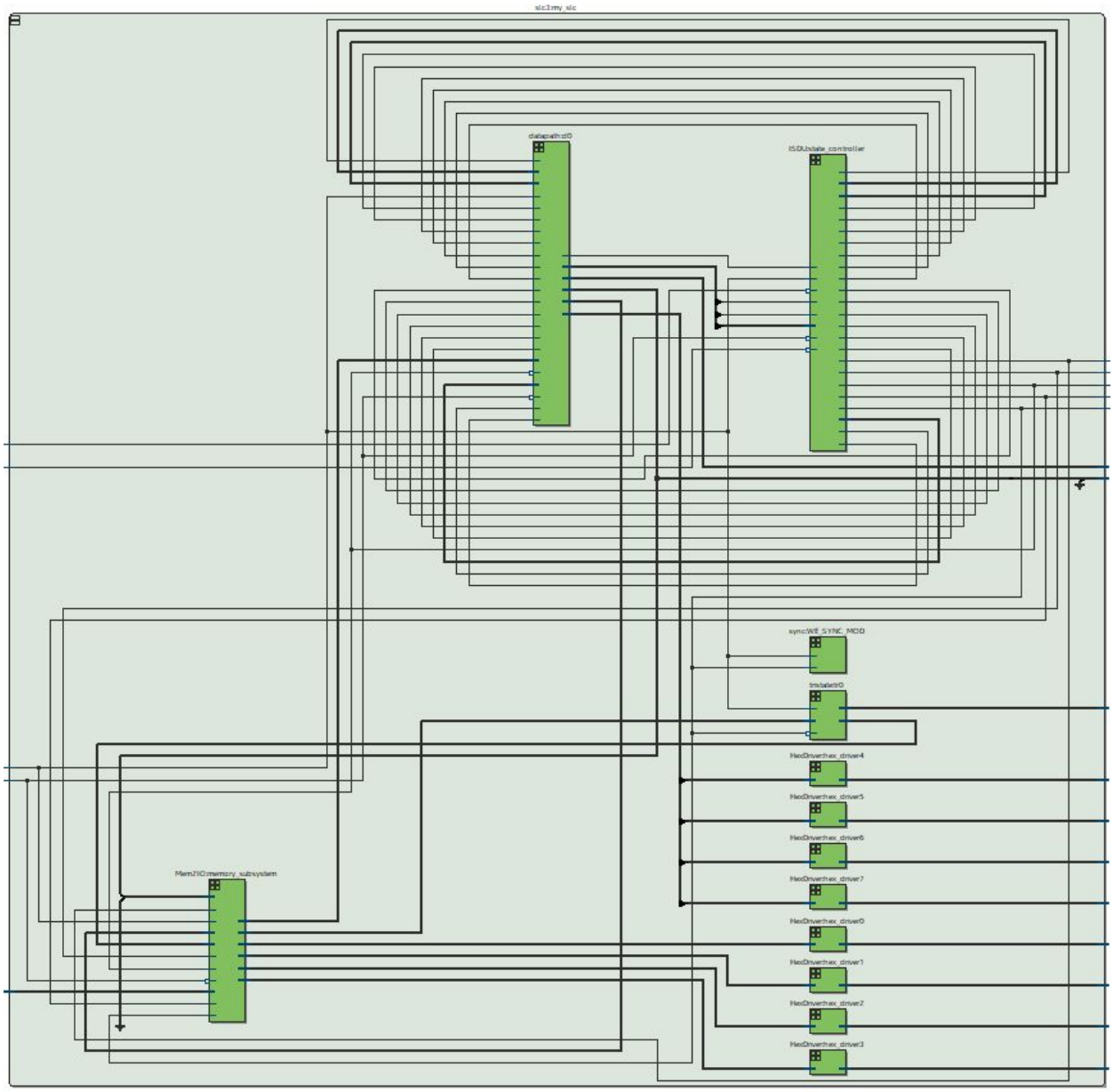
  First, instructions are FETCHed from memory and stored in the IR. From there, we DECODE the opcode from the IR and thus go to the specified next state from the OPCODE. The OPCODE is responsible for choosing between the following operations: ADD/ADDi, AND/ANDi, NOT, BR, JMP, JSR, LDR, STR or PAUSE.

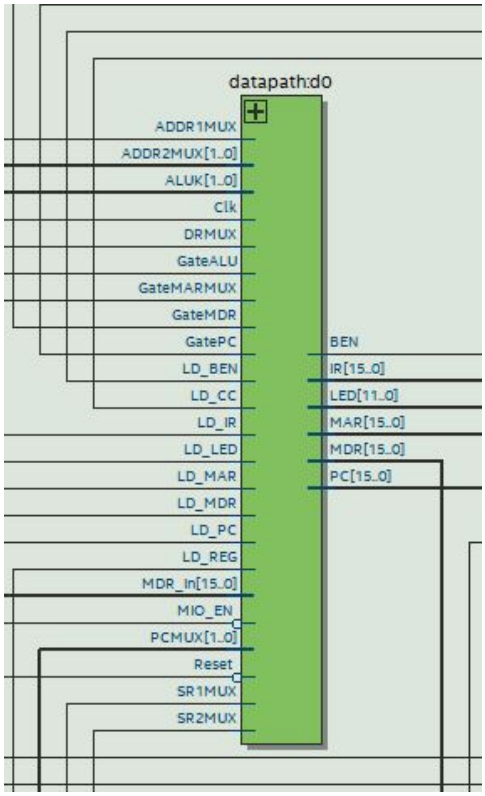
  The EXECUTE cycle is responsible for actually performing the operation, then looping back to initial state.

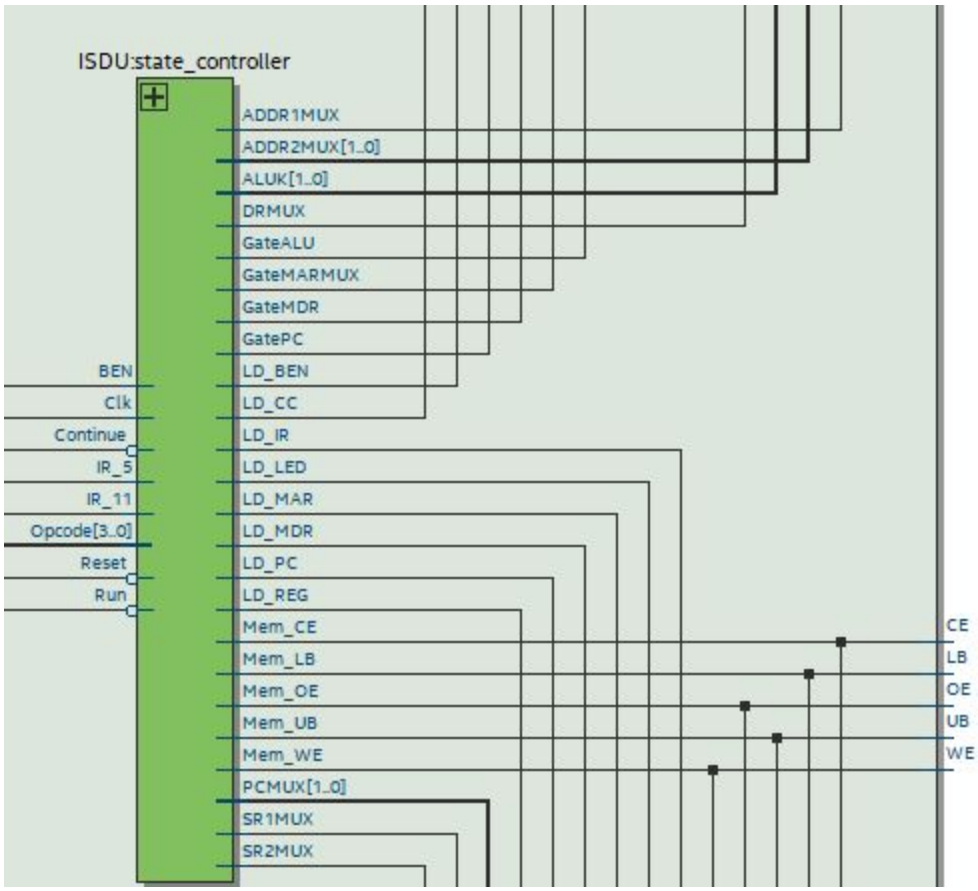
  In each state the FSM outputs control signals which get inputted into the various components such as muxes, registers, etc. This is how we perform the operations.

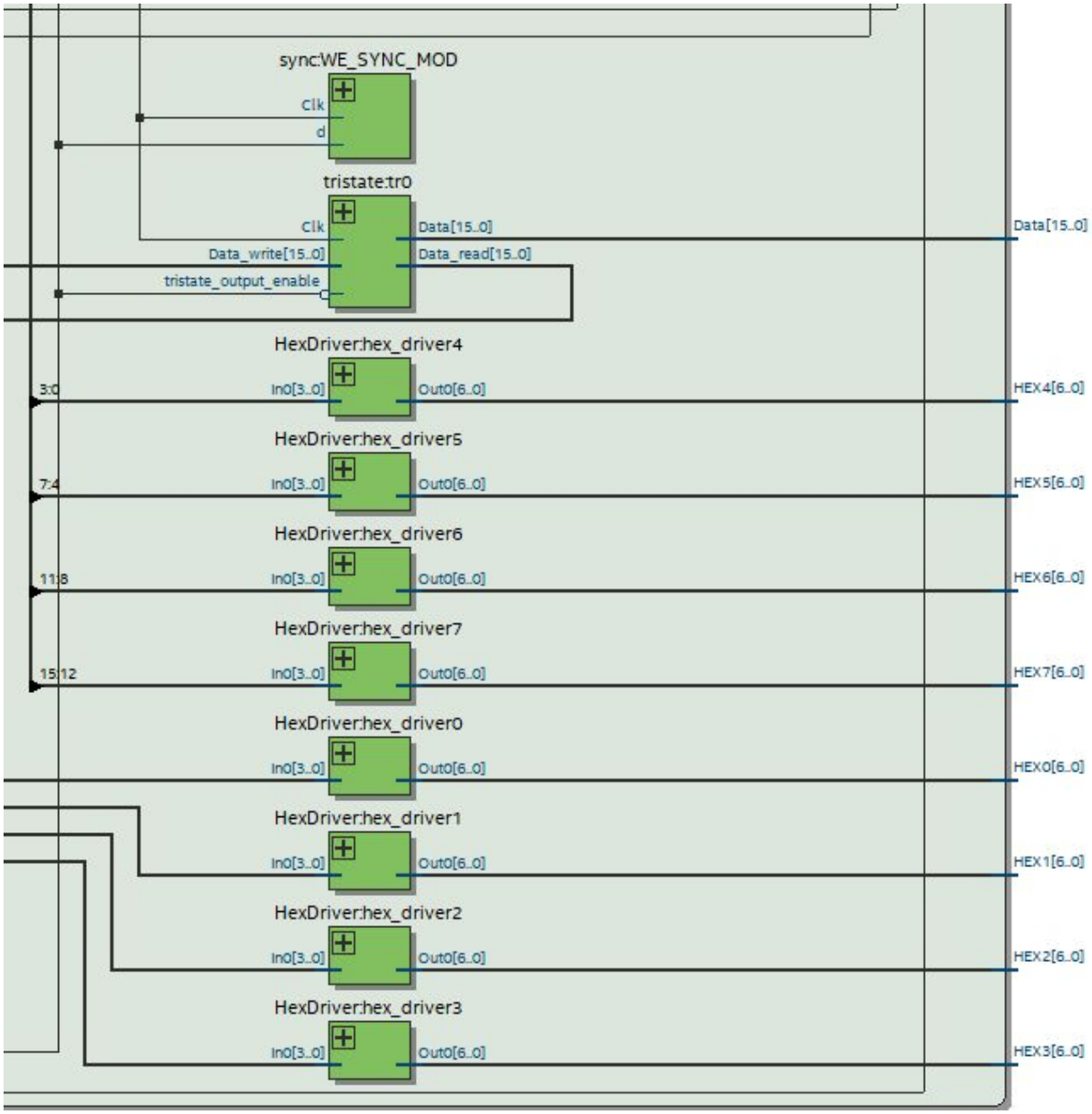
6.2.3 Block Diagram

- 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 (this can go into the individual module descriptions).




