Processor Project

Evan Cooper, Steven Tieu

TCES330

Abstract

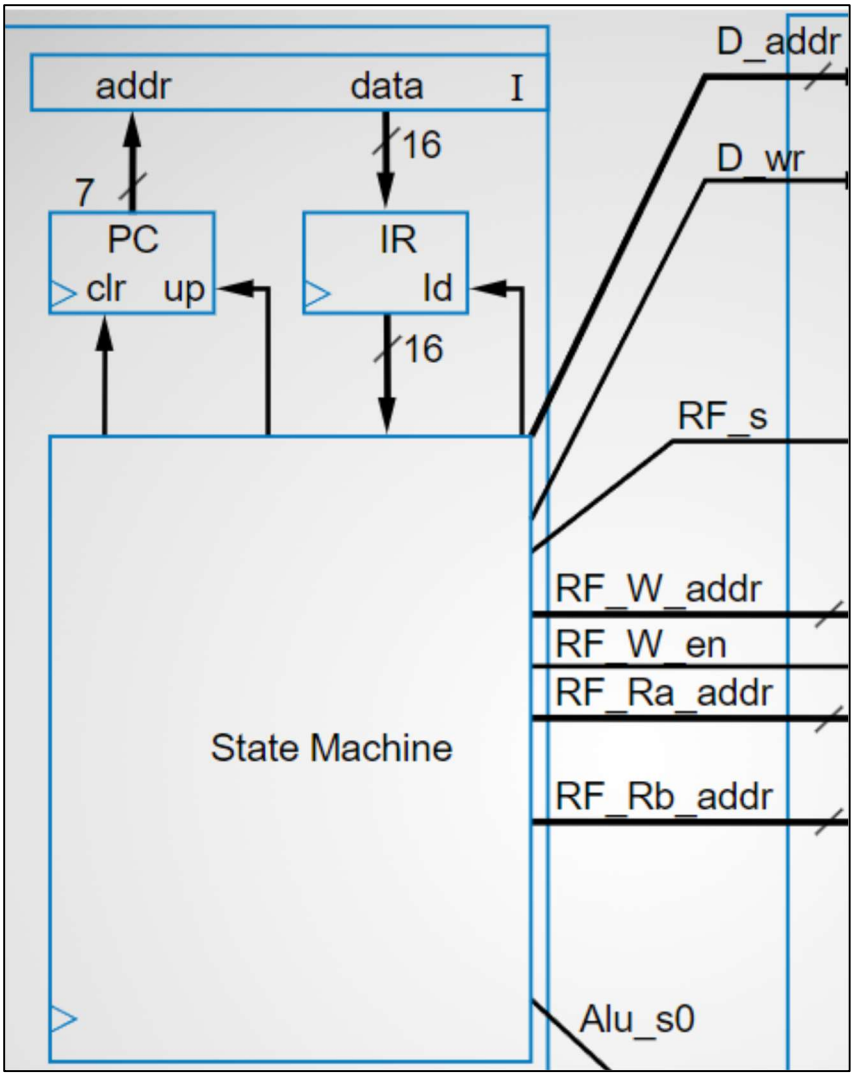
The purpose of this project was to understand the architecture that a processor is built on and to demonstrate this knowledge by creating a simple processor in System Verilog. Our team members for this project are Evan Cooper and Steven Tieu. We divided the workload as follows; Evan did the control unit side of the CPU while Steven did the data path side. We ended up working together for the processor itself and the project file that we program onto the DE2 board. Overall, this project was a great opportunity to gain working knowledge of a processor and to combine our knowledge of computer architecture with the design experience we have learned from this class.

1.Requirements

The design of this CPU requires two modules to be created that are instantiated from one high level module. This high-level module is the ‘processor.sv’ file, which calls upon the ‘Controller.sv’ and ‘Datapath.sv’ files. The controller handles the control signals for the operations of the modules inside of the CPU, such as the ALU. The data path on the other hand controls how the data in the processor flows, and where it should go to next.

1.1 Requirements for Controller.sv

To create the controller, we needed to create more low-level modules that handled the functions of the controller. We created a finite state machine to handle the instructions that were given to the ‘Datapath.sv’. We also created an instruction register to hold the instruction as a latch for the state machine. We needed a counter as well to drive the controller, so Steven created the ‘PC.sv’ module that acts as a counter and tells the controller as to what instruction it needs to read in a sequential order. Finally, we created a ROM for the controller by using Intel Quartus to create a ‘.mif’ file that stored the instructions. This is read by the 1-port ROM Verilog that we created using the Quartus IP-Catalog. You can view all of this in the Figure below

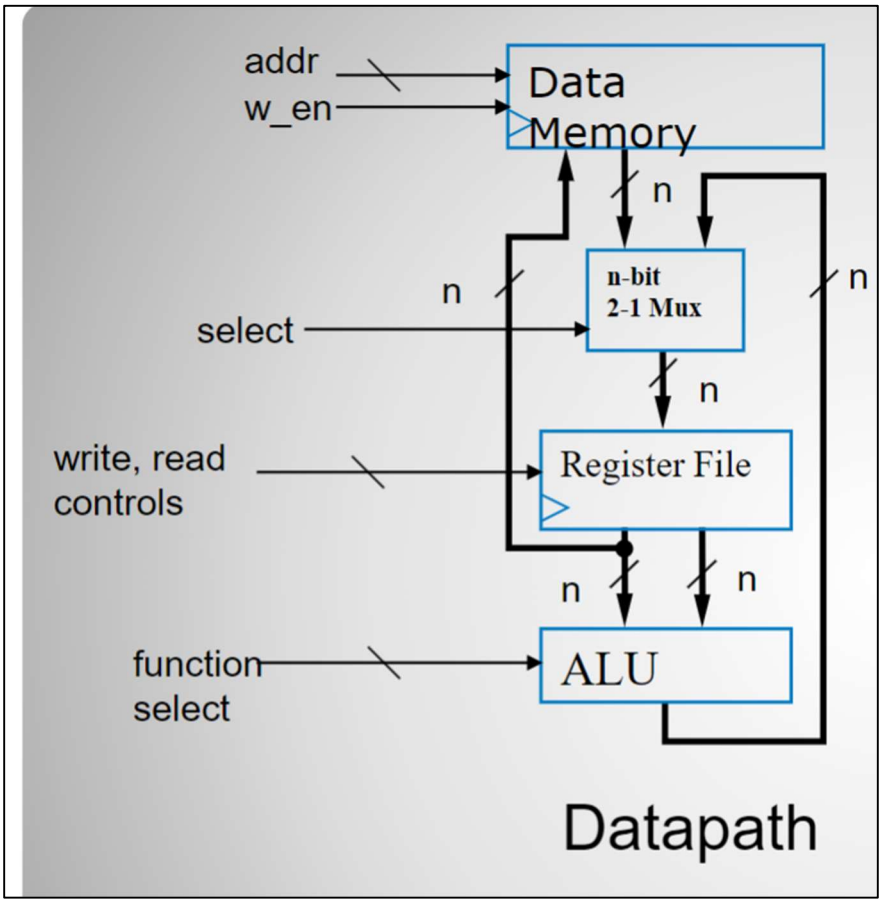


Control unit's submodules and their connections. Right wires go to Datapath Unit.

1.2 Requirements: Datapath.sv

The Datapath requires four low-level modules to operate: Data memory, Mux, Register File, and an ALU unit. The Data memory acts as our non-volatile memory for the CPU, it contains initial data that we will operate in our Processor and where we will store data, this module be created through Quartus's LPM library and using a Memory initialization file as storage for data. The Mux module will be a 16-bit wide 2 to 1 mux which takes inputs from the

Data memory and ALU then outputs them to the Registers. Register files will hold volatile data that we will perform operations on during runtime in the ALU. The ALU will perform one of 8 possible functions on data coming from the register files and return them back to the register if the mux selects it. Control signals will be sent from the Control unit to tell what each low-level module should do. You can view all of this in the Figure below.



Datapath unit's submodules and their connections. External wires come from Control.

1.3 Requirements: Processor.sv

The processor module was created by instantiating the ‘Controller.sv’ and ‘Datapath.sv’ modules. Then, the next step was to create the wiring between the two modules so that the data could be passed between them, and they could work as a processor. This required both Controller and Datapath to work correctly beforehand, and we made sure of this by using our test benches.

1.4 Requirements: Project.sv

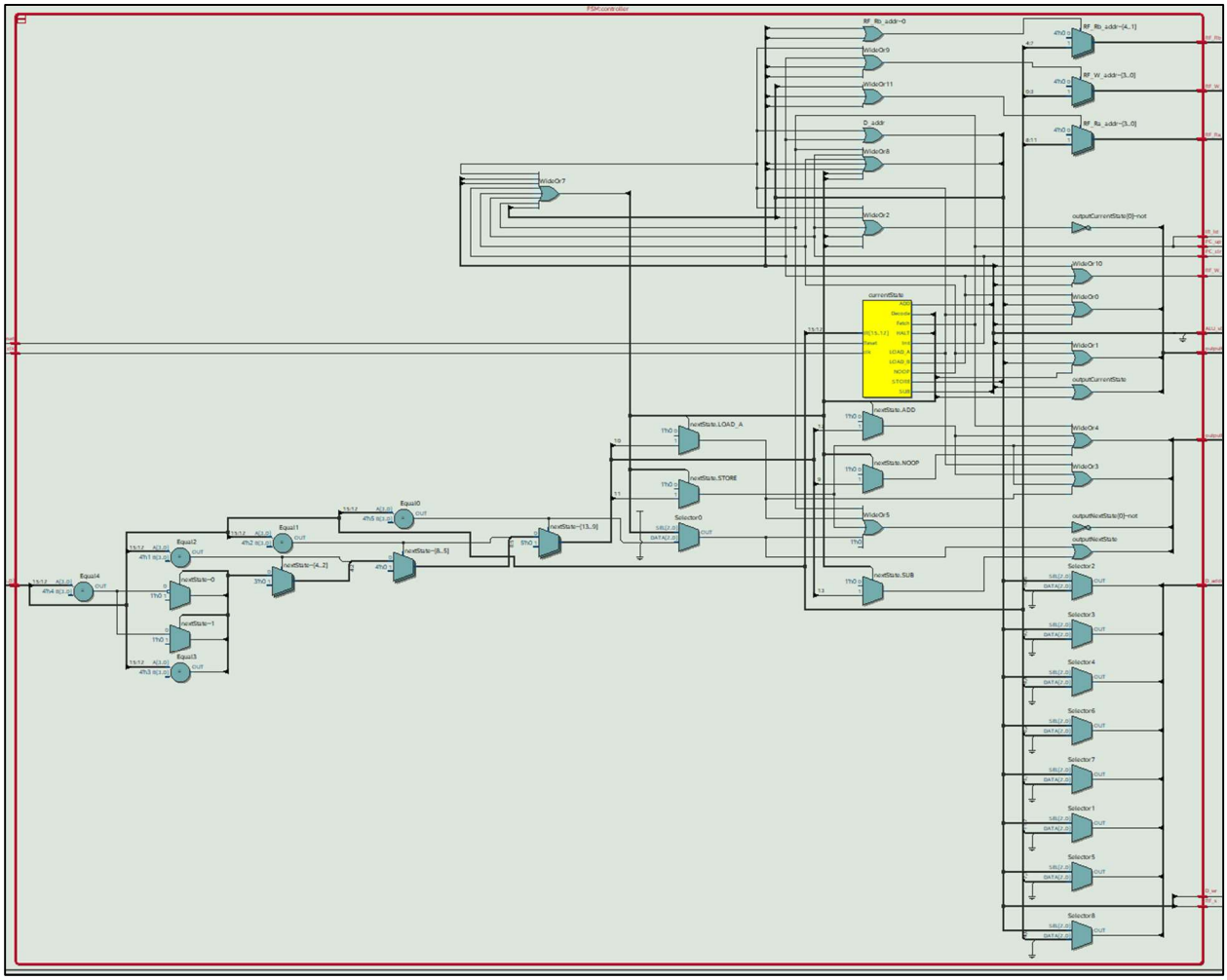
Finally, the project file is the highest-level in the hierarchy of the modules and is called the ‘Processor.sv’ file. It uses a 16-bit wide 8 to 1 multiplexer that will change the HEX displays based on the input provided on the first 3 switches. Also, it was necessary to include a filter for the buttons so that there would not be accidental extra presses while testing. We also included the provided ‘ButtonSyncReg.sv’ module to ensure that a button press is only 1 clock cycle long.

2. Design

2.1 FSM.sv

The creation of the FSM was quite simple and was based off the state machines we created in class. It uses case statements to list all the states that are used in the controller. It begins with an initialization state ‘Init’ that just sets the ‘PC_clr’ to 1. The next state of this is the ‘Fetch’ state, which increments the counter by 1 and sets the instruction register load signal to 1. The next state is ‘Decode’, which reads the first 4 bits of the instruction register and determines what operation to do. If there is no operation to be done, it simply goes to a ‘NOOP’, but if there is instruction, it will go to either ‘HALT’, ‘LOAD_A’, ‘STORE’, ‘ADD’, or ‘SUB’. The halt operation completely stops the CPU and requires the processor to be reset to run again. The load

instruction for A stores a set of 8 bits from the instruction register into the data address variable. The register file then stores the data in the instruction register. ‘LOAD_A’ then calls ‘LOAD_B’, which performs the same operation except with the write function enabled for the register file. ’STORE’ writes data back from the register file into the data memory and goes back to fetch once completed. ‘ADD’ and ‘SUB’ are essentially the same operations in the fact that they both send 2 4-bit data addresses into the ALU, but with the add operation the ALU select bit is set to 1, whereas the subtract function is set to 2. The 1 and 2 represent the select bits for add and subtract respectively.



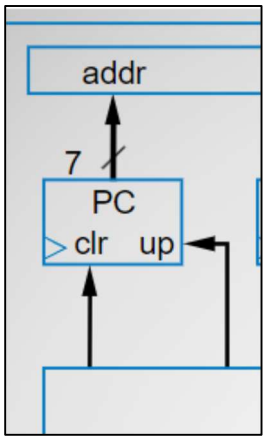
RTL view of the FSM.sv module

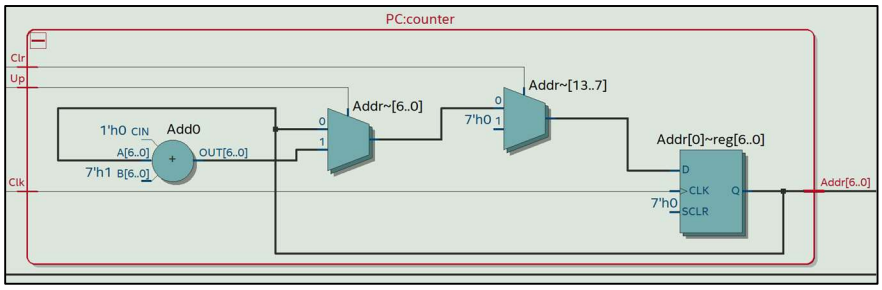
2.2 PC.sv

The PC or program counter is a counter increments through all instructions from our 128x16 instruction memory (InstMemory.v). The PC requires three 1-bit input signals and sends out one 7-bit output signal. Clock (Clk), Clear (Clr), and Counter up (Up) are our three input signals and Instruction Address (Addr) is our 7-bit output. See Figure below for a diagram.

Clear is a synchronous active high signal and resets the counter to 0 when the finite state machine (FSM) is reset to its initial state. If Clear is low and the Up signal is high, it increments the counter by 1 on clock positive edge, the up signal is high when the FSM is in the FETCH stage. Clock input is the clock signal that the PC will synchronize to, where it will operate on a positive edge.

The PC only has one output signal which is the 7-bit instruction address (Addr), which starts from 0 after clearing and increments up to 127 when signaled up. This will allow us to send instruction addresses such that instruction memory performs sequentially.





Block Diagram Snippet of Program Counter. Bottom arrows come from FSM unit.

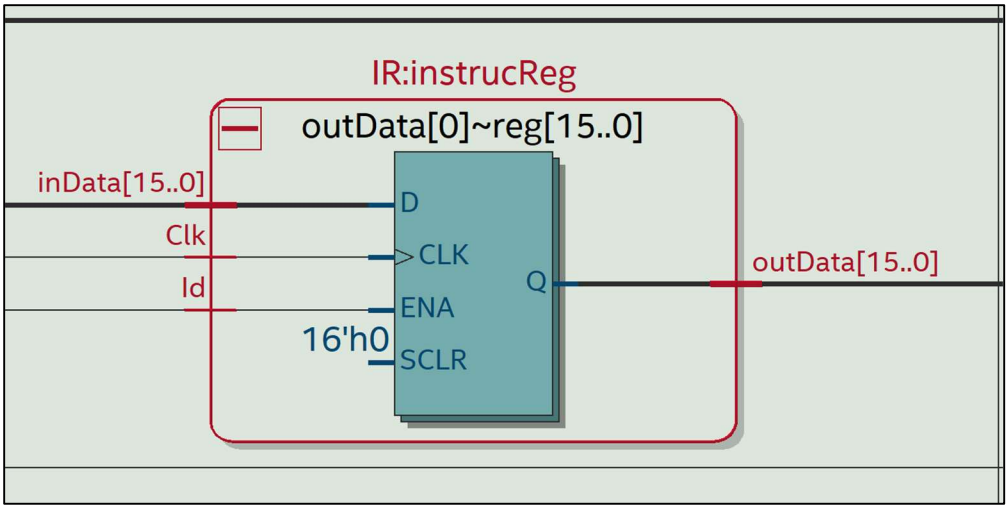
RTL view of the PC.sv module

2.3 IR.sv

The IR or the instruction register is simply a latch that holds the next instruction that will be performed in the processor. This latch is implemented through a simple synchronous if statement in the SystemVerilog code, which you can view below in Figure below.

```
//Based on Lab 5 Part 2 flip-flop.
always_ff @(posedge Clk) begin
        if(Id) outData <= inData;
        // else outData <= outData; Not necessary
end
```

SystemVerilog code used to implement IR latch

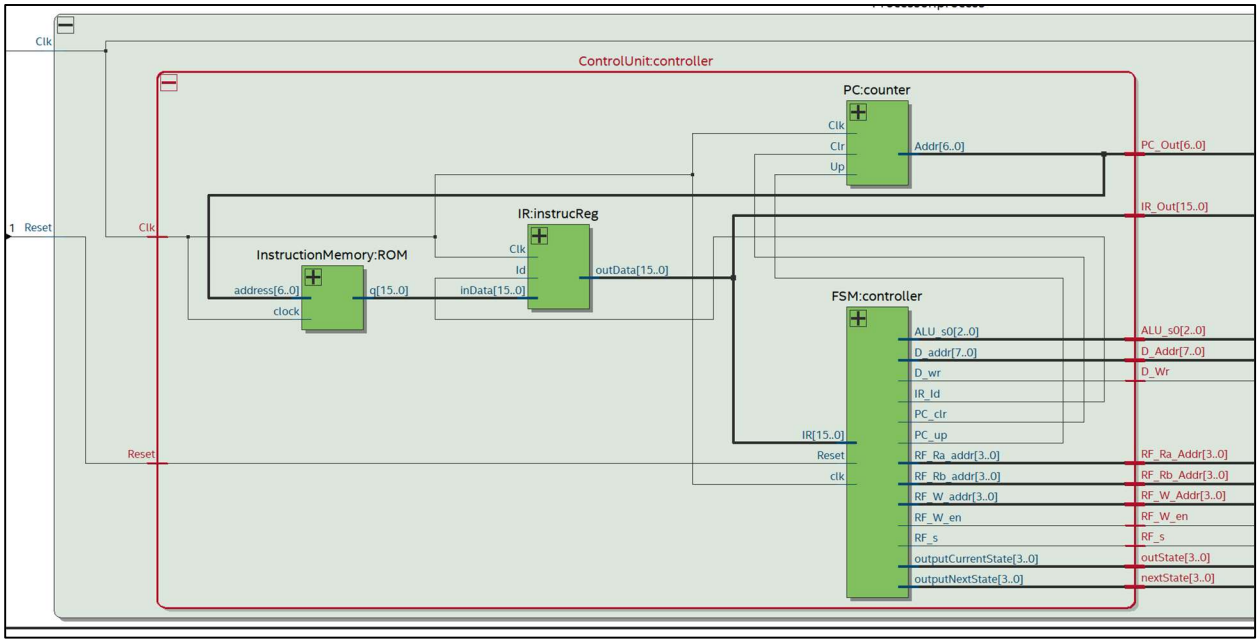


RTL view of the IR.sv module

Once the IR receives a high signal for Load (Ld) from the FSM unit, it will assignment the 16-bit input data (inData) to the 16-bit output port (outData) on the positive edge of the clock signal attached to this IR unit, where the instruction will be sent to the FSM for execution.

