Course Project

Programmable Processor

TCES330 Digital System Design

Spring 2022

**Authors:** Jisu Park



Submission Date: June 5, 2022

**Table of Contents:**

1: Requirements \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 3

1.1 Controller.sv

1.2 Datapath.sv

1.3 Processor.sv

1.4 Project.sv

2: Design ­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 4

2.1 FSM.sv

2.2 PC.sv

2.3 IR.sv

2.4 Controller.sv

2.5 regfiler8x16a.sv

2.6 Mux2to1.sv

2.7 ALU.sv

2.8 Datapath.sv

2.9 Processor.sv

2.10 Project.sv

3: Test Procedures \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 13

3.1 FSM\_tb.sv

3.3 Controller\_tb.sv

3.4 Datapath\_tb.sv

3.2 testProcessor.sv

4: Test Results \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 17

5: Observations \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 17

6: Conclusion \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ page number 17

Appendix

The purpose of this project was to build a simple processor. I divided the workload as follows Datapath module and Controller module.

1. **Requirements**

The simple processor is for handling multiple additions and subtractions. The data are stored in the Datapath memory (RAM). The instructions are stored in the Controller memory (ROM).

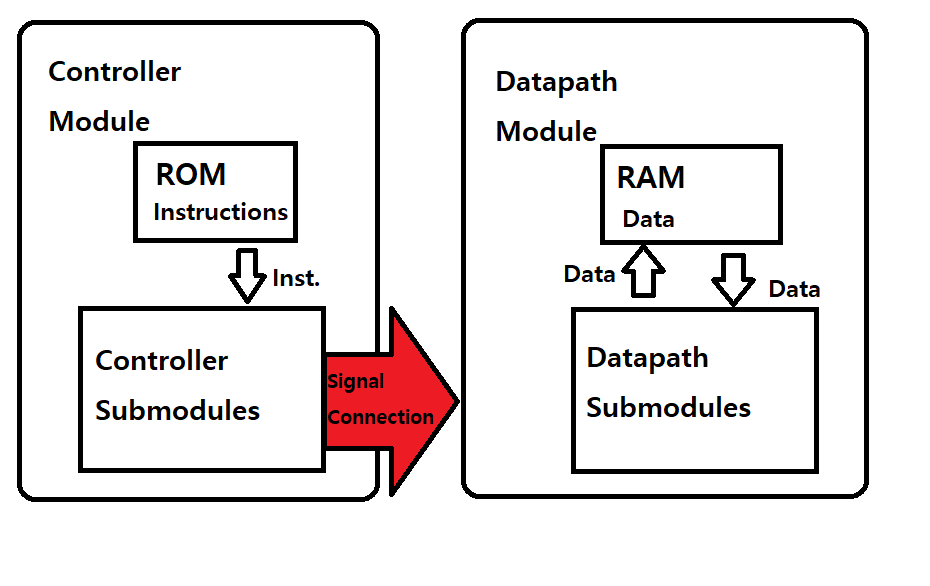
The first hexadecimal digit of an instruction code is the operation code, while the other bits are the destination or source addresses in the RAM.

Figure 1. The Processor module basic structure diagram. RAM can load and store data, and ROM can only load instructions. The controller module and Datapath module are connected with signal connections.

* 1. **Requirements for Controller.sv**

The controller module understands the instructions in the ROM, converts them to signals, and sends them to the datapath module. The controller module only takes clock and reset inputs. The ROM already contains instruction lines and only sends a current instruction line to other controller submodules.

* 1. **Requirements: Datapath.sv**

The datapath module stores, loads, and calculates data in the RAM based on the signal from the Controller module. There is preset data already in the RAM. However, I should be able to store data to RAM from RF or load data to RF from RAM.

* 1. **Requirements: Processor.sv**

The processor module is the top-level module of the controller module and the datapath module. It connects those modules. The outputs are the current address of the instruction memory (PC), current state, next state, and current instruction line from the controller module. It also sends out the result of the arithmetic logic unit, the value of arithmetic logic unit input A and B from the datapath module.

* 1. **Requirements: Project.sv**

The project module installs and connects the processor module on the DE2 bot. It has a button synchronizer and a key filter module to regulate the clock input signal. It also has an 8 to 1 mux module to display the selected output of the processor module to decoder modules wired with hex displays. The filtered clock input signal is the clock input for the processor module.

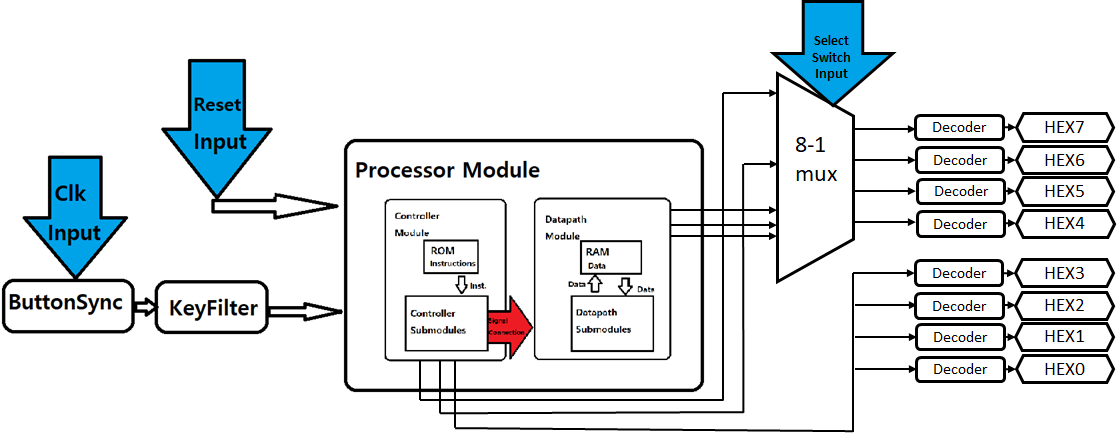


Figure 2. In the project's basic structure diagram, the blue arrows are the input from the switches. HEX 7~0 is the output to hex displays in hexadecimal value.

**2. Design**

In this section, I explain the details of each module. The controller module includes finite state machine(SM.sv), 0~127 program counter(PC.sv), and 16-bit instruction register (IR) modules. The Datapath module has 16x16 Register file(RF.sv), 16bit 2 to 1 mux(Mux\_16w\_2\_to\_1.sv), and arithmetic logical unit(ALU.sv) modules

**2.1 SM.sv**

This finite state machine takes a 16-bit instruction input from the IR module, a clock input, and a rest input, and it starts with all outputs at 0 (see Figure 3). Figure 4 shows the flow of the finite state machine. Figure 5 shows the corresponding state to the instruction operation code and output.

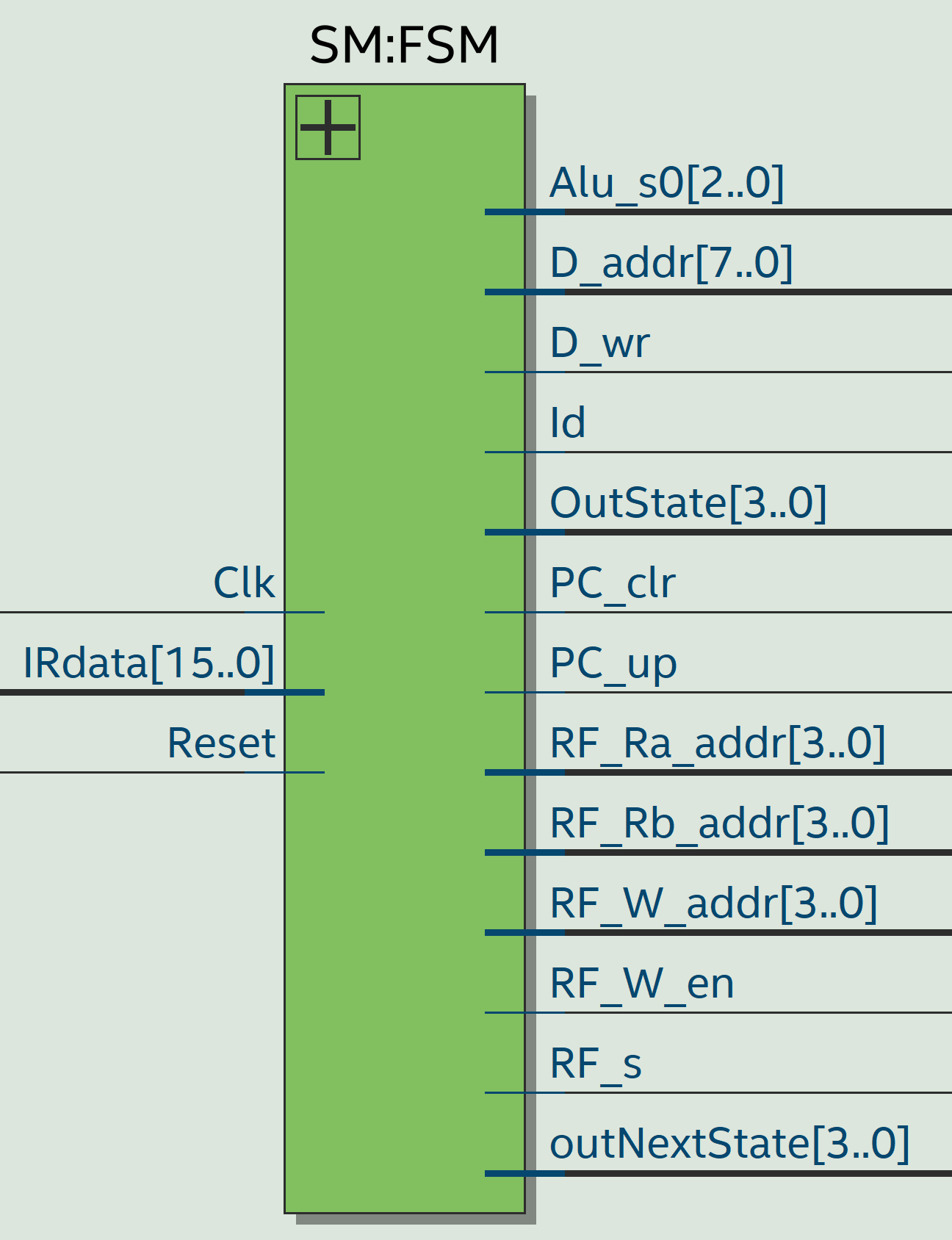


Figure 3. The input and output of SM.sv

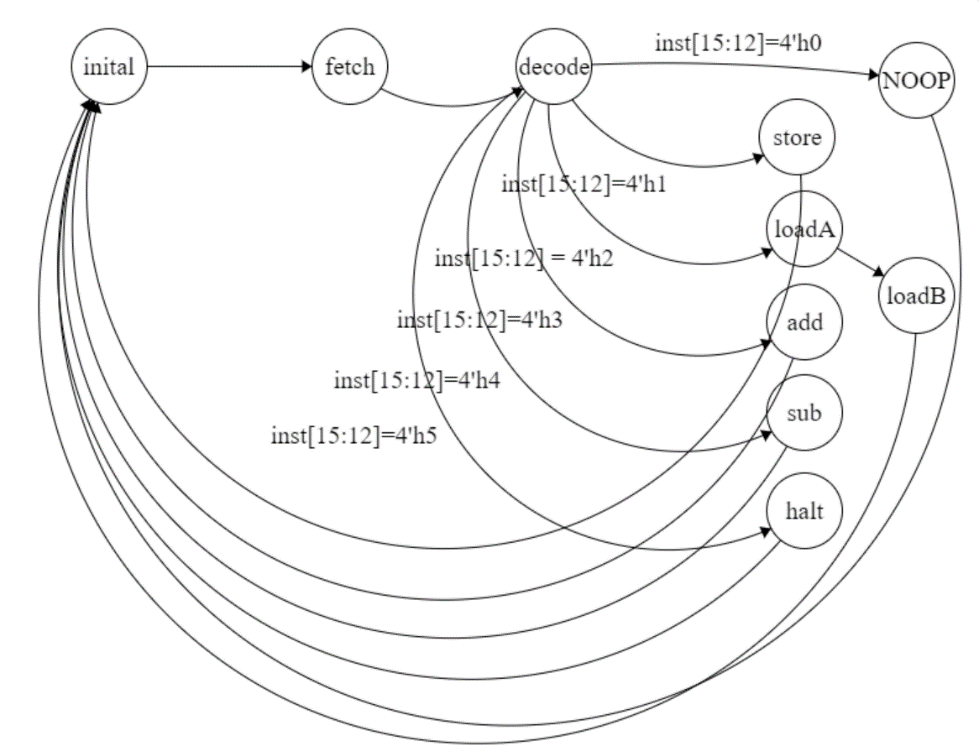


Figure 4. The finite state machine diagram. It proceeds to the next state when a positive edge clock happens.

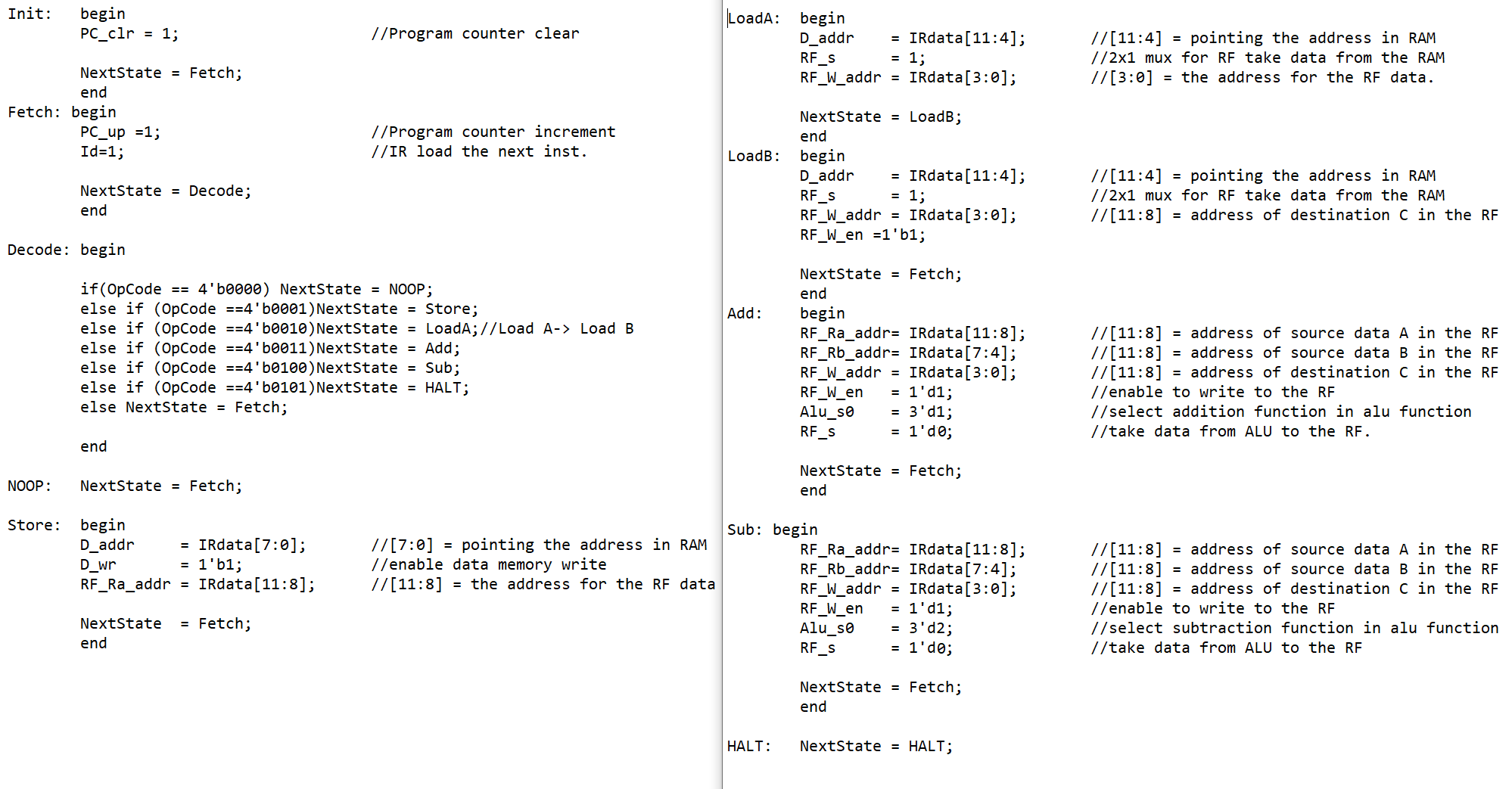


Figure 5. The case statements show the output of each state in the SM.sv file. All outputs’ default value is 0.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Figure 6. The list of the state variables in the SM.sv. These numbers will be displayed on the FPGA 7-segment display according to the current or next state.

**2.2 PC.sv**

The program counter counts the number of instructions processed. It is purely a 128 counter which takes 1-bit input from the SM module. The output is connected to the address of the ROM.

**2.3 IR.sv**

The instruction register module takes a 16-bit instruction or holds the instruction depending on the 1-bit signal from the SM module. The output is the 16-bit instruction currently saved.

**2.4 Controller.sv**

The controller module takes a clock input and a reset input. This module contains a ROM module with instruction lines stored; thus, I don’t need to put any other inputs (see Figure 7). Figure 8 is the diagram of the controller module. All of the submodules and Controller module changes output at the positive edge.

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 7. The data in myROM.mif. I explain saved instructions in Appendix A.

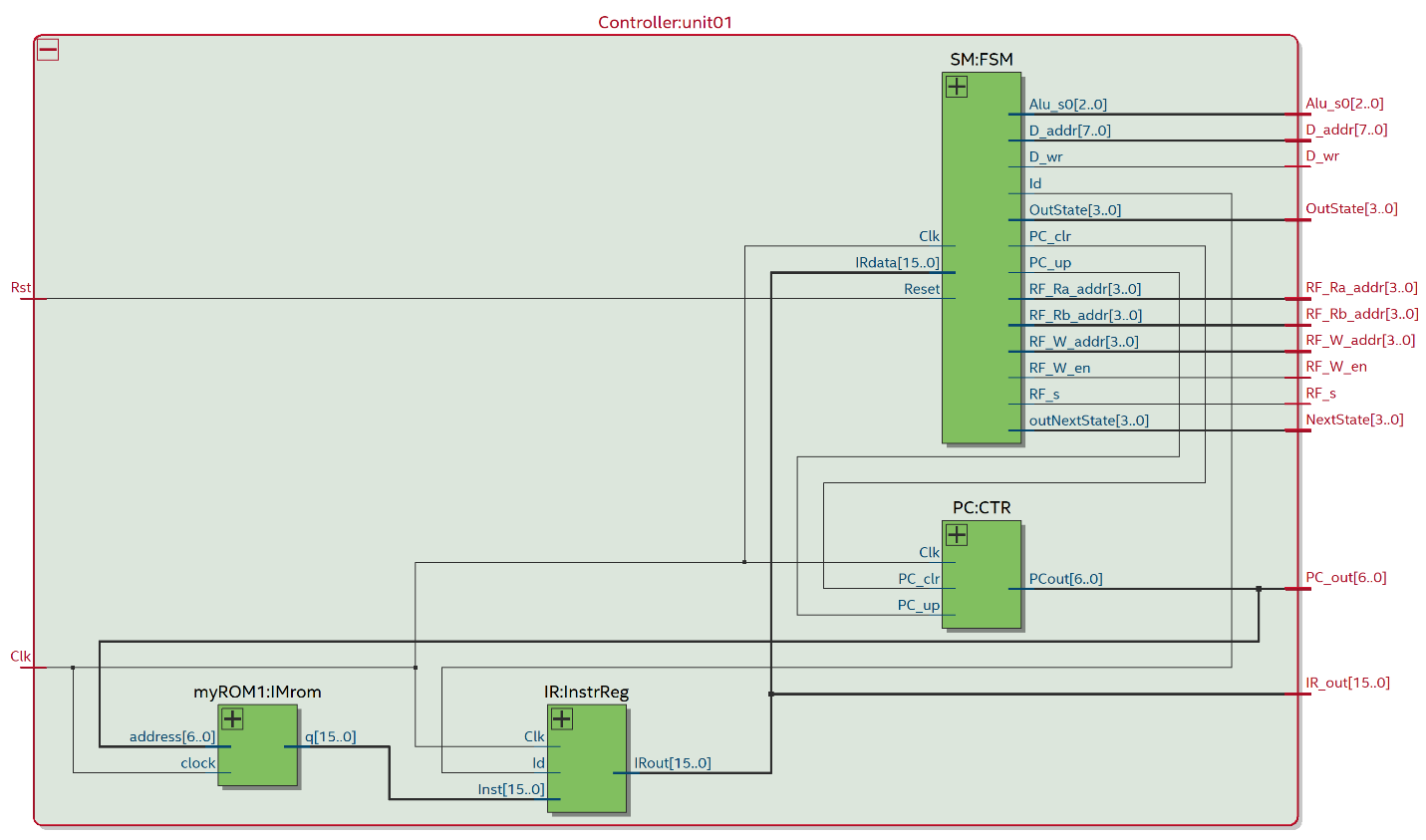


Figure 8. The RTL view of Controller module with Quartus. This shows the inputs, the outputs to the datapath module, and the internal connections between submodules.

**2.5 RF.sv**

   I built a 16x 16 bits register file module. It can hold up to 16 words of 16 bits. Hence, it has 4-bit address input. It also has a 1-bit select input signal, which determines whether to save the 16-bit input data from the 16-bit 2-1 mux or hold the data. Finally, it sends 2 of 16 bits of data as output to ALU.sv. Finally, it takes 2 data address inputs for those addresses of the outputs saved in the RF (see Figure 9).

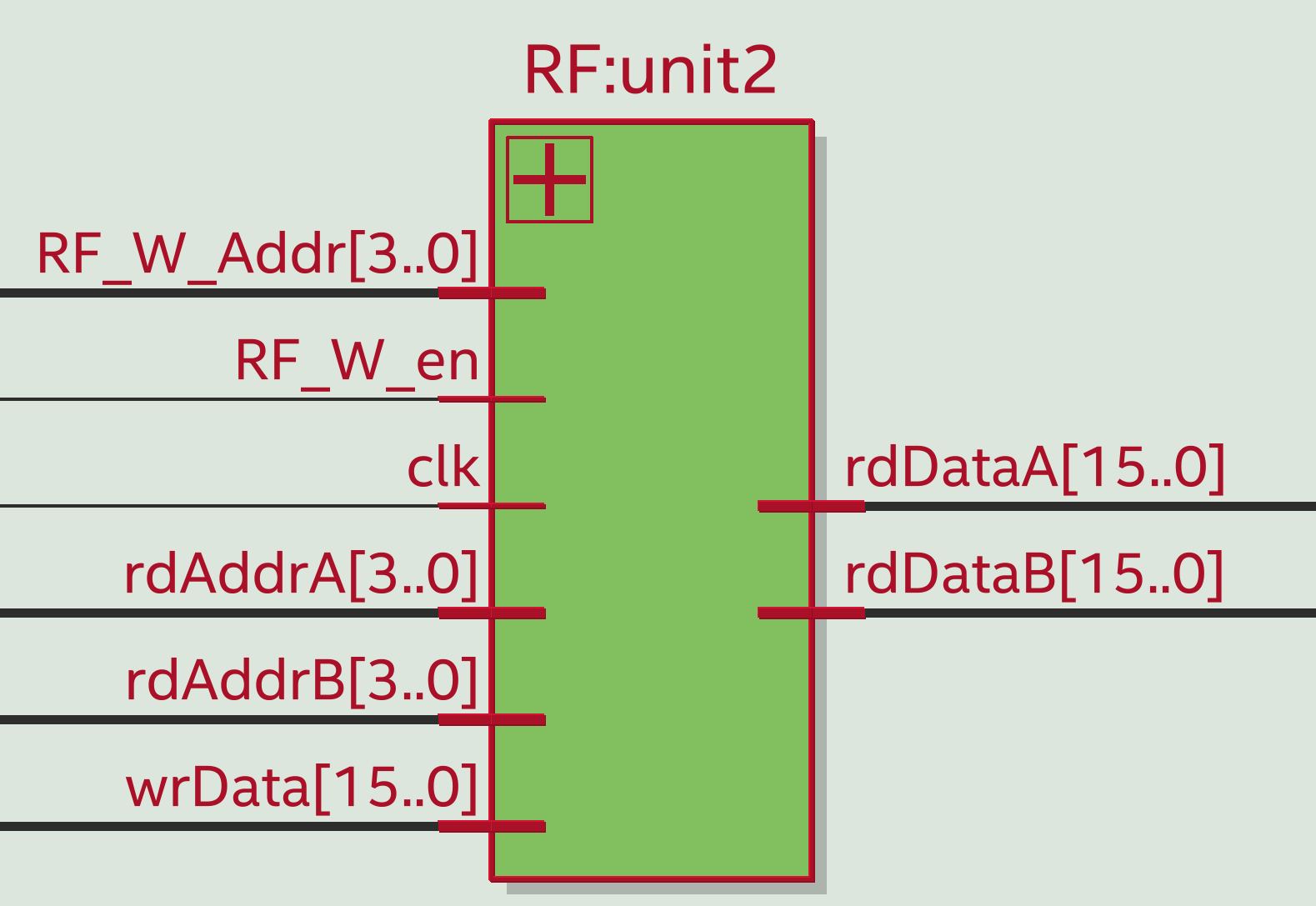


Figure 9. The diagram of RF.sv. RF\_W\_Addr is a target address to write in RF. Data is the data to the target address. RF\_W\_en is for write enable. Clk is the clock. rdAddrA and rdAddrB are selecting output data A and data B.

**2.6** **Mux\_16w\_2\_to\_1.sv**

This Mux has the input data and the output as 16 bits, and the select bit as 1-bit. The input data is from the RAM or ALU result. The outcome is for writing data in RF.sv(see Figure 10).

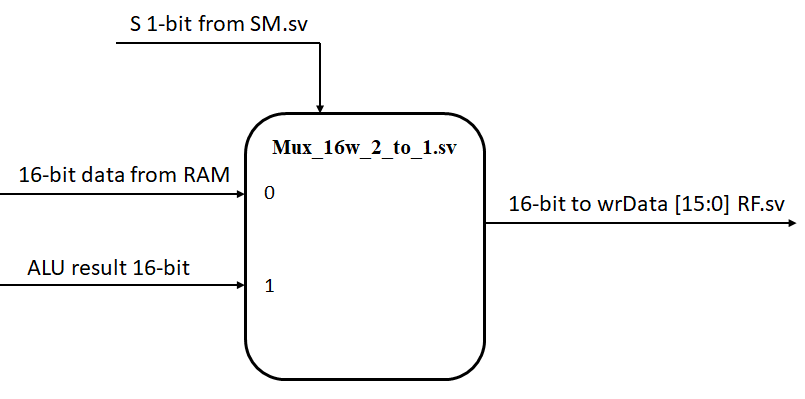


Figure 10. This is the diagram of the Mux\_16w\_2\_to\_1.sv. If the select bit is 1’b0, the output data will be from the RAM. When the chosen bit is 1’b1, the output data is the ALU result.

**2.7 ALU.sv**

This simple processor has addition and subtraction operations. The output of the subtraction operation is a bigger number subtracting a smaller number. The addition operation adds two inputs and returns the result as output.

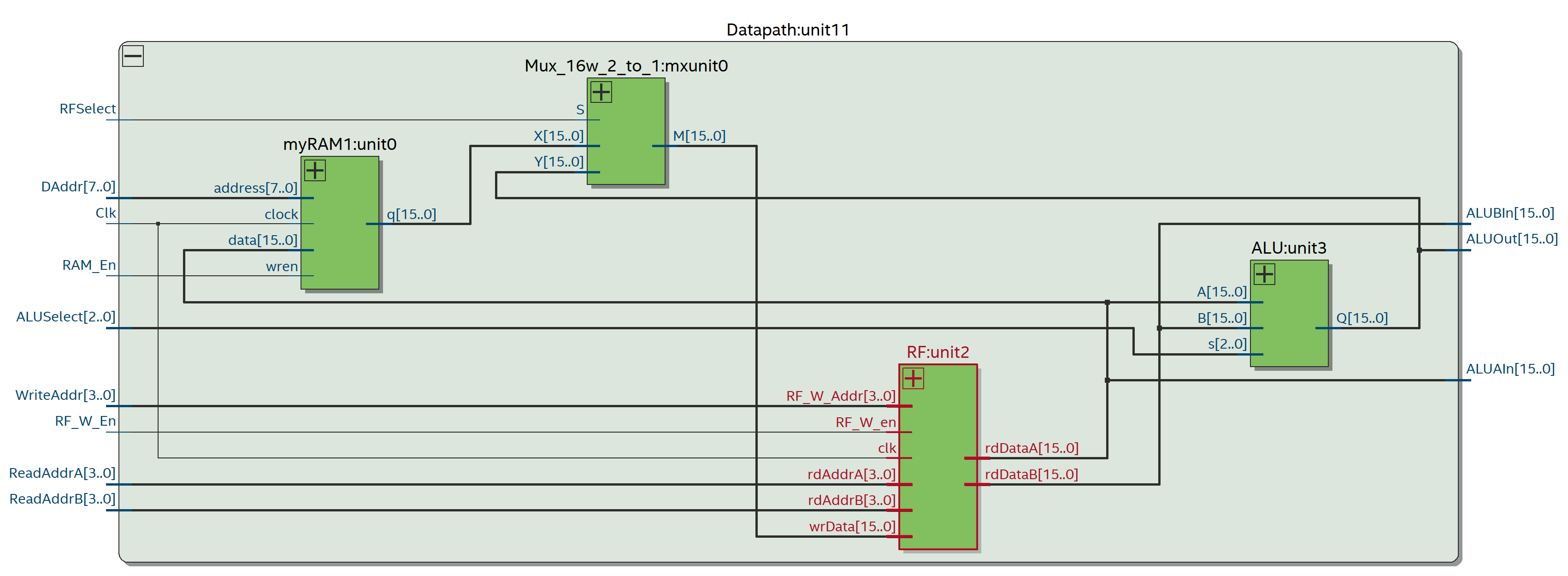
**2.8 Datapath.sv**

For the datapath, the challenge was connecting all the appropriate modules and feeding the correct wires to each to obtain the best possible output to fit the specification of the projectWhen instantiating our RAM module, I needed to concatenate the 4-bit incoming data to provide the RAM’s 16-bit port so the resulting in the following instantiation, *myRAM1 unit0( DAddr, Clk, {12'h0,ReadAddrA}, Write\_En, RDataOut )*. I used six wires for our datapath module to appropriately pass the necessary values to each submodule. I used assignment statements for ALU’s outputs ALUOut, ALUAIn, and ALUBIn (see Figure 12). This was an essential step since the values of the ALUAIn and ALUBIn are provided by the Register File module’s outputs, and the ALUOut is used for the second input to the 2-to-1 MUX.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Figure 11. This is the input and output of the datapath module.

Figure 12. The datapath module diagram shows the internal wire connections between the submodules.

**2.9 Processor.sv**

The processor module has a clock input and a reset input. Outputs are the addition or subtraction data, the current and following states, the current instruction code, and the program counter number (see Figure 13). The processor module has the controller module and the datapath module as submodules. The processor module connects those modules (see Figure 14).

텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 13. The list of input and output of Processor.sv.

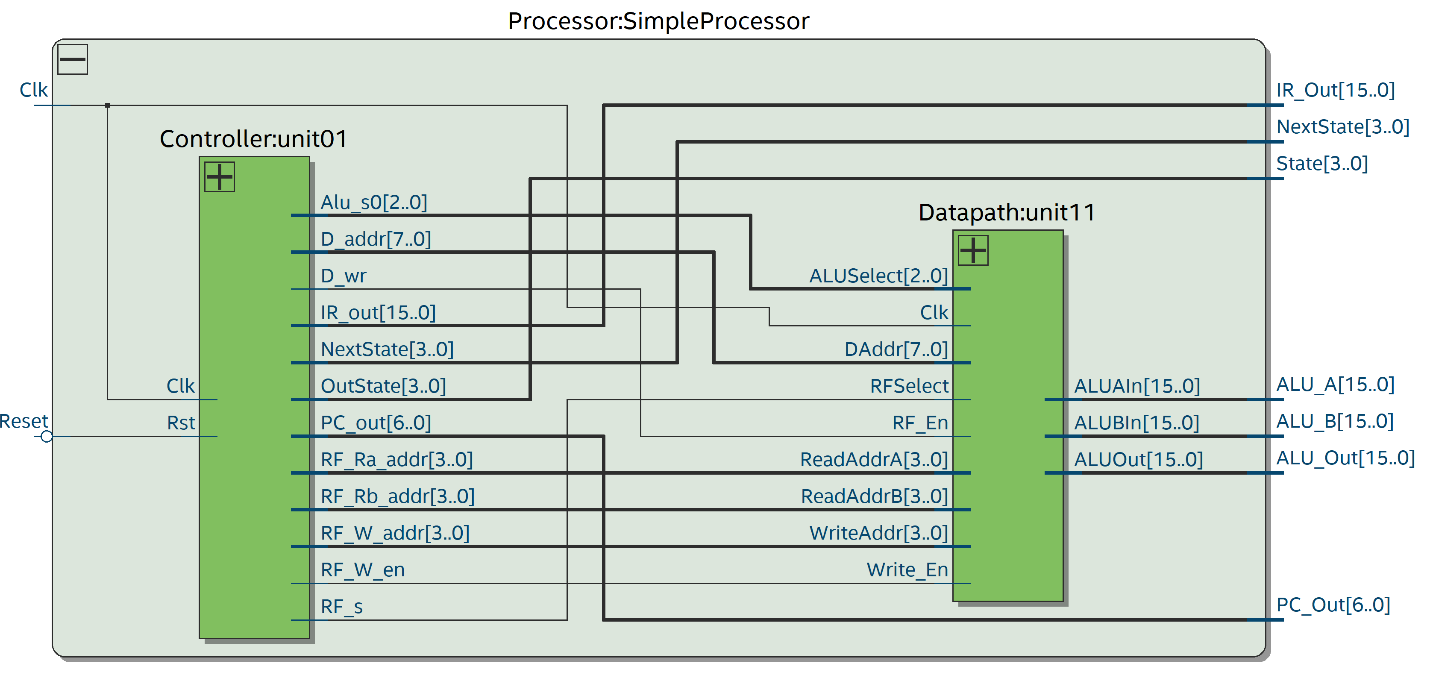


Figure 14. Processor module internal diagram.

**2.10 Project.sv**

Project.sv takes the clock and reset input from the user, processes it, and returns the output on the hex displays and LEDs. Project.sv includes a Processor module, Button Synchronizer module, Key Filter module, Hex Decoder module, 8 to 1 Mux module, and its submodules (see Figure 15). Button Synchronizer and Key Filter regulate the input clock signal to make the circuit process as I designed. The hex display 3~0 always shows the current instruction. 8-1 Mux takes inputs from switches 17,16 and 15. Finally, it displays selected outputs on hex display 7~4 (see Figure 16).

