Course Project

Programmable Processor
TCES 330 Digital System Design
Spring 2021

Authors:

Anastasia Staroverova: X___ 

Ajay Matto: X___ 

Submission Date: June 5, 2021

Table of Contents:

Project:

The purpose of this project is to implement the six instruction programmable processor using System Verilog. Our team members consisted of Ajay Matto and Anastasia Staroverova. To divide the workload, one of us designed and tested half of the modules needed and the other one designed and tested the second half.

1. Requirements

The purpose of the Controller, Datapath, Processor and Project are described in this section of our report. In the image below, the left side shows the Control and the components inside the Control: FSM, PC, IR, and instruction memory . The right side shows the Datapath and the components that make up that module: data memory, 2 to 1 MUX, 16x16 register file, and ALU. The Processor instances both the Control and the Datapath. The Project instances the processor, button sync, key filter, 8 to 1 MUX, and the decoder to test the modules on the D2 board.

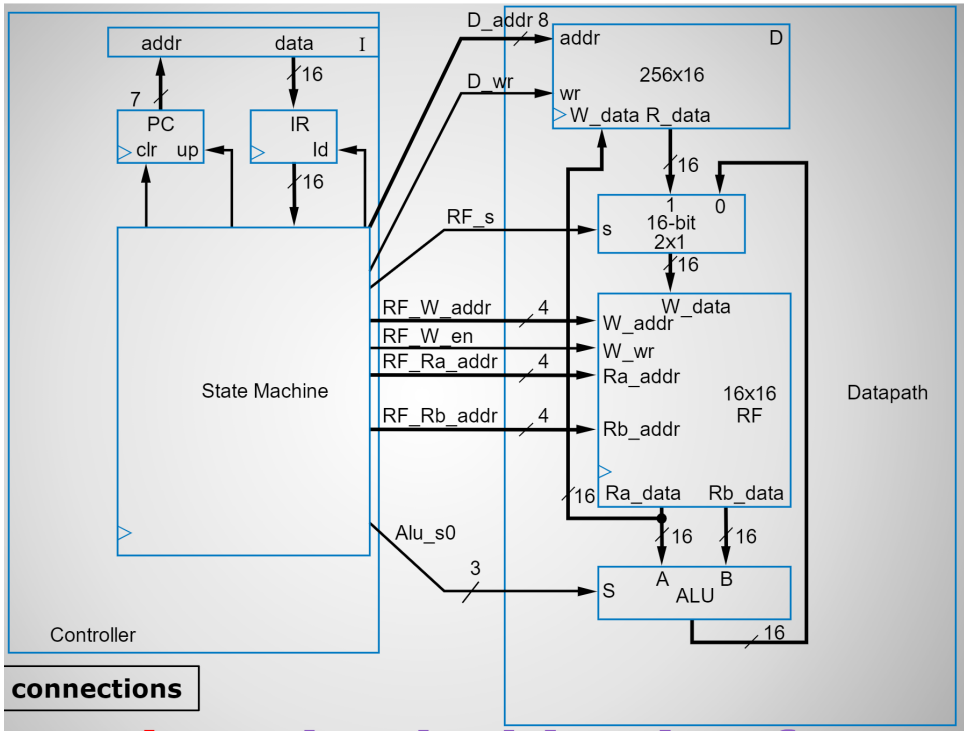

Figure 1:Modules needed

1.1 Control.sv:

The Control module instances 4 modules. Those modules include the Instruction Memory, PC, IR, and the Statemachine. The purpose of the Control module is to instance the PC which is a

counter that counts up in the instruction memory. Then, it instances the IR which is a flip flop that latches out the input. The IR takes the 16 bit output. Lastly, the state machine takes the output of the IR as an input then goes through the corresponding stages given to output the variables the datapath needs to execute the instruction.

The instruction memory is a 1 port ROM that uses “A.mif” with a size of 7 bit input and 16 bit output. The instruction memory holds the instructions. The PC tells it what instruction count it is on and it outputs the instruction to the IR. To figure out what to place in the A.mif as well as the location, we used Figure 2.

```
RF[0] = D[0B] - D[1B] + D[06] - D[8A];
D[CD] = RF[0];
HALT
```

Figure 2: Equation for Instruction Memory

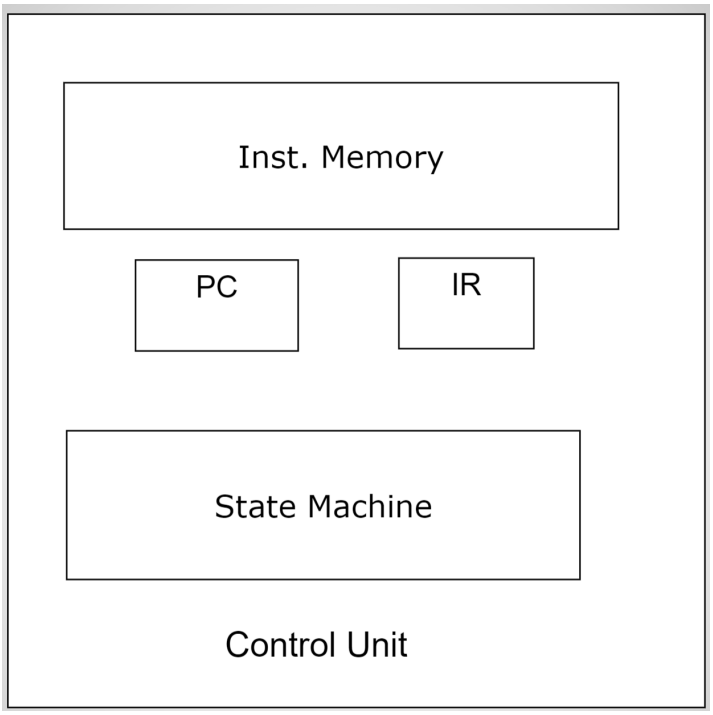

Figure 3:Control Unit outline

1.2 Datapath.sv

The Datapath constructs 4 modules: Data Memory, Mux2to1, Regfile16x16a, and the ALU. The purpose of the datapath is to execute the instruction given by the control. It has a memory called data memory where the values are stored. Values are temporarily stored in the register file.

The Datapath memory was used by creating a DataMemory.v(1-port RAM)with a D.mif which was a size of 16x256. We used the following equation given to understand where to place the bits.

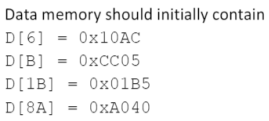

Figure 4: Data Memory components

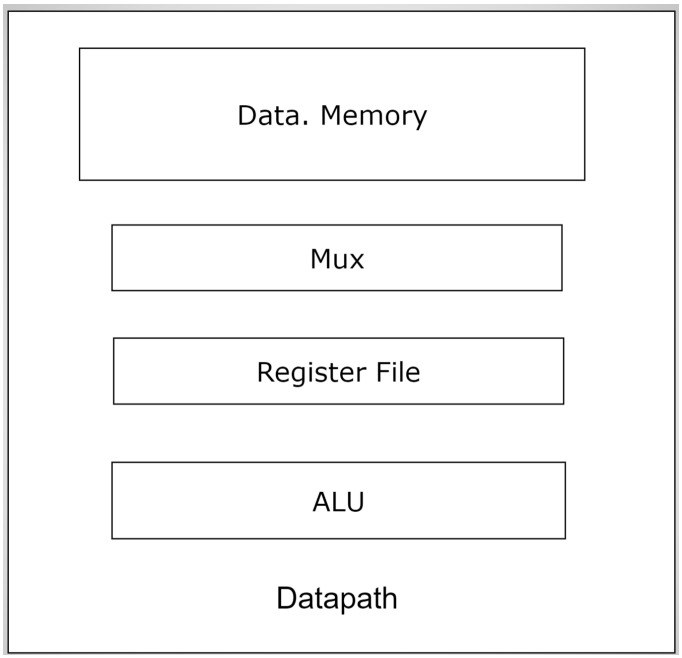

Figure 5:Data path submodules

1.3 Processor.sv

The purpose of the Processor was to instance both the Control and the Datapath. It is also used in the upper level module when creating the Project in Quartus. The Processor has 2 inputs: clock and reset and 7 outputs :IR_Out, PC_Out, State, NextState, ALU_A, ALU_B, and ALU_Out. The processor begins by instancing the Control Unit then the Datapath.

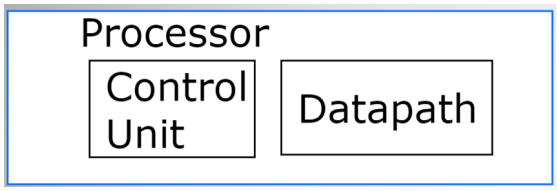

Figure 6: Outline of the Processor

1.4 Project.sv

The purpose of the project is to test the circuit we created to verify the validity. The project portion consisted of the Processor, Buttonsync, Keyfilter, Decoder and the Mux_3W_8_to_1. We instanced the processor we designed previously. The buttonsync as well as the KeyFilter was given in class. The decoder and Mux_3W_8_to_1 were designed in previous assignments so we just used those. We made a change to the MUX for its inputs and output to be 16 bits.

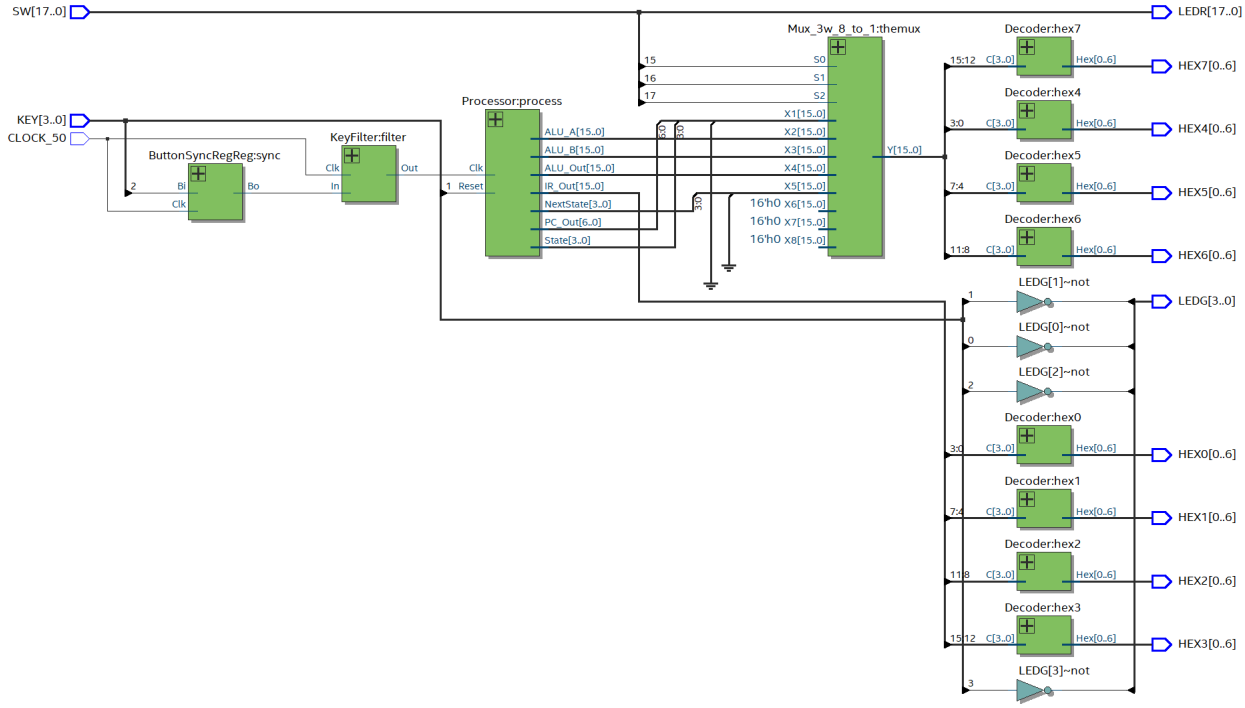

Figure 7: RTL view of the project.sv

2. Design

We used 16 different modules to help design and construct the project accordingly. The modules consist of: FSM.sv, PC.sv, IR.sv, Controller.sv, regfile16x16a.sv, DataMemory.v, InstructionMemory.v, Mux2to1.sv, ALU.sv, Datapath.sv, Processor.sv, ButtonSyncRegReg.sv, KeyFilter.sv, MUX_16w_8_to_1.sv, Decoder.sv, and Project.sv. We used the following image, of the processor, as a guideline when pursuing the solution needed to complete the project.

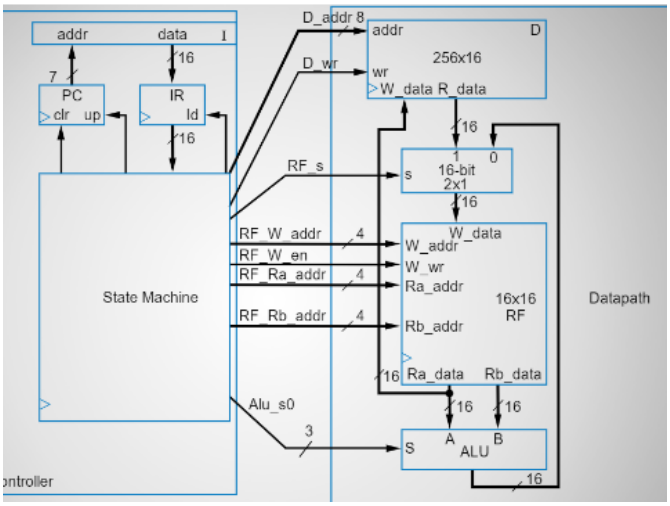

Figure 8: Multi-Module Connector

2.1 FSM.sv

The finite state machine submodule takes a 16 bit input from the IR module and assigns the corresponding 12 outputs accordingly. The output variables carry the information to the datapath needed to execute the instruction. We used figure 10 to help us determine the correct assignments. This submodule is instanced when creating the Control unit. The first thing we had to do for the FSM was to declare the 10 different states.

```
localparam Init=4'b0000,
                Fetch=4'b0001,
                Decode=4'b0010,
                NOOP= 4'b0011,
                Load_A= 4'b0100,
                Load_B= 4'b0101,
                STORE= 4'b0110,
                ADD= 4'b0111,
                SUB=4'b1000,
                HALT=4'b1001;
```

Figure 9 : FSM local param

```
Decode: begin
        if (IR[15:12] == 4'b0000) NextState = NOOP;
        else if (IR[15:12] == 4'b0010) NextState = Load_A;
        else if (IR[15:12] == 4'b0001) NextState = STORE;
        else if (IR[15:12] == 4'b0011) NextState = ADD;
        else if (IR[15:12] == 4'b0100) NextState = SUB;
        else if (IR[15:12] == 4'b0101) NextState = HALT;
        else NextState = Init;
end
```

Figure 10 : FSM decode case

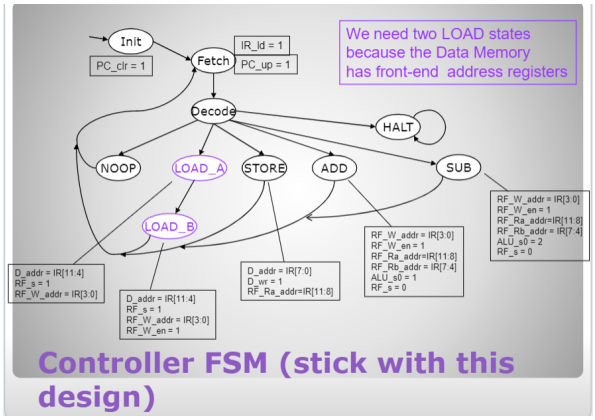

Figure 11: FSM Controller Map

2.2 PC.sv

The PC submodule is a counter with 3 inputs and 1 output. Figure 11 shows the code for the counter. The inputs consist of a clock, clear, and up signal. The output signal is the address to access the instruction memory. This module is instanced by the Control module.

```
always_ff @(posedge Clock) begin
        if (Clr) Im_Out <= 0;
        else begin
                if (Up) Im_Out <= Im_Out + 1'b1;
                else Im_Out <= Im_Out;
        end
end
```

Figure 12 :PC code

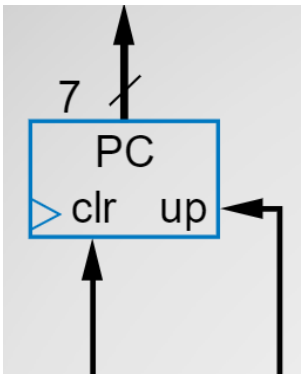

Figure 13: PC outline

2.3 IR.sv

This submodule has a 16 bit input from the instruction memory as well as a 1 bit clock and Id. It also contains a 16 output to the finite state machine. This submodule helps us design the control unit. The purpose of the IR is that it will latch out the input signal using a flip flop. Figure 14 shows the code used to implement the flip flop.This module is instanced by the Control module.

```
always_ff @(posedge Clock) begin
        if(Id) OutFSM<=instruction;
        else OutFSM<=OutFSM;
end
```

Figure 14: IR code

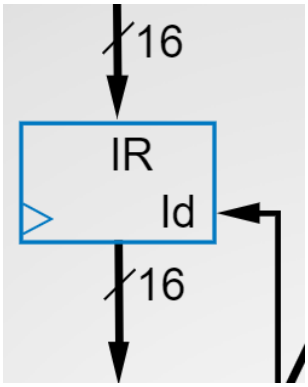

Figure 15: IR outline

2.4 Control.sv

This Control module instances the Instruction Memory, PC, IR, and the State machine accordingly. The Control module requires only 2 inputs: a clock and reset but requires 12 outputs: IR_Out, OutState, NextState, D_Addr, D_Wr, RF_s, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, RF_W_Addr, and Alu_s0. Figure 15 shows the code used to design the Control.

```
//InstructionMemory (address,clock,q);
InstructionMemory Inst(PC_Hold, Clk, IR_In);

//module PC(Clock, Clr, Up, Im_Out);
PC Count(Clk, PC_clr, PC_up, PC_Hold);

//module IR(Clock,Id,instruction,OutFSM);
IR instr (Clk, IR_Id, IR_In, IR_Hold);

//module FSM(Clk,Rst,IR,OutState,OutNext,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);
FSM SM(Clk, Rst, IR_Hold, OutState, NextState, 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);
```

Figure16 : Control Instantiation Code

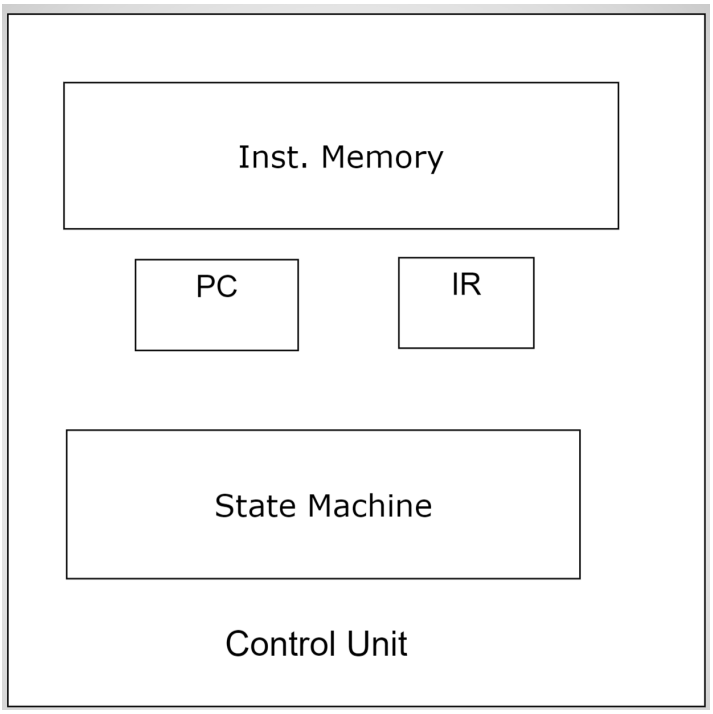

Figure 17: Submodules instanced by the Control

2.5 regfiler16x16a.sv

This submodule consists of 5 inputs and 2 outputs. The inputs include a clock, w write, write address, write data, A-side read address and B side read address. The output is the A side read data and the B side read data. The purpose of this register file is to read and write the controls. Then, the outputs go into the ALU. The register file is instanced by the Datapath. Figure 18 shows the code we used to describe this module.

```
assign Ra_data = regfile[RF_Ra_addr];
assign Rb_data = regfile[RF_Rb_addr];
always @(posedge clk) begin
       if(RF_W_en) regfile[RF_W_addr] <= W_Data;
end
```

Figure 18 : regfiler16x16a code

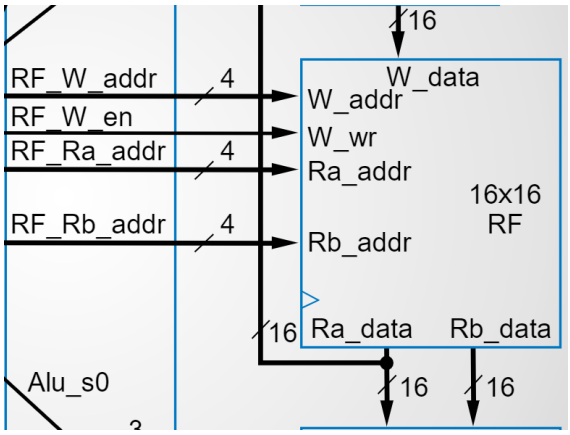

Figure 19: Regfiler16x16a outline

2.6 Mux2to1.sv

This 2 to 1 mux is similar to the one we designed in class. This module consisted of 3 inputs: R_data, ALU_Q, and RF_s along with 1 output: W_data. For this module, we used conditional assignment to complete which is shown in the figure below. The register file is instanced by the Datapath.

```
assign W_data = (RF_s)?R_data:ALU_Q;
```

Figure 20 : Mux2to1 assignment

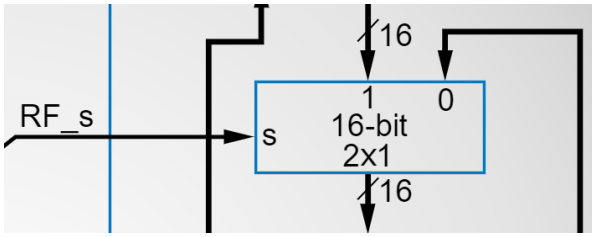

Figure 21: Mux2to1 outline

2.7 ALU.sv

This submodule is the first one we created. The ALU module has 3 inputs and 1 output. For a specific 3 bit input S, the output is an expression consisting of the remaining two 16-bit inputs A and B. We used the figure below to set up the portlist. The register file is instanced by the Datapath. The ALU works as followed: If s == 0 the output is 0. If s == 1 the output is A + B. If s == 2 the output is A –B. If s == 3 the output is A (pass-through). If s == 4 the output is A ^ B. If s == 5 the output is A | B. If s == 6 the

output is A & B. If s == 7 the output is A + 1. Figure 21 shows the code used to implement the ALU.

```
always @(S) begin
        case(S)
                3'b000: Q = 0;
                3'b001: Q = A + B;
                3'b010: Q = A - B;
                3'b011: Q = A;
                3'b100: Q = A ^ B;
                3'b101: Q = A | B;
                3'b110: Q = A & B;
                3'b111: Q = A + 1;
                default: Q = 0;
        endcase
end
```

Figure 22 : ALU code

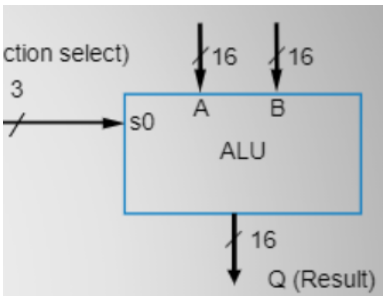

Figure 23: ALU Outline

2.8 Datapath.sv

The Datapath instances the Data Memory, Mux2to1, RegFiler16x16a, and the ALU. The inputs are clk, D_Addr, D_Wr, RF_s, RF_W_Addr, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, and ALU_s0. The outputs are ALU_inA, ALU_inB, and ALU_out. The Datapath is then instanced later by the Processor. The code below shows the instantiation used to complete the Datapath.

```
//DataMemory      (address, clock, data, wren, q);
DataMemory Datamem(D_Addr, clk, RaData, D_Wr, R_data);
//Mux_2_to_1  (R_data, ALU_Q, RF_s, W_data);
Mux_2_to_1 Mux(R_data, Rout, RF_s, W_Data);
//regfile16x16a      (clk, RF_W_en, RF_W_addr, W_Data, RF_Ra_addr, Ra_data, RF_Rb_addr, Rb_data);
regfile16x16a Regfile(clk, RF_W_en, RF_W_Addr, W_Data, RF_Ra_Addr, RaData, RF_Rb_Addr, RbData);
//ALU (A, B, Sel, Q);
ALU Aludata(RaData, RbData, ALU_s0, Rout);
```

Figure 24 : Datapath code

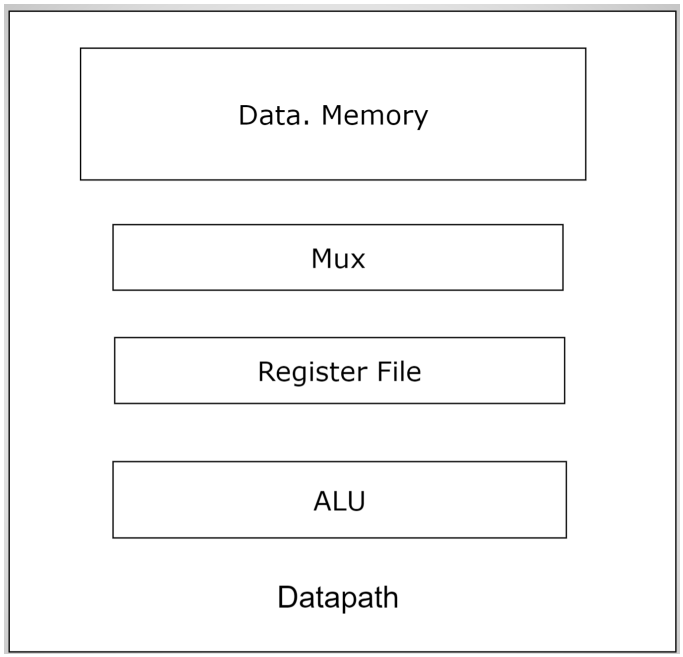

Figure 25: Submodules instanced by Datapath

2.9 Processor.sv

The design of the Processor was simpler because it instantiated the Control and the Datapath which was already designed.The code for the implementation is shown below. The inputs are Clk and Reset. The outputs are IR_Out, PC_Out, State, NextState, ALU_A, ALU_B, and ALU_Out. The first figure shows the code for instancing

```
//Control  (Clk,Rst, 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);
Control con(Clk,Reset, PC_Out, IR_Out, State, NextState, D_Addr, D_Wr, RF_s, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, RF_W_Addr,ALU_s0);

//Datapath   (clk, 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);
Datapath data(Clk, D_Addr, D_Wr, RF_s, RF_W_Addr, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, ALU_s0, ALU_A, ALU_B, ALU_Out);
```

Figure 26 : Processor Instantiation Code

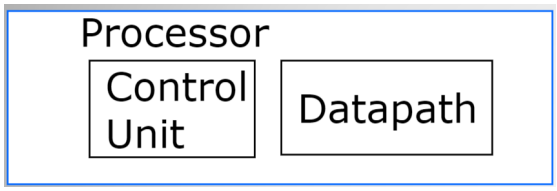

Figure 27: Components of Processor

2.10 Project.sv

The Project module is the top level module. It instantiates the button sync, key filter, processor, MUX 16w 8 to 1, and decoder. The inputs are CLOCK_50, SW, and KEY. The outputs are LEDG, LEDR, HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0. The figure below shows the code used to instance those modules.

```
//ButtonSyncRegReg (Clk, Bi, Bo);
ButtonSyncRegReg sync(CLOCK_50, KEY[2], syncout);

//KeyFilter (Clk, In,Out);
KeyFilter filter(CLOCK_50, syncout, filterout);

//Processor      (Clk, Reset, IR_Out, PC_Out, State, NextState, ALU_A, ALU_B, ALU_Out);
Processor process(filterout, KEY[1], IR_Out, PC_Out, State, NextState, ALU_A, ALU_B, ALU_Out);

//Mux_3w_8_to_1(S2, S1, S0, X1, X2, X3, X4, X5, X6, X7, X8, Y);
Mux_3w_8_to_1 themux(SW[17], SW[16], SW[15], X0, ALU_A, ALU_B, ALU_Out, X4, 16'h0, 16'h0, 16'h0, MUXOut);

//Decoder(C, Hex);
Decoder hex7(MUXOut[15:12], HEX7);
Decoder hex4(MUXOut[3:0], HEX4);
Decoder hex5(MUXOut[7:4], HEX5);
Decoder hex6(MUXOut[11:8], HEX6);
Decoder hex0(IR_Out[3:0], HEX0);
Decoder hex1(IR_Out[7:4], HEX1);
Decoder hex2(IR_Out[11:8], HEX2);
Decoder hex3(IR_Out[15:12], HEX3);
```