2.4 Controller.sv

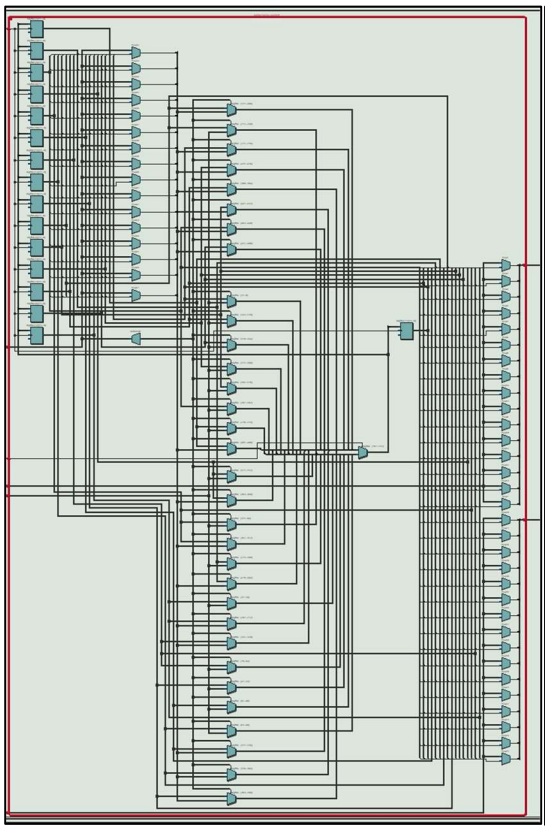
To design the controller, we had to create the ROM file and instruction memory to read the data from a .mif file. We used the Quartus IP-catalog to achieve this, and we were able to read the data from the file. We then instantiated the instruction memory, instruction register, FSM, and counter. With the use of some temporary wires, we were able to read the instructions using instruction memory, which then sent these instructions to the instruction register to store them. The state machine then reads the instructions and determines the operation to be executed and the counter moves onto the next instruction set.



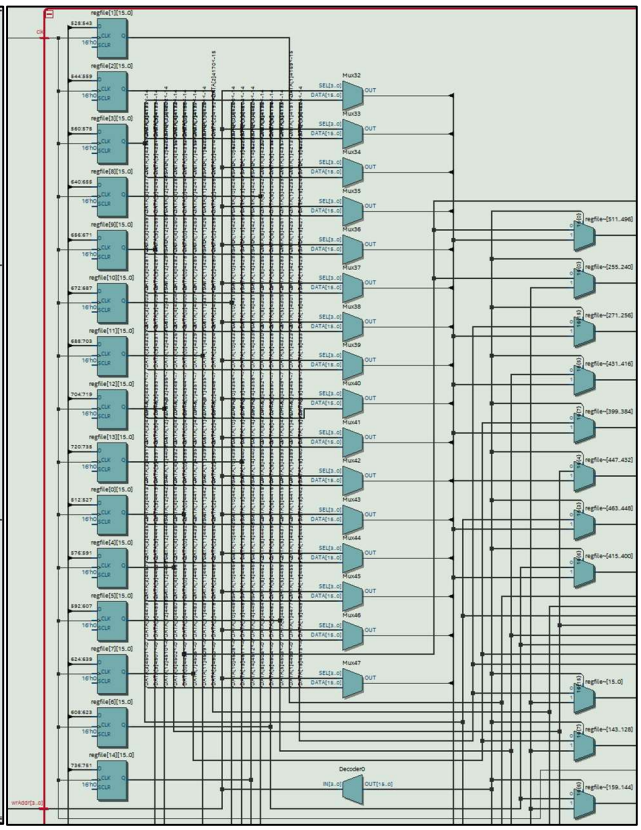
RTL view of the Controller.sv

2.5 regfile16x16a.sv

The register file used in our design contains 16 registers that are 16 bit each. It allows for data to be read or written over if the write signal is enabled. It assigns the output to the data in the address for A and B that are passed into it.



RTL view of the regfile16x16a.sv module

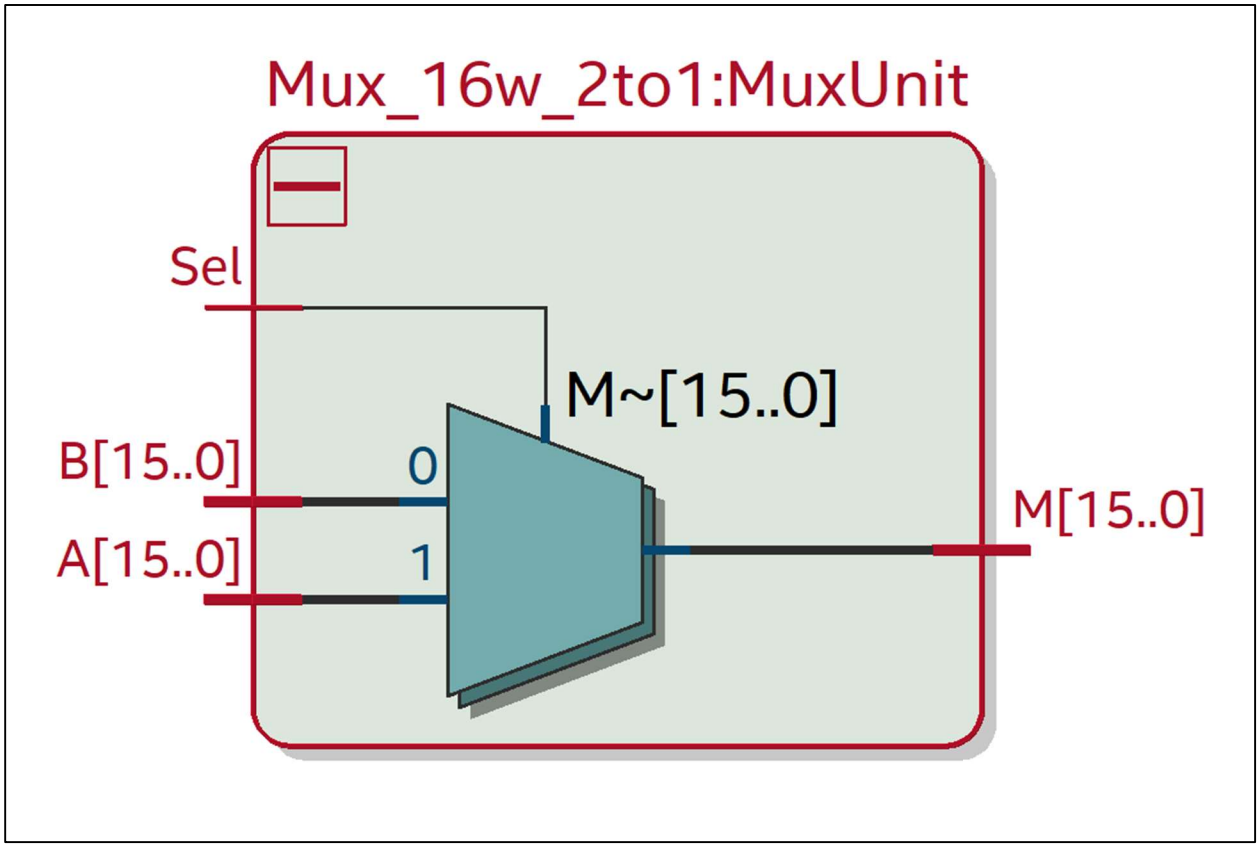


Closer view of the register file RTL

2.6 Mux_n_2to1.sv

The Mux unit is a 2-to-1 multiplexer that is 16-bit wide with a 1-bit select signal. The 1-bit select signal of this multiplexer is received from the FSM, this select signal decide whether the Register file (regfile16x16a.sv**)** will receive 16-bit write data from either the Data memory (DataMemory.v) when select signal is high or ALU (ALU.sv) when select signal is low.

Implementation of this Mux unit is a simple combinational circuit with an if/else statement describing that when the select signal is high, assign output M to A otherwise assign M to B.



RTL view of the Mux_16w_2to1.sv module

2.7 ALU.sv

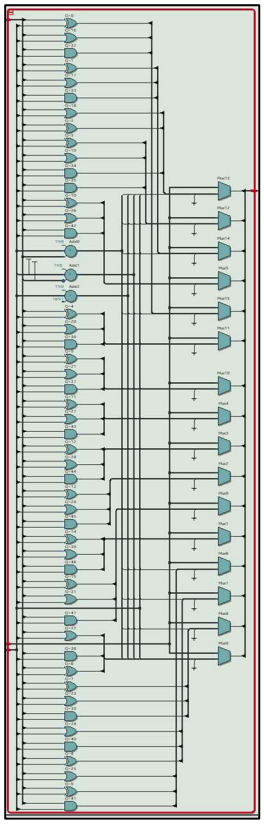
The ALU or Arithmetic Logic Unit is a purely combinational circuit that performs operations on data received from the register files (regfile16x16a.sv). The ALU takes two 16-bit data as inputs (A, B), a 3-bit select signal (SelectFunc), and a 16-bit output (Q) from operating with the two inputs. The eight functions in the ALU are selected through a combinational block with a case statement in SystemVerilog, these are shown in program code below.

```
case (SelectFunc)
        3'b000: Q = 3'b000;      //0
        3'b001: Q = A + B;       //ADD
        3'b010: Q = A - B;       //SUBTRACT
        3'b011: Q = A;           //BYPASS
        3'b100: Q = A ^ B;       //XOR
        3'b101: Q = A | B;       //OR
        3'b110: Q = A & B;       //AND
        /*3'b111*/ default: Q = A + 3'b001; //ADD ONE
endcase
```

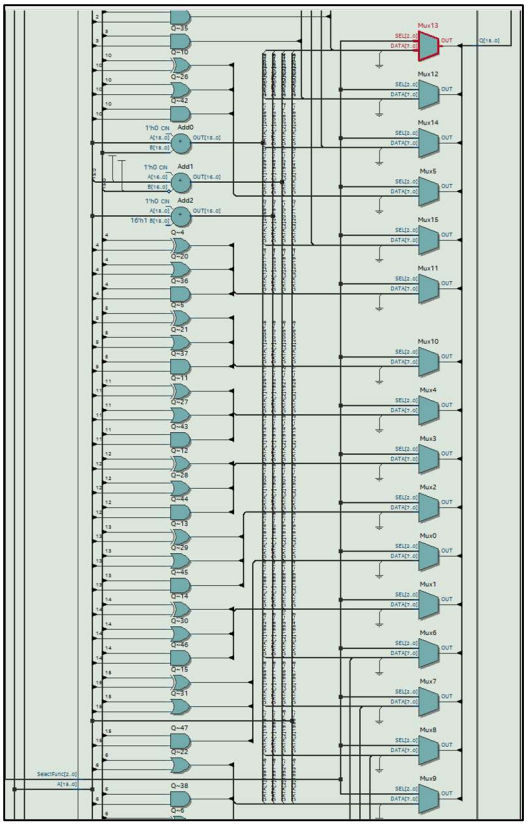
All available ALU operations in System Verilog

The ALU will either perform addition (SelectFunc = 001), subtraction (SelectFunc = 010), exclusive or (SelectFunc = 100), “or” (SelectFunc = 101), or a “and” (SelectFunc = 110) operation depending on the signal sent into the Select port. There are also options to simply output the data on port A (SelectFunc = 011), output a 0 (SelectFunc = 000), or add 1 to A then output it (SelectFunc = 111).

It is worth noting that the ALU is instantiated with a “Bits” parameter to decide how many bits wide the ALU will be, this is set to be 16 bits in the Datapath.sv file.



RTL view of the ALU.sv module

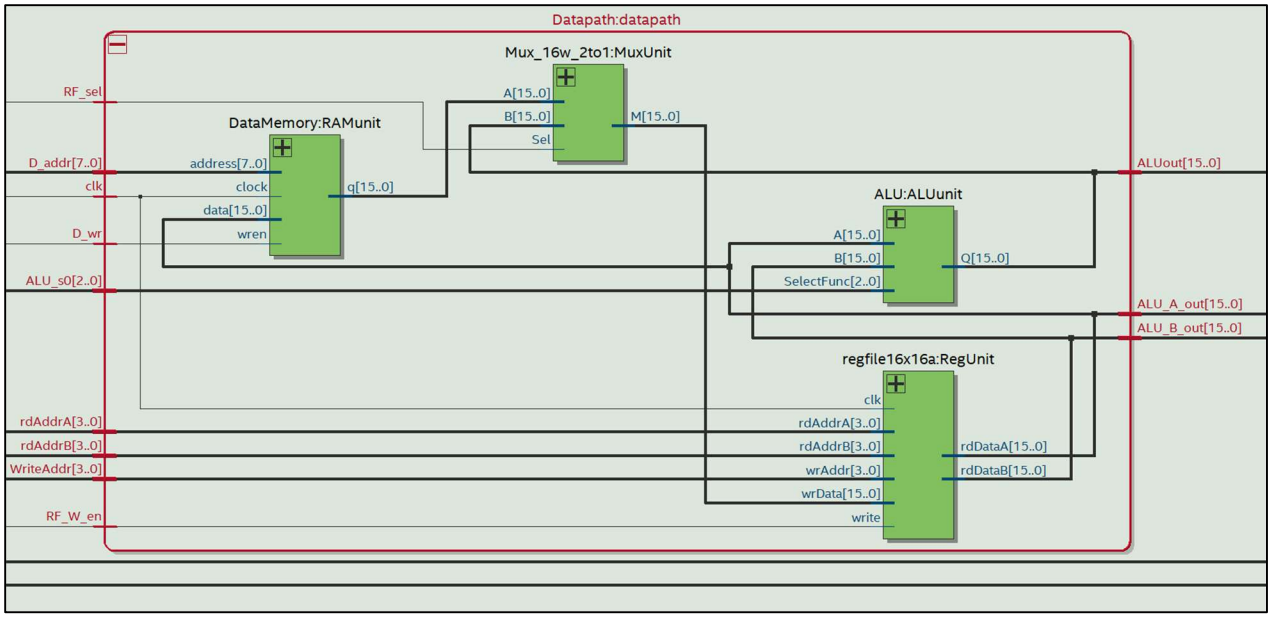


Closer view of the ALU module

2.8 Datapath.sv

The Datapath module is implemented by simply instantiating its required submodules and making the right connections between their ports. We used the RTL diagrams provided on the Canvas page to guide us. The block diagram told us how to generally connect all the Datapath submodules while the RTL diagram told exactly which ports are connected to which.

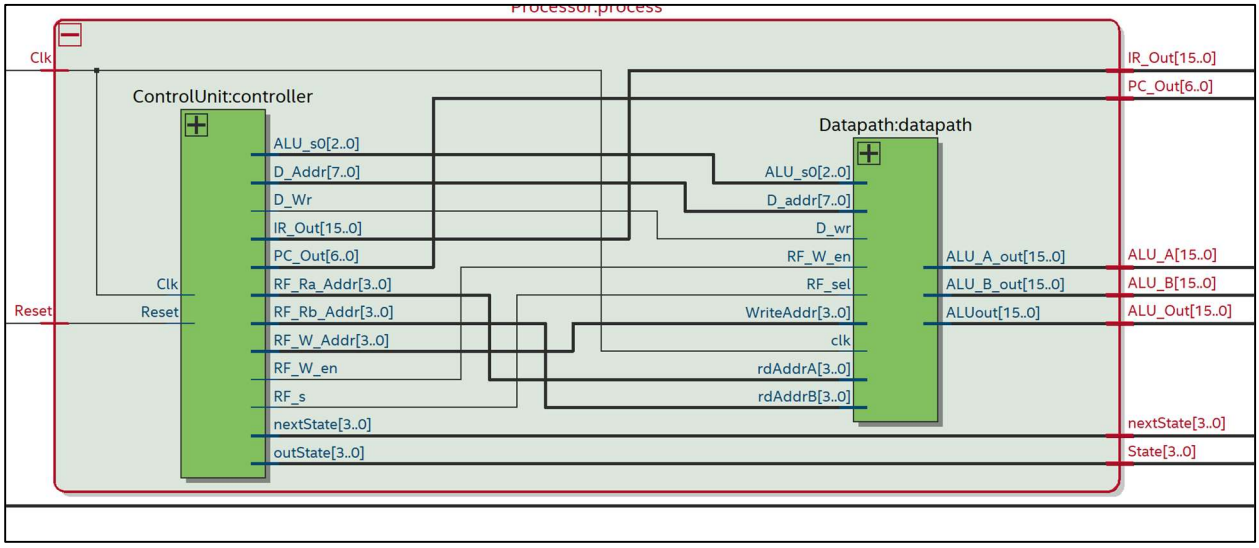
The Register file would output register data to either the ALU or Data memory. The Mux would take inputs from data memory or ALU then output to the Register File. The ALU would take two inputs from the Register file and only output back to the Mux to be potentially stored in the register. Finally, the Data memory would take inputs from register file and output to Mux. All modules receive signals from the control unit to the modules to operate in certain ways.



RTL view of the Datapath.sv module

2.9 Processor.sv

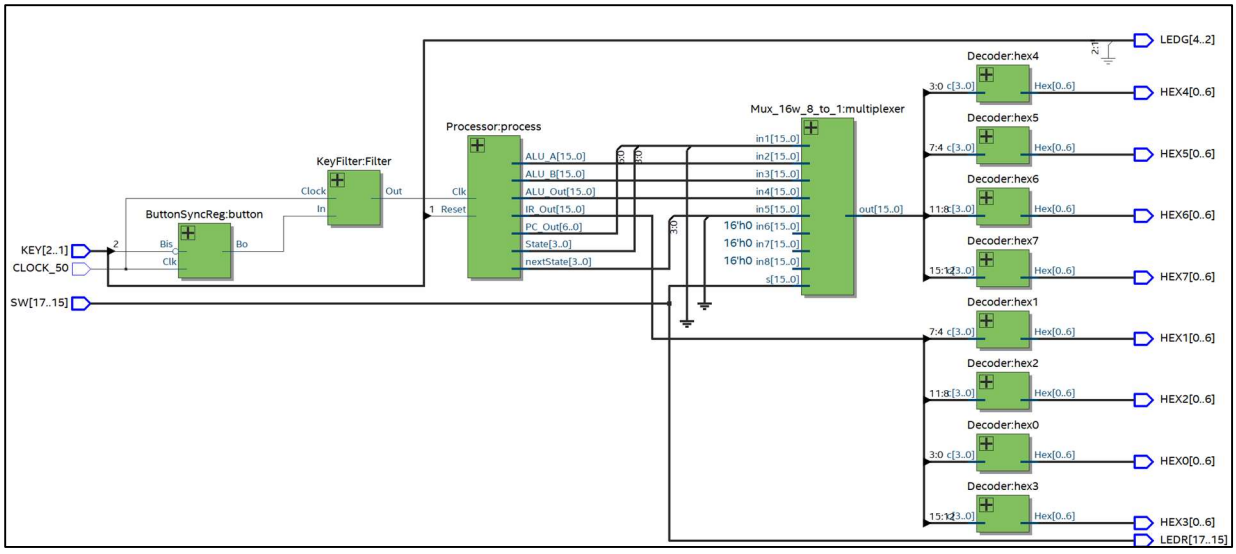
The design of the processor was relatively simple and mostly relied on the previous modules to be working as intended for the processor itself to work. We instantiated the 'Controller.sv' and the 'Datapath.sv' and used more wires to connect the outputs of the controller into the data path. By doing so, the data path can receive signals from the controller to execute specific operations.



RTL view of the Processor.sv module

2.10 Project.sv

The project design required the creation of a new MUX module that is 16 bits wide and an 8 to 1. This is so that the HEX displays can be changed to display different data about the status of the CPU by reading in a select bit from the switches on the board. After creating this, we also had to include the 'Decoder' from our previous labs to decode the registers into hexadecimal. We then instantiated the button sync module as well as the filter. Button sync uses the on board 'CLOCK_50' clock and the KEY2 to create an output that is fed into the filter. Filter then produces the output that we can use for the clock in our CPU, which is KEY2, but it is now filtered so that one click will equal to one clock cycle. 'Processor.sv' and the MUX are then called, and the output of the processor is fed into the MUX, where it will be chosen to be displayed on the HEX via switches 17-15. We then used 8 wires to store the 16-bit register and data into 4-bit chunks, so that each HEX could be fed 4-bits. Finally, we called 'Decoder' 8 times, one for each HEX display and it displays the hexadecimal of the 4-bit chunk. Also, we assigned the green LEDs above the keys to the keys, and we assigned the red LEDs to their corresponding switches.



RTL view of the Project.sv module

3. Test Procedures

3.1 FSM_tb

For the design of the FSM testbench, I created a variety of instruction register values that cycle through each state, and I created display statement to keep track of the data and addresses as well as the counter. By doing so, it was simple to validate with assert statements that the FSM was working as intended. All operations of the processor were tested and show the correct state sequence in testing.

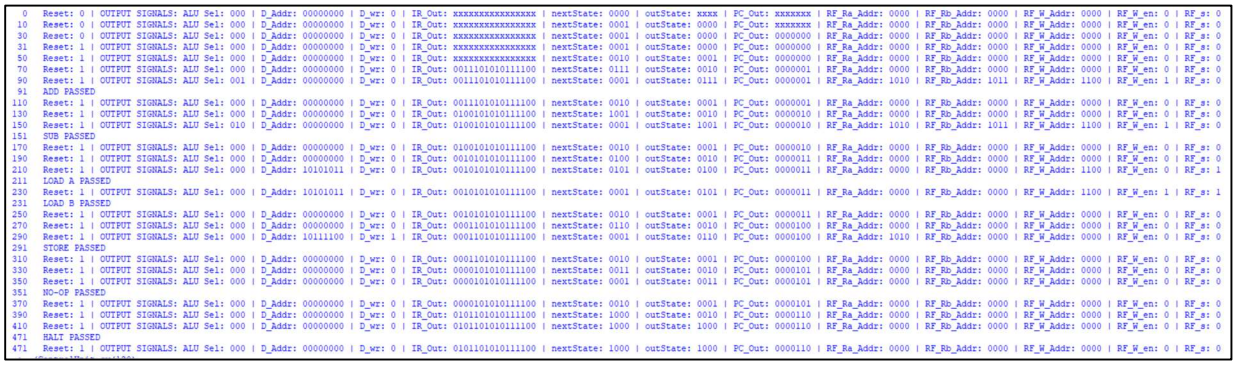
```
# Instruction Register Address: 0011000000000001 | PC_clr = 1
# IR_Id = 1, PC_up = 1
# Add Operation: Write Address = 0001 | Write Enable = 1 | A Address = 0000 | B Address = 0000 | ALU Select = 001
# Sub Operation: Write Address = 0001 | Write Enable = 1 | A Address = 0000 | B Address = 0000 | ALU Select = 010
# Load Operation | Data Address: 00000000 | Register File Select Bit: 1 | Write Address: 0001 | Write Enable: 1
# Store Operation | Data Address: 00000001 | Data Write: 1 | A Address: 0000
# Previous Sub Operation w/ NOOP: Write Address = 0001 | Write Enable = 1 | A Address = 0000 | B Address = 0000 | ALU Select = 010
# Store Operation After Halt (Should not Change from Previous Store) | Data Address: 00000001 | Data Write: 1 | A Address: 0000
# ** Note: $stop    : ./FSM.sv(169)
#    Time: 600 ps  Iteration: 0  Instance: /FSM_tb
# Break in Module FSM_tb at ./FSM.sv line 169
```

Test bench results for the FSM_tb.sv

3.2 Controller_tb

The Controller testbench was designed to ensure that all control signals output by the Controller would be correspond correctly with the entered instruction from the ROM. The testbench goes through all states of the FSM unit, that being ADD, SUB, LOAD A, LOAD B, STORE, NO-OP, and finally HALT. Each operation was given a 60-picoseconds to allow each operation to go through the Fetch, Decode, and the test operation itself in the FSM unit. All output signals were monitored at this time and assert statements were coded to ensure that outputs were correct.

It is worth noting that when changing the Reset input state from active low to high, we needed to add 30 picosecond delay before changing, as this would notably interpret first instruction as NO-OP rather than ADD. We approached our professor for help on solving a solution, but we were unable to locate the source of this problem this time and continued the project with the professor's permission.



Test bench results for the Controller_tb.sv

3.3 Datapath_tb

The Datapath testbench was designed to ensure that when receiving a specific signal from the control unit, the Datapath unit would perform the correct operation on the right data. All signals sent between modules were monitored each time a change occurred in the testbench.

```
# Run the simulation
# run -all
#                    5 LOAD (A) 1111 in REG 1:
#                    5   SELECT SIGNALS | ALU Sel: xxx | RF Sel: 1 | D_wr: 0 | RF_wrEn: 0
#                   20   NEGCLK: Load into REG 1 A | IR: 000000010001 | DataToMux: 0001000100010001 | write: 0 | wr data: 0001000100010001
#                   25 LOAD (B) 1111 in REG 1:
#                   25   SELECT SIGNALS | ALU Sel: xxx | RF Sel: 1 | D_wr: 0 | RF_wrEn: 1
#                   40   NEGCLK: Load into REG 1 A | IR: 000000010001 | DataToMux: 0001000100010001 | write: 1 | wr data: 0001000100010001
#                   40   LOAD_1111 PASSED
#
#                   45 Store Operation
#                   45   SELECT SIGNALS | ALU Sel: xxx | RF Sel: 1 | D_wr: 1 | RF_wrEn: 0
#                   60   NEGCLK: IR: 000101101010 | Wr Addr: 1010
#                   60   STORE PASSED
#
#                   65 LOAD (A) 2222 in REG 2:
#                   65   SELECT SIGNALS | ALU Sel: xxx | RF Sel: 1 | D_wr: 0 | RF_wrEn: 0
#                   80   NEGCLK: Load 2222 into REG 2 A | IR: 000000100010 | DataToMux: 0010001000100010 | write: 0 | wr data: 0010001000100010
#                   85 LOAD (B) 2222 in REG 2:
#                   85   SELECT SIGNALS | ALU Sel: xxx | RF Sel: 1 | D_wr: 0 | RF_wrEn: 1
#                  100   NEGCLK: IR: 000000100010 | DataToMux: 0010001000100010 | write: 1 | wr data: 0010001000100010
#                  100   LOAD_2222 PASSED
#
#                  105 Add 1 Operation
#                  105   SELECT SIGNALS | ALU Sel: 001 | RF Sel: 0 | D_wr: 0 | RF_wrEn: 1
#                  120   NEGCLK: Add Operation: Wr Addr = 0011 | A Addr = 0001 | B Addr = 0010
#                  120   ADD 1 PASSED
#
#                  125 Add 2 Operation
#                  140   NEGCLK: Add Operation: Wr Addr = 0100 | A Addr = 0010 | B Addr = 0011
#                  140   ADD 2 PASSED
#
#                  145 Sub 1 Operation
#                  145   SELECT SIGNALS | ALU Sel: 010 | RF Sel: 0 | D_wr: 0 | RF_wrEn: 1
#                  160   NEGCLK: Sub Operation: Wr Addr = 0000 | A Addr = 0010 | B Addr = 0001
#                  160   SUB PASSED
#
#                  165 EDGE CASE LOAD Operation
#                  165   SELECT SIGNALS | ALU Sel: 010 | RF Sel: 1 | D_wr: 0 | RF_wrEn: 0
#                  180   NEGCLK: Load into REG 1 A | IR: 111111111111 | DataToMux: 1111111111111111 | write: 0 | wr data: 1111111111111111
#                  185   SELECT SIGNALS | ALU Sel: 010 | RF Sel: 1 | D_wr: 0 | RF_wrEn: 1
#                  200   NEGCLK: Load into REG 1 A | IR: 111111111111 | DataToMux: 1111111111111111 | write: 1 | wr data: 1111111111111111
#                  200   EDGE CASE LOAD PASSED
#
#                  205 EDGE CASE Store Operation
#                  205   SELECT SIGNALS | ALU Sel: 010 | RF Sel: 1 | D_wr: 1 | RF_wrEn: 0
#                  220   NEGCLK: IR: 111111111110 | Wr Addr: 1110
#                  220   EDGE CASE STORE PASSED
#
#                  225 NO-OP Operation
#                  225   SELECT SIGNALS | ALU Sel: 000 | RF Sel: 0 | D_wr: 0 | RF_wrEn: 0
#                  240   NEGCLK: IR: 111111111110 | Data Addr: 11111110 | A Addr: 1111
#                  245   NO-OP PASSED
# ** Note: $stop    : ./Datapath.sv(244)
#    Time: 245 ns  Iteration: 0  Instance: /Datapath_tb
# Break in Module Datapath_tb at ./Datapath.sv line 244
```

Test bench output for the Datapath_tb.sv module

The test bench begins with performing LOAD operations through setting Data address and RF write signals to move data from the Data memory into the register unit. We then tested ADD, SUB, and STORE operations on them by setting input signals to select the correct register data and mux output while also sending the corresponding ALU signal. At the end we tested the NO-OP operation to ensure no data changes during it.

3.4 testProcessor

For the processor test bench, we just used the provided module on Canvas. We did have to change the delay of the module to 30 picoseconds rather than 10, since we were having issues with the processor not reading the first instruction with the lower delay.

```
# Begin Simulation.
# Time is 0 : Reset = 0   PC_Out = xx   IR_Out = xxxx  State = x  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 10000 : Reset = 0   PC_Out = xx   IR_Out = xxxx  State = 0  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 30000 : Reset = 0   PC_Out = 00   IR_Out = xxxx  State = 0  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 31000 : Reset = 1   PC_Out = 00   IR_Out = xxxx  State = 0  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 50000 : Reset = 1   PC_Out = 00   IR_Out = xxxx  State = 1  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 70000 : Reset = 1   PC_Out = 01   IR_Out = 21b0  State = 2  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 90000 : Reset = 1   PC_Out = 01   IR_Out = 21b0  State = 4  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 110000 : Reset = 1   PC_Out = 01   IR_Out = 21b0  State = 5  ALU A = xxxx  ALU B = xxxx ALU Out = 0000  RA Address = 0000
# Time is 130000 : Reset = 1   PC_Out = 01   IR_Out = 21b0  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 150000 : Reset = 1   PC_Out = 02   IR_Out = 22a1  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 170000 : Reset = 1   PC_Out = 02   IR_Out = 22a1  State = 4  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 190000 : Reset = 1   PC_Out = 02   IR_Out = 22a1  State = 5  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 210000 : Reset = 1   PC_Out = 02   IR_Out = 22a1  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 230000 : Reset = 1   PC_Out = 03   IR_Out = 23c2  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 250000 : Reset = 1   PC_Out = 03   IR_Out = 23c2  State = 4  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 270000 : Reset = 1   PC_Out = 03   IR_Out = 23c2  State = 5  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 290000 : Reset = 1   PC_Out = 03   IR_Out = 23c2  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 310000 : Reset = 1   PC_Out = 04   IR_Out = 27e3  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 330000 : Reset = 1   PC_Out = 04   IR_Out = 27e3  State = 4  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 350000 : Reset = 1   PC_Out = 04   IR_Out = 27e3  State = 5  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 370000 : Reset = 1   PC_Out = 04   IR_Out = 27e3  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 390000 : Reset = 1   PC_Out = 05   IR_Out = 4015  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 410000 : Reset = 1   PC_Out = 05   IR_Out = 4015  State = 9  ALU A = 21ba  ALU B = a04e ALU Out = 816c  RA Address = 0000
# Time is 430000 : Reset = 1   PC_Out = 05   IR_Out = 4015  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 450000 : Reset = 1   PC_Out = 06   IR_Out = 3526  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 470000 : Reset = 1   PC_Out = 06   IR_Out = 3526  State = 7  ALU A = 816c  ALU B = 71ac ALU Out = f318  RA Address = 0101
# Time is 490000 : Reset = 1   PC_Out = 06   IR_Out = 3526  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 510000 : Reset = 1   PC_Out = 07   IR_Out = 463a  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 530000 : Reset = 1   PC_Out = 07   IR_Out = 463a  State = 9  ALU A = f318  ALU B = b17f ALU Out = 4199  RA Address = 0110
# Time is 550000 : Reset = 1   PC_Out = 07   IR_Out = 463a  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 570000 : Reset = 1   PC_Out = 08   IR_Out = 1a6a  State = 2  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
# Time is 590000 : Reset = 1   PC_Out = 08   IR_Out = 1a6a  State = 6  ALU A = 4199  ALU B = 21ba ALU Out = 0000  RA Address = 1010
# Time is 610000 : Reset = 1   PC_Out = 08   IR_Out = 1a6a  State = 1  ALU A = 21ba  ALU B = 21ba ALU Out = 0000  RA Address = 0000
#
# End of Simulation.
#
```

Test bench output for testProcessor.sv

4. Test Results

Overall, from using our testbenches we were able to directly see how the data was being addressed and manipulated in real time. It made it much easier to be able to follow along with the data using the waveforms, display statements, and assertions for debugging issues we had with our code. It also gave us insight into how to debugging works in large scale hardware design language projects such as this one. Each module worked with their corresponding test benches after taking time to debug.

5. Observations

The module hierarchy implementation of our processor helps with the debugging and design process of the Processor. We were able to isolate bugs to certain modules or units through testing and testing ensured that modules instantiated generally worked when we are testing higher level modules such as the Controller or Datapath. For instance, when control unit wasn't working with PC and IR not incrementing, we figured out that we had designed the PC and IR wrong, while they worked by themselves, state of signals sent by the FSM did not match and hence we redesigned them from there.

Additionally, warnings given in ModelSim were useful in test benches. These warnings would warn us when ports were connected incorrectly with non-matching port sizes or throw assertion errors in our testbenches to help streamline the testing process.

Warning messages in both ModelSim and Quartus were useful in helping us debug our code and we were quickly able to address warning message during Quartus and ModelSim compilation to quickly get our modules up and running.

6. Conclusion

This project and this quarter in general have been incredibly beneficial to our learning and development to become computer engineers. We learned how to collaborate and work together to meet and to work on separate modules while still being able to come together and understand each other's work. This project was also an invaluable experience to apply our knowledge of computer architecture and logic design into a practical and hands-on experience. Being able to see how the content we learn applies to a real-world scenario or project is arguably one of the most important aspects of learning, especially when it comes to content that is not necessarily easy. We are both excited to take our new experiences with System Verilog and digital design into our senior year as well as into the field of computer engineering and digital design itself.