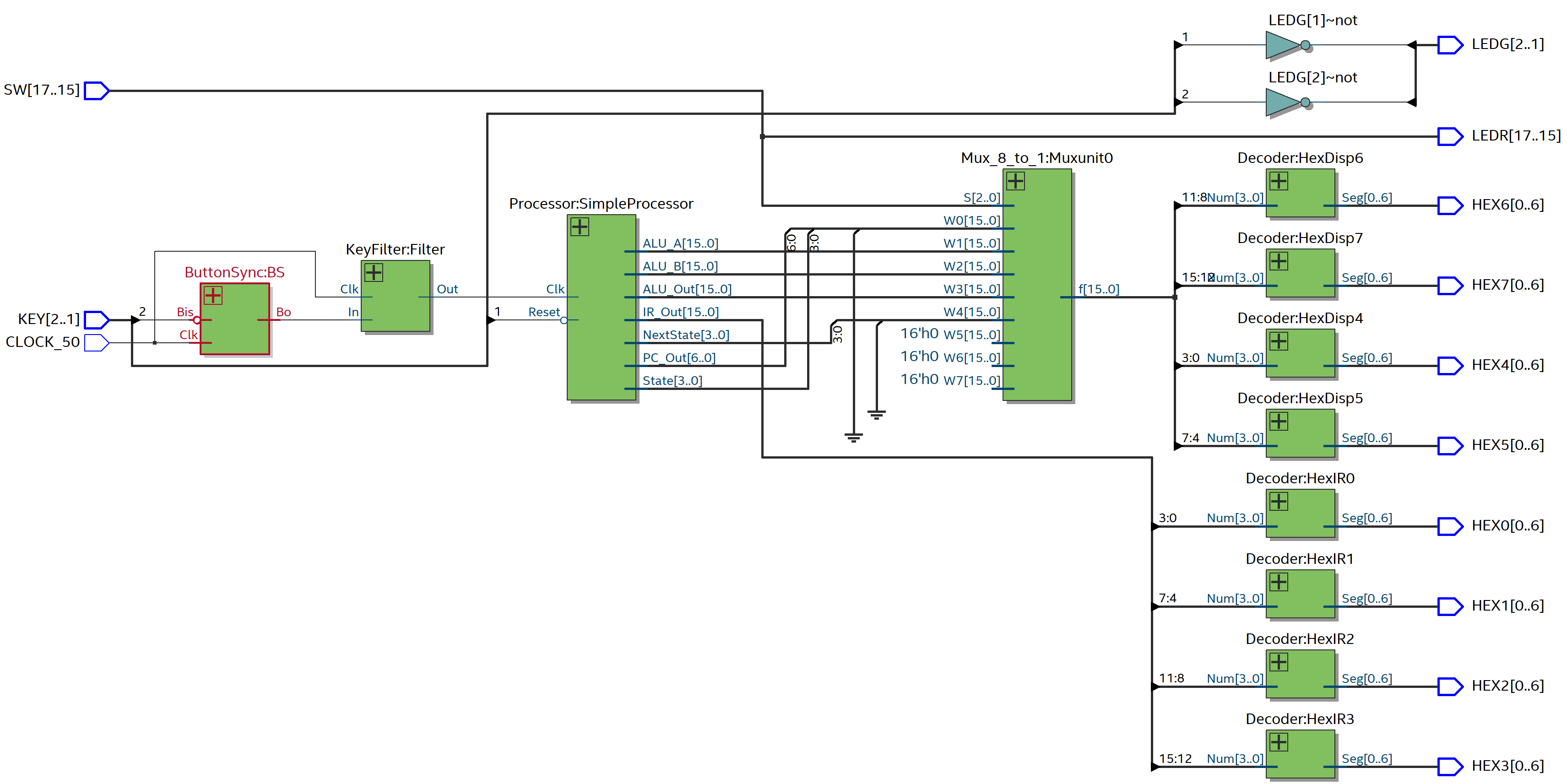


Figure 15. The diagram of the project module. Key 2 is clock input which the user creates. Key 1 is the reset input by the user. the DE2 bot automatically generates CLOCK\_50 which is the 50Mhz clock

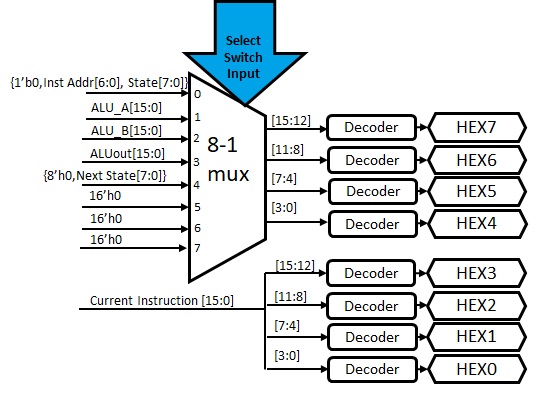


Figure 16. Detailed diagram of the displayed hex and 8-1 Mux. This shows the input data and the mux select bit. The inst address is the current ROM address (PC). State means the current state. ALU\_A is input A to the ALU. ALU\_B is the input B to the ALU. ALUout is the output of ALU. Next State is the next state. The select switch input is 3’b {SW[17],SW[16],SW[15]}.

**3. Test Procedures**

I tested each module. I have set instructions and data in the ROM and the RAM to test modules. I have given pseudo code; I interpret the given code to instructions codes. Then, I explain and show the instruction codes in Appendix A.

**3.1 SM\_tb**

This module does not have (or is connected with) ROM; I put the instruction codes in Appendix A. Figure 17 shows the output.

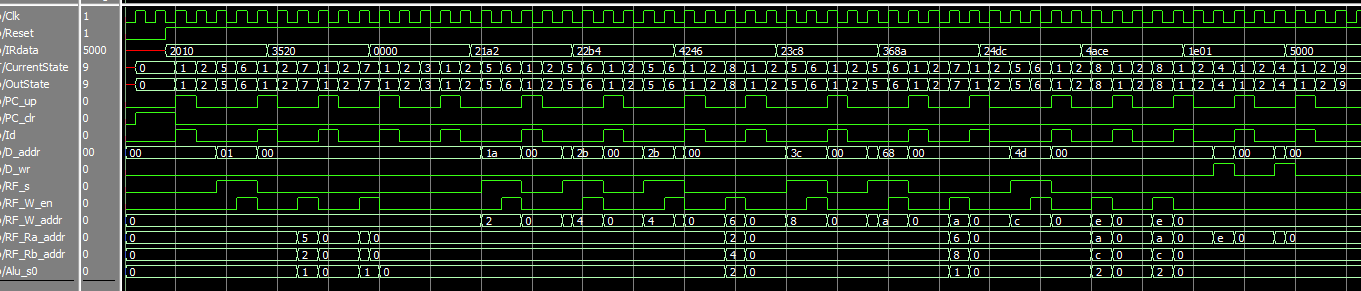
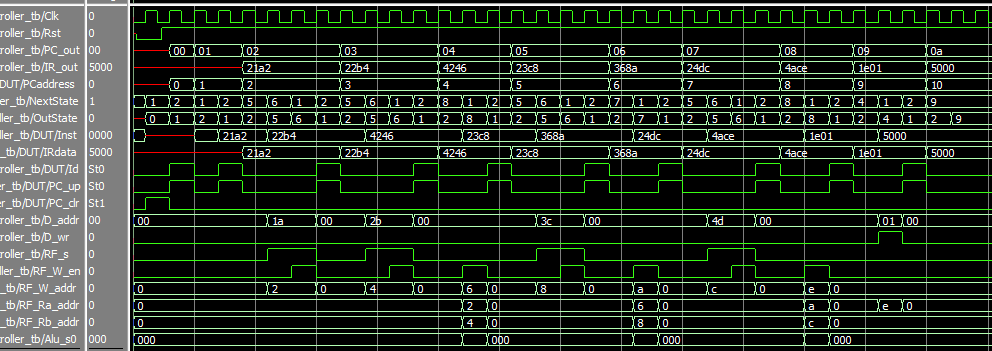


Figure 17. The wave graph of SM\_tb.sv. It interprets each instruction line and sends outputs as I designed in Figure 5. The first 2 IRdata are random numbers to test the modules.

**3.2 Controller\_tb**

This testbench only takes clock and reset inputs since it has all submodules. It sends outputs as I set the clock is shifting and reset to 0 (see Figure 18). I checked the IR out with the ROM; it works as expected (see Figure 19).

****Figure 18. The wave graph of Controller\_tb is similar to Figure 17.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Figure 19. The memory List of the ROM. It is the same as the R\_out in Figure 18.

**3.3 Datapath\_tb**

This testbench needs input from the controller module, but it is not connected to the controller module. Check the testbench in Figure 20. Figure 21 and Figure 22 are the output result and the memory list.

텍스트이(가) 표시된 사진

자동 생성된 설명

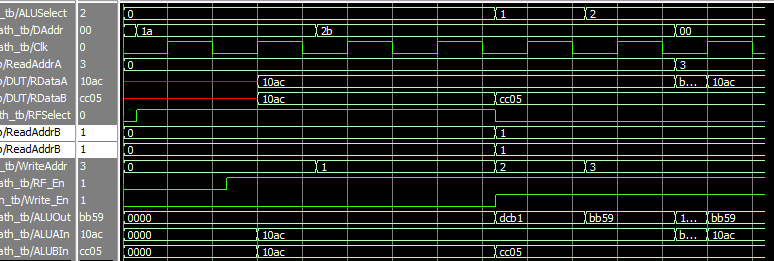
Figure 20. The testbench in datapath. Those are: Load RAM[1A] to RF[0]; Load RAM[2B] to RF[1];Add RF[0] & RF[1], save it to RF[2]; SubstractSubtractRF[1], save it to RF[3]; Store RF[3] to RAM[01]****

Figure 21. The wave graph of the testbench inputs from Figure 20.

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 22. The memory list of RAM. It stores RF[3] to RAM[01], which is the same as Figure 21, the last value of ALUOut.

**3.4 testProcessor**

I use this testbench module to test processor.sv. The output is the same as expected(see Figure 23). Also, I checked the RAM and RF to verify that it works as I built (see Figure 24 and Figure 25).

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

Figure 23. The wave graph of testProcessor.sv. the outputs are the same as RAM and RF values in Figure 24 and Figure 25.

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 24. The memory list of RAM.

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 25. The memory list of RF.

**4. Test Results**

I set the Project.sv to DE2 bot. DE2 shows the output same as Figure 23. I have a demonstration video. The video is attached in the ZIP file. Quartus tells Project.sv has 0 error and 9 warnings.

테이블이(가) 표시된 사진

자동 생성된 설명

Figure 26. The compilation report.

**5. Observations**

A processor has multiple components and modules, and the connection between modules is essential.

**6. Conclusion**

It was building a

**Appendix A**

The project requirement is

Instruction pseudo code:

RF[0A] = D[1A] - D[2B] + D[3C] - D[4D];

D[EF] = RF[0A];

HALT

I divide the pseudo-code into instruction codes and save them into ROM.

0000 : The first address in the ROM doesn’t do anything since PC.sv need initiation time. Thus, I put it as NOOP.

21A2 : Load data from RAM [1A] to RF [2].

22B4 : Load data from RAM [2B] to RF [4].

4246 : Subtract RF [2] & RF [4] , save it to RF [6]

23C8 : Load data from RAM address 3C to RF address 8.

368A : Add RF [6] & RF [8] , save it to RF [A]

24DC : Load data from RAM [4D] to RF [C].

4ACE : Subtract RF [A] & RF [C] , save it to RF [E]

1E01 : Save data from RF [E] to RAM [01]

5000 : HALT

I save this data in RAM

RAM[1A] = 0x10AC

RAM[2B] = 0xCC05

RAM[3C] = 0x01B5

RAM[4D] = 0xA040