Course Project Programmable Processor

TCES330 Digital System Design Spring 2023

Authors:

Student A: Jotkamal Jawal

Student B: Souleymane Dembele

Submission Date: 6,06, 2023

Table of Contents:

Requirements

For this project my groupmate and I were assigned to create a programmable cpu. In order to make this cpu we need to first create two separate units which are the control unit and the datapath. In the control unit we were tasked to create the instruction memory, PC, IR, and a state machine. After these 4 modules have been completed and connected properly, the control unit is finished. Furthermore, we also had to create a data memory file, a mux, register file and an ALU in order to create the data path. After the creation of these files we then created another file which connects the two modules together and works as the processor.

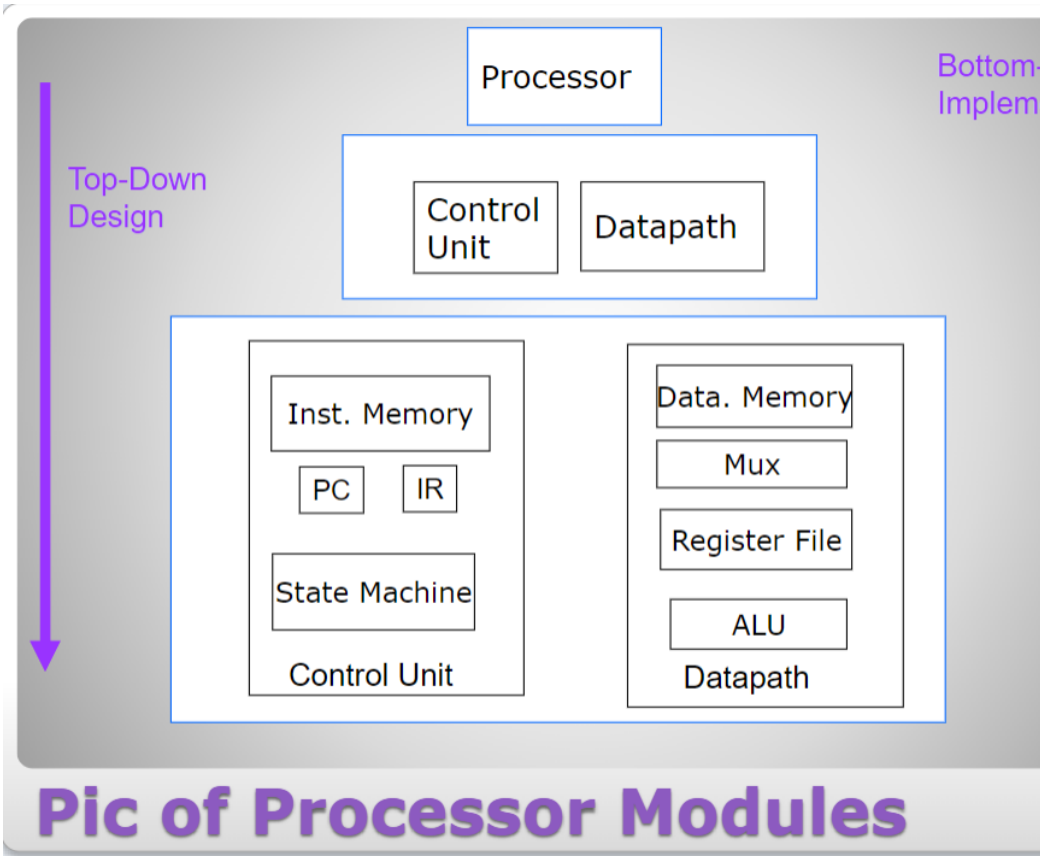

Figure 1: Processor Structure

Requirements for Controller.sv

For this part of the project we were tasked to create 4 separate files. These 4 separate files include the instruction memory, PC, IR and a state machine. The goal of the instruction memory was to hold instructions that our cpu will carry out. Moving along, the PC is a simple counter that increments its count as long as clear is active. The IR job of the IR is to hold the encoded instruction and assign it to our instructionOut. The final part of our control unit is a state machine that decides the states that our cpu will go through. Combining all these files is the purpose of our Controller.sv. In our Controller.sv we instantiate all 4 of the lower-level modules that we created.

Requirements: Datapath.sv

In this part of the project we also had to create 4 separate files. These files included the data memory, a mux, a register file and the alu. The data memory file that we created served the purpose of storing and holding data. The purpose of our Mux 2 to 1 was to decide whether the write data needed to equal the ALU input or the read data. Our register file was used to hold an

instruction or store an address and our ALU worked as the arithmetic logic unit; in order words it decided what kind of mathematical approach was needed. Combining these files together created our datapath.

Requirements: Processor.sv

The processor has two main parts and they were the two parts that we completed earlier which were the control unit and the datapath. Our goal for this part of the project was to connect the two modules together with proper connections and wires. We did this by instantiating both the control unit and data path and passed information in between them. Some of the inputs in the datapath needed to be accessed from the control unit so we created wires in order to pass the data.

Requirements: Project.sv

The main requirement of our project consists of building a simple programmable processor that can compute six instructions. The instructions are load, store, add, subtract, noop, halt. Our goal for the Project.sv was to instantiate 5 different modules and connect them using internal wires. The main modules are:

- ButtonSyncReg
- Filter
- Processor
- Mux_Nw_8_to_1
- Decoder

In addition to the modules, our Project.sv interface with the DE2 board as follows:

- KEY[2] acts as our system clock
- KEY[1] is a synchronous system reset (Make sure to press reset while cycling the clock)
- HEX3, 2, 1 and 0 always displays the current contents of IR (current instruction)
- SW[17:2] determines what HEX 7, 6, 5 and 4 displays as follows:
  - 0 => HEX7, HEX6 = PC; HEX5, HEX4 = Current State;
  - 1 => HEX7, 6, 5, 4 = ALU_A (A-side input to ALU)
  - 2 => HEX7, 6, 5, 4 = ALU_B (B-side input to ALU)
  - 3 => HEX7, 6, 5, 4 = ALU_Out (ALU output)
  - 4 => Next State (FSM next state variable, if you use one)
  - 5 Unused
  - 6 Unused
  - 7 Unused

Design

ControllerStateMachine.sv (FSM.sv)

The ControllerStateMachine.sv is our Finite State Machine module. It was straightforward during the design and implementation process since we had the chance to work on a similar design during a previous lab and we also had a great schematic of how it should look. The FSM takes in 3

inputs: the clock, the reset signal as well as the current 16-bit wide instruction. Then configures and outputs signals as per **Figure 2.1 a**.

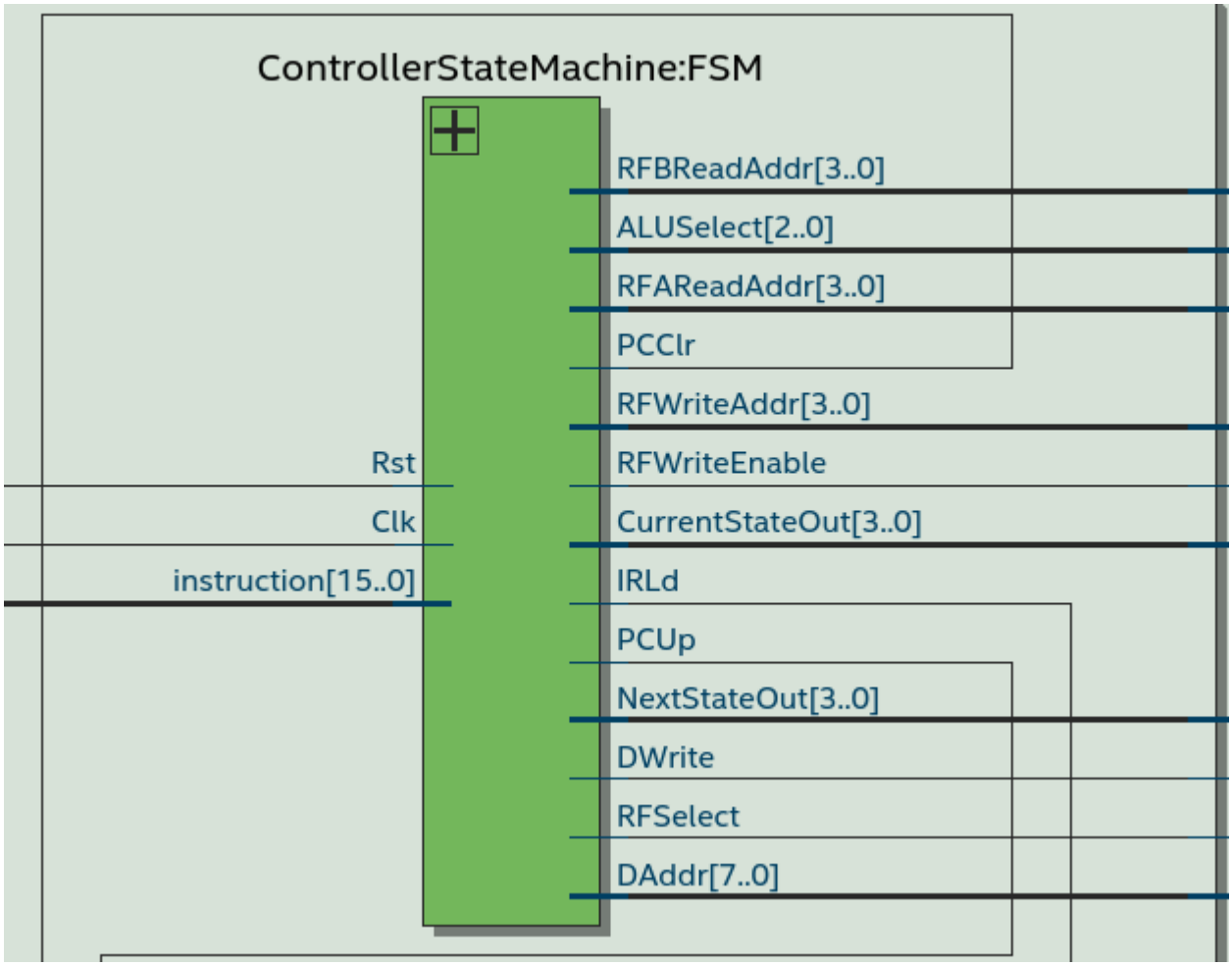

Figure 2.1 a - Represents the rtlview of our FSM module.

Our FSM module is composed of 3 major portions. A sequential circuit, a combinational circuit, and continuous assignment of required and test signals. In addition, we defined our states as follows:

- INIT = 4'b1000, // initial state
- FETCH = 4'b1111, // fetch instruction
- DECODE = 4'b1100, // decode instruction
- NOOP = 4'b0000, // no operation
- LOAD_A = 4'b0010, // load A
- STORE = 4'b0001, // store
- ADD = 4'b0011, // ADD
- SUB = 4'b0100, // SUB
- HALT = 4'b0101, // halt
- LOAD_B = 4'b0110; // load B

For the sequential side in the always_ff, for each positive edge of the clock, if reset is equal to zero, we set the CurrentState to INIT otherwise we set the Current State to the Next State. For the combinational portion of the FSM (always_comb), we started by initializing all the necessary outputs to their default value. For example, we want our ALUSelect to be 0 initially and PCClear 0 as well. We used the case statement with the sensitivity of the currentState. As stated above depending on the CurrentState, we perform the task in our FSM then we go to the nextState. For instance, in the case of the LOAD_A, we set the DAddr to the 11:4 bit of the current instruction, RFSelect to zero, RFWriteAddr to bit 3:0 of the instruction then got to LOAD_B. For LOAD_B, we set the DAddr to the 11:4 bit of the current instruction, RFSelect to zero, RFWriteAddr to bit 3:0 of the instruction then got back to FETCH state. For testing and requirement, we assigned the specific outputs for CurrentStateOut and NextStateOut using the continuous assignment.

PC.sv

The PC module is a pretty straightforward module as it is purely a counter. The required inputs and output for the PC are a clock signal, clear, Up and adder. In our pc we use Up as the opposite input to clear. If the pc sees the input to clear is 1 it resets the counter total to 0 otherwise if the signal to Up is 1 the counter gets incremented by 1.

```
5   module PC (
6       Clk,
7       Clr,
8       Up,
9       Addr
10  );
11
12    input Clk, Clr, Up;
13    output reg [6:0] Addr;
14
15    always @(posedge Clk) begin
16      if (Clr) Addr <= 0;
17      else if (Up) Addr <= Addr + 1;
18    end
19
20  endmodule
```

Figure 2.2: The code design for the PC

IR.sv

The design for the IR was pretty simple as all it does is hold the instruction that is currently being executed or decoded. We began the design by giving it 2 inputs and one output. The two inputs we used were clock and load; and the output we used was instruction. We also had another piece which was called the instructionOut. We then wrote a simple always block in which if there is a load, our instructionOut is equal to instruction.

```
 9  module IR (
10      Clock,
11      Ld,
12      instruction,
13      instructionOut
14  );
15    input Clock, Ld;
16    input [15:0] instruction;
17    output [15:0] instructionOut;
18    reg [15:0] instructionOut;
19    always_ff @(posedge Clock) begin
20      if (Ld) begin
21        instructionOut <= instruction;
22      end
23    end
24  endmodule
```

Figure 2.3: IR code design

Controller.sv

The purpose of the Controller.sv file was to combine all 4 of the lower-level modules into one module. When creating this module the inputs that we gave the controller were clock and reset. For the outputs we had 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. We also created 4 wires in order to help pass some data relating the instruction memory, pc up, pc clear and instruction load. We then wired our modules together based on the figure below. We first called our instruction memory and gave it the PC_Out as the address, a clock signal and a wire to hold its output. We then instantiated the PC module and gave it its respective inputs and saved its output into a wire. We then called our instruction register and passed our output that we saved from the instruction memory into it. After this step we passed the output of the instruction register into the state machine along with a clock and reset signal.