datapath:d0
ADDR1MUX
ADDR2MUX[1..0]
ALUK[1..0]
Clk
DRMUX
GateALU
GateMARMUX
GateMDR
GatePC
LD_BEN
LD_CC
LD_IR
LD_LED
LD_MAR
LD_MDR
LD_PC
LD_REG
MDR_In[15..0]
MIO_EN
PCMUX[1..0]
Reset
SR1MUX
SR2MUX
BEN
IR[15..0]
LED[11..0]
MAR[15..0]
MDR[15..0]
PC[15..0]


ISDU:state_controller
ADDR1MUX
ADDR2MUX[1..0]
ALUK[1..0]
DRMUX
GateALU
GateMARMUX
GateMDR
GatePC
LD_BEN
LD_CC
LD_IR
LD_LED
LD_MAR
LD_MDR
LD_PC
LD_REG
Mem_CE
Mem_LB
Mem_OE
Mem_UB
Mem_WE
PCMUX[1..0]
SR1MUX
SR2MUX
BEN
Clk
Continue
IR_5
IR_11
Opcode[3..0]
Reset
Run
CE
LB
OE
UB
WE



6.2.4 Written Description of modules

Adder.sv

Inputs:

[15:0] ADDR2_OUT, ADDR1_OUT,

Outputs:

[15:0] out

Description:

This module outputs the addition of two 16 bit inputs, ADDR2_OUT and ADDR1_OUT.

Purpose:

We need this adder to take in the original address (from ADDR1MUX) and add the offset (ADDR2MUX), and the output goes into PCMUX and GateMARMUX.

ALU.sv

Inputs:

[1:0] sel,
[15:0] A, B,

Outputs:

[15:0] out

Description:

The ALU takes two 16 bit input signals, A and B, and returns an operation based on the select signal (A+B, ~A, A&B, A)

Purpose:

We need the ALU because the SLC3 needs to be able to perform these 4 operations.

BEN_reg.sv

Inputs:

[2:0] IR_nzp,
N, Z, P,
Clk, Reset, LD_BEN,

Outputs:

BEN

Description:

This module takes inputs N, Z and P from the NZP registers and the NZP bits from the IR, outputting BEN = 1 if there are any cases where a N, Z, or P bit are both 1 in the register and IR (i.e if the IR has N=1 and the N register value =1, then we want BEN to equal 1).

Purpose:

We need this module because we need to generate a control bit that will determine whether we branch in a BR instruction or not.

Datapath.sv

Inputs:

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

Outputs:

[11:0] LED,
[15:0] MAR, MDR, IR, PC,
BEN

Description:

This module initializes all the components (adders, muxes, sign extends, registers, bus, ALU, etc) and connects them together.

Purpose:

This module is essentially the “building” of the circuit, we are creating different components and connecting them all together to follow the datapath of the SLC-3

Extend.sv

Inputs:

Input	Module
[10:0] in	sext10_0
[8:0] in	sext8_0
[5:0] in	sext5_0
[4:0] in	sext4_0

Outputs:

[15:0] out

Description:

This files has multiple modules that will sign-extend a number to 16 bits from the original size of the input.

Purpose:

There are various parts of the SLC-3 datapath that require 16 bit input such as the ALU, mux, etc. If we have any sort of input that we want to pass through those components that have less than 16 bits, we need to sign extend them to 16 bits. This is specifically used for when we extract a portion of bits from the IR.

HexDriver.sv

Inputs:

[3:0] In0

Outputs:

[6:0] Out0

Description:

The HexDriver was provided to us during Experiment 4 and was simply recycled for this experiment. The HexDriver translates a binary input representing a number into the specific segments of a 7-segment display that would need to be able to display that number in Hex

Purpose:

This is used to display calculations on the FPGA board in run-time.

ISDU.sv

Inputs:

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

Outputs:

LD_MAR, LD_MDR, LD_IR, LD_BEN,
LD_CC, LD_REG, LD_PC, LD_LED,

GatePC, GateMDR, GateALU, GateMARMUX,
[1:0] PCMUX,
DRMUX, SR1MUX, SR2MUX, ADDR1MUX,
[1:0] ADDR2MUX, ALUK,
Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE

Description:

This module is the main control unit responsible for the finite state machine in addition to controlling output signals depending on the state and current input. The finite state machine will first FETCH the instruction from memory and increment PC, then it will decode the opcode and transition to the state corresponding to that opcode, and finally execute the operation.

Purpose:

The ISDU is the backbone of the SLC-3 processor, it will navigate through its finite state machine as part of the FETCH, DECODE, and EXECUTE cycle and control all the components with the control signals it generates.

Lab6_toplevel.sv

Inputs:

input logic [15:0] S,
input logic Clk, Reset, Run, Continue,
inout wire [15:0] Data;

Outputs:

output logic [11:0] LED,

```
output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4,
                   HEX5, HEX6, HEX7,
 output logic [19:0] ADDR,
output logic CE, UB, LB, OE, WE,
```

Description:

Here the slc3 module is initialized with the input/output variables corresponding to the pin assignments

Purpose:

In order to program the FPGA we need to initialize the slc3 module with correct variable names corresponding to the pin assignments on the board.

Mem2IO.sv

Inputs:

```
input logic Clk, Reset,
input logic [19:0]  ADDR,
input logic CE, UB, LB, OE, WE,
input logic [15:0]  Switches,
input logic [15:0] Data_from_CPU, Data_from_SRAM,
```

Outputs:

```
output logic [15:0] Data_to_CPU, Data_to_SRAM,
output logic [3:0]  HEX0, HEX1, HEX2, HEX3 );
```

Description:

Load data from switches when address is xFFFF, and from SRAM otherwise. It will also pass data from CPU to SRAM and vice versa.

Purpose:

We need MEM2IO to interface with the SRAM since the SRAM is an external component not on the FPGA chip. MEM2IO accounts for things such as the SRAM data pins being bidirectional by using tristate buffers

Memory_contents.sv

Inputs:

Outputs:

output logic[15:0] mem_array[0:size-1]

Description:

This file is essentially the C programming that we are having our SLC3 process. The PC keeps track of which line we are in for the file and grabs the instructions from there.

Purpose:

We use the memory_contents.sv file so that we can see how store/ load operations work in simulation before we try it on the FPGA

NZP_reg.sv

Inputs:

input logic LD_CC, Clk, Reset,

input logic [15:0] in,

Outputs:

output logic N, Z, P

Description:

This module takes a 16 bit input and will update the NZP register output values when LD_CC is high based on the new input.

Purpose:

In various states of our finite state machine of the SLC-3 we have to set CC, i.e after ADD,

Reg_16.sv

Inputs:

input logic [15:0] in,
input clk, reset, load,

Outputs:

output logic [15:0] out

Description:

This register store 16 bits and when a load signal is passed as "1" it will store what is in the 16 bit input and output it.

Purpose:

We need to use this register to be able to store values- specifically the IR, MAR, MDR, PC values respectively in their own instances.

RegFile.sv

Inputs:

input logic LD_REG, Reset, Clk,
input logic [15:0] in,
input logic [2:0] SR1, SR2, DR,

Outputs:

output logic [15:0] SR1_OUT, SR2_OUT)

Description:

The register file is a unit with eight 16-bit registers that with input logic DR which will choose which register to load and input logic LD_REG for when to load. SR1, SR2 select which registers to output.

Purpose:

For our operations in the SLC-3 we need to be able to store our "answers" into our 8 data registers- this is how we implement it.

Muxes.sv

The Muxes.sv file has multiple module definitions of muxes that fall into a single category: 4:1, 2:1 and 16:1.

16:1 - bus_mux

Input/Output example: bus_mux

Inputs: input gate_PC, gate_MDR, gate_ALU, gate_MARMUX,
input logic [15:0] PC, MDR, ALU, MARMUX

Outputs: output logic [15:0] out

4:1 - pc_mux, adr2_mux

Input/Output example: pc_mux

Input: input logic [1:0] sel,
input logic [15:0] curr_pc, bus, adder,

Outputs: output logic [15:0] out

2:1 - dr_mux, mdr_mux, sr2_mux, sr1_mux, adr1_mux

Input/Output example: dr_mux

Inputs: input logic sel,
input logic [2:0] IR11_9

Outputs: output logic [15:0] out

Description:

A mux is used to choose between multiple inputs into a single output. Out select bit(s) determine what input to use, with a 2:1 requiring 1 select bit, a 4:1 requiring 2 select bits, and a 16:1 requiring 4 select bits.

Purpose:
PC_Mux is used to select if next PC value is PC+1, from BUS, or ADDER output.

Bus_mux is used in replacement of a tristate buffer since the FPGA has no internal tri-state buffers. We need to make sure that out of PC, MDR, ALU and MARMUX, only one of them is outputting to the mux.

Dr_mux is used to select whether our DR value is extracted from the IR or if we output 7.

Adr1_mux is used to select whether we output PC or SR1_out

Adr2_mux is used to select which sign extended version of the IR portions to use

Sr1_mux is used to select whether to choose our SR1 from IR[11:9] or IR[8:6]

Sr2_mux is used to select whether to output SR2_out or SEXT4_0 version of IR

Mdr_mux is used to select whether MDR is loading from bus or from mem2IO

slc3.sv
Inputs:

input logic [15:0] S,
input logic Clk, Reset, Run, Continue,

Outputs:

output logic [11:0] LED,
output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
HEX6, HEX7,
output logic CE, UB, LB, OE, WE,
output logic [19:0] ADDR,

Description:

This module is the definition of the SLC3 unit with all of the necessary connections generated. We initiate the hex drivers, datapath, mem2IO, and datapath here.

Purpose:

This module is what we need to instantiate the SLC3 in the top level to actually use it for processing the memory and performing operations.

Test_memory.sv

Inputs:

```
input  Clk,
input  Reset,
inout  [15:0]  I_O,
input  [19:0]  A,
input          CE,
               UB,
               LB,
               OE,
```

WE

Outputs:

Description:

This module is used to interface with the memory contents file in simulations, simulating the SRAM

Purpose:

We need to be able

Testbench.sv

Inputs:

NONE

Outputs:

NONE

Description:

The testbench is not synthesized and is only used in simulation purposes. It instantiates a top level module and we manually set signal values at certain times.

Purpose:

For simulating the program to see if it works, we need to be able to simulate user inputs such as RUN, Continue, switch values, etc.

Tristate.sv

Inputs:

input logic Clk,
input logic tristate_output_enable,
input logic [N-1:0] Data_write, // Data from Mem2IO

Outputs:

output logic [N-1:0] Data_read,

Description:

This module will set a specific side, output or input to high impedance (Z) so that we don't connect two wires to the same pin.

Purpose:

We need a tristate buffer when accessing the SDRAM because it is bidirectional- we both read and write from the same pin but we can only have one wire connected to it at a time.

6.2.5 ISDU

- Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.

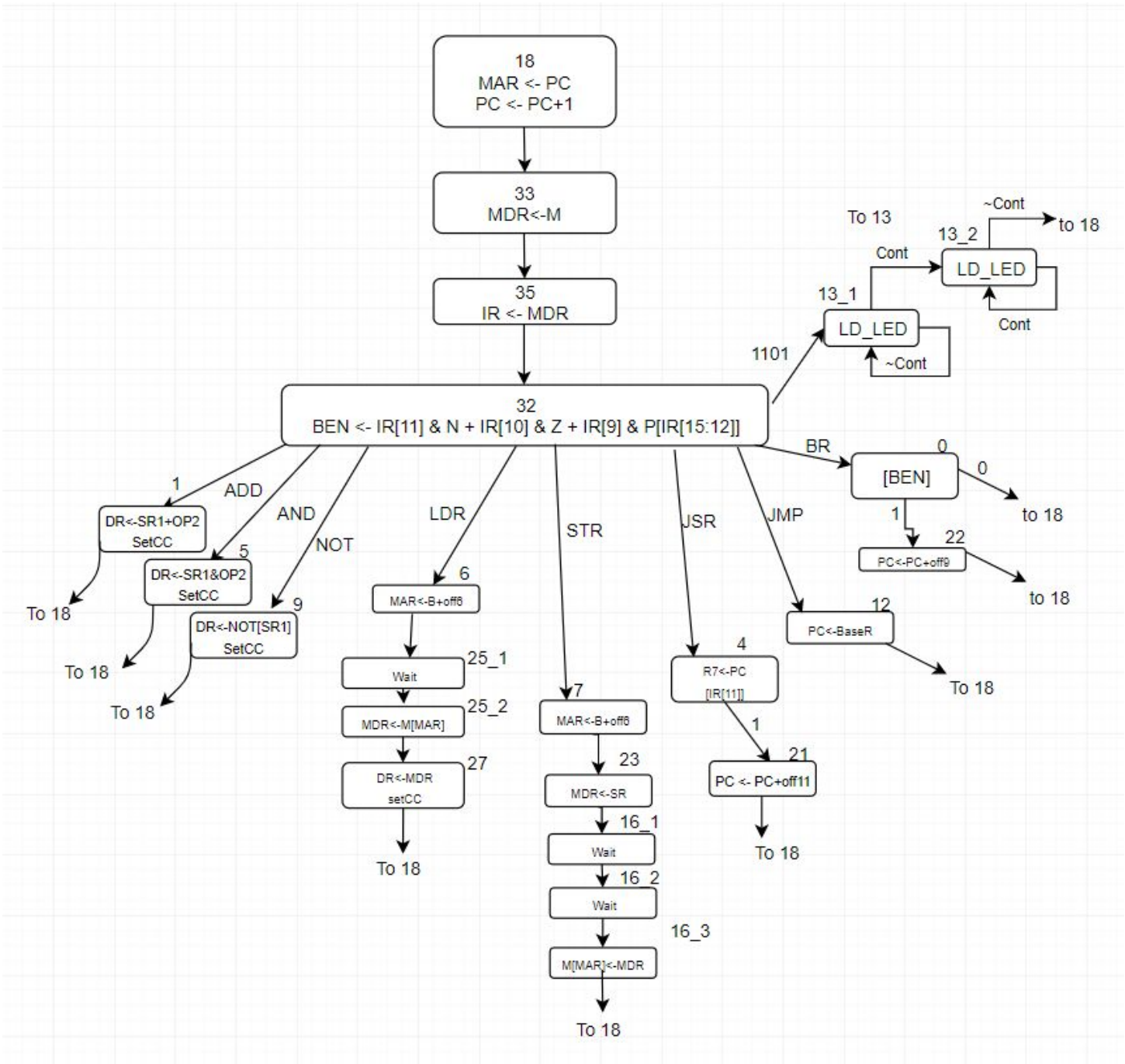
There are 3 different cycles that the ISDU cycles through- FETCH, DECODE, and EXECUTE. FETCH is responsible for loading the instruction from memory, DECODE selects what the next state to go to based on opcode (i.e. what ADD, NOT, etc). EXECUTE performs the actual operation.

There are default control signals for every state, such as disabling READ, disabling WRITE, etc. In addition to that, every state will have specific control signals that can overwrite default to control the components. The muxes and ALUs are controlled with select control signals, registers are controlled with Load control signals, and RegFiles use both select signals and Load signals.

If an instruction has multiple steps, it is broken up into multiple steps if we need to wait for parts of the circuit to propogate first (i.e any time we read we need to wait an extra cycle before loading the value into MDR).

6.2.6 State Diagram of ISDU

- This should represent all states present in the ISDU and their transitions. The diagram from Patt & Patel Appendix C can be used as a starting point but would need to be modified to be representative of the ECE385 implementation of the LC-3. You will lose points if you just copy the diagram.


18
MAR <- PC
PC <- PC+1
33
MDR<-M
35
IR <- MDR
32
BEN <- IR[11] & N + IR[10] & Z + IR[9] & P[IR[15:12]]
1101
To 13
13_2
~Cont
to 18
Cont
LD_LED
13_1
Cont
LD_LED
~Cont
BR
0
[BEN]
0
to 18
1
22
PC<-PC+off9
to 18
1
ADD
DR<-SR1+OP2
SetCC
To 18
5
AND
DR<-SR1&OP2
SetCC
To 18
9
NOT
DR<-NOT[SR1]
SetCC
To 18
6
LDR
MAR<-B+off6
25_1
Wait
25_2
MDR<-M[MAR]
27
DR<-MDR
setCC
To 18
7
STR
MAR<-B+off6
23
MDR<-SR
16_1
Wait
16_2
Wait
16_3
M[MAR]<-MDR
To 18
4
JSR
R7<-PC
[IR[11]]
1
21
PC <- PC+off11
To 18
12
JMP
PC<-BaseR
To 18

State NUMBER	Control Signal
S_00	
S_01	ALUK = 00
	GateALU = 1
	SR2MUX = IR_5
	SR1MUX = 1
	LD_REG = 1
	LD_CC = 1
S_04	GatePC=1
	LD_REG =1
	DRMUX= 1
S_05	ALUK = 10
	GateALU = 1
	SR2MUX = IR_5
	SR1MUX = 1
	LD_REG = 1
	LD_CC = 1
S_06	ADDR1MUX = 1
	ADDR2MUX= 10
	SR1MUX= 1
	GateMARMUX= 1
	LD_MAR = 1
S_07	GateMARMUX =1
	LD_MAR =1
	SR1MUX = 1
	ADDR1MUX= 1
	ADDR2MUX= 10
S_09	ALUK = 2'b01;
	GateALU = 1
	SR2MUX = IR_5;
	SR1MUX = 1
	LD_REG = 1
	LD_CC = 1
S_12	LD_PC =1

