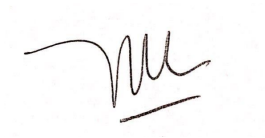


# Project B

## Programmable Processor

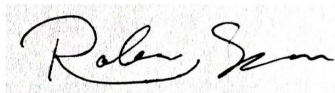
TCES 330 Digital Systems Design Spring 2020

**Authors:**



TRUNG DO

---



ROHAN SEAM

---

**Submission Date:** June 4, 2020

## **Table of Contents**

### **Project B**

1.	Introduction.....	3
a.	Objective	
b.	Requirements	
c.	Group members	
d.	Workload distribution	
2.	Design.....	4
a.	Controller.....	4
b.	Datapath.....	6
c.	Processor.....	7
d.	Top level module.....	7
3.	Test Procedures/Results.....	7
a.	PC module.....	8
b.	IR module.....	9
c.	Instruction memory testbench.....	9
d.	FSM module.....	10
e.	Controller module.....	12
f.	ALU module.....	13
g.	MUX 2 to 1 module.....	14
h.	Data memory testbench.....	15
i.	Register file module.....	16
j.	Datapath module.....	17
k.	Processor module.....	18
l.	ButtonSync module.....	19
m.	KeyFilter module.....	19
n.	MUX 8 to 1 module.....	19
o.	Top level module RTL viewer.....	20
p.	DE2-115 board.....	21
4.	Observations.....	22
5.	Conclusions.....	22

## PROJECT B: 6 Instruction Processor

### 1. INTRODUCTION

**a. Objective:** The purpose of this project is to learn how to write System Verilog code and test our created circuits using Modelsim, through designing and implementing the six-instruction programmable processor on the DE2-115 board.

**b. Requirements:** We were required to create multiple submodules for this processor that involved significant testing. Each module required a corresponding testbench that tested all possibilities of inputs. After this, we abstracted our modules and used them as part of the Control Unit (Controller) and Datapath. We have to ensure the functionality of the processor then test it on the DE2-115 board using the HEX displays, LEDs, and switches. Verifying the functionality on the board was a challenging and time consuming task.

**c. Group members:** Rohan Seam and Trung Do

**d. Workload distribution:**

- Quartus Controller module and its sub modules: Trung
- Quartus Datapath module and its sub modules: Rohan
- Quartus Processor module, Mux 8 to 1 and Key filter: Trung
- Quartus top level module, Button sync: Rohan
- DE2-115 Board testing: Rohan
- Project report: Trung (Introduction, design, observations), Rohan (test procedures/results, conclusions)

The work load divided equally among both members.

### 2. DESIGN

**a. Controller**

To make the controller unit, we created four submodules: InstMemory.v, PC.sv, IR.sv, FSM.sv. The InstMemory.v is associated with a mif file that holds our instructions. Figure 1 explains how we translated the given instructions into hex and decimal numbers.

Next is the PC.sv module, it is simply a counter, its count is the address to access the instruction memory. The address produced is based on the required inputs: Clk, Clr and Up. When Clr = 1 it will reset and restart from 0. When Up = 1, the PC value is incremented by 1 at each positive edge of the clock cycle. The output of the InstMemory is the 16-bit wide instruction stored at the address given from the PC and

to be sent to the Instruction Register (IR). We implemented IR with a group of flip-flops which will latch out the input signal. IR will just store the instruction until it receives a command to send the instruction out.

The most complicated module has to be the FSM.sv. There are a total of 10 states in the FSM. We denoted each state using a 4-bit binary number. The next state and output for each state is set using the state diagram given from the Processor2020cont.pdf (Figure 3). At the state “Fetch”, the instruction is to be fetched from IR. At the state “Decode”, the next state is based on the bit [15:12] of the instruction received from IR. These 4 bits represented the opcode. Then after the instruction is executed, it will go back to state “Fetch” and fetch in the new instruction until the instruction with opcode Halt is fetched in. From the state diagram we see that state “HALT” will not go to any other state but itself, unless the state machine is reinitialized. So basically no other steps can be done after HALT unless we return to the Init stage as shown in Figure 3.

```

LOAD D[B] --> RF[1]
LOAD D[1B] --> RF[2]
LOAD D[06] --> RF[3]
LOAD D[8A] --> RF[4]

SUB RF[1] - RF[2] --> RF[5]
ADD RF[5] + RF[3] --> RF[6]
SUB RF[6] - RF[4] --> RF[0]

STORE D[CD] = RF[0]
HALT

0010 00001011 0001 = 0x20B1 = 8369
0010 00011011 0010 = 0x21B2 = 8626
0010 00000110 0011 = 0x2063 = 8291
0010 10001010 0100 = 0x28A4 = 10404

0100 0001 0010 0101 = 0x4125 = 16677
0011 0101 0011 0110 = 0x3536 = 13622
0100 0110 0100 0000 = 0x4640 = 17984

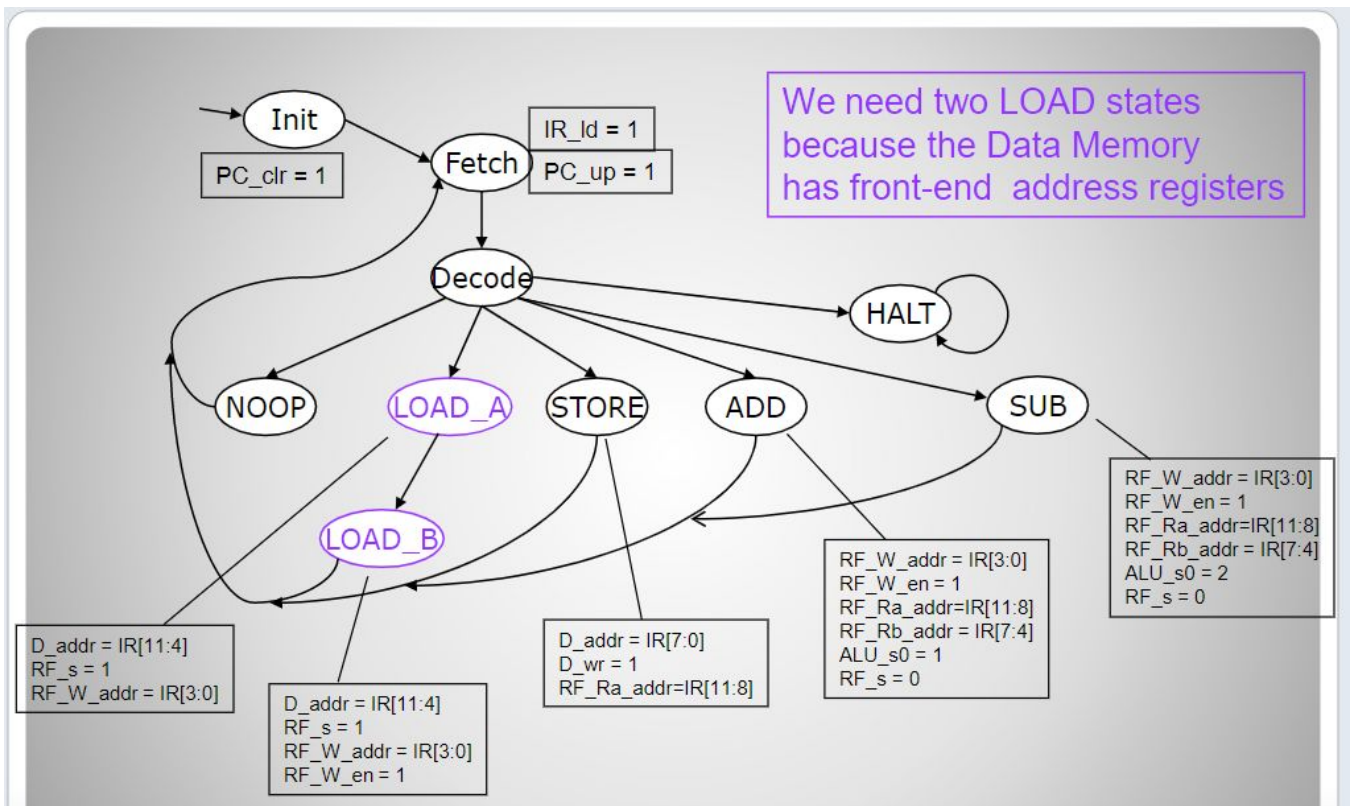
0001 0000 1100 1101 = 0x10CD = 4301
0101 0000 0000 0000 = 0x5000 = 20480

```

**Figure 1.** Changing instruction into decimal numbers

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	8369	8626	8291	10404	16677	13622	17984	4301
8	20480	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0	0

**Figure 2.** Instruction memory (A.mif)



**Figure 3.** State transition diagram for FSM module

As mentioned before, the controller module instantiates the sub-modules above. The inputs of the PC will be sent from the FSM, the FSM will decide whether to let the PC count up or not, as well as resetting the PC. The input of InstMemory will be sent from PC, the output of

PC is the address to access the instructions. The input of IR will be sent from the InstMemory, IR will store this instruction, and wait for the load command from the FSM to send out the instruction. The input of the FSM will be sent from IR. The opcode of the instruction will decide the next state for the “decode” state of the FSM.