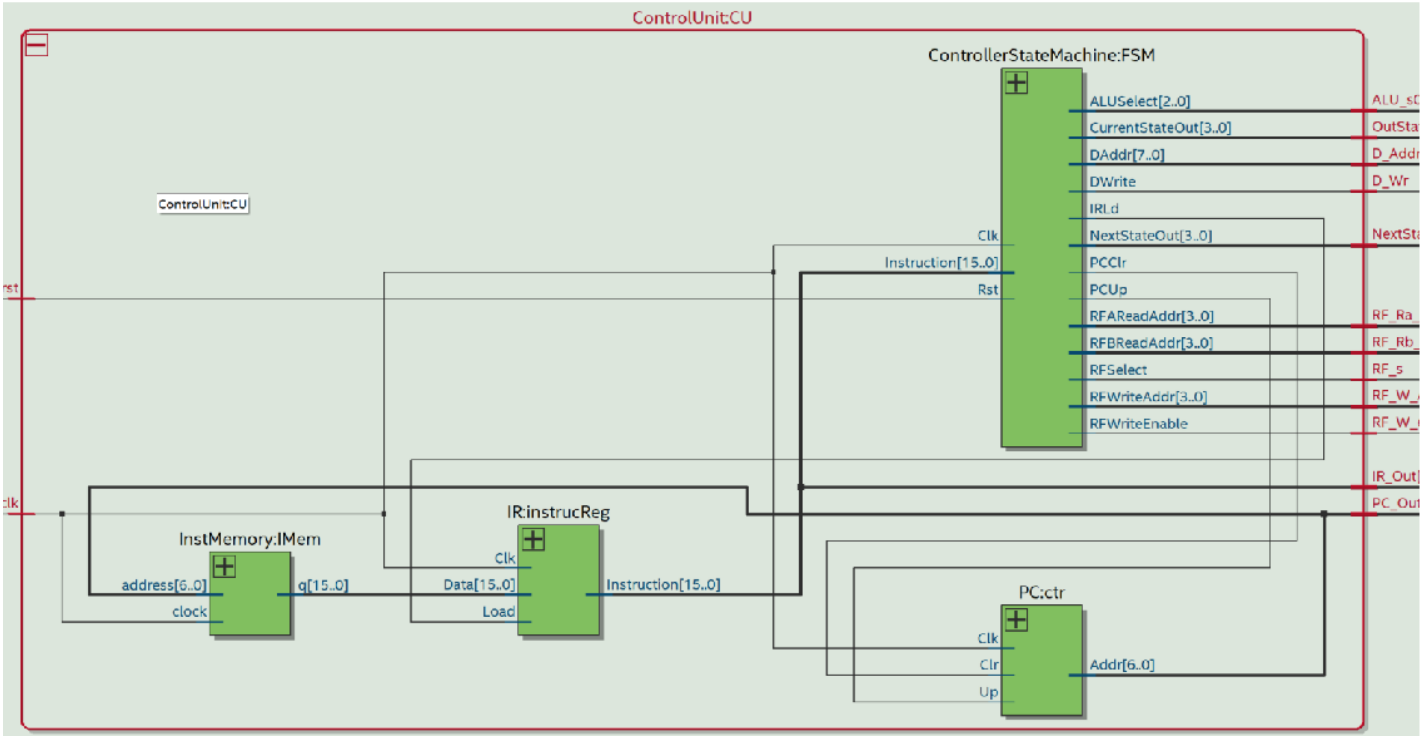

Figure 2.4: Control Unit Blueprint

regfile16x16a.sv

Like the other modules, our register file was a great challenge. During the process, we learned to design the register file as well. The register file is one of the main structures of our Datapath. It consists of a set of registers that can be read and written by supplying a register number (address) to be accessed. Our register takes in a 16-bit wide data to be written called wrData, a 4-bit wide write address (wrAddr), two 4-bit wide read data addresses (A-side read address and B-side read address), a clock, and a write enable signal then it outputs two 16-bit wide signals: the A-side read data (rdDataA) and the B-side read data (rdDataB). Figure 2.5 a represents the code snippet of our regFile16x16.sv.

```
 9  module regFile16x16 (
10      clk,  // system clock
11      write,  // write enable
12      wrAddr,  // write address
13      wrData,  // write data
14      rdAddrA,  // A-side read address
15      rdDataA,  // A-side read data
16      rdAddrB,  // B-side read address
17      rdDataB  // B-side read data
18  );
19    input clk, write;
20    input [3:0] wrAddr, rdAddrA, rdAddrB;
21    input [15:0] wrData;
22    output [15:0] rdDataA, rdDataB;
23
24    reg [15:0] regFile[0:15];
25
26    // read the registers
27    assign rdDataA = regFile[rdAddrA];
28    assign rdDataB = regFile[rdAddrB];
29
30    // write to the registers
31    always_ff @(posedge clk) begin
32      if (write) begin
33        regFile[wrAddr] <= wrData;
34      end
35    end
36  endmodule
```

Figure 2.5 a - represents the code snippet of our register file (regFile16x16.sv)

Mux_n_2to1.sv

The design of the Mux2to1 was pretty straightforward as all it does is just decide what our write data will equal. To create the mux we gave it 3 inputs and one output. The 3 inputs were ALUQ, RFSelect, and ReadData. The output for the mux was write data. The way that the mux works is that based on our RFSelect it will decide our write data. If our RF select value is 0 our write data will equal our ALUQ input, otherwise if it is equal to 1 our write data will equal read data.

```
7   module Mux2to1 (
8       ALUQ,
9       RFSelect,
10      ReadData,
11      WriteData
12  );
13
14    input [15:0] ALUQ;
15    input RFSelect;
16    input [15:0] ReadData;
17    output reg [15:0] WriteData;
18
19    always @(ALUQ, ReadData, RFSelect, WriteData)
20      case (RFSelect)
21        0: WriteData = ALUQ;
22        1: WriteData = ReadData;
23      endcase
24
25  endmodule
```

Figure 2.6: Code design for Mux2to1

ALU.sv

The alu is an arithmetic logic unit and we use it to perform mathematical operations. For this module we gave it an input of A, B and a select bit value. It also has an output which we named Q. A and B hold two separate data points and will have operations done on them based on the select bit. We then assign operations to each of our select bits. If our select bit is 0 the output is 0. If our select bit is 1 the output is A + B. If the select bit is 2 the output is A - B. If the select bit is 3 then our output is A. If the select bit is 4 then our output is the xor of A and B. If our select bit is 5 our output is A or B. If our select bit is 6 then our output is A anded with B. Finally if our select bit is 7 our output is A + 1.

```
7   module ALU (
8       A,
9       B,
10      Sel,
11      Q
12  );
13
14    input [2:0] Sel;
15    input [15:0] A, B;
16
17    output reg [15:0] Q;
18
19    always @(Sel, A, B)
20      case (Sel)
21        0: Q = 16'h0;
22        1: Q = A + B;
23        2: Q = A - B;
24        3: Q = A;
25        4: Q = A ^ B;
26        5: Q = A | B;
27        6: Q = A & B;
28        7: Q = A + 1;
29      endcase
30
31  endmodule
```

Figure 2.7: Code design for ALU

Datapath.sv

In order to create the datapath we followed the blueprint that was given to use below. We first created the 4 respective individual files and tested each to insure that they worked. After that we created our data path file and initialized all 4 files in the datapath file. We first began by calling our data memory module and passing it its respective inputs and saved its output into DMEM_Q. We then called our mux and passed our output from the data memory into it and saved its output into MUX_Q. This output was then used in our regFile which gave two outputs (RDataA and RDataB) and these two outputs served also as the A and B to our ALU. Once we passed these two pieces of data and a select bit to our ALU it gave us our ALU output.

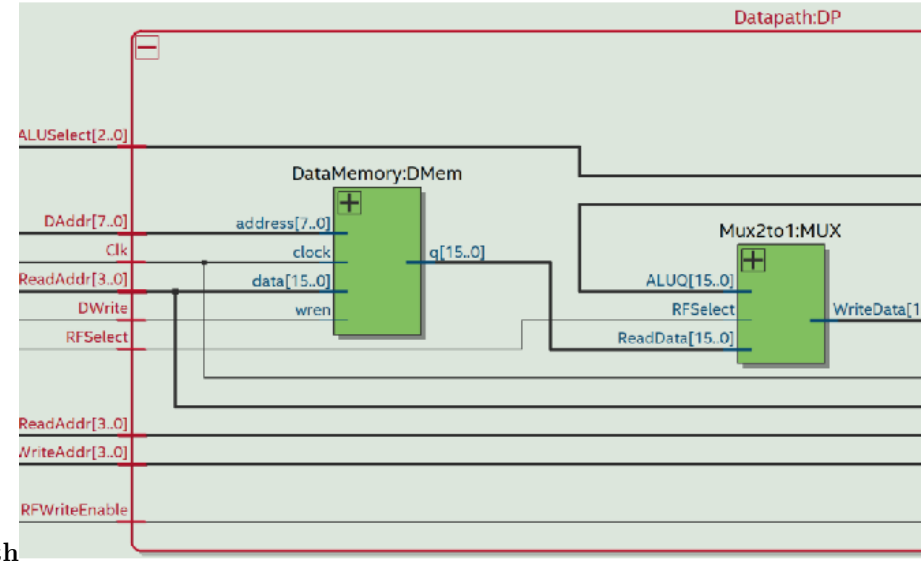

Figure 2.8: Design blueprint for Datapath

Processor.sv

The processor file was the combination of our control unit and our datapath. In this module we had two inputs which were the clock and reset. There were also several outputs which were the IR_Out, Pc_Out, state, nextState, alu_a, alu_b and alu_out. We almost created several wires in order to use some of the data that we would get from certain modules. To properly wire and connect our processor we followed the figure given below. We began by connecting the Control unit first and gave it its two inputs which were the clock and reset. After that we assigned its outputs to their respective values and saved some outputs to wires. For example, D_addr, RF_Ra_Addr, RF_Rb_Addr, RF_W_Addr, and Alu_s0 were saved to a wire. The wires were then used in our datapath module in order to be able to pass data to certain inputs. After that we connected aluAin and aluBin and aluOut to send output. Adding on to that IRout, NextState, CurrentState and PCOut also are outputs to the processor module.

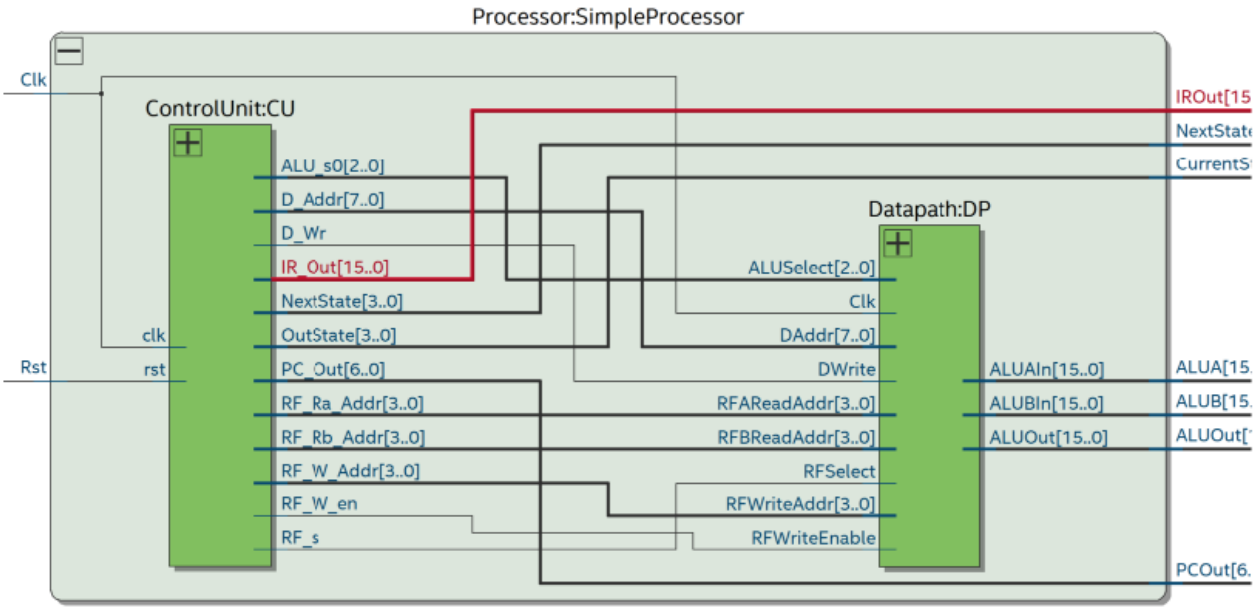

Figure 2.9: Blueprint design for Processor

Project.sv

In this file our goal was to set it up so that we could use this file on our d2 board. On our board we needed to be able to use 2 keys, switches 17-15 and be able to see our output on HEX0-7. In order to do this we added key, HEX0-7 and SW as our inputs. We also added LEDR and LEDG as our outputs because we wanted to be able to see when the switch and key are on. We also used a button synchronizer and a key filter in order to ensure our inputs were correct. We used the button synchronizer as a solution to the button debouncing problem and then paired it with a key filter to only allow for a certain amount of signals per second. We then created a 8_to_1 mux in order to be able to decide what we wanted to output onto our fpga board. For example if A was to be selected then we want our HEX's to display our pc out and state out. If b or c is selected then we want either Alu_A or Alu_B to print out; the same procedure applies to selecting D or E. In our case, we have f, g, and h as 0s because according to our instructions those are supposed to do nothing. After that we also instantiated our decoder allowing for our HEX values to be printed onto our fpga. Once we had everything combined we compiled the circuit and began to create the processor clocks to get rid of the warnings from our compilation. We created three clocks and after running the compilation again we saw that there were no more warnings.

```
69    Mux_Nw_8_to_1 #(
70        .N(16)
71    ) Mux_Nw_8_to_1 (
72        .S(SW[17:15]),
73        .A({1'h0, PC_Out, 4'h0,StateOut}),
74        .B(ALU_A),
75        .C(ALU_B),
76        .D(ALU_Out),
77        .E({12'h0, NextStateOut}),
78        .F(16'h0),
79        .G(16'h0),
80        .H(16'h0),
81        .M(Mux8tlNw_Out)
82    );
```

Figure 2.10 : Mux_8_to_1 example

Test Procedures and Test Results

ControllerStateMachine_tb (FSM_tb)

The ControllerStateMachine_tb tests our Finite State Machine. During the test, we checked the output of the state machine and made sure that it was correct. In addition, we started by running all 65536 possible inputs to the state machine. We then checked the output of the state machine and made sure that it was correct. Then we removed the 65536 cases and made use of $random, then reduced the number of inputs to 200 and checked the output of the state machine. We noticed that the state machine was working as expected. For example, when the bit 15:12 of the instruction, which is the opcode, is 0000, the state machine goes to the state 0000 (NOOP). In addition, when the bit 15:12 of the instruction is 0011, the state machine goes to state (0011) ADD. We are also able to see the ALUSelect as well as all the wires set according to the instruction.

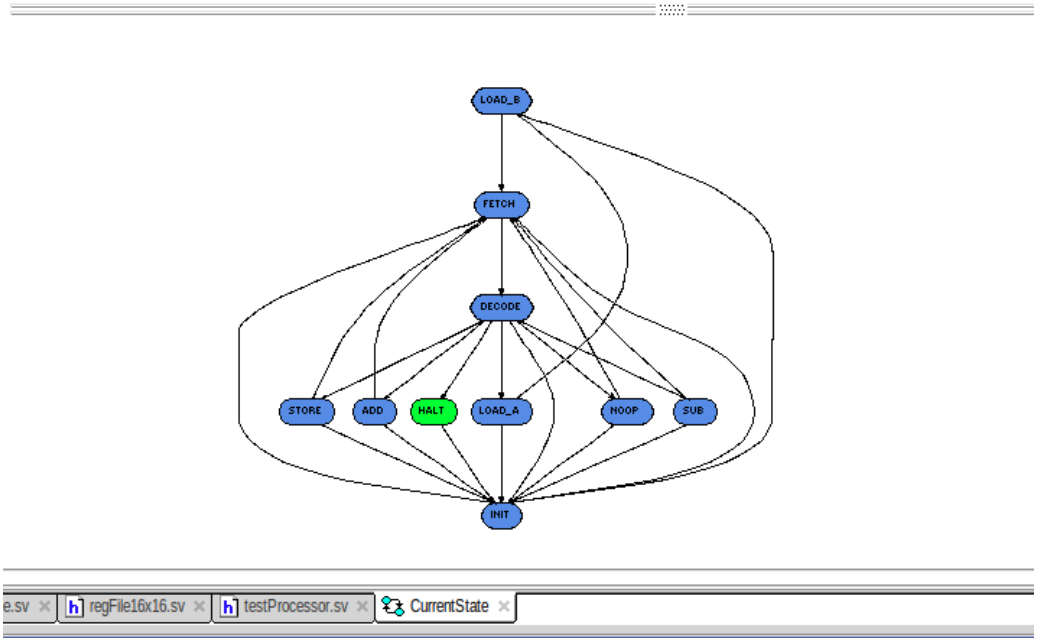

Figure 3.1 a - Representation of our FSM view from ModelSim.

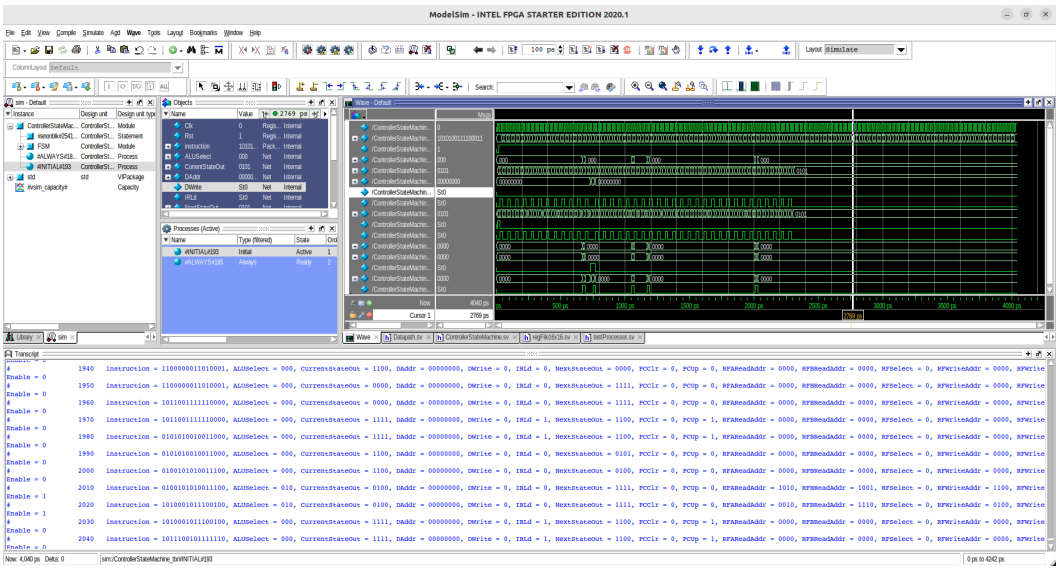

Figure 3.1 - b Representation of the modelsim view of the testbench of our FSM.