State NUMBER	Control Signal
S_13_1	LD_LED = 1
S_13_2	LD_LED = 1
S_16_1	Mem_WE =1
S_16_2	Mem_WE =1
S_16_3	Mem_WE =0
S_18	MAR = PC
	GatePC = 1
	LD_MAR = 1
	PCMUX = 00
	LD_PC = 1
S_21	ADDR1MUX = 00
	ADDR2MUX= 00
	PCMUX = 10
	LD_PC = 1
S_22	LD_PC=1
	ADDR1MUX= 0
	ADDR2MUX = 01
	PCMUX = 10
S_23	GateALU =1
	LD_MDR = 1
	SR1MUX = 0
	ALUK= 11
S_25_1	MEm_OE = 0
S_25_2	MEm_OE = 0
	LD_MDR = 1
S_27	GateMDR = 1
	LD_REG = 1
	LD_CC = 1
S_32	LD_BEN = 1
S_33_1	Mem_OE = 0
S_33_2	Mem_OE = 0
	LD_MDR = 1
S_35	GateMDR = 1

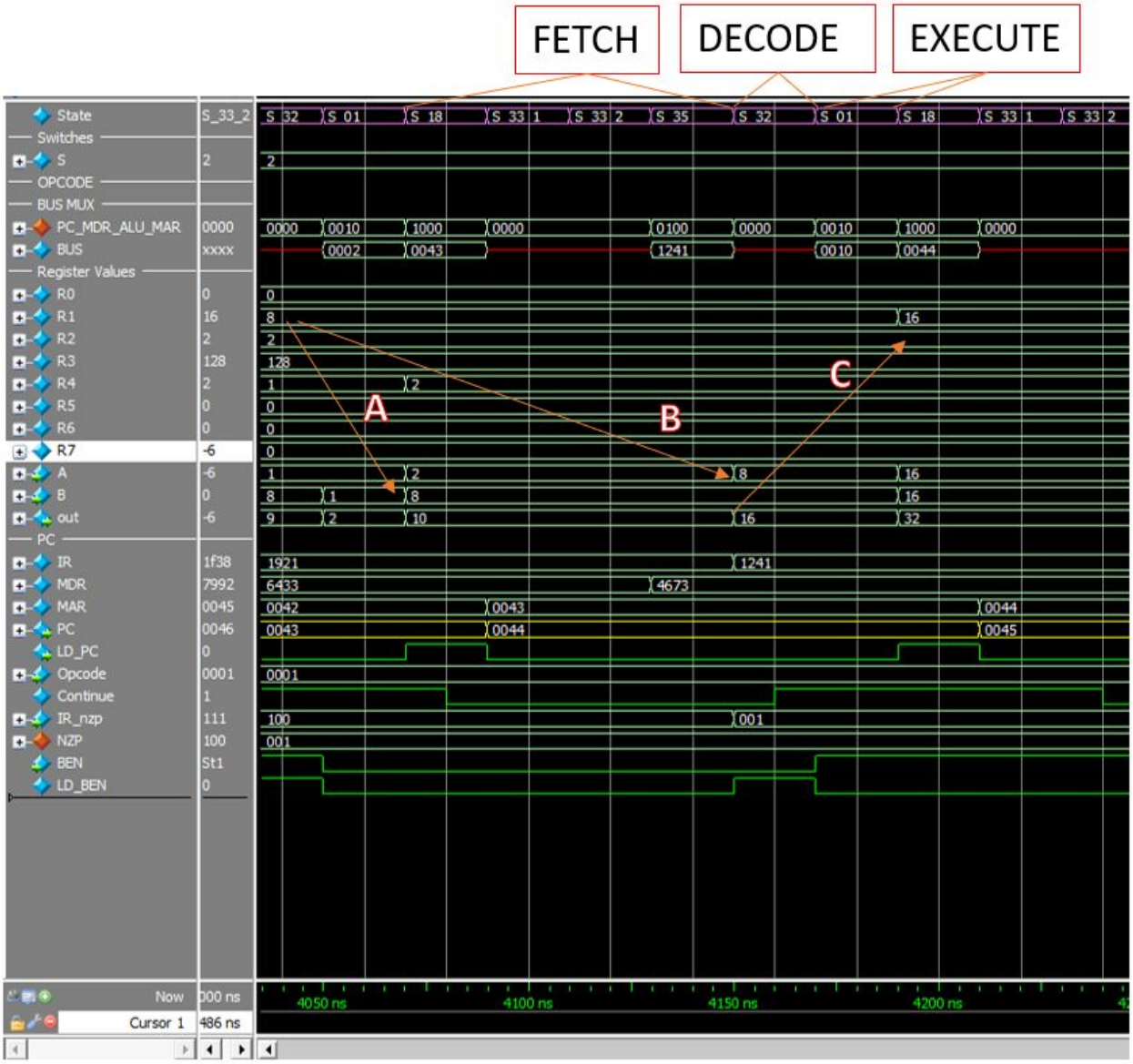
	PCMUX =10			LD_IR = 1
	SR1MUX = 1			
	ADDR1MUX = 1			
	ADDR2MUX = 11			

6.3 Simulations of SLC-3 Instructions

6.3.1 Simulation of Completion

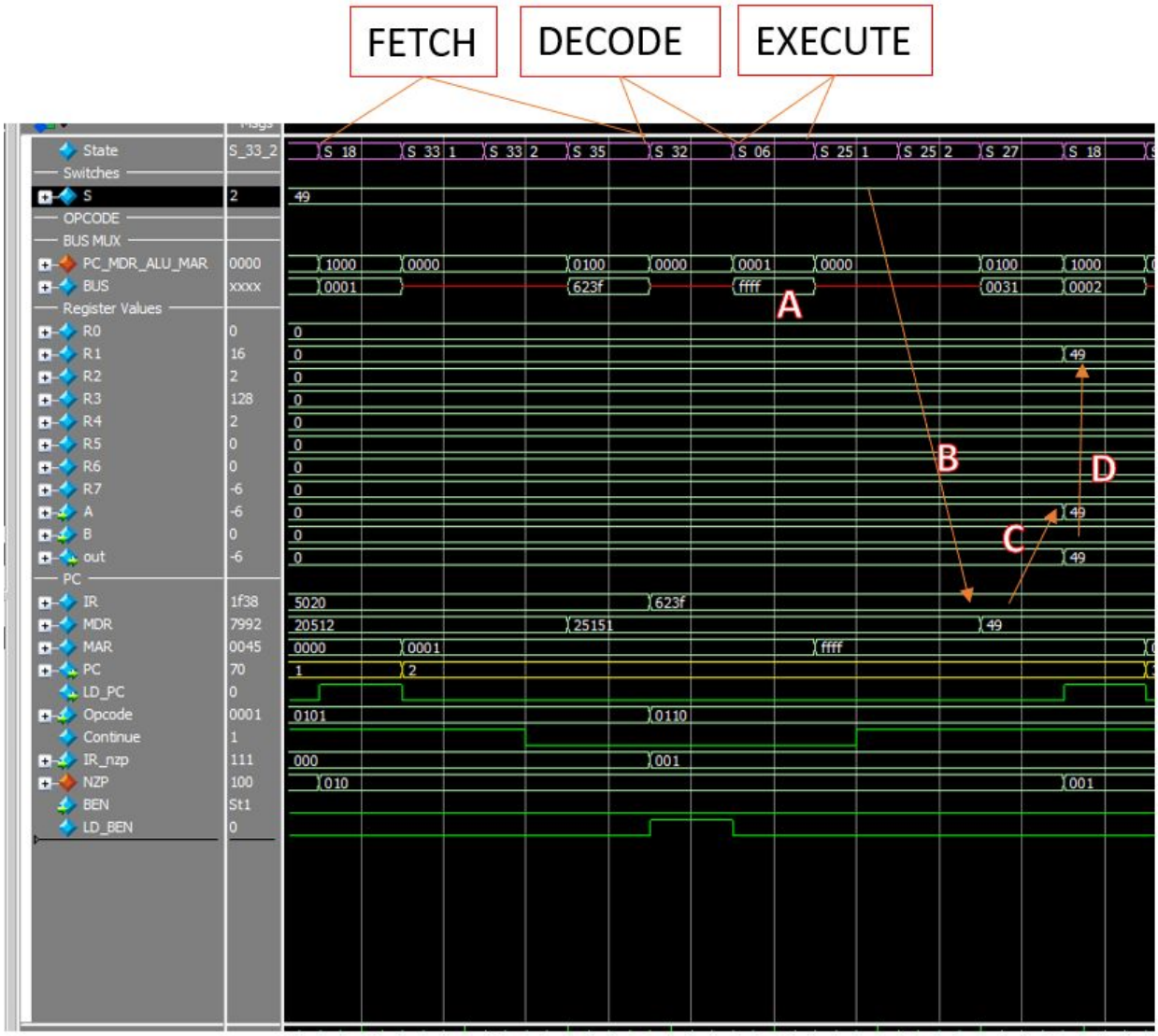
- Simulate the completion of 3 instructions from the following groups: ADD/ADDi/AND/ANDi/NOT; BR/JMP/JSR; LDR/STR. For example, consecutively simulating ADD, BR and then LDR would be an acceptable simulation. You must annotate this diagram (for instance, label where instructions begin, where the answer is stored, etc.)

opADD(R1, R1, R1)



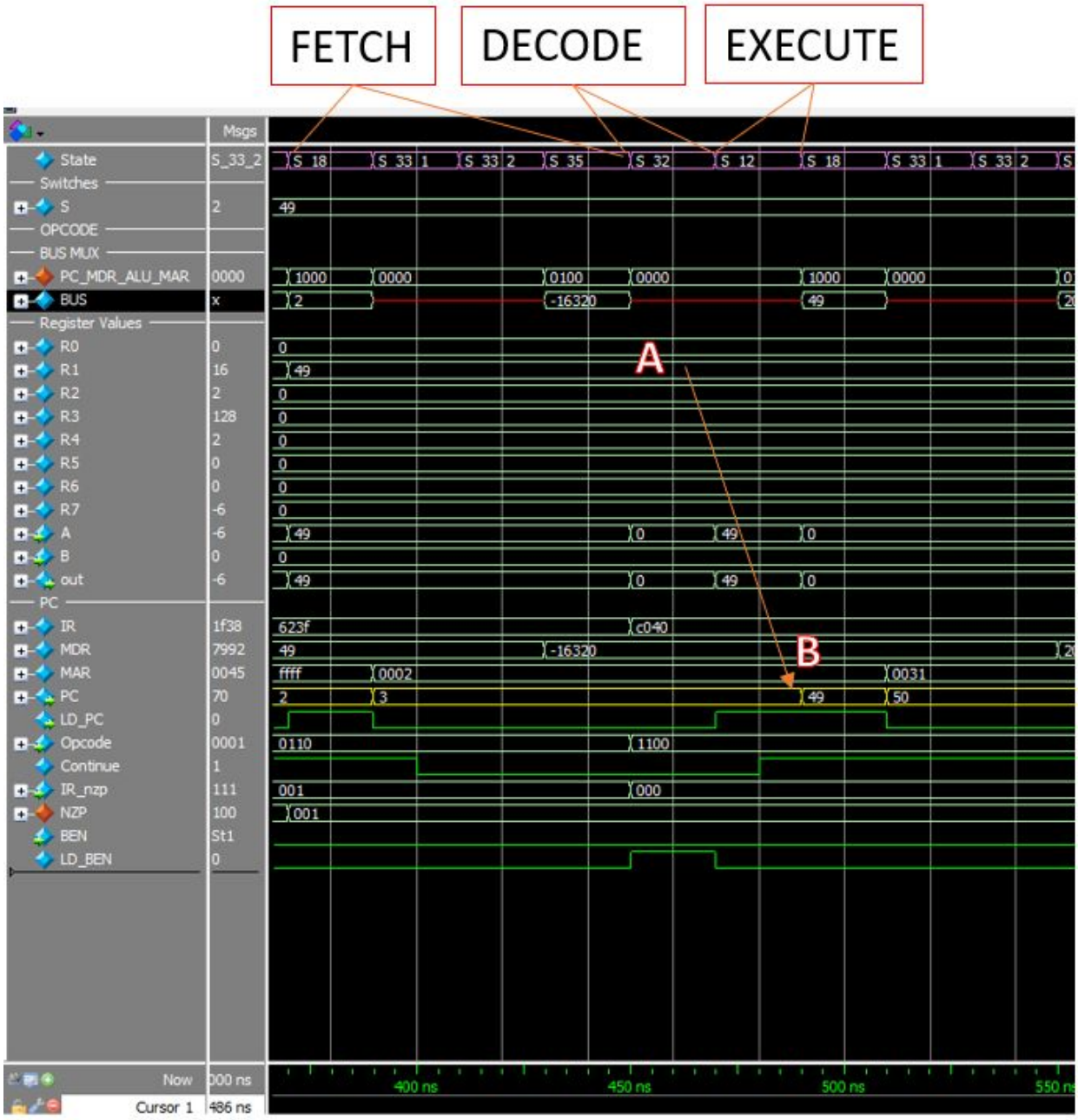
At A we see that 8 from R1 goes into port B of the ALU (from SR2MUX output). At B we see that 8 from R1 goes into port A of ALU (from SR1 OUT). At C, we see that 16 is stored back into R1

opLDR(R1, R0, inSW)



At A, we see that xffff in the bus indicates we will be reading from switches. At B, The value from the switches is loaded into MDR. At C this goes into the ALU (ADD), so 49 + 0 (R0 value) = 49, and at D the output goes into R1

opJMP(R1)



At A we see the value at R1 is 49. When the LD_PC signal goes high right before we see B, PC gets updated the next clock cycle to 49.

6.4 Post Lab Questions

6.4.1 Question A

a. Fill out the Design Resources and Statistics table from Post-Lab question one

6.4.2 Question B

b. Answer the post lab questions

6.4.3 Design Resources and Statistics

LUT	504
DSP	0
Memory (BRAM)	0

Flip-Flop	269
Frequency	96.1 MHz
Static Power	98.63 mW
Dynamic Power	0.00mW
Total Power	172.28 mW

Document any problems you encountered and your solutions to them, and a short conclusion.

The main issue that we ran into in week 1 was that using the QSF file given to us on Lab 6. Under the lab description, we were told that we can use the QSF file provided to simply the pin assignment process and have to avoid manually typing in pin assignment. When we imported the QSF file it imported a bunch of unnecessary pin assignments (VGA pins, extra clocks, etc) which had conflicting pin locations configured as the different variable names used in the lab. There is no easy or quick way to bulk-delete pin assignments so we had to make a new project, import the top level file, have it generate the qsf file with the variable names used in our top level file, and manually input everything

As for week 2, we ran into a couple of issues in the case of forgetting small things. For example, we were not using the BEN bit to decide whether during a BR execution, we were always branching no matter what. Even though the first four tests used BR and passed, this was because they used BRnzp which is unconditional just like our old code.

We also had a few cases where we forgot to set a LD control signal in our state machine so weren't loading values into register when we should have.

We had some issues programming SRAM since we did not correctly use the SD card method at first. We flashed the SD card .sof but immediately after we flashed the Lab6.sof file. For that reason we were getting garbage readings in our ram. We realized that after we flash the SD card .sof file we weren't actually navigating with the FPGA buttons to load flash the memory with the .sof file we flashed to the FPGA

6.4.4 Other Lab Report questions

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

Mem2IO is used as a tool for the FPGA to interface with the SDRAM and the switches. If the address given is xFFFF it will read from the switches, else it will read from SDRAM. In the case that we do choose to read or write from SDRAM, Mem2IO interfaces with tristate buffers so that only the CPU to SRAM part of the circuit is connected or the SRAM to CPU part of the circuit is connected. Since the SRAM is bidirectional, we read and write to the same pins and thus need to set one of the portions to high impedance.

• What is the difference between BR and JMP instructions?

Br is conditional based on the NZP bits of the instruction register with max PCoffset size of 9 bits, jmp is unconditional with a max PCoffset

size of 11 bits. JMP is essentially BRnzp with 2 more bits space for PCoffset.

• 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?

In the actual LC3, the memory it interfaces with has an “R” or ready signal. Whenever the LC3 reads or writes from memory, it waits until it gets the indication from the memory before doing so.

The SRAM we are working with does not have a ready signal, so we have to compensate it by adding extra states where we don’t do anything and just wait- we delay reading or writing so that even in worst case scenarios we will always have valid read/writes.

In the LC3, the “R” signal is generated from the memory is asynchronous- it is generated separate from the FSM and thus is not synchronised to the rising edge of the clock. In the actual LC3 this needs to be accounted for- that the “R” signal can change within a current state and lead to unintended behaviour.

Since in the SLC-3 we are not relying on an asynchronous signal for determining when to read/write, we do not have to deal with the synchronization issues that occur when dealing with asynchronous signals

6.5 Conclusion

- a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.

  Our design was fully functional, it worked both in simulations and on the FPGA board. We got full demo points in lab and passed the XOR test and SORT test which are all inclusive for determining whether all our operations worked. While we were able to get everything working, we were not able to get it working with a synchronizer for write enable which would have been better practice.

- b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

  The biggest issue was the provided qsf file and the claims made about it. There is no reason it should be given to us for lab 6, saying that we can use it to "avoid manually inputting pins" as that is extremely misleading. It took at least an hour to figure out why we were having issues with something that seems that should have

worked right off the bat. We finally realized that this qsf file was intended for the final project with a bunch of unnecessary or conflicting pin assignments for Lab 6, and would require us to delete and reassign conflicting assignments and input our own pin locations. This completely kills the purpose of giving us a file with the claim that it we can avoid manually importing pins. The worst part was that it would be faster to just start from the beginning and manually import the assignments in the table we were given, but importing the assignments overwrote our original qsf file. In order to restore the generated qsf file with only our top level variables, we had to make a new project with the same .sv files, generate a qsf file, then go back to the original project and import that qsf file (in addition to manually inputting all the pin assignments form the table).

There is simply no practical way to use it for Lab 6 and I have no idea why were giving the file, especially without any such disclaimer. I know that I am not the one that ran into this issue, I talked to someone else in my lab section and said it took him 4 hours just troubleshooting why nothing was working after importing the qsf file.