## b. Datapath

The Datapath module is made up of 4 sub modules, they are DataMemory.v, ALU.sv, mux\_16w\_2\_to\_1.sv and Register.sv. The DataMemory is associated with the D.mif file which store the following data: D[6] = 0x10AC, D[B] = 0xCC05, D[1B] = 0x01B5, D[8A] = 0xA040. D[6] represents address 6 in the data memory, while D[B] represents address 11 in data memory, etc. Figure 4 shows how we inserted our data (D[8A] is not included in the screenshot).

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0000	0000	0000	0000	0000	0000	10AC	0000
8	0000	0000	0000	CC05	0000	0000	0000	0000
16	0000	0000	0000	0000	0000	0000	0000	0000
24	0000	0000	0000	01B5	0000	0000	0000	0000
32	0000	0000	0000	0000	0000	0000	0000	0000
40	0000	0000	0000	0000	0000	0000	0000	0000
48	0000	0000	0000	0000	0000	0000	0000	0000
56	0000	0000	0000	0000	0000	0000	0000	0000
64	0000	0000	0000	0000	0000	0000	0000	0000
72	0000	0000	0000	0000	0000	0000	0000	0000

**Figure 4.** Data memory (D.mif)

In the ALU module, we implemented eight different arithmetic functions. The inputs are A, B and s0. A and B are 16 bits binary number and s0 is the select signal that will select the function to operate upon A and B. The output of the ALU module is of course the results obtained from the operation.

Next, we created the 16-bits Mux 2 to 1 module, this module takes in 2 input values (each 16-bits) and selects one of them to send out based on a select signal RF\_s. The input values include one from the data memory and one from the result of the ALU operation. When RF\_s = 1 the output data will be from the data memory and when RF\_s = 0, the output data will be the result from the ALU operation.

Finally is the datapath module. After we finish the four sub modules, we instantiate them in the datapath module. In the datapath module, the outputs of the data memory (data) and the ALU (result from ALU operation) are the inputs of the MUX 2 to 1. The output of the MUX 2 to 1

1 are the inputs to the Register File (data selected from the MUX 2 to 1). Then the two outputs of the Register File (data of A and B) go in as inputs of the ALU.

### **c. Processor**

Once we have the Controller and Datapath modules done, we made the Processor module that instantiated both of them. The outputs of the Controller went in as inputs of the Datapath. The controller will give the address to access the data memory, select the data to write in the register, give address to store data in the register and also give address to read data for A and B, and finally select the arithmetic functions for A and B.

### **d. Top Level Module**

The top level module (ProjectB.sv) instantiated the Processor module as well as numerous other modules for display and debugging purposes. The ButtonSync and KeyFilter modules were included to help stabilize the button press from the KEY input as our clock. The ButtonSync and KeyFilter modules both operated off of the 50MHz clock on the DE2-115 board while the Processor operated off of the KEY input. The output of the processor was displayed on the 8 HEX displays on the board based on the orientation of the switches SW[17:15]. HEX displays 3, 2, 1, and 0 always display the contents of the IR, while HEX displays 6, 5, and 4 varied based on SW[17:15]. If the switches were in the orientation of 000, HEX 7 and 6 would display PC, while HEX 5 and 4 would display the current state of the state machine. If the switch orientation was 001, HEX 6, 5, and 4 would display the 1st input of the ALU. If the orientation was 010, it would display the 2nd input of the ALU, but if the orientation was 011, it would display the output of the ALU. Finally, if the orientation was 100, it would display the Next State of the state machine.

The red LEDs on the DE2-115 board displayed the orientation of the switches while the green LEDs illustrated the state of the KEY inputs. KEY[2] acted as a clock signal to our processor module that allowed the processor to operate. On the other hand, KEY[1] acted as a synchronous system reset that would reset the state machine.

## **3. TEST PROCEDURES/RESULTS**

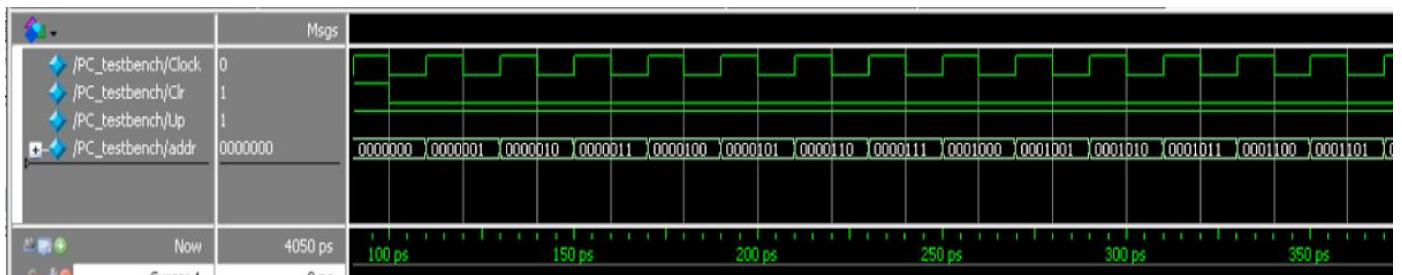
When we completed our design, we were able to test its operation in ModelSim through the testbench modules we created. This played a significant role in the functionality of the processor as a whole, as each upper module depended on the lower ones. Because we carefully tested each module before moving to the upper level ones, it became much easier to debug the upper modules. The importance of the testbench modules was illustrated deeply in this project.

### a. PC module

Time	Clr	Up	addr				
0	1	1	0	3290	0	1	100
100	0	1	0	3310	0	1	101
110	0	1	1	3330	0	1	102
130	0	1	2	3350	0	1	103
150	0	1	3	3370	0	1	104
170	0	1	4	3390	0	1	105
190	0	1	5	3410	0	1	106
210	0	1	6	3430	0	1	107
230	0	1	7	3450	0	1	108
250	0	1	8	3470	0	1	109
270	0	1	9	3490	0	1	110
290	0	1	10	3510	0	1	111
310	0	1	11	3530	0	1	112
330	0	1	12	3550	0	1	113
350	0	1	13	3570	0	1	114
370	0	1	14	3590	0	1	115
390	0	1	15	3610	0	1	116
410	0	1	16	3630	0	1	117
430	0	1	17	3650	0	1	118
450	0	1	18	3670	0	1	119
470	0	1	19	3690	0	1	120
490	0	1	20	3710	0	1	121
510	0	1	21	3730	0	1	122
530	0	1	22	3750	0	1	123
550	0	1	23	3770	0	1	124
570	0	1	24	3790	0	1	125
590	0	1	25	3810	0	1	126
600	1	0	25	3830	0	1	127
610	1	0	0	3850	0	1	0
800	0	0	0	3870	0	1	1
1300	0	1	0	3890	0	1	2
1310	0	1	1	3910	0	1	3
1330	0	1	2	3930	0	1	4
				3950	0	1	5

**Figure 5.** Modelsim testing transcript of PC module

Looking at Figure 5 we see that when the input Clr=0 and Up=1, the PC is incrementing 1 at each clock cycle, it can count up to the maximum of 127 and start again from 0. When Clr=1 it will reset to 0 at the next clock cycle and counting up again from 0. When Up=0, the PC holds its current value. The results agreed with our expectations.



**Figure 4.** Waves of PC module in Modelsim

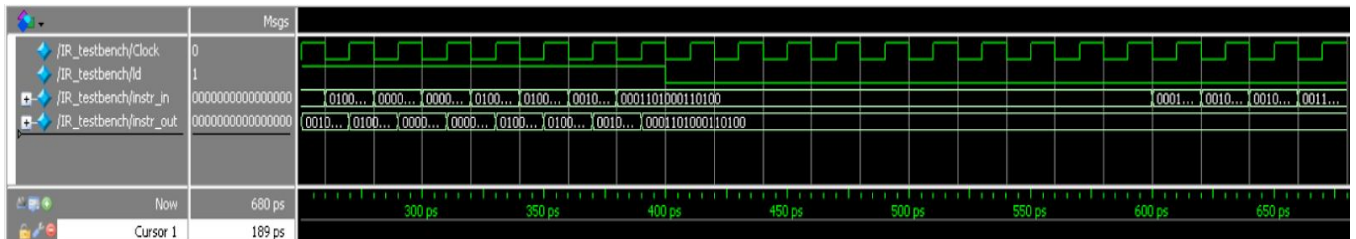


## b. IR module

Time	ld	instr_in	instr_out
0	1	0	0
200	1	1215	0
210	1	1215	1215
220	1	17116	1215
230	1	17116	17116
240	1	12180	17116
250	1	12180	12180
260	1	19318	12180
270	1	19318	19318
280	1	3588	19318
290	1	3588	3588
300	1	1420	3588
310	1	1420	1420
320	1	17926	1420
330	1	17926	17926
340	1	16419	17926
350	1	16419	16419
360	1	10822	16419
370	1	10822	10822
380	1	6708	10822
390	1	6708	6708
400	0	6708	6708
600	0	8107	6708
620	0	9873	6708
640	0	9011	6708
660	0	13460	6708

**Figure 6.** Modelsim testing transcript of IR module

The transcript shows that when the input load is on, the input instruction will be sent out at the next clock cycle and hold the current instruction if load is off. IR worked correctly.



**Figure 7.** Waves of IR module in Modelsim

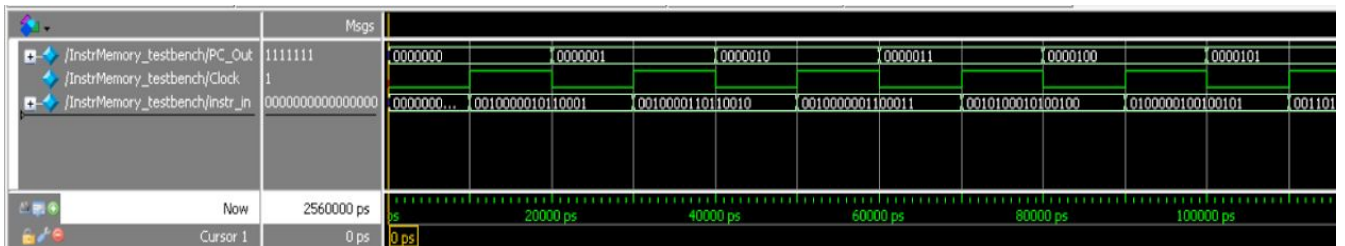
## c. Instruction memory testbench

	8369	8626	8291	10404	16677	13622	17984	4301	20480
0									
14	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0
84	0	0	0	0	0	0	0	0	0
98	0	0	0	0	0	0	0	0	0
112	0	0	0	0	0	0	0	0	0
126	0	0							

**Figure 8.** Instruction memory from ModelSim

Time	Address	Instruction
0	0	0
10	0	8369
20	1	8369
30	1	8626
40	2	8626
50	2	8291
60	3	8291
70	3	10404
80	4	10404
90	4	16677
100	5	16677
110	5	13622
120	6	13622
130	6	17984
140	7	17984
150	7	4301
160	8	4301
170	8	20480
180	9	20480
190	9	0
200	10	0
220	11	0
240	12	0
260	13	0
280	14	0
300	15	0
320	16	0
340	17	0
360	18	0

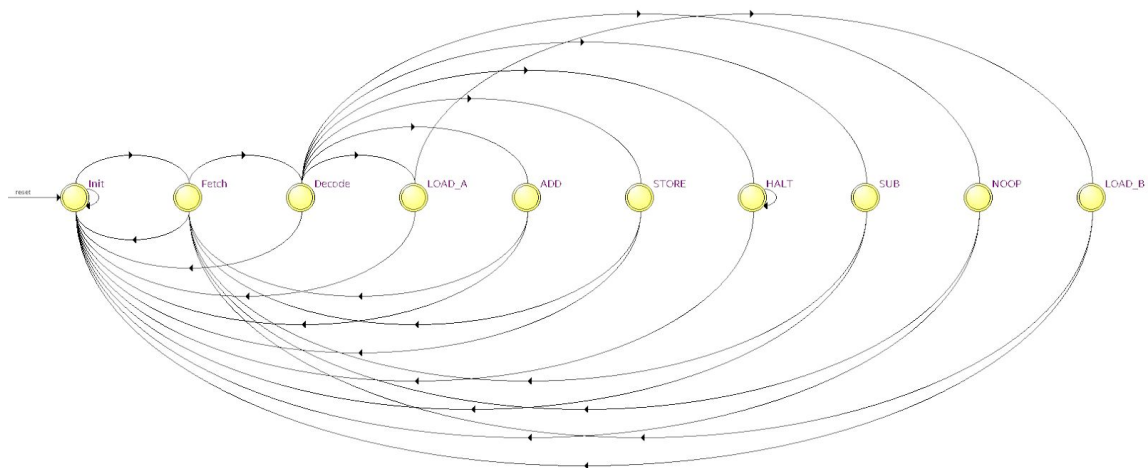
**Figure 9.** Modelsim testing transcript of Instruction memory testbench module



**Figure 10.** Waves of InstMemory testbench module in Modelsim

Comparing the results in the transcript and the instruction memory we see that it is reading correctly the content in each address.

#### d. FSM module



**Figure 11.** State transition diagram of FSM in Quartus

For the testbench of the FSM module, we let reset=1 (active low) and assigned all possible combinations for the opcode of the instruction (IR[15:12]) and set specific value for the rest of the bits of the instruction (IR[11:8] = 4, IR[7:4]=4, IR[3:0]=4) to see if the state transition operated correctly and produced the expected outputs corresponding to each state. Using the “assert” statement we were able to confirm the results.

Time	reset	IR	OutState	PC_clr	PC_up	IR_id	D_addr	D_wr	RF_s	RF_W_addr	RF_W_en	RF_Ra_addr	RF_Rb_addr	ALU_s0	NextState
0	1	0444	0	1	0	0	0	0	0	0	0	0	0	0	1
10	1	1444	1	0	1	1	0	0	0	0	0	0	0	0	2
20	1	2444	1	0	1	1	0	0	0	0	0	0	0	0	2
30	1	2444	2	0	0	0	0	0	0	0	0	0	0	0	4
50	1	2444	4	0	0	0	68	0	1	4	0	0	0	0	5
55	1	3444	4	0	0	0	68	0	1	4	0	0	0	0	5
70	1	3444	5	0	0	0	68	0	1	4	1	0	0	0	1
90	1	3444	1	0	1	1	0	0	0	0	0	0	0	0	2
110	1	3444	2	0	0	0	0	0	0	0	0	0	0	0	7
130	1	3444	7	0	0	0	0	0	0	4	1	4	4	1	1
135	1	4444	7	0	0	0	0	0	0	4	1	4	4	1	1
150	1	4444	1	0	1	1	0	0	0	0	0	0	0	0	2
170	1	4444	2	0	0	0	0	0	0	0	0	0	0	0	8
190	1	4444	8	0	0	0	0	0	0	4	1	4	4	2	1
195	1	5444	8	0	0	0	0	0	0	4	1	4	4	2	1
210	1	5444	1	0	1	1	0	0	0	0	0	0	0	0	2
230	1	6444	2	0	0	0	0	0	0	0	0	0	0	0	3
240	1	7444	2	0	0	0	0	0	0	0	0	0	0	0	3
250	1	8444	3	0	0	0	0	0	0	0	0	0	0	0	1
270	1	8444	1	0	1	1	0	0	0	0	0	0	0	0	2
290	1	9444	2	0	0	0	0	0	0	0	0	0	0	0	3
300	1	a444	2	0	0	0	0	0	0	0	0	0	0	0	3
310	1	b444	3	0	0	0	0	0	0	0	0	0	0	0	1
330	1	b444	1	0	1	1	0	0	0	0	0	0	0	0	2
350	1	c444	2	0	0	0	0	0	0	0	0	0	0	0	3
360	1	d444	2	0	0	0	0	0	0	0	0	0	0	0	3
370	1	e444	3	0	0	0	0	0	0	0	0	0	0	0	1
390	1	e444	1	0	1	1	0	0	0	0	0	0	0	0	2
410	1	f444	2	0	0	0	0	0	0	0	0	0	0	0	3
420	0	0444	2	0	0	0	0	0	0	0	0	0	0	0	3

Figure 12. Testing transcript of FSM module in Modelsim

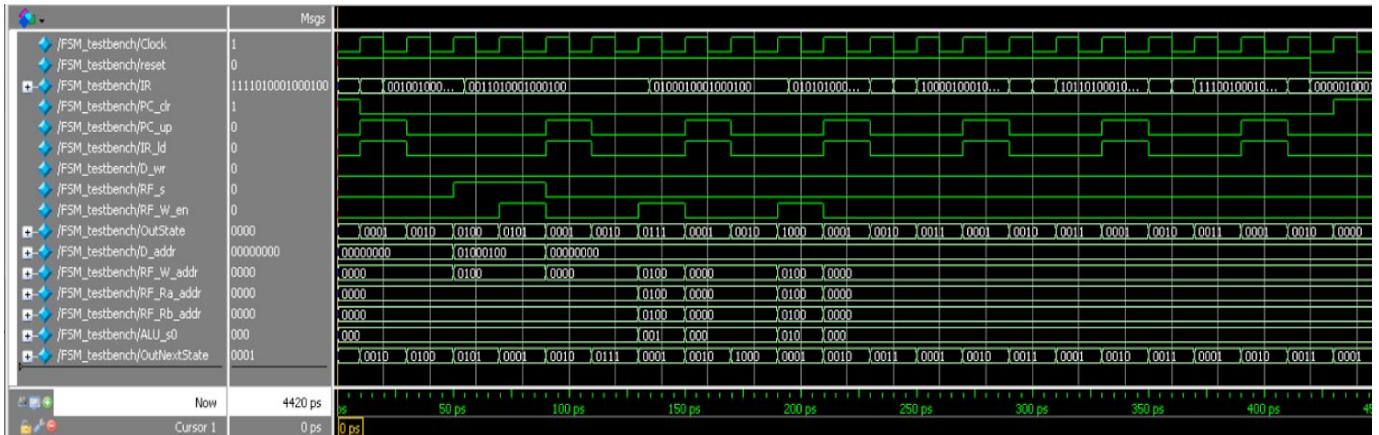


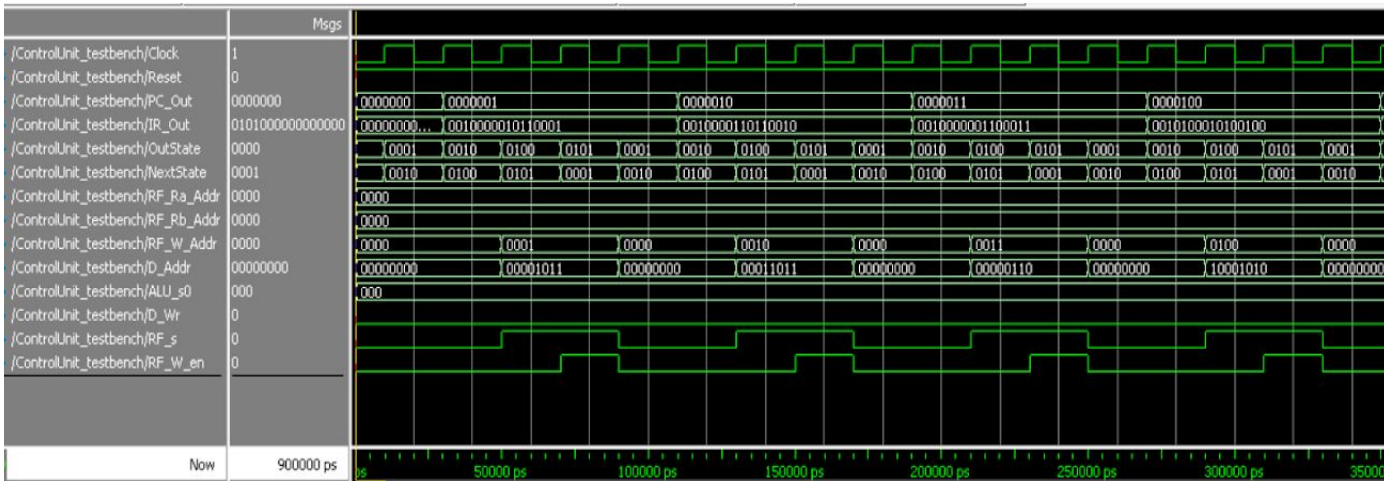
Figure 13. Waves of FSM module in Modelsim

### e. Controller module

To test the controller's operation, we set the PC's value to go from 0 to 9 to access all the instructions we stored in our instruction memory, and check if the IR\_Out will send out the correct instruction it received from the Instruction memory. And based on the instruction, we can also check the state transitions happening in the FSM and the corresponding outputs of each state.

Time	Reset	PC_Out	IR_Out	OutState	NextState	D_addr	D_wr	RF_s	RF_W_en	RF_Ra_addr	RF_Rb_addr	RF_W_addr	ALU_s0
0	1	0	0	0	1	0	0	0	0	0	0	0	0
10	1	0	0	1	2	0	0	0	0	0	0	0	0
30	1	1	8369	2	4	0	0	0	0	0	0	0	0
50	1	1	8369	4	5	11	0	1	0	0	0	1	0
70	1	1	8369	5	1	11	0	1	1	0	0	1	0
90	1	1	8369	1	2	0	0	0	0	0	0	0	0
110	1	2	8626	2	4	0	0	0	0	0	0	0	0
130	1	2	8626	4	5	27	0	1	0	0	0	2	0
150	1	2	8626	5	1	27	0	1	1	0	0	2	0
170	1	2	8626	1	2	0	0	0	0	0	0	0	0
190	1	3	8291	2	4	0	0	0	0	0	0	0	0
210	1	3	8291	4	5	6	0	1	0	0	0	3	0
230	1	3	8291	5	1	6	0	1	1	0	0	3	0
250	1	3	8291	1	2	0	0	0	0	0	0	0	0
270	1	4	10404	2	4	0	0	0	0	0	0	0	0
290	1	4	10404	4	5	138	0	1	0	0	0	4	0
310	1	4	10404	5	1	138	0	1	1	0	0	4	0
330	1	4	10404	1	2	0	0	0	0	0	0	0	0
350	1	5	16677	2	8	0	0	0	0	0	0	0	0
370	1	5	16677	8	1	0	0	0	1	1	2	5	2
390	1	5	16677	1	2	0	0	0	0	0	0	0	0
410	1	6	13622	2	7	0	0	0	0	0	0	0	0
430	1	6	13622	7	1	0	0	0	1	5	3	6	1
450	1	6	13622	1	2	0	0	0	0	0	0	0	0
470	1	7	17984	2	8	0	0	0	0	0	0	0	0
490	1	7	17984	8	1	0	0	0	1	6	4	0	2
510	1	7	17984	1	2	0	0	0	0	0	0	0	0
530	1	8	4301	2	6	0	0	0	0	0	0	0	0
550	1	8	4301	6	1	205	1	0	0	0	0	0	0
570	1	8	4301	1	2	0	0	0	0	0	0	0	0
590	1	9	20480	2	9	0	0	0	0	0	0	0	0
610	1	9	20480	9	9	0	0	0	0	0	0	0	0
800	0	9	20480	9	9	0	0	0	0	0	0	0	0
810	0	9	20480	0	1	0	0	0	0	0	0	0	0
830	0	0	20480	0	1	0	0	0	0	0	0	0	0

**Figure 14.** Testing transcript of the Controller module in Modelsim



**Figure 15.** Wave of Control Unit from ModelSim



f. ALU module

Time	Select	A	B	Q
0	0	0	0	0
40	0	1093	0	0
45	0	1093	7440	0
50	0	8877	7440	0
55	0	8877	2464	0
60	0	8273	2464	0
65	0	8273	7652	0
70	1	8273	7652	15925
110	1	6377	7652	14029
115	1	6377	2849	9226
120	1	6241	2849	9090
125	1	6241	5223	11464
130	1	1797	5223	7020
135	1	1797	882	2679
140	2	1797	882	915
180	2	2982	882	2100
185	2	2982	6880	61638
190	2	6110	6880	64766
195	2	6110	9768	61878
200	2	6508	9768	62276
205	2	6508	5896	612
210	3	6508	5896	6508
250	3	1992	5896	1992
255	3	1992	4509	1992
260	3	3294	4509	3294
265	3	3294	4026	3294
270	3	6887	4026	6887
275	3	6887	5134	6887
280	4	6887	5134	3817
320	4	4864	5134	1806
325	4	4864	5502	1662
330	4	8085	5502	2795
335	4	8085	4180	4033
340	4	3514	4180	7662
345	4	3514	3882	656
350	5	3514	3882	4026

Figure 16. Testing transcript of the ALU module in Modelsim

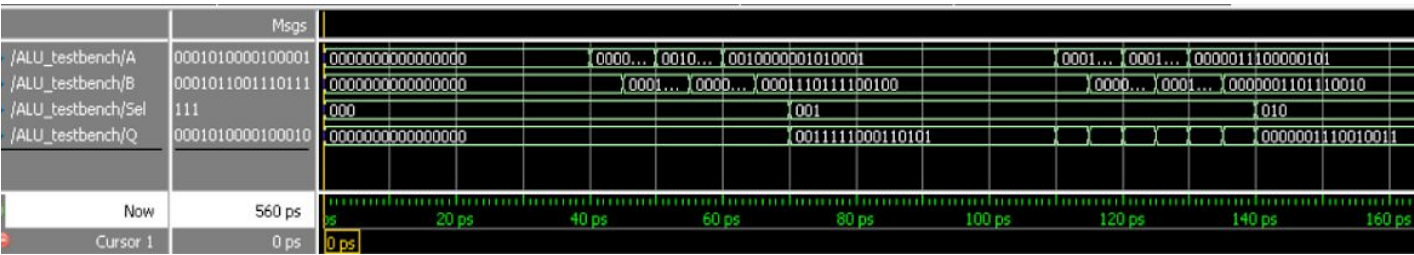
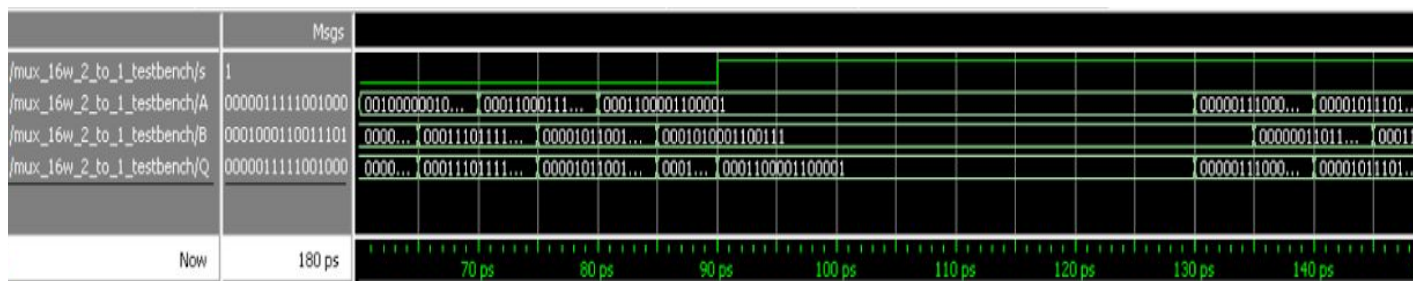


Figure 17. Waves of ALU module in Modelsim

### g. MUX 2 to 1 module

A	B	s	Q
0	0	0	0
1093	0	0	0
1093	7440	0	7440
8877	7440	0	7440
8877	2464	0	2464
8273	2464	0	2464
8273	7652	0	7652
6377	7652	0	7652
6377	2849	0	2849
6241	2849	0	2849
6241	5223	0	5223
6241	5223	1	6241
1797	5223	1	1797
1797	882	1	1797
2982	882	1	2982
2982	6880	1	2982
6110	6880	1	6110
6110	9768	1	6110
6508	9768	1	6508
6508	5896	1	6508
1992	5896	1	1992
1992	4509	1	1992

**Figure 18.** Testing transcript of the MUX 2 to 1 module in ModelSim



**Figure 19.** Waves of the MUX 2 to 1 module in Modelsim

Looking at both transcript and waves, we see that given values for A and B, select signals will decide which value to be produced as the output. The module operated as expected.

## h. Data memory testbench

0	002a	0000	0000	0000	0000	0000	0001	0000	0000	0000	0000	cc05
19	0000	0000	0000	0000	0000	0000	0000	0000	01b5	0000	0000	0000
38	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
57	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
76	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
95	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
114	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
133	0000	0000	0000	0000	0000	a040	0000	0000	0000	0000	0000	0000
152	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
171	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
190	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

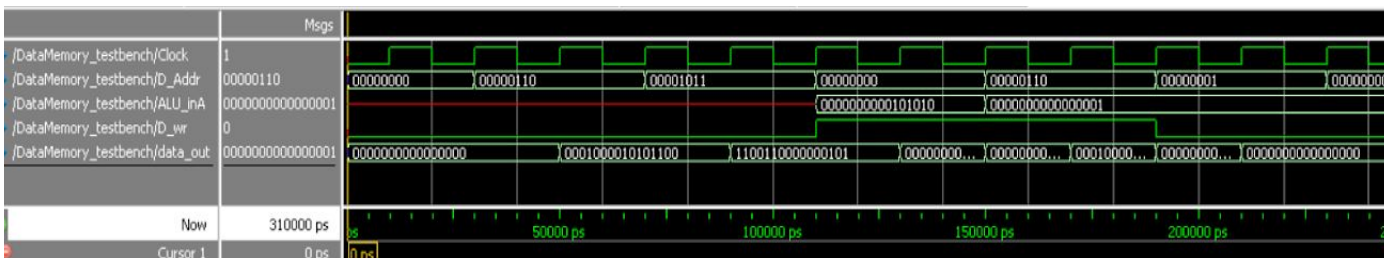
**Figure 20.** Data Memory from ModelSim

```

Reading Ram
  30: Address =  0, Data Read = 0000
  70: Address =  6, Data Read = 10ac
 110: Address = 11, Data Read = cc05
Writing Ram
 150: Address =  0, Data Read = 0000
 190: Address =  6, Data Read = 10ac
Reading Ram
 230: Address =  1, Data Read = 0000
 270: Address =  0, Data Read = 002a
 310: Address =  6, Data Read = 0001

```

**Figure 21.** Testing transcript of the Data memory in Modelsim



**Figure 22.** Waves of the Data Memory testbench module in ModelSim

One observation from the testbench of the Data Memory was that the ALU\_inA value was initially red. This makes sense as there is no value for the ALU at that current time. There is only a value ready for the ALU after the registers are populated with the correct information for the ALU to use.

## i. Register file module

In order to test this module, we randomly assigned values to each of the 16 registers in the register file. We used the assert statement to ensure that the contents of the register file matched the data entered. We then tested reading the register file and ensured that the value read by the addresses given by Ra\_addr and Rb\_addr was the same as the value we wrote to that specific register. We also verified that the write enable worked correctly, as write=1, it allows us to write new data in the register, otherwise it will just hold the current value and won't let it write in the register.

Time	write	W_addr	W_data	Ra_addr	Ra_data	Rb_addr	Rb_data
0	1	0	1215	0	0	0	0
10	1	0	1215	0	1215	0	1215
405	1	1	17116	1	0	1	0
410	1	1	17116	1	17116	1	17116
810	1	2	12180	2	12180	2	12180
1215	1	3	19318	3	0	3	0
1230	1	3	19318	3	19318	3	19318
1620	1	4	3588	4	0	4	0
1630	1	4	3588	4	3588	4	3588
2025	1	5	1420	5	0	5	0
2030	1	5	1420	5	1420	5	1420
2430	1	6	17926	6	17926	6	17926
2835	1	7	16419	7	0	7	0
2850	1	7	16419	7	16419	7	16419
3240	1	8	10822	8	0	8	0
3250	1	8	10822	8	10822	8	10822
3645	1	9	6708	9	0	9	0
3650	1	9	6708	9	6708	9	6708
4050	1	10	8107	10	8107	10	8107
4455	1	11	9873	11	0	11	0
4470	1	11	9873	11	9873	11	9873
4860	1	12	9011	12	0	12	0
4870	1	12	9011	12	9011	12	9011
5265	1	13	13460	13	0	13	0
5270	1	13	13460	13	13460	13	13460
5670	1	14	17909	14	17909	14	17909
6075	1	15	16477	15	0	15	0
6090	1	15	16477	15	16477	15	16477
6480	0	15	16477	0	1215	8	10822
6885	0	15	16477	1	17116	9	6708
7290	0	15	16477	2	12180	10	8107
7695	0	15	16477	3	19318	11	9873
8100	0	15	16477	4	3588	12	9011
8505	0	15	16477	5	1420	13	13460
8910	0	15	16477	6	17926	14	17909
9315	0	15	16477	7	16419	15	16477

Figure 23. Testing transcript of the Register module in Modelsim

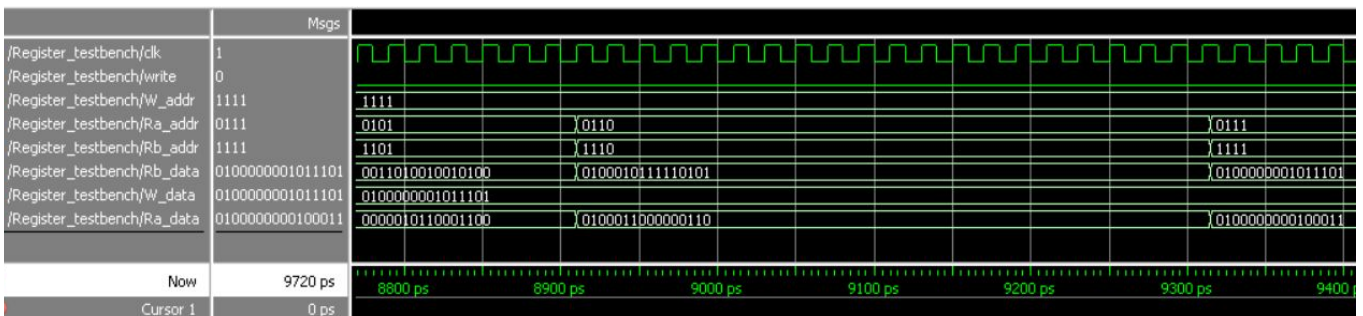


Figure 24. Waves of the Register module in Modelsim



## j. Datapath module

In the testbench of the datapath, we wanted to test the full operation of the data memory, register file, multiplexer, and ALU together. Since there are many inputs, we couldn't test all possible combinations of inputs, so we only tested a few cases that we set a specific value for each of the inputs and verified the expected outputs. For example, as we set the input  $ALU\_s0 = 1$ , we expect the output  $ALU\_Out = ALU\_inA + ALU\_inB$ . We tested a specific case of reading an input from memory address 6, and performed an ADD operation in the ALU and verified the correct output. We also verified the value of  $ALU\_inA$  was written back to the specified memory address. We did this by using the "assert" statement to verify that the  $ALU\_inA$  value was equal to the memory value.

Time,	D_Addr,	D_Wr,	RF_s,	RF_W_Addr,	RF_W_en,	RF_Ra_Addr,	RF_Rb_Addr,	ALU_s0,	ALU_inA,	ALU_inB,	ALU_out
0	6	1	1	1	1	1	1	1	0000	0000	0000
Memory[6] = 10ac											
30	6	0	1	1	1	1	1	1	10ac	10ac	2158
50	6	0	1	1	1	1	1	1	0000	0000	0000
70	6	1	0	1	1	1	1	1	0000	0000	0000
190	8	1	0	13	1	0	4	1	0000	0000	0000
210	6	1	0	12	1	14	5	1	0000	0000	0000
230	11	1	0	6	1	0	4	1	0000	0000	0000
250	0	1	0	2	1	6	0	1	0000	0000	0000
270	0	0	1	2	1	6	0	0	0000	0000	0000
390	15	0	1	6	1	4	0	0	0000	0000	0000
410	11	0	1	11	1	7	1	0	0000	0000	0000
430	14	0	1	6	1	8	1	0	0000	0000	0000
450	9	0	1	12	1	1	12	0	0000	0000	0000
470	9	0	1	12	1	1	12	2	0000	0000	0000
590	6	0	1	11	1	12	12	2	0000	0000	0000
610	6	0	1	13	1	0	9	2	0000	0000	0000
630	14	0	1	1	1	5	5	2	0000	0000	0000
650	7	0	1	14	1	12	1	2	0000	0000	0000

Figure 25. Testing transcript of the Datapath module in Modelsim

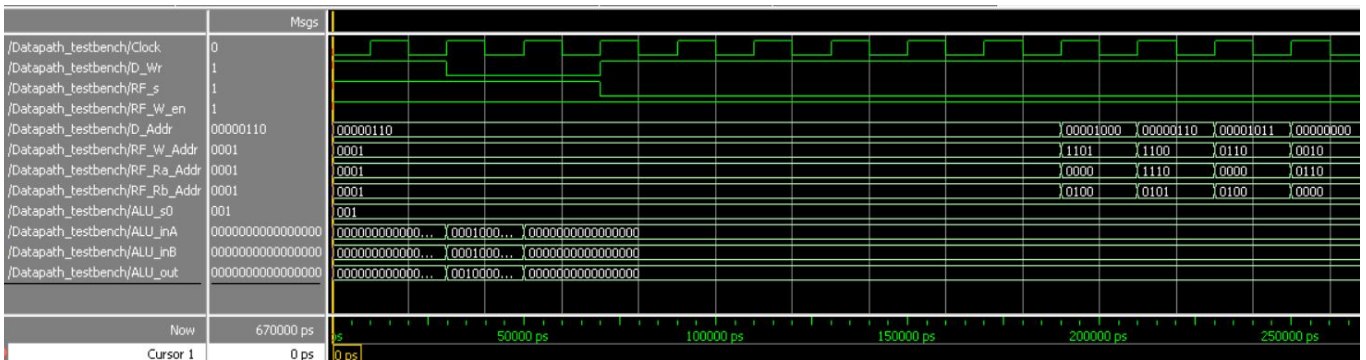
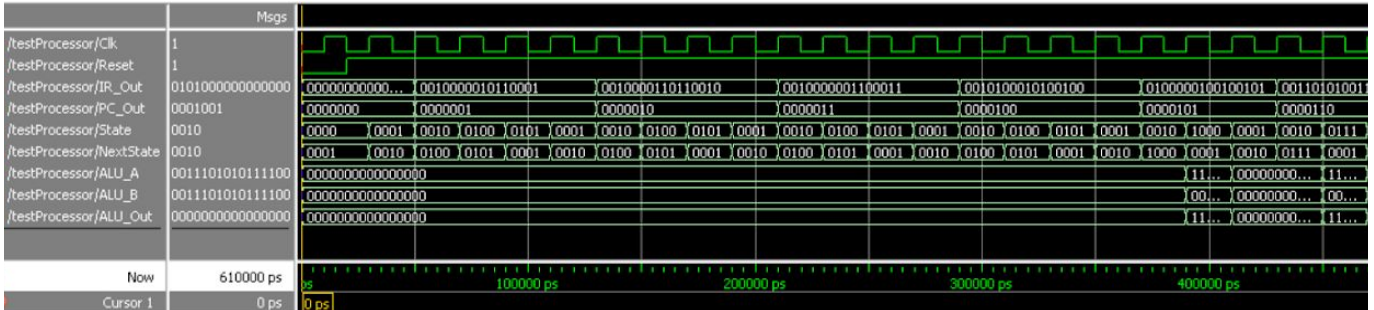


Figure 26. Waves of the Datapath module in ModelSim

## k. Processor module

To test the operation of the Processor, we used the testbench given by professor Jie Sheng. We observed the correct functionality of the reset (active low), the current states, and the ALU inputs and output. This matches what we observed during our testing on the DE2-115 board.



**Figure 27.** Waves for Processor module in Modelsim

```

Begin Simulation.
Time is 0 : Reset = 0 State = 0 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 20000 : Reset = 1 State = 0 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 30000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 50000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 70000 : Reset = 1 State = 4 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 90000 : Reset = 1 State = 5 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 110000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 130000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 150000 : Reset = 1 State = 4 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 170000 : Reset = 1 State = 5 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 190000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 210000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 230000 : Reset = 1 State = 4 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 250000 : Reset = 1 State = 5 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 270000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 290000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 310000 : Reset = 1 State = 4 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 330000 : Reset = 1 State = 5 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 350000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 370000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 390000 : Reset = 1 State = 8 ALU A = cc05 ALU B = 01b5 ALU Out = ca50
Time is 410000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 430000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 450000 : Reset = 1 State = 7 ALU A = ca50 ALU B = 10ac ALU Out = dafc
Time is 470000 : Reset = 1 State = 1 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 490000 : Reset = 1 State = 2 ALU A = 0000 ALU B = 0000 ALU Out = 0000
Time is 510000 : Reset = 1 State = 8 ALU A = dafc ALU B = a040 ALU Out = 3abc
Time is 530000 : Reset = 1 State = 1 ALU A = 3abc ALU B = 3abc ALU Out = 0000
Time is 550000 : Reset = 1 State = 2 ALU A = 3abc ALU B = 3abc ALU Out = 0000
Time is 570000 : Reset = 1 State = 6 ALU A = 3abc ALU B = 3abc ALU Out = 0000
Time is 590000 : Reset = 1 State = 1 ALU A = 3abc ALU B = 3abc ALU Out = 0000

End of Simulation.

```

**Figure 28.** Testing Transcript of the Processor module in ModelSim

I. ButtonSync module

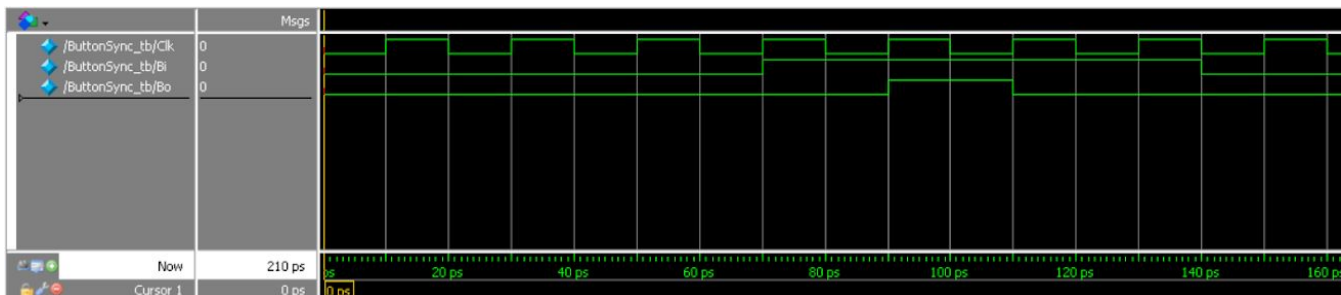


Figure 29. Waves for the Button Sync module in ModelSim

m. KeyFilter module

Time	In	Out	Strobe
0	0	x	x
10	0	0	1
40	1	0	1
50	1	1	1
70	1	0	0
90	0	0	0
150	1	0	0

Figure 30. Testing transcript of KeyFilter module in Modelsim

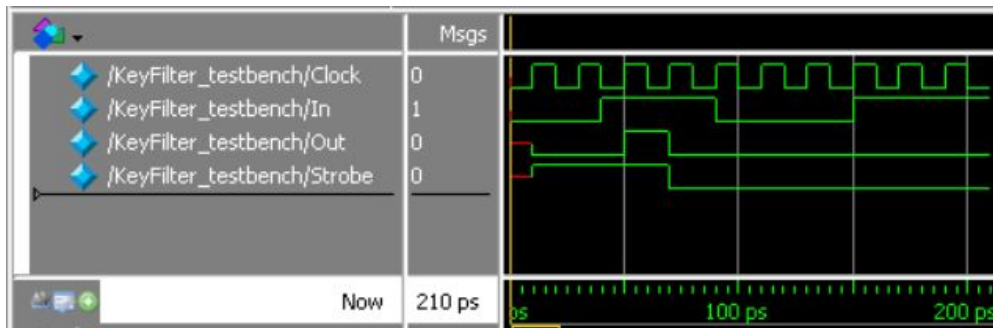
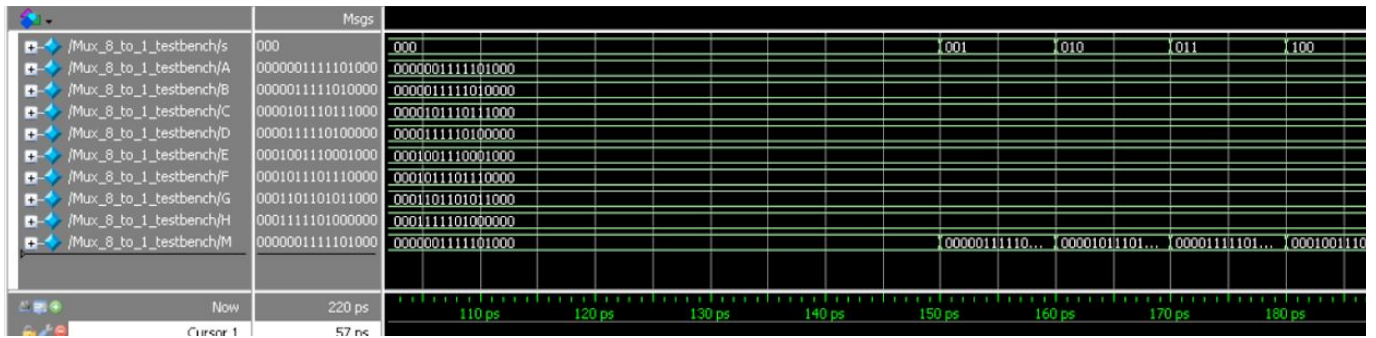


Figure 31. Waves for the Key Filter module in ModelSim

n. MUX 8 to 1 module

s	A	B	C	D	E	F	G	H	M
0	0	0	0	0	0	0	0	0	0
0	1000	2000	3000	4000	5000	6000	7000	8000	1000
1	1000	2000	3000	4000	5000	6000	7000	8000	2000
2	1000	2000	3000	4000	5000	6000	7000	8000	3000
3	1000	2000	3000	4000	5000	6000	7000	8000	4000
4	1000	2000	3000	4000	5000	6000	7000	8000	5000
5	1000	2000	3000	4000	5000	6000	7000	8000	6000
6	1000	2000	3000	4000	5000	6000	7000	8000	7000
7	1000	2000	3000	4000	5000	6000	7000	8000	8000

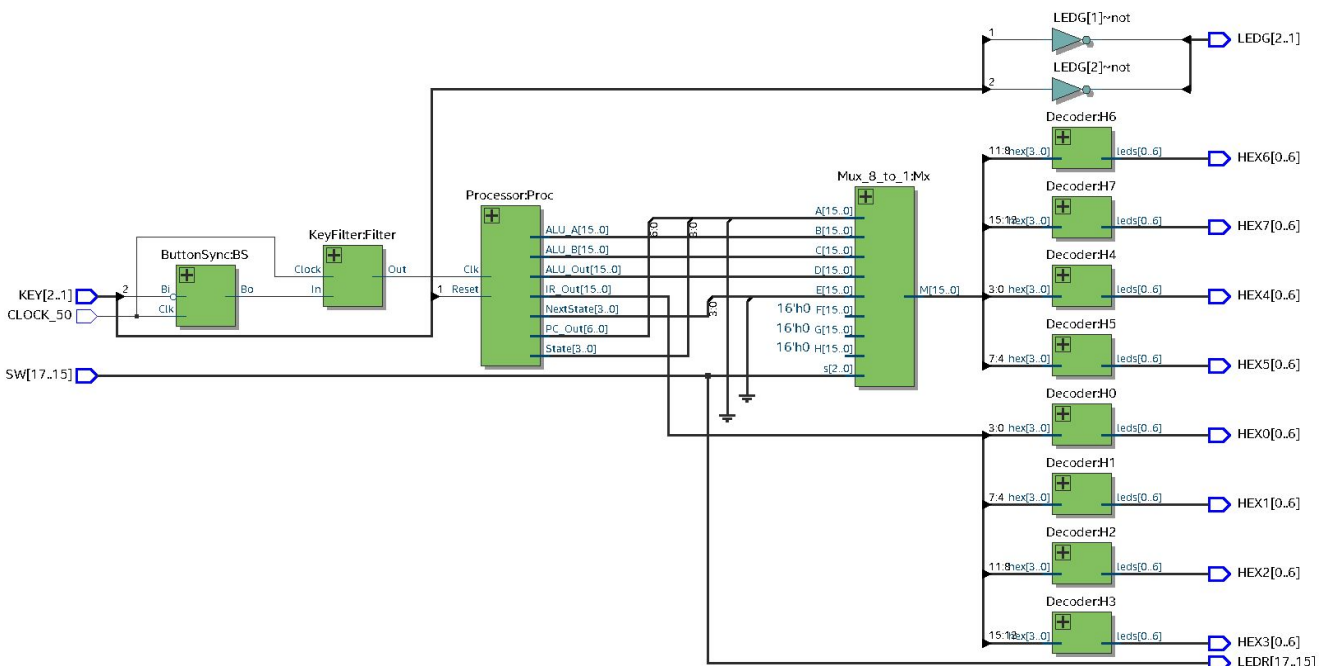
Figure 32. Testing transcript for the MUX 8 to 1 module in ModelSim



**Figure 33.** Waves for the MUX 8 to 1 module in ModelSim

Looking at both transcript and waves, we see that for each value of the select-signal, the output produced a corresponding output that matched with our expectations. Which proved that the MUX is working properly.

#### o. Top Level Module RTL Viewer



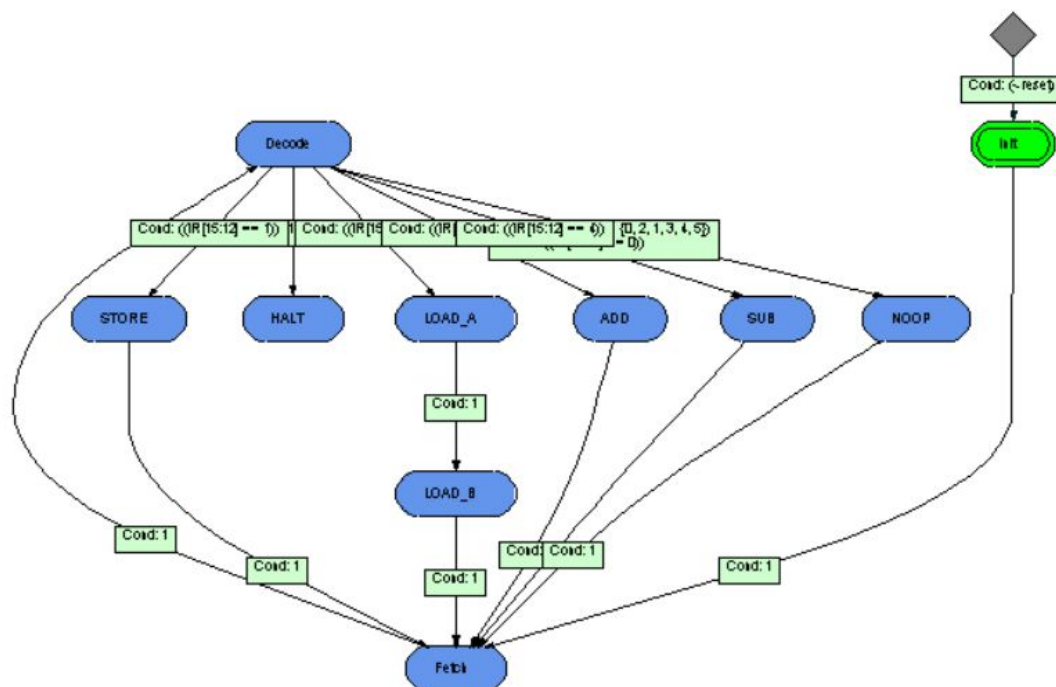
**Figure 34.** RTL Viewer of the Processor from Quartus

We can see that this figure matches what was expected from us in the project description. The main difference is that we only assigned switches that were needed in the project (SW[17:15]). We also only used the necessary red LEDs (17:15) that are needed within the project. The same idea was used for the KEY inputs as well. We only needed KEY[2:1] which is why we only included those two instead of all 4 of them.



Type	ID	Message
⚠	18236	Number of processors has not been specified which may cause overloading on shared machines.
⚠	18236	Number of processors has not been specified which may cause overloading on shared machines.
⚠	292013	Feature LogicLock is only available with a valid subscription license. You can purchase a s
⚠	15714	Some pins have incomplete I/O assignments. Refer to the I/O Assignment Warnings report for
⚠	171167	Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilati
> ⚠	169177	1 pins must meet Intel FPGA requirements for 3.3-, 3.0-, and 2.5-V interfaces. For more info
⚠	18236	Number of processors has not been specified which may cause overloading on shared machines.
⚠	18236	Number of processors has not been specified which may cause overloading on shared machines.
⚠	18236	Number of processors has not been specified which may cause overloading on shared machines.

**Figure 35.** Final warning messages of the project from Quartus



**Figure 36.** State transition diagram of the FSM from Modelsim

We can see from Figure 36 that this state transition diagram matches the one that was generated in Quartus (Figure 11). The conditions match what we have implemented in the code which makes sense to what we are seeing.

#### p. DE2-115 board

Finally, the design is downloaded to the DE2-115 board and tested in hardware. The following link provides a physical demonstration of our implementation of this 6-instruction processor:

<https://www.youtube.com/watch?v=whlhjFock6U>

Our design worked correctly as expected. We were able to display the current contents of IR, the current state and PC, the next state, the value for A and B of the ALU, and the ALU\_Out result. We were able to display the reset function working correctly, which would reset the state machine and PC.

#### **4. OBSERVATIONS**

Throughout this project, many issues appeared because of minor mistakes or misunderstanding of the project implementation. One of the issues involved the ALU, regarding the case where the output is equal to the first input plus 1. When we got to the top level module, we noticed a warning mentioning a truncated value. This is because we implemented the operation as the first input + 1, when it should have been the first input + 1'b1. This issue took some time to resolve. Another issue involved the provided testProcessor.sv file. When we created our processor and were ready to test it, we noticed that our module was stuck in the init stage of the state machine. To resolve this, we realized that the testProcessor.sv operates on an active low reset, while our FSM operates on an active high. This was why our module wasn't executing correctly. To resolve this, we fixed our FSM and testbench as well as the ControlUnit testbench.

#### **5. CONCLUSION**

The project was completed successfully. We were able to design and implement a 6-instruction processor that can be verified on the DE2-115 board. The hardest part of the project was making sure that all of us understood the operation of each module, especially the state machine. Another challenge involved the interconnections between multiple modules (Control Unit and Datapath). After understanding the function and guidelines of each module, we were able to implement it fairly quickly with considerable communication.

Throughout the project, we learned to cooperate and work together as a team. The importance of communication and team chemistry played an important role in our success to the project completion. We learned the basic idea of how a team project is designed, implemented, and tested while working together to achieve the common goal of creating and implementing this 6-instruction processor.