Controller_tb

The Controller_tb is the test module for our Controller module. In order to run, we create runrtlController.do as well as Controller_wave.do files as for each of our modules. The test itself was pretty easy since the only inputs were the Clock and Reset. We then instanciate the Controller module and connected with all the required wires. When we run the simulation, we can see that the PC is initialized to 0x00. The next state out of our current state is always our current state. The instruction memory get loaded with the instruction that we gave it soon as the PC increment to 0x01 we see that IR_Out contains 0x21b1. At the time of the first instruction, the PC is 0x01 and the instruction memory is loaded with the first instruction. Furthermore, for the instruction in IR_Out we see that our state machine started at the first state and then went to the second state (Fetch). Then it went to the third state (Decode) and then to the fourth state (0x2 which is the load state). After the load state, we see the next state that goes to fetch state again. After the fetch state, we see that the PC is incremented to 0x02 and the instruction memory is loaded with the second instruction. We see that the IR_Out contains 0x21a2. We see that the state machine went to the decode state and then to the load state. After the load state, we see that the next state is the fetch state. After the fetch state, we see that the PC is incremented to 0x03 and the instruction memory is loaded with the third instruction. We see that the IR_Out contains 0x21c3. We see that the state machine went to the decode state and then to the load state. After the load state, we see that the next state is the fetch state. After the fetch state, we see that the PC is incremented to 0x04 and the instruction memory is loaded with the fourth instruction. So one and so forth and we see that our Controller works as expected.

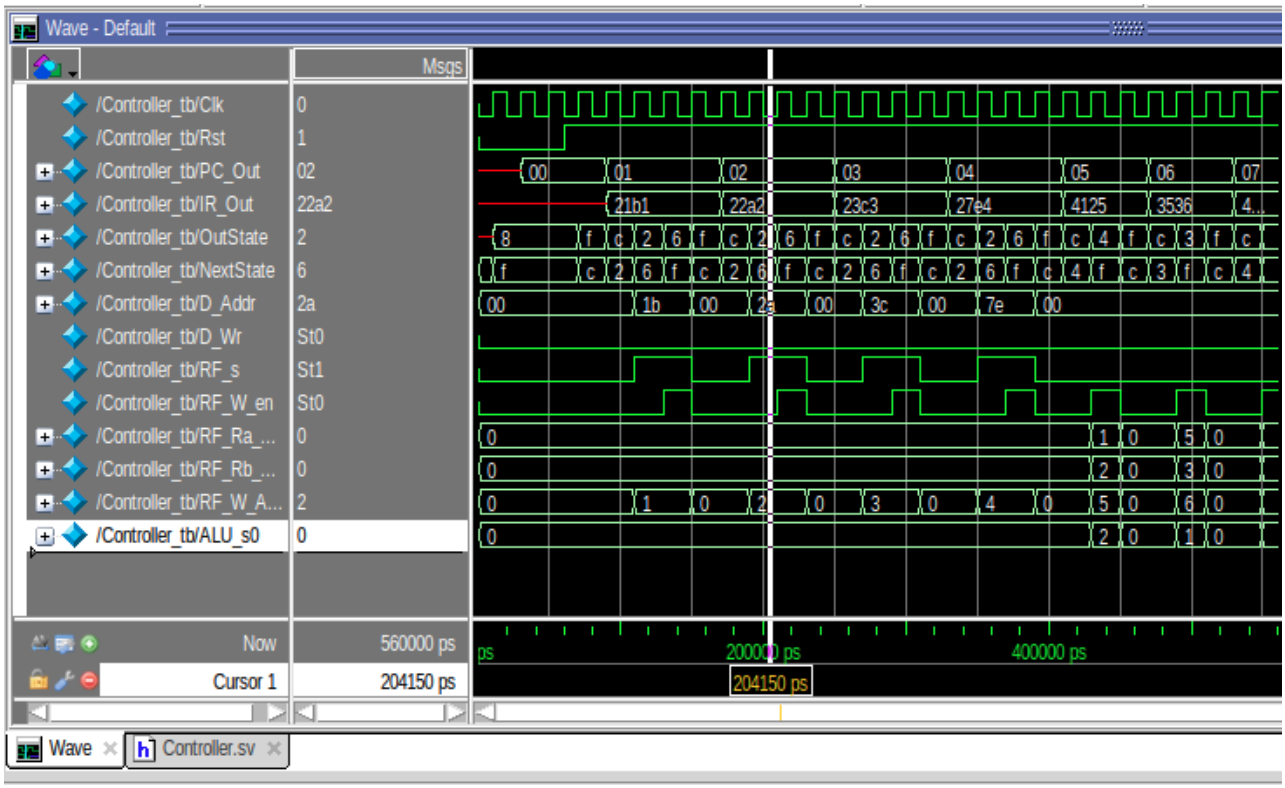

Figure 3.2 a - Represents the wave form of the testbench for our Controller_tb

```
# run -all
#                    0  IMem_Q = 0000000000000000, PC_Out=xxxxxxx
#                   10  IMem_Q = xxxxxxxxxxxxxxxx, PC_Out=xxxxxxx
#                   30  IMem_Q = xxxxxxxxxxxxxxxx, PC_Out=0000000
#                   50  IMem_Q = 0010000110110001, PC_Out=0000000
#                   90  IMem_Q = 0010000110110001, PC_Out=0000001
#                  110  IMem_Q = 0010001010100010, PC_Out=0000001
#                  170  IMem_Q = 0010001010100010, PC_Out=0000010
#                  190  IMem_Q = 0010001111000011, PC_Out=0000010
#                  250  IMem_Q = 0010001111000011, PC_Out=0000011
#                  270  IMem_Q = 0010011111100100, PC_Out=0000011
#                  330  IMem_Q = 0010011111100100, PC_Out=0000100
#                  350  IMem_Q = 0100000100100101, PC_Out=0000100
#                  410  IMem_Q = 0100000100100101, PC_Out=0000101
#                  430  IMem_Q = 0011010100110110, PC_Out=0000101
#                  470  IMem_Q = 0011010100110110, PC_Out=0000110
#                  490  IMem_Q = 0100011001001010, PC_Out=0000110
#                  530  IMem_Q = 0100011001001010, PC_Out=0000111
#                  550  IMem_Q = 0001101001101010, PC_Out=0000111
# ** Note: $stop    : ./Controller.sv(118)
```

Figure 3.2 b - Represents the console output of the testbench for our Controller_tb

Datapath_tb

Our datapath_tb is the test bench that we used for our datapath. To be able to use this testbench we created a runrtlDatapath file and a waveDatapath file. In order to test it we had to add some inputs such as Alu_select, Daddr, clock, readAddr, Dwrite, RFselect, ReadAddr, WriteAddr and RFWrite enable. We also had some other inputs but those were based on the other lower level

modules. After setting up the beginning of our testbench we then instantiated our datapath file. We began the test by first setting Alu_s0 and RF_s to be 0, RF_w_en, D_wr, RF_W_adder, RF_Ra_addr, RF_Rb_addr and D_addr to 1. After setting those to 0 and 1 we waited 20 seconds. We then also added cases where we switched our Alu_s0 to be a 3 and 7, and RF_w_addr to be equal to 2. We then ran the test to see if we got the expected outcomes. In the beginning of the waves our ALU_A has a value and ALU_B does not. However, our ALU_out is still being incremented because we have our select bit at 7. When the select bit is 7 our A is supposed to increment by 1 which it is doing. Carrying on when we set our rf_w_addr we get a value in our alu_b and when we set our select bit to 3, we can now do addition. From our wave file we saw that the addition was occurring. From these tests we concluded that our datapath was working.

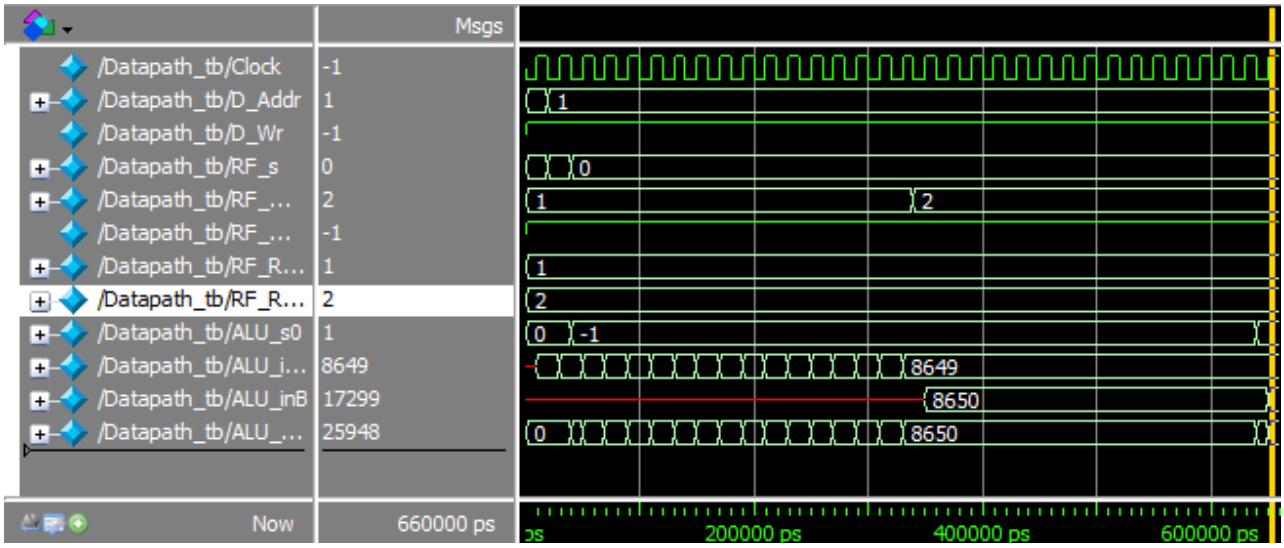

Figure 3.3: Datapath_tb wave file

testProcessor

The testProcedure.sv tests the procedure module. First we tested all the submodules of our processor to ensure that they all worked correctly. After we saw that they worked correctly we downloaded the provided testProcessor file from canvas and began to test our processor with the file. Initially we ran into some issues but using a display statement helped fix our issue. Once we fixed our problems we ran our code and got the result shown below. From the results shown below we can see that our PC is counting correctly as it is incrementing properly. Furthermore we can see that with the instruction for subtraction is active our state machine is in the state of 4 which is the correct state that it needs to be in. We can also see that the output to Alu_A is 21ba and the output to Alu_B is a04e. That is also correct because that is what we loaded into our memory. In the next case we can see that the Alu_A is 816c and the Alu_B is 71ac. That is correct because Alu_A is supposed to be what we calculated and Alu_B is what we loaded into D[3C] which is 71AC. In this step we are doing addition so we should be in state 3 which we are in. In the next occurrence we are subtracting so we should be in state 4 again and based on our testbench we are in state 4. Also our Alu_A is f318 and our Alu_B is b17f which is the correct value that we loaded in previously. Based on these results we concluded that our processor was working correctly.

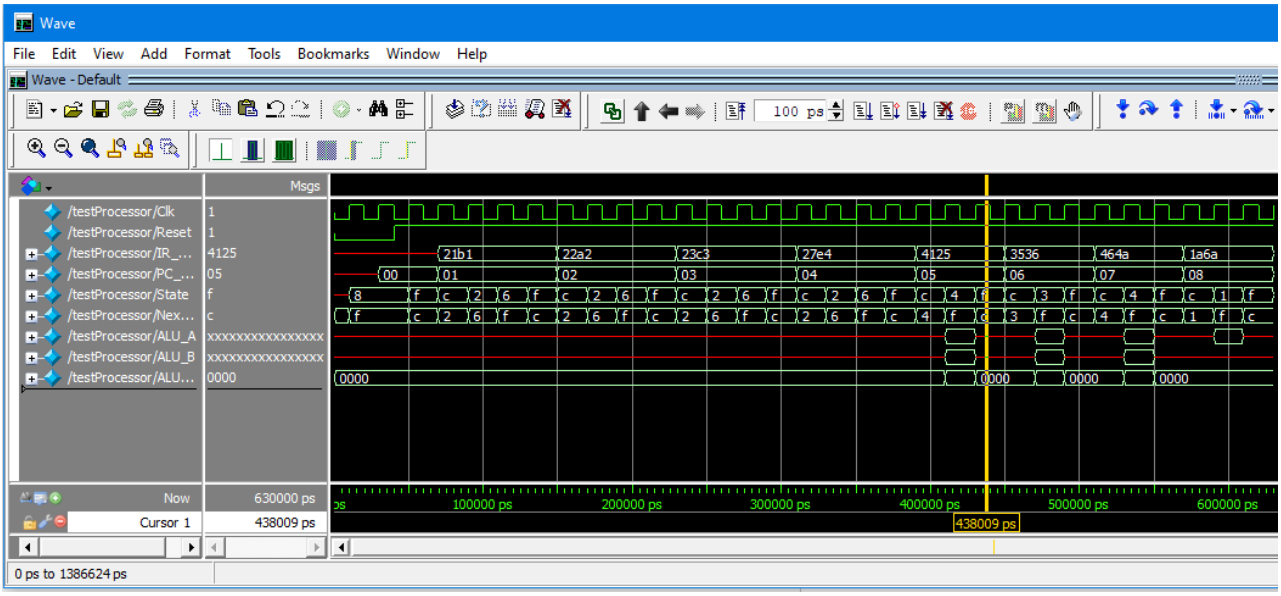

Figure 3.4: Waves for testProcessor

Observations

On top of the observations we made during the modular test, we noticed that we could write a testbench and have a good observation; however, if we do not follow the requirements, the could still be a final bug that can slow down the rest of the development process. For example, when building our datapath and the Mux-2-to-1, we were using 2-bit wide for RFSelect while we needed 1-bit for RFSelect. We had to debug with the use of multiple techniques we learned in class. Other than that there really weren't any error messages that we were not unfamiliar with. Listed below is our rtl view of the processor and when comparing it with the one provided by our instructor we can see that it is pretty much the same.

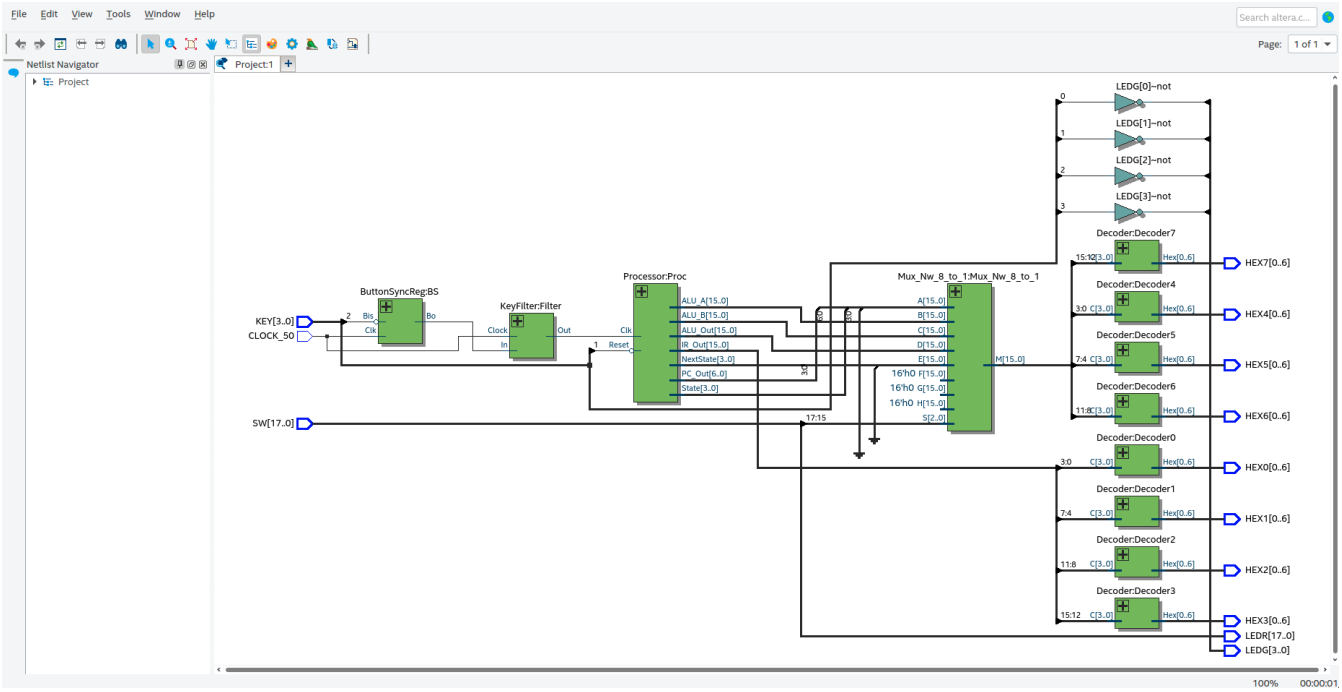

Figure 4: RTL of our processor

Conclusion

All in All, this was a very educational project and an amazing learning experience. Going into this project we thought it would be a lot harder to write and code a cpu however after breaking it down we realized that it is not as hard as it seems. Breaking everything down into functional modules made it a lot easier to understand what each part does and helped with the understanding of combining everything. Learning earlier subjects from this class such as the alu, creating counters, muxs, and test benches really helped speed up the process of getting the work done. Furthermore, learning how to use the display statement helped when it came to debugging and finding the answers to our errors. Being able to work with a team was a great experience as it lowered the amount of stress that this project could have caused. Working with a team allowed us to brainstorm and come up with solutions faster than we could have alone.

References

- Computer Organization and Design Mips Edition

Appendix

- **Work-Split**
  - **Datapath**
    - Souleymane: Data Memory and Register File
    - Jotkamal Jaswal: Mux and Alu
    - Together: Datapath combination of files
  - **Control Unit**
    - Souleymane: IR and State Machine
    - Jotkamal Jaswal: Instruction Memory and PC
    - Together: Control Unit combination of files
  - **Processor.sv:**
    - Jotkamal and Souleymane
  - **Project.sv:**
    - Souleymane and Jotkamal
  - **Keyfilter and ButtonSync:**
    - Souleymane
- **Video Demonstration:**
  - **Souleymane:** https://drive.google.com/file/d/1AqmRo3hyO7uKR_LuKsf1v-YFNvS39kAh/view?usp=sharing
  - **Jotkamal:** https://drive.google.com/drive/folders/1nCfMvkX4NndpeiJjm8Ir95_GARAfSG21?usp=sharing