Figure 28: Project code

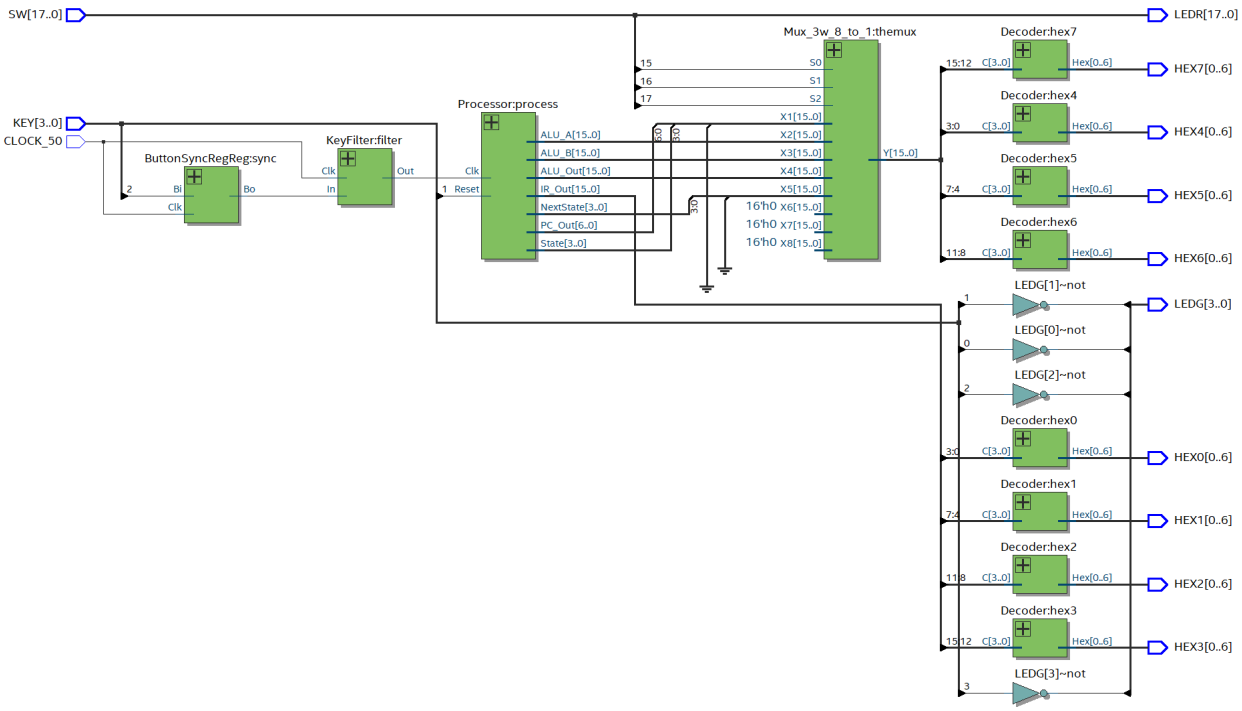

Figure 29: RTL view of the project

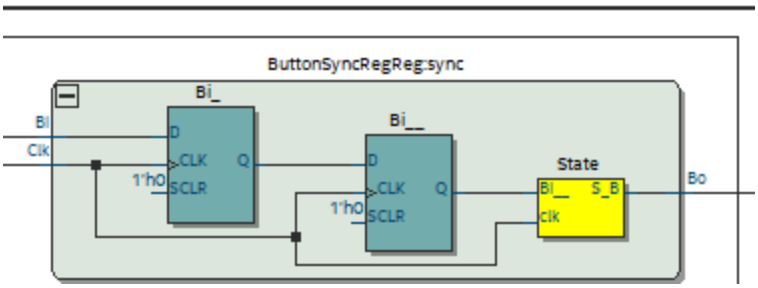

Figure 30: RTL view of ButtonSync

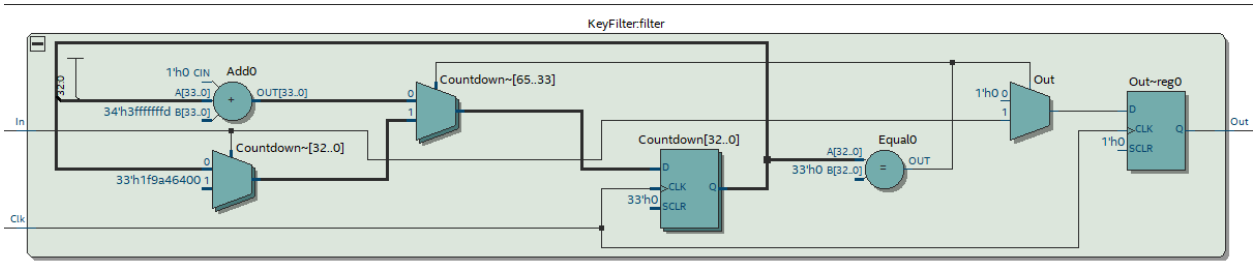

Figure 31: RTL view of Keyfilter

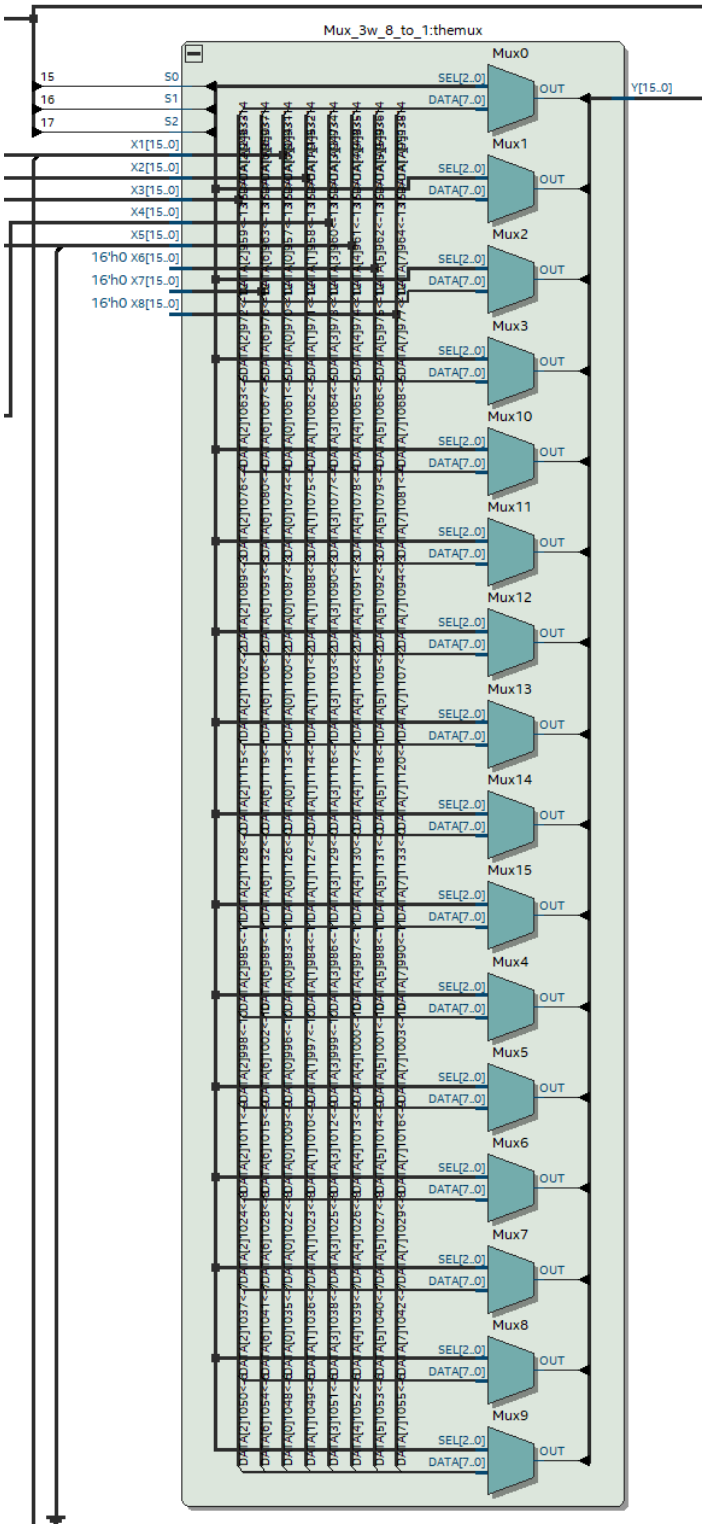

Figure 32: RTL view of mux_16w_8_to_1

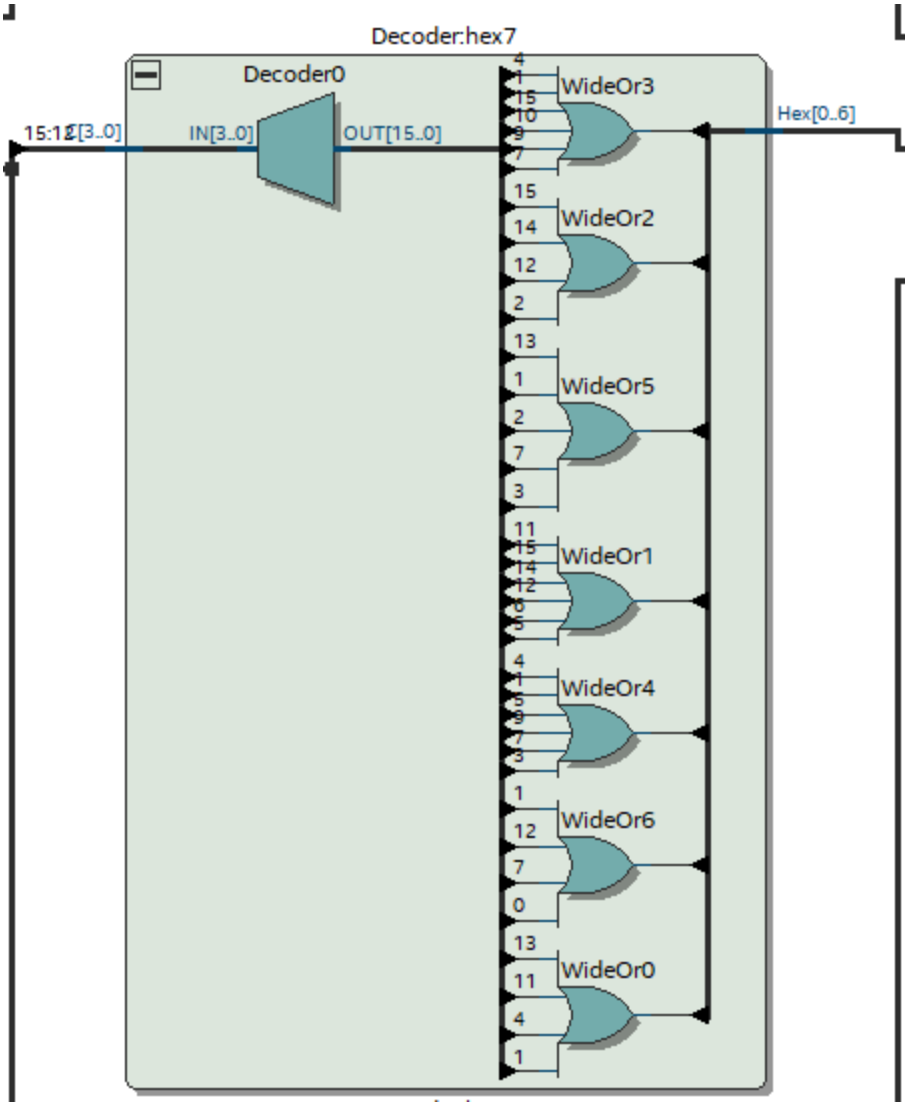

Figure 33: RTL view of decoder

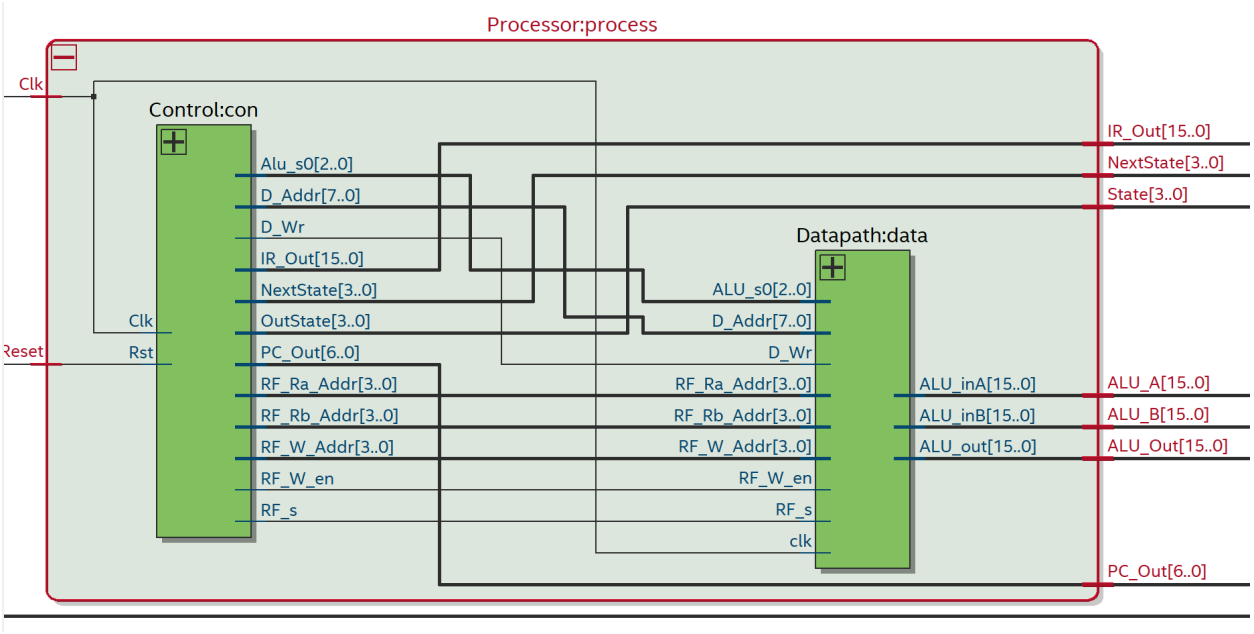

Figure 34: RTL view of Processor

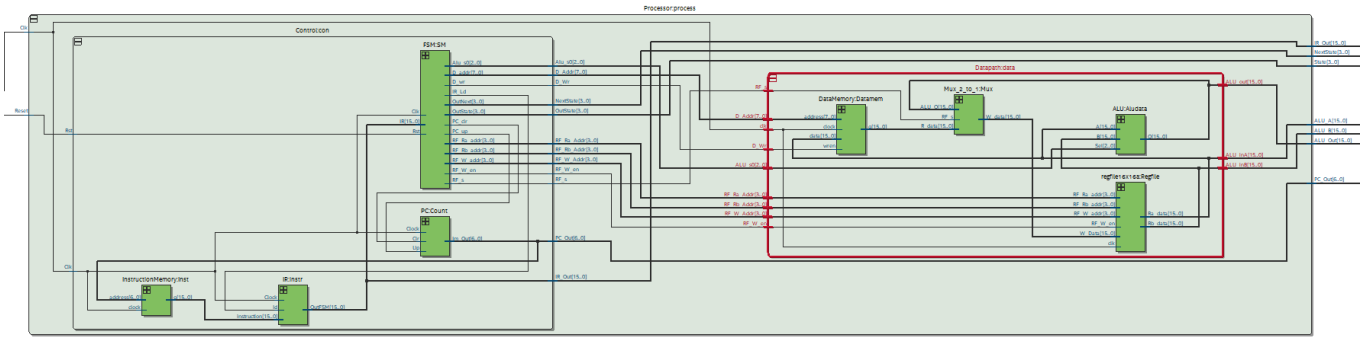

Figure 35: RTL view of processor

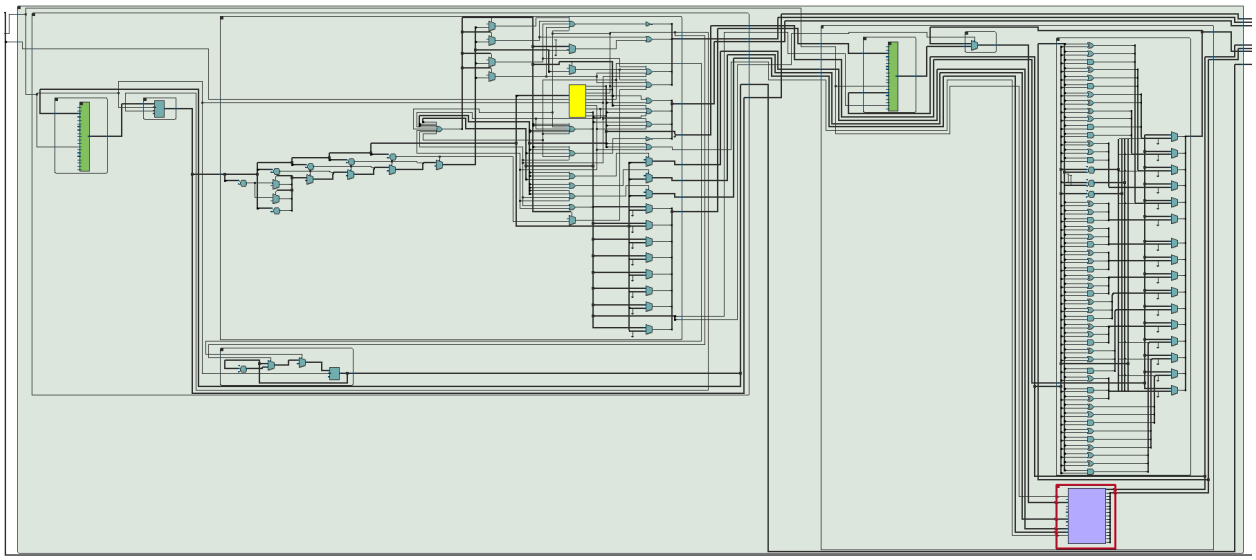

Figure 36: RTL view of processor

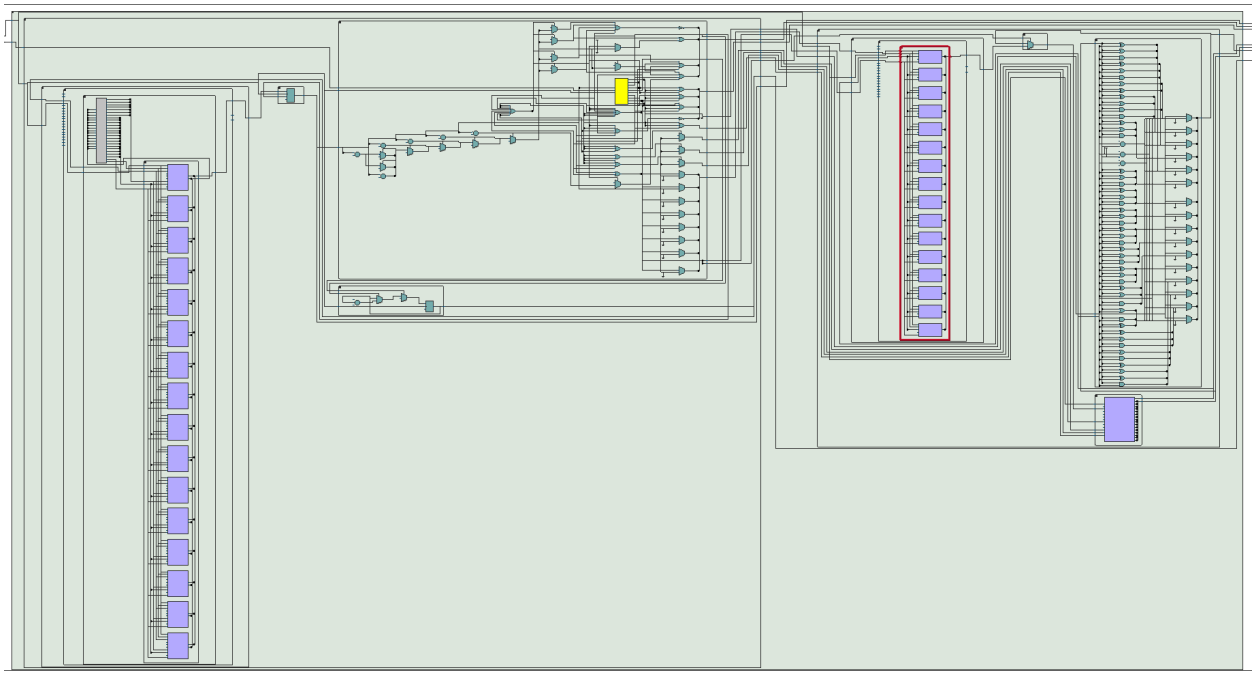

Figure 37: RTL view of processor

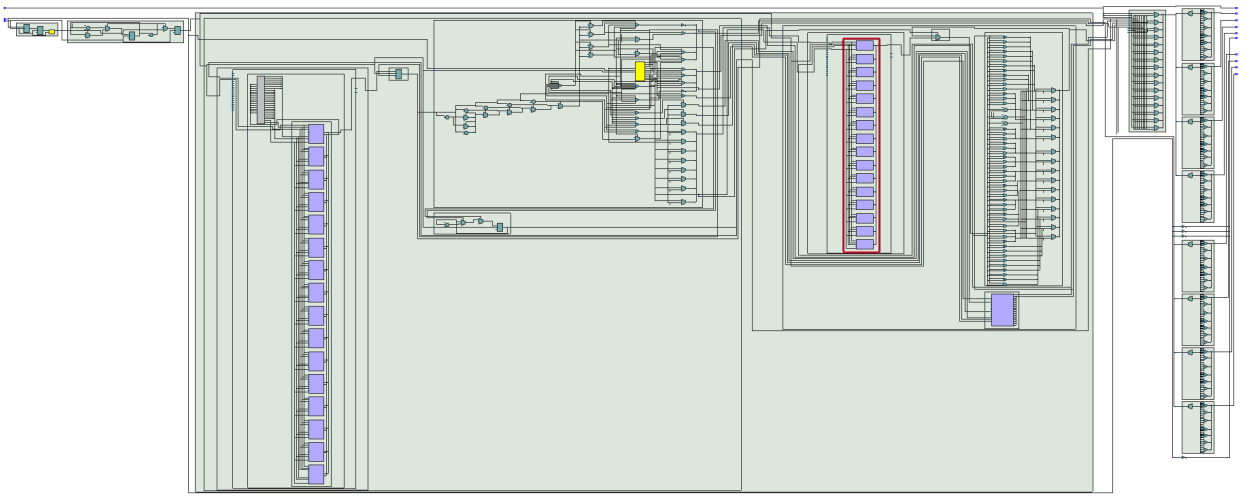

Figure 38: RTL view of Project

3. Test Procedures

This section focuses on the test procedures and the test results of the FSM, Control, Processor, and the Project. By creating our runlab.do, runrtl.do and wave.do, we were able to test the results of each module. Each subsection shows the corresponding wave output of the modules.

3.1 FSM_tb

To test the FSM the testbench sets the input IR's 15th to 12th bit to call each instruction and sets the rest of the bits to random.

```
always begin
        Clk=0; #10;
        Clk=1; #10;
end //always end

initial begin
        Rst = 1; IR[15:12] = 4'b0000; IR[11:0] = $urandom(); #63;
                 IR[15:12] = 4'b0001; IR[11:0] = $urandom(); #60;
                 IR[15:12] = 4'b0010; IR[11:0] = $urandom(); #90;
                 IR[15:12] = 4'b0011; IR[11:0] = $urandom(); #60;
                 IR[15:12] = 4'b0100; IR[11:0] = $urandom(); #60;
                 IR[15:12] = 4'b0101; IR[11:0] = $urandom(); #60;
        Rst = 0; #30;
        $stop;
end
```

Figure 39: FSM testbench

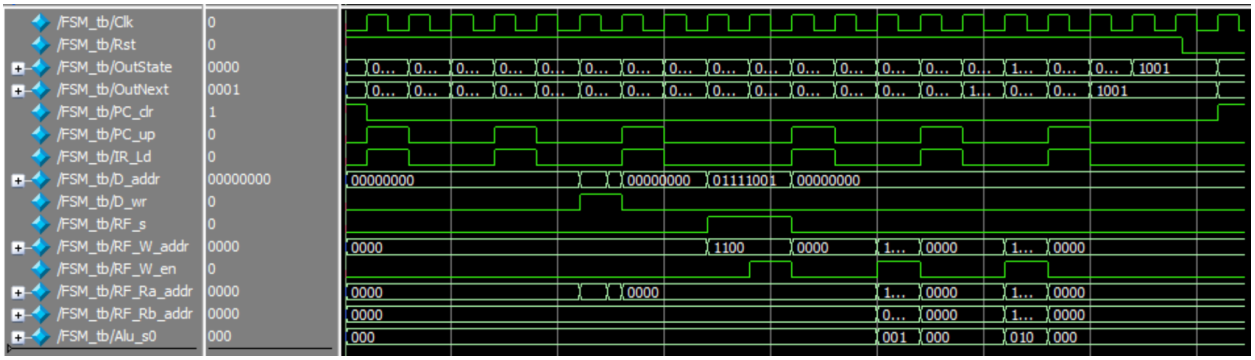

Figure 40: FSM wave output

3.2 Controller_tb

To test the controller the testbench sets the clock and reset to 1 then 0. While the reset is 1 the module goes through the necessary instructions from the instruction memory.

```
//clock
always begin
        Clk = 0; #10;
        Clk = 1; #10;
end
initial begin
        Rst = 1; #810;
        Rst = 0; #40;
        $stop;
end
```

Figure 41: Control testbench

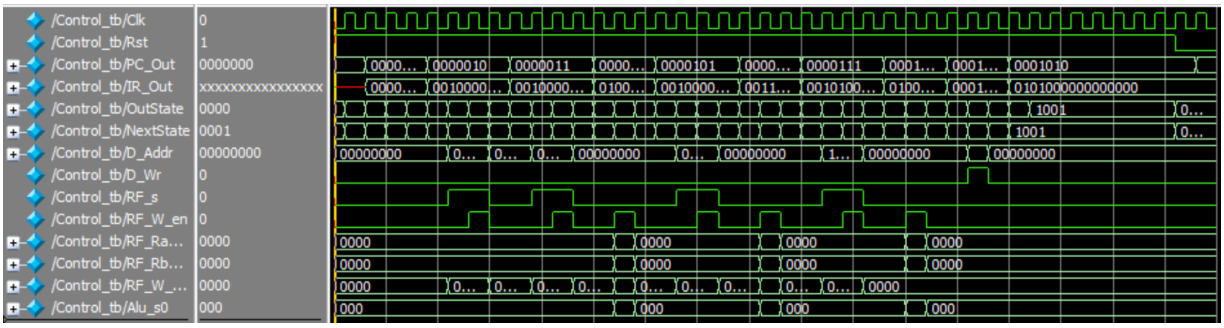

Figure 42: Control wave output

3.3 Datapath_tb

To test the datapath the testbench sets the D_Addr, D_Wr, RF_s, RF_W_Addr, RF_W_en, RF_Ra_Addr, RF_Rb_Addr, and ALU_s0 to execute the necessary instructions.

```
always begin
        clk = 0; #10;
        clk = 1; #10;
end
initial begin
        //load Reg[1] = Dmem[0B] = 52229 = CC05         0B=11
        D_Addr = 8'h0B; D_Wr = 1'b0; RF_s = 1'b1; RF_W_Addr = 4'd1; RF_W_en=1'b1; RF_Ra_Addr = 4'd1; RF_Rb_Addr = 4'd1; ALU_s0=3'b000; #120;
        //load Reg[2] = Dmem[1B] = 437 = 01B5           1B=27
        D_Addr = 8'h1B; D_Wr = 1'b0; RF_s = 1'b1; RF_W_Addr = 4'd2; RF_W_en=1'b1; RF_Ra_Addr = 4'd2; RF_Rb_Addr = 4'd2; ALU_s0=3'b000; #120;
        //load Reg[4] = Dmem[06] = 4268 = 10AC          06=6
        D_Addr = 8'h06; D_Wr = 1'b0; RF_s = 1'b1; RF_W_Addr = 4'd4; RF_W_en=1'b1; RF_Ra_Addr = 4'd4; RF_Rb_Addr = 4'd4; ALU_s0=3'b000; #120;
        //load Reg[5] = Dmem[8A] = 41024 = A040         8A=138
        D_Addr = 8'h8A; D_Wr = 1'b0; RF_s = 1'b1; RF_W_Addr = 4'd5; RF_W_en=1'b1; RF_Ra_Addr = 4'd5; RF_Rb_Addr = 4'd5; ALU_s0=3'b000; #120;
        //sub Reg[3]=Reg[1] - Reg[2]    51792 = 52229 - 437
        D_Addr = 8'h00; D_Wr = 1'b0; RF_s = 1'b0; RF_W_Addr = 4'd3; RF_W_en=1'b1; RF_Ra_Addr = 4'd1; RF_Rb_Addr = 4'd2; ALU_s0=3'b010; #120;
        //add  Reg[6]=Reg[4] + Reg[5]    45292 = 4268 + 41024
        D_Addr = 8'h00; D_Wr = 1'b0; RF_s = 1'b0; RF_W_Addr = 4'd6; RF_W_en=1'b1; RF_Ra_Addr = 4'd4; RF_Rb_Addr = 4'd5; ALU_s0=3'b001; #120;
        //add Reg[0]=Reg[3] + Reg[6]     97084 = 51792 + 45292
        D_Addr = 8'h00; D_Wr = 1'b0; RF_s = 1'b0; RF_W_Addr = 4'd0; RF_W_en=1'b1; RF_Ra_Addr = 4'd3; RF_Rb_Addr = 4'd6; ALU_s0=3'b001; #120;
        //store D[CD]=Reg[0] = 97084             CD=205
        D_Addr = 8'hCD; D_Wr = 1'b1; RF_s = 1'b0; RF_W_Addr = 4'd0; RF_W_en=1'b0; RF_Ra_Addr = 4'd0; RF_Rb_Addr = 4'd0; ALU_s0=3'b000; #120;
        $stop;
end
```

Figure 43: Datapath testbench

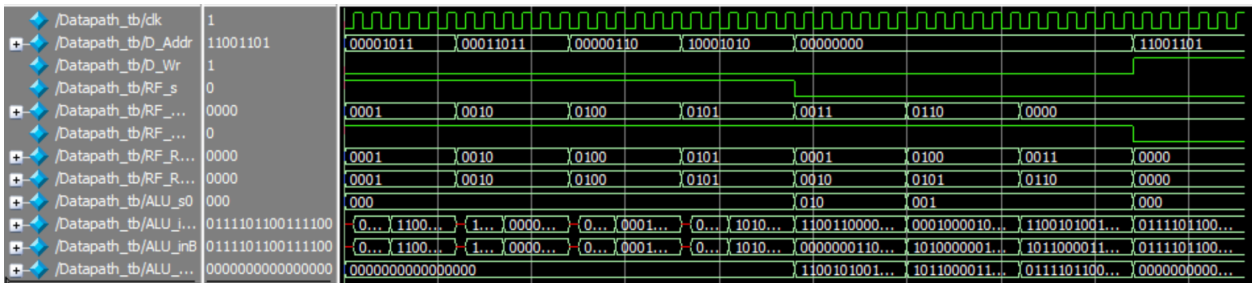

Figure 44: Datapath wave output

3.4 testProcessor

To test the processor the test processor module sets the clock and reset while the processor runs through the instructions manually set in the instruction memory until IR Out reaches 0X5000 which is the halt state

```
begin
  $display( "\nBegin Simulation." );
  Reset = 0;          // reset for one clock
  @ ( posedge Clk )
  #10 Reset = 1;
  wait( IR_Out == 16'h5000 ); #10; // halt instruction
  $display( "\nEnd of Simulation.\n" );
  $stop;
end
```

Figure 45: testProcessor

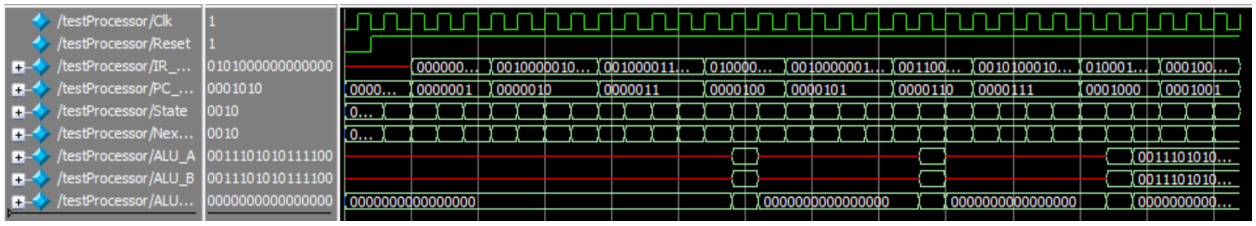

Figure 46: testProcessor wave output

4. Test Results

The simulation results of the PC, IR, regfilier8x16a, Mux2to1 and ALU are shown below. Each module has a corresponding wave output that describes the circuit. We created run.do and wave.do files for each module to test them. The sections below describe the results we achieved through the waves.

4.1 PC.sv

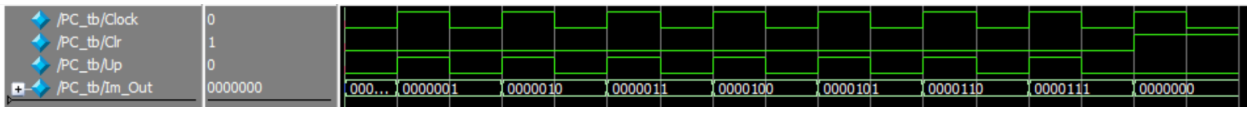

Figure 43::PC testbench

4.2 IR.sv

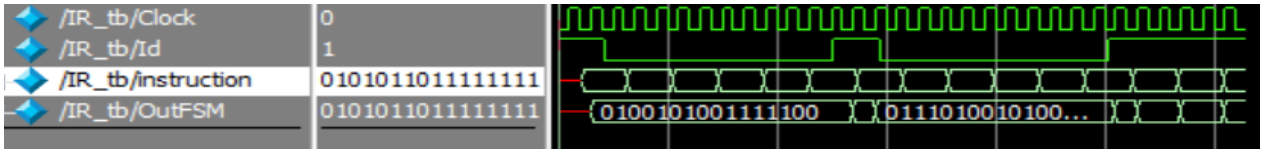

Figure 44:IR testbench

4.3 regfiler8x16a.sv

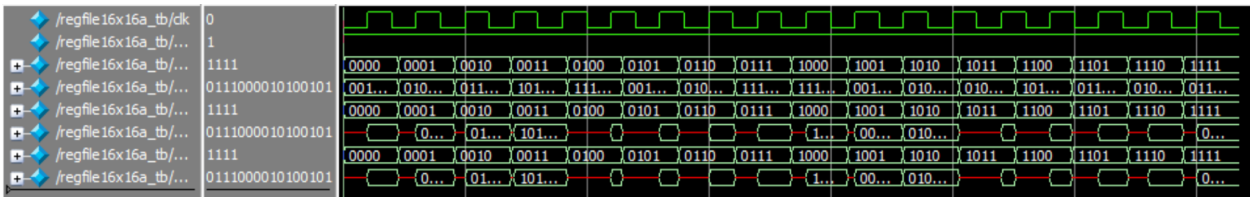

Figure 45: regfiler8x16a testbench

4.4 Mux2to1.sv

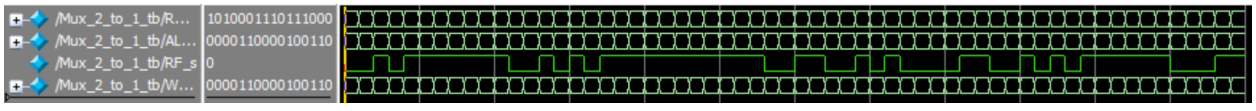

Figure 46: Mux2to1 testbench

4.5 ALU.sv

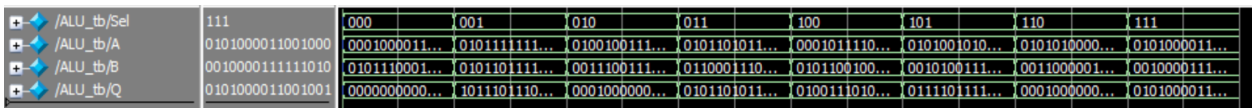

Figure 47: ALU testbench

5. Observations

This project was complicated because there were alot of submodules needed to be created and instanced. There were also a lot of small problems we faced when creating each module. Each problem took a fair amount of troubleshooting. The description on some of the submodules weren't very clear. We were able to create the ALU, IR, PC, regfilier16x16, and datamemory modules without a problem. However, we began to have some issues with the FSM because the template given had an extra line (NextState=Next) that wasn't necessary and because of that, our output was incorrect. We also had our output states in the wrong location. We had a problem with our processor too. While it was a simple module that instantiates the datapath and control it wasn't working. We spent hours trying to figure it out with the professor, the control and datapath worked correctly separately but not together in the processor. The problem was the clock in the instruction memory module had to be changed to UNREGISTERED. Our last major problem was in the project module with the altera clock. Without creating an altera clock in the sdc file there was an error and our project would not compile. When we did create the clock there was a warning saying the altera clock was being overwritten and the project did not work correctly. With the professors help we discovered that we had to comment out the altera clock but keep the altera clock's uncertainties.

We were able to instance the submodules in the datapath but we had issues with the Control because of the Instruction memory. Our size of the A.mif was wrong and because of that, the Control wasn't able to compile and run. We had some difficulties understanding the differences between the Instruction Memory and the Data Memory because there wasn't much clarification so we spent a good amount of our time trying to figure that out. We were also unsure as to what instructions to manually input into the A.mif. Despite there being lab and office hours almost every day, it was still difficult to ask simple questions because almost every other group had questions of their own and time was limited for those sessions. We would end up waiting almost an hour every time just to ask a question. Because of that, we spend more time on the project than need be. We started the project on May 13th and we have worked on it almost everyday for multiple hours including the weekends.While the project did take a lot of time and effort we believe we have done our best work.

6. Conclusion

To conclude, the Simple Processor project consisted of a Processor that instanced 2 submodules (Control and Datapath) and those 2 submodules instances 4 additional modules each. (Instruction Memory, PC, IR, FSM, DataMemory, regfiler8x16a, Mux2to1, ALU). In total, we created 11 modules and 1 upper level module . We tested each module as well and were able to verify them by observing the waves. The upper level module Project instanced the Processor, decoder, ButtonSyncregreg, Keyfilter, and Mux_3w_8_to_1. Overall, this project was a bit tedious and difficult to do in the amount of time given. Throughout the weeks of the project we had other classes with a fair amount of homeworks and a midterm. Through the process of this project we have learned a lot about modelsim, systemverilog, and working together in a group.

Demo video:
https://drive.google.com/file/d/1u8MZtb8KOUeJwVSamV5cuQmNx7RJVo65/view?usp=sharing