Course Project

Programmable Processor

TCES330 Digital System Design

Spring 2022

**Authors:**

Student A: Thomas Halford (Datapath/Project)

Student B: Kader Khafif (Control Unit/Processor)

Submission Date: June 5, 2022

**Table of Contents:**

1: Requirements Page 3

1.1 CU.sv

1.2 Datapath.sv

1.3 Processor.sv

1.4 Project.sv

2: Design ­­­­­ Page 4

2.1 FSM.sv

2.2 PC.sv

2.3 IR.sv

2.4 CU.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 14

3.1 FSM\_tb.sv

3.2 Controller\_tb.sv

3.3 Datapath\_tb.sv

3.4 testProcessor.sv

4: Test Results Page 18

4.1 FSM\_tb.sv

4.2 Controller\_tb.sv

4.3 Datapath\_tb.sv

4.4 testProcessor.sv

5: Observations Page 21

5.1 regfiler8x16a.sv

5.2 Mux2to1.sv

5.3 ALU.sv

5.4 Datapath.sv,

5.5 FSM.sv

5.6 PC.sv

5.7 IR.sv

5.8 CU.sv

5.9 Processor.sv

5.10 Project.sv

6: Conclusion Page 24

7: References Page 24

Appendix Page 25

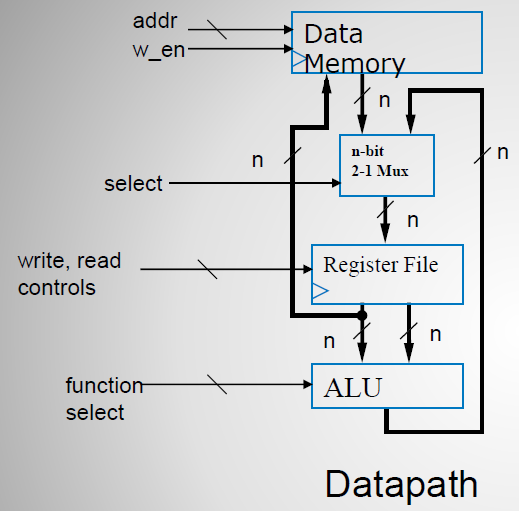
The purpose of this project was to implement the six-instruction programmable processor. The project was developed in SystemVerilog using Quartus and ModelSim with a hierarchy depth of 4 levels. The Quartus project is composed of the top-level module (called project) that instantiates the Button Sync, the Key Filter, the Processor, the 8-to-1 Mux, and several Hex modules. On the 3rd level, the processor instantiates the Control Unit and Datapath modules. On the 2nd level, the Control Unit instantiates a range of submodules which are the Program Counter (PC), Instruction Register (IR), Instruction Memory (ROM), and the State Machine (FSM). Also on the 2nd level, the Datapath instantiates the 2-to-1 Mux, the Arithmetic Logic Unit (ALU), and the Register File 16x16 (RF). To execute an instruction, the processor takes in a 16-bit input from the instruction memory file (ROM) and performs the corresponding operation with the input. The processor output is displayed using eight 7-segments displays.

1. **Requirements**

In this section, we will lay out the basic requirements for all of the major modules in the project.

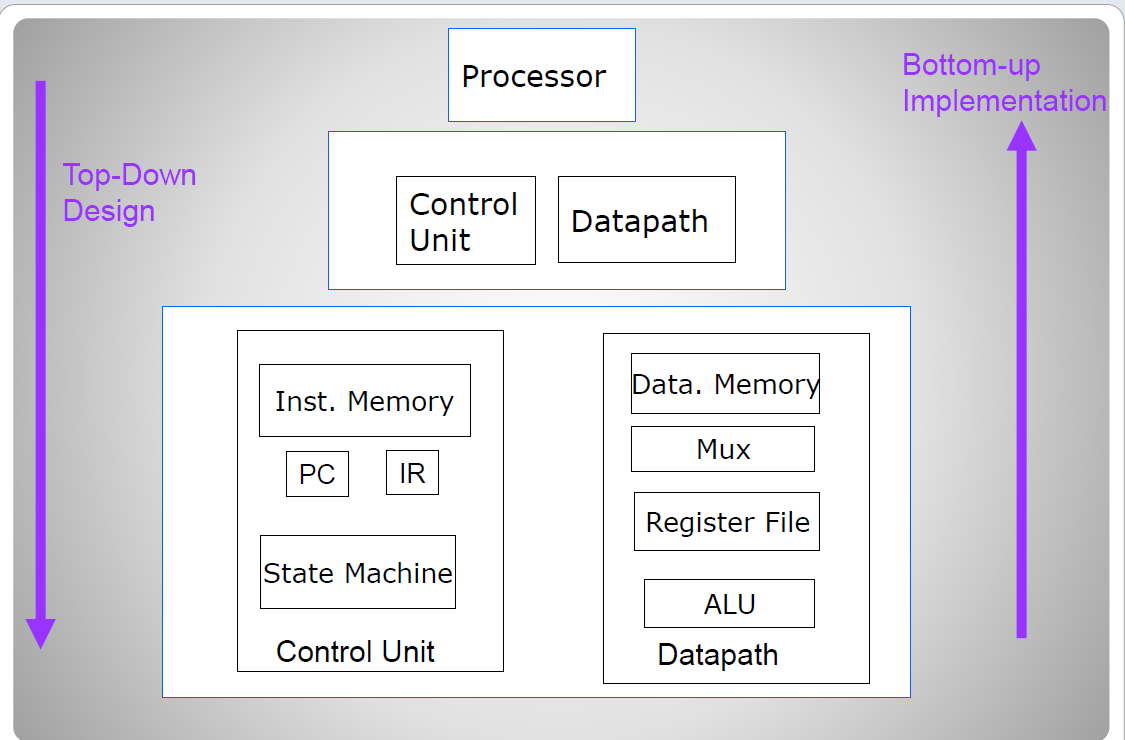
* 1. **Requirements for Controller.sv**

For designing the Control Unit, we were required to use the proper inputs and output in order to properly connect the Control Unit and the Datapath. To successfully build the Control Unit, it must load instructions from the instruction memory, where the addresses must be provided by a program counter via a shift register to the state machine (FSM). The state machine must decode the instruction and assign the different state and output values to process the current instruction.

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

When designing the Datapath module, we were required to properly connect the register file, the ALU, the data memory, and the 2-to-1 multiplexer. The Datapath module must be able to load values from the data memory to the register file, perform operations with values from the register file using the ALU, and store the result from the ALU back into the data memory by taking it from the register file.

**Figure 1:** Datapath interconnection schematic

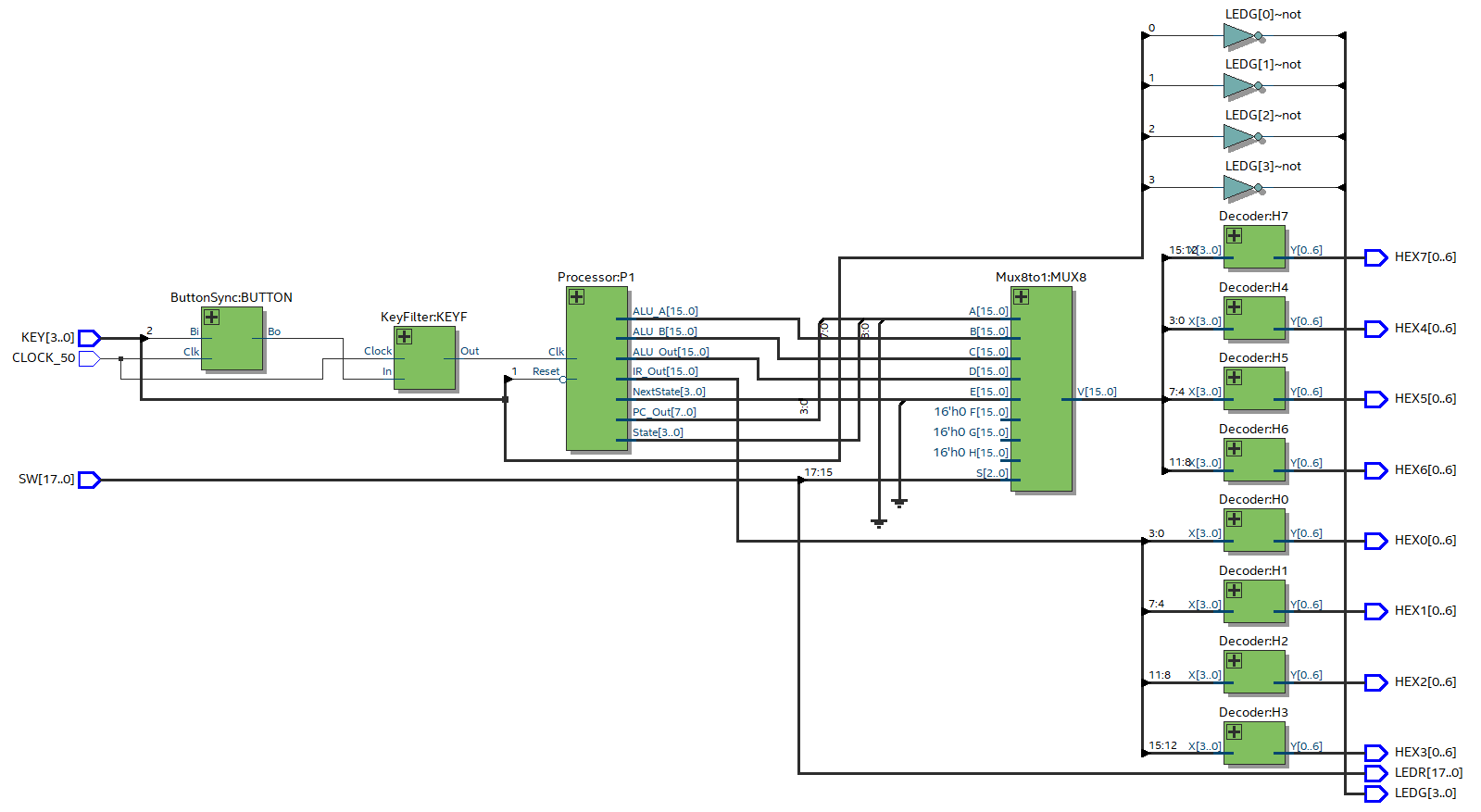


* 1. **Requirements: Processor.sv**

The upper-level module, Processor, must instantiate the Control Unit and Datapath so that they properly interact with each other. They must together perform the 7 different instructions that are discussed later in section 2.1 (NOOP, HALT, LOAD, STORE, SUB, ADD).

**Figure 2:** Processor hierarchy diagram

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

The top-level module, Project, must instantiate the Processor, KeyFilter, ButtonSync, and Decoder modules. This module allows us to interface with the DE2 board and display the contents of the instruction register, the PC counter, the current/next state, and the ALU inputs/output.

**Figure 3:** RTL view of Quartus Project

**2. Design**

The project was designed using the layout shown in Figure 1, 2, 3, and 4, provided in Programable Processor document. The objective of this project was to design a Processor that will execute a range of operations by instructing an ALU on which operation to execute in time.

**2.1 FSM.sv**

The FSM is a state machine that takes a 16-bit instruction as input from the instruction memory through the shift register. Before the initial state, all outputs are set to their default value. At its initial state, the FSM clears the program counter (PC), by assigning the output PC\_Clr to 1 and sets the next state to Fetch. The Fetch state assigns the output PC-Up to 1 along with the IR\_ld in order to proceed to the load of the next instruction, while setting the next state to Decode. In Decode, the FSM deciphers the four most significant bits of the instruction, which represent the Opcode and the remaining digits contain relevant memory address. According to the Opcode, the decode state sets the next state to the appropriate state. The state machine contains 10 states:

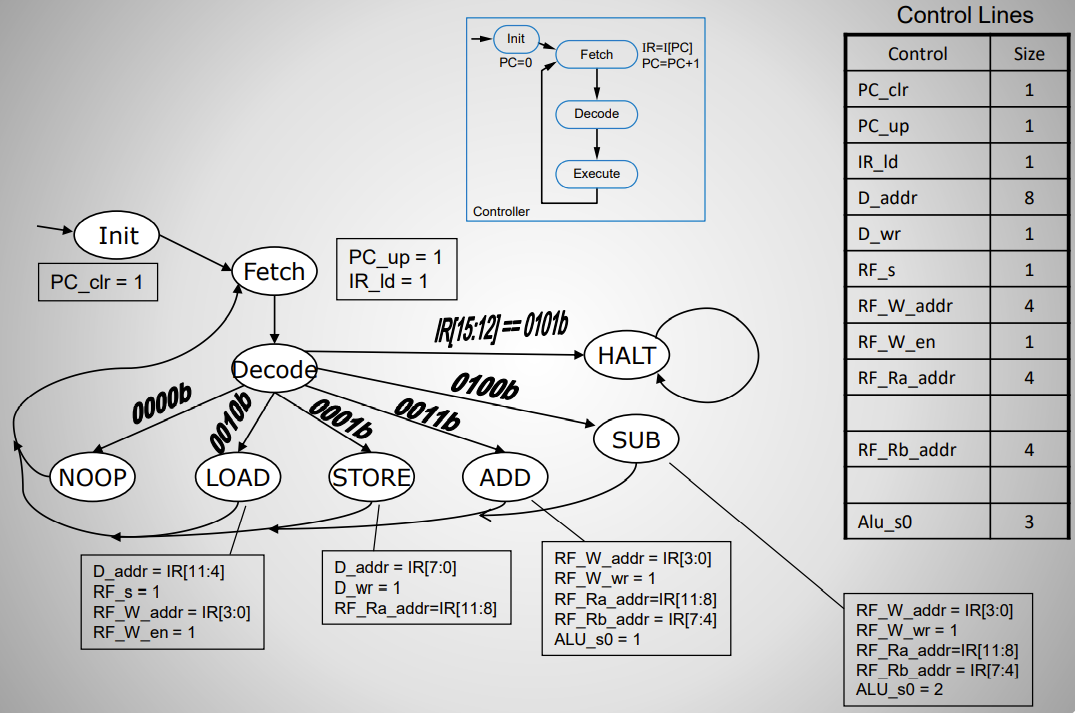
* Init – Clears the PC and sets the next state to Fetch
* Fetch – Loads the next instruction from the instruction register and sets the next state to Decode.
* Decode – Increments the PC and decides which state is next based in the first 4 bits of the instruction.
* NOOP – No operations are made and set next to Fetch.
* Load\_A – Pulls the data address from the instruction and sets next to Load\_B.
* Load\_B – Pulls the data address and the register file address from the previous state and sets the write enable to true. Set the next to Fetch.
* Store – Gets the register file address from the instruction and sets data address from the instruction. Sets data write to true and sets the next state to Fetch.
* ADD– Gets the register file address for A and B from the instruction and sets the ALU instruction to perform (ADD operation). Sets write address, write enable and next state to Fetch.
* SUB – Gets the register file address for A and B from the instruction and sets the ALU instruction to perform (SUB operation). Sets write address, write enable and next state to Fetch.
* Halt – Does nothing and sets next to Halt, causing the processor to stop and stay in this state unless it is reset.

Each instruction is described in more detail in Table 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 16 bits Instruction code | | | | |
| Instruction | OPcode | 4 bits | 4 bits | 4 bits |
| NOOP | 4 | 0 | 0 | 0 |
| STORE | 7 | Source Register | Destination Data Memory | |
| LOAD A | 5 | Source Data Memory | | Destination Register |
| LOAD B | 6 |
| ADD | 8 | Source Register A | Source Register B | Destination Register C |
| SUB | 9 | Source Register A | Source Register B | Destination Register C |
| HALT | 3 | 0 | 0 | 0 |

**Table 1:** 16-bit Instruction code description

Each FSM state assigns values to a certain number of outputs destined for the program counter, the shift register and the DataPath Unit. The list of outputs of each state is summarized in Figure 4.



**Figure 4:** State Machine (FSM) description

**FSM port list:**

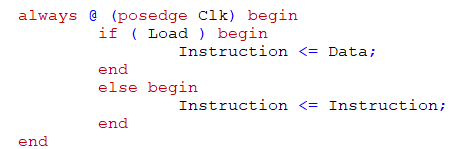
* clk – system clock input
* IR\_ld – instruction load command
* PC\_up – PC increment command
* D\_addr – Data memory address (8 bits)
* D\_wr – Data memory write enable
* RF\_s – Mux select signal
* RF\_Ra\_addr – Register file A-side read address (4 bits)
* RF\_Rb\_addr – Register file B-side read address (4 bits)
* RF\_W\_en – Register file write enable
* RF\_W\_Addr – Register file write address (4 bits)
* ALU\_s0 ALU – function select (3 bits)

**2.2 PC.sv**

The lower-level PC.sv module (level 1) is a simple counter that is incremented by 1 when at the Fetch State of the FSM. The range of the PC counter is from 0 to 127 (8-bit counter). The output value of the counter is used as address of the next instruction to read from the instruction memory file (myROM.mif), to be processed by the Processor. A clear input (from FSM) will reset the count to zero while the clock input will trigger the count, and the output provided will be a positive integer.

**2.3 IR.sv**

The lower level IR.sv module (level 1) is a simple shift register that will latch the instruction (16-bit) from the instruction memory, as an input to the FSM. The shift register loads the Instruction to the FSM at the rising edge of the clock when the load input (from the FSM) is true. We used a procedural assignment (always block) that implements the 16-bit shift register in Figure 5.

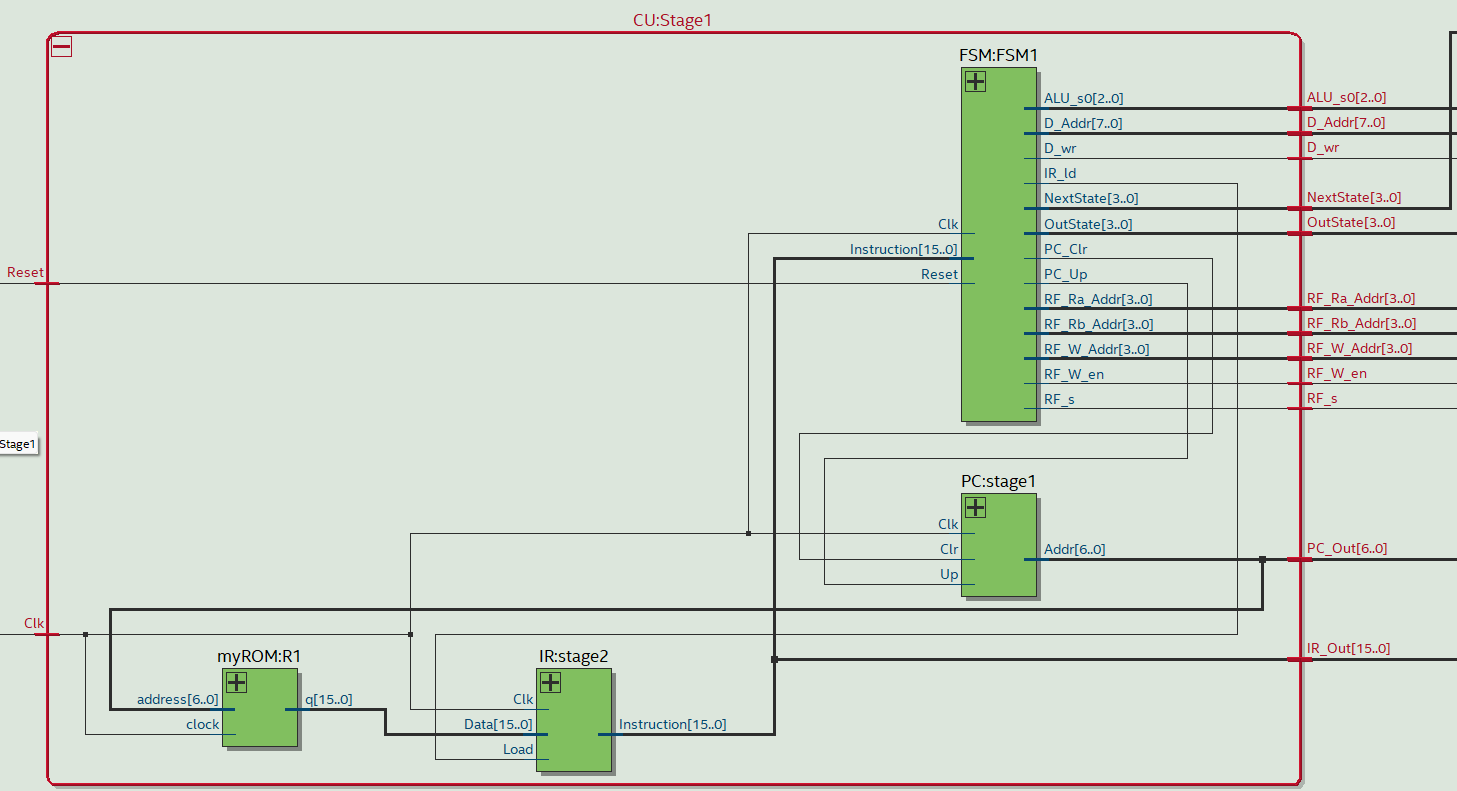


**Figure 5:** Procedural assignments to implement the 16-bit shift register

**2.4 CU.sv (Control Unit)**

The Control Unit (CU) drives the processor and instantiates the instruction memory as a 128 X 16 ROM LPM module. The CU.sv is a module of higher level (see Figure 6) that instantiates the FSM, PC, IR, and Instruction memory (myROM.v) module. The Control Unit clock and reset inputs will set and control the lower-level module, where its outputs will provide input to the DataPath unit.

**CU port list:**

* clk – system clock input
* Reset – necessary to reset the FSM
* D\_addr – Data memory address (8 bits)
* D\_wr – Data memory write enable
* RF\_s – Mux select signal
* RF\_Ra\_addr – Register file A-side read address (4 bits)
* RF\_Rb\_addr – Register file B-side read address (4 bits)
* RF\_W\_en – Register file write enable
* RF\_W\_Addr – Register file write address (4 bits)
* ALU\_s0 ALU – function select (3 bits)

**Figure 6:** RTL view of Control Unit module

**2.5 regfile16x16a.sv**

The Register File module has sixteen 16-bit slots that hold values that can be used for operating in the ALU or holding values loaded from the data memory. The register file can both read and write data. It will read data in a store operation and write data in a load or arithmetic operation.

**The port list:**

* clk – system clock input
* WriteEnable – Tells the register file whether it needs to read data or be written to
* WAddr – The address or index that the value is written to in the register file
* WData – The data being written to the register file
* RAddrA – The address of the A-side value being output from the register file
* RAddrB – The address of the B-side value being output from the register file
* RDataA – The data taken from the A-side
* RDataB – The data taken from the B-side

The register file also uses a wire called *regfile* which is a 16-bit by 16-bit wire that holds the register file contents. *RDataA* and *RDataB* are assigned to the contents of *regfile* at the indexes *RAddrA* and *RAddrB*, respectively.

Diagram, schematic

Description automatically generated**2.6 Mux\_n\_2to1.sv**

The 2-to-1 Multiplexer module is designed to take two 16-bit inputs and select between them to output one 16-bit value. When the select bit is 1, the output of the Mux is the data read from the data memory module. When the select bit is 0, the output is the output of the ALU as shown in Figure 3. We used a simple case statement inside an always block to determine the logic as shown in Figure 4

Text

Description automatically generated

**Figure 8:** Datapath interconnection schematic

**Figure 7:** Mux16w2to1 always block code

**2.7 ALU.sv**

The ALU module is designed to take two 16-bit inputs and perform one of 8 possible logic operations using a 3-bit select signal:

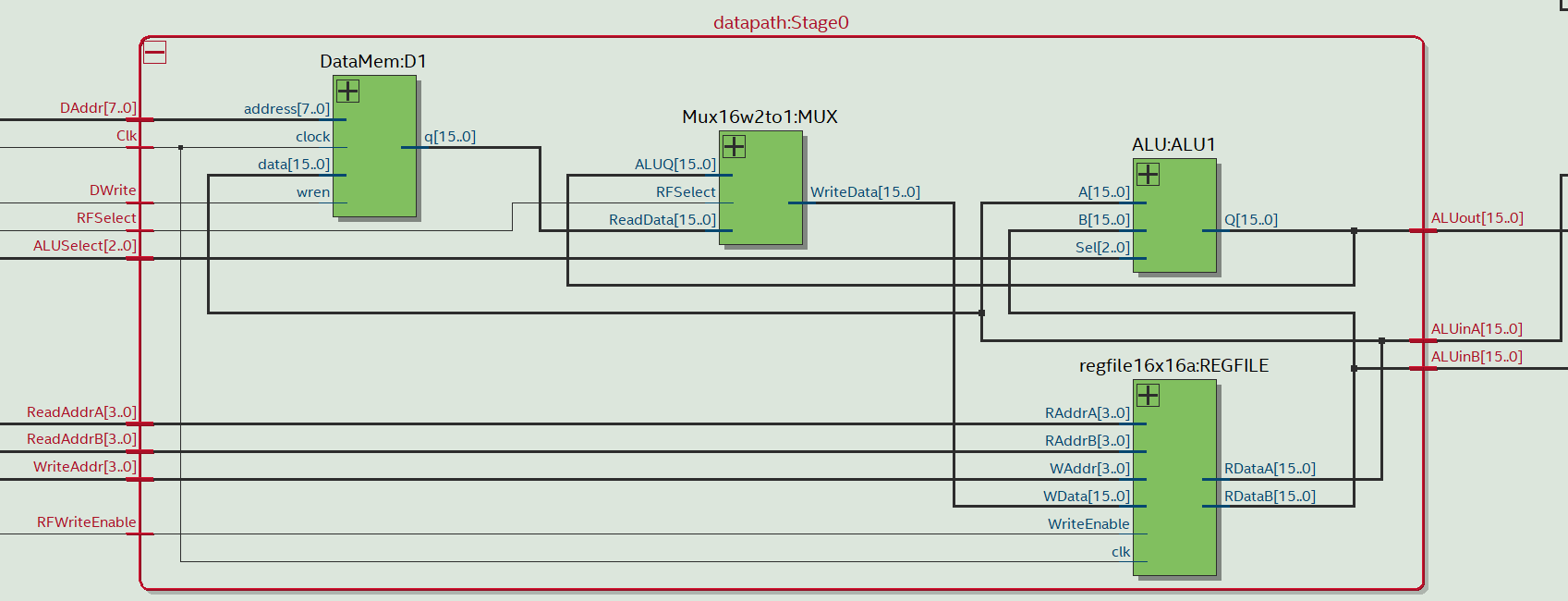
1. if s == 0 the output is 0
2. if s == 1 the output is A + B
3. if s == 2 the output is A - B
4. if s == 3 the output is A (pass-through)
5. if s == 4 the output is A ^ B
6. if s == 5 the output is A | B
7. if s == 6 the output is A & B
8. if s == 7 the output is A + 1

The two ALU inputs *RDataA* and *RDataB* come from the register file. The output of the ALU is looped back to the 2-to-1 Mux where it can be filtered into the register file. The ALU operations are done using procedural assignment with a case statement, similar to the 2-to-1 Mux.

**2.8 Datapath.sv**

The Datapath module uses the following port list:

* Clk – Clock Signal (1-bit)
* DAddr – Address of DRAM (8-bit)
* DWrite – Determines if the DRAM is writing or reading (1-bit)
* RFSelect – Register file select bit (1-bit)
* WriteAddr – Writing address of RF (4-bit)
* RFWriteEnable – Determines if RF is writing (1-bit)
* ReadAddrA – Reading address of A (4-bit)
* ReadAddrB – Reading address of B (4-bit)
* ALUSelect – Select bit of ALU (3-bit)
* ALUinA – ALU A input (16-bit)
* ALUinB – ALU B input (16-bit)
* ALUout – ALU output (16-bit)

All the input signals of the Datapath come from the Controller module, where the instructions are fed into the finite state machine which outputs the values needed by the Datapath, depending on the current state.

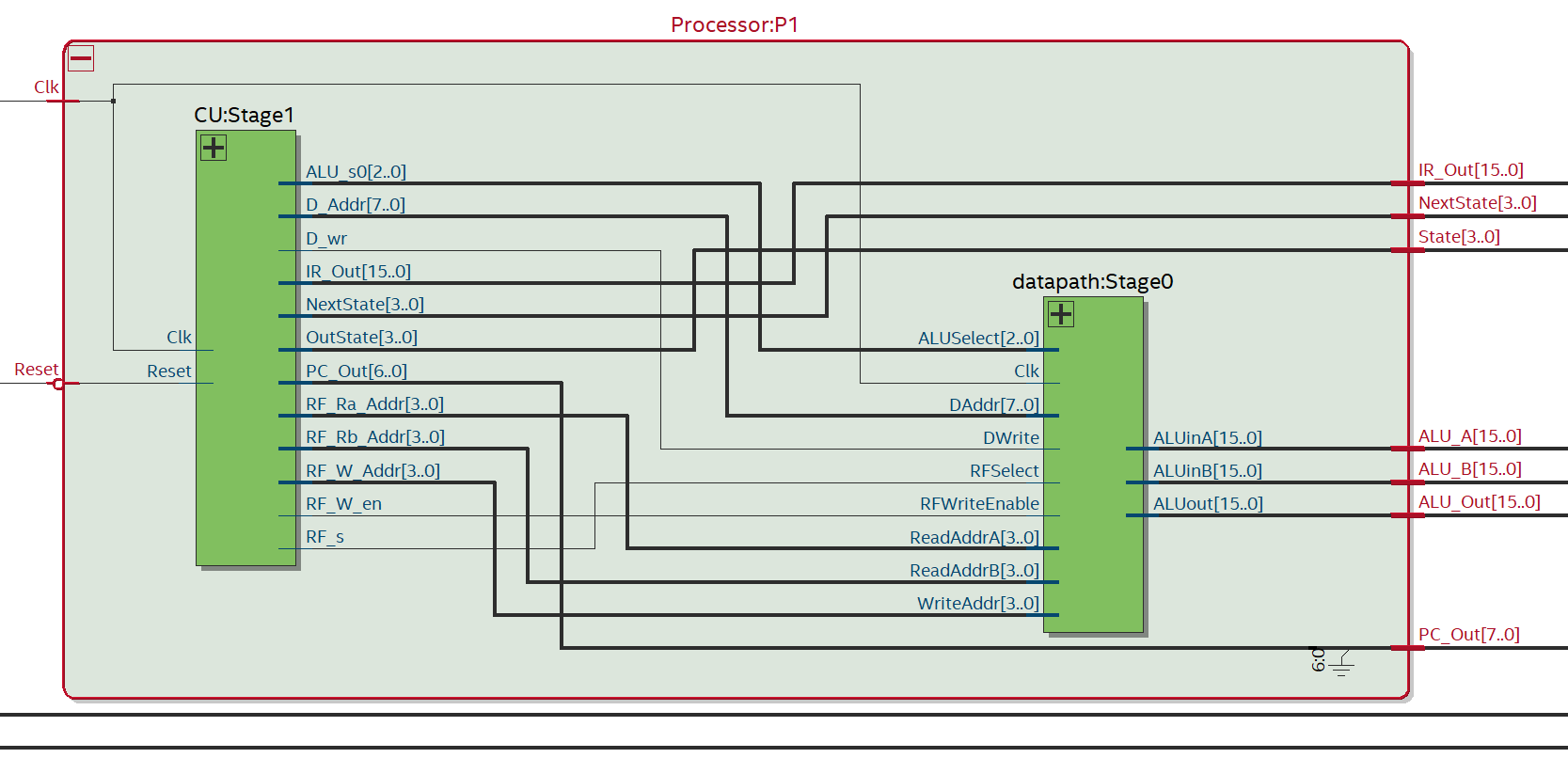
**Figure 9:** RTL view of Datapath module

**2.9 Processor.sv** (Kader)

The Processor.sv is a higher-level module that instantiates the Control Unit and the Datapath. Using Clock and Reset inputs in order to orchestrate the Control Unit and Datapath, the Processor output signals destined to the 8-to-1 Multiplexer.

**The port list:**

* clk – system clock input
* Reset – reset for the FSM
* ALUinA – ALU A input (16-bit)
* ALUinB – ALU B input (16-bit)
* ALUout – ALU output (16-bit)
* IR\_Out – Instruction in process (16-bit)
* PC\_Out – Instruction memory addresses (8-bit)
* State – Current state (4-bit)
* NextState – Next state (4-bit)

****To match the output of the Processor, *PC\_Out* (8-bits), defined by the skeleton given in the Project\_SimpleProcessor.pdf, to the *PC\_Out* (7-bits) of the Control Unit, we created an additional wire called *PC\_Out7* (7-bits) to act as the output of the Control Unit and then concatenated that wire with a 1-bit 0. This allowed us to assign the 8-bit *PC\_Out* of the Processor to the concatenated wire. Other than that, connecting this module was straightforward.

**Figure 10:** RTL view of Processor module

**2.10 Project.sv**

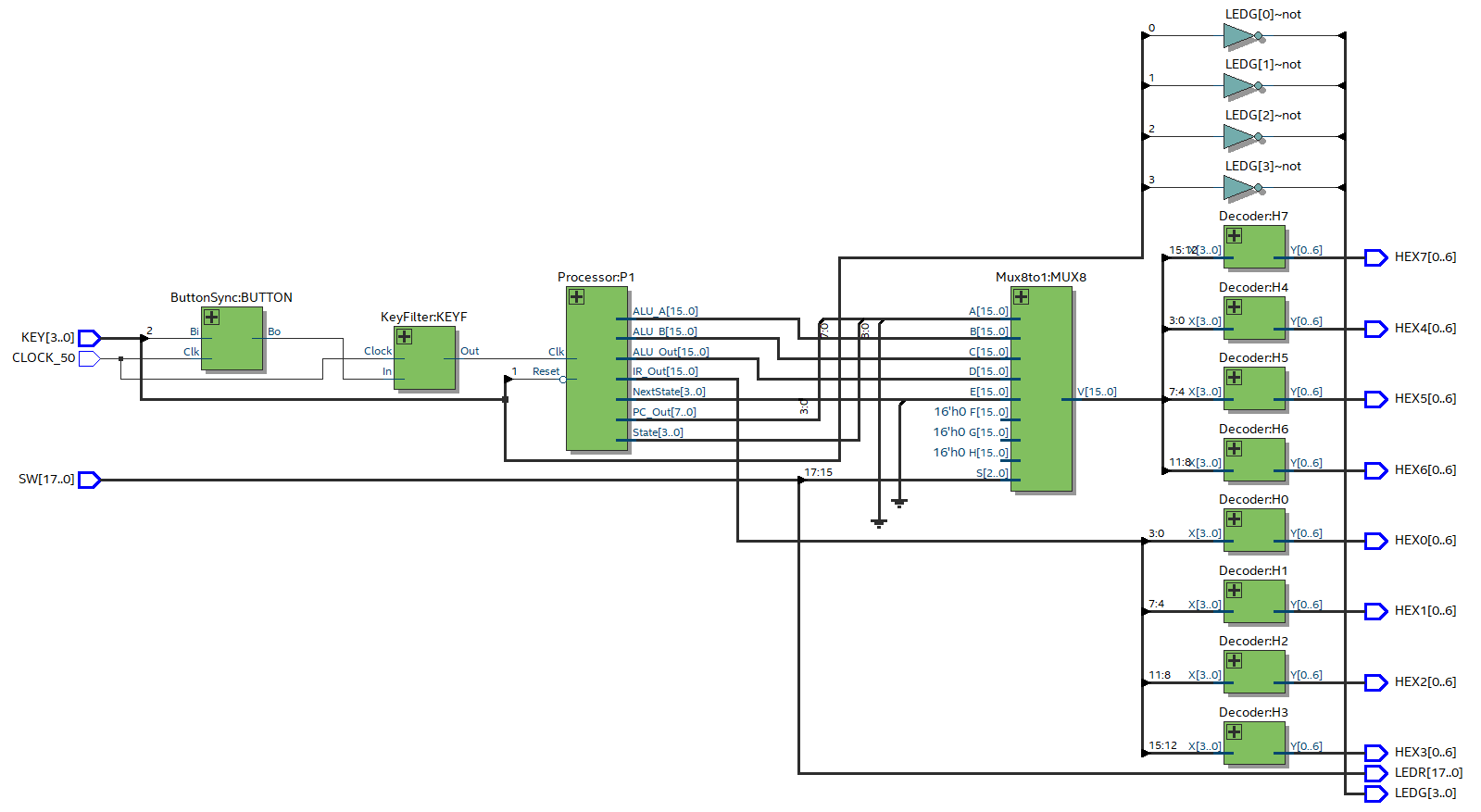
**The port list:**

* SW – Main board switches (SW[17:15] controls what is displayed on the board)
* KEY – Button Keys (KEY[2] used for clock cycling, KEY[1] used as the reset)
* HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7
* CLOCK\_50 – 50 MHz clock signal
* LEDG – Green LEDs (Used to indicate when a button key is pressed)
* LEDR – Red LEDs (Used to indicate when a main board switch is turned on)

The first module in the line is the ButtonSync module, which is designed so that a button press is only one clock period long. This means that no matter how short or long you hold the button down while pressing it, the clock will only cycle once. The output of the ButtonSync module then feeds into the KeyFilter module, which is used to combat pushbutton debouncing. Pushbutton debouncing is when the metal contacts of a switch will bounce from in-contact to not in-contact several times for a brief time when the button is initially pressed. This can cause inaccuracies in the button pressing. The output of the KeyFilter is then fed into the clock input of the Processor. The Processor module will cycle through the states in the state machine, reading instructions from the instruction memory and then enacting the instructions with the help of the Datapath. The outputs of the processor feed into an 8-to-1 multiplexer, where a 3-bit select input, controlled by the main board switches, is used to display different output values from the Processor. Next, the outputs of the Mux feed into 8 different decoder modules that interact with the 7-segment displays. HEX0 through HEX3 always show the contents of the IR.

SW[17:15] determines what HEX 7, 6, 5, and 4 display as follows:

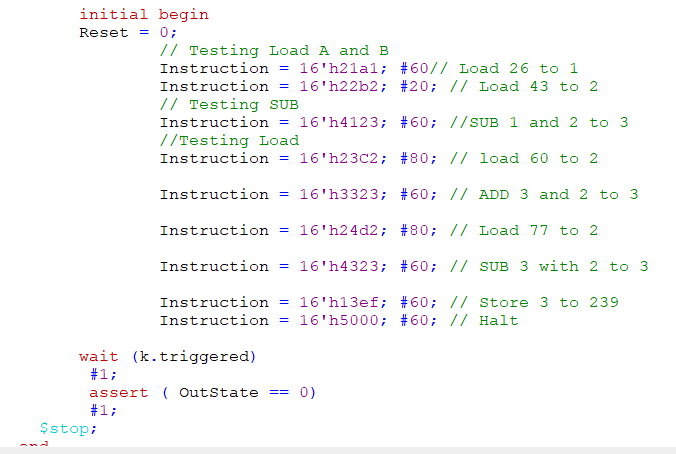
* 0 => HEX7, HEX6 = PC; HEX5, HEX4 = Current State
* 1 => HEX7, 6, 5, 4 = ALU\_A (A-side input to ALU)
* 2 => HEX7, 6, 5, 4 = ALU\_B (B-side input to ALU)
* 3 => HEX7, 6, 5, 4 = ALU\_Out (ALU output)
* 4 => Next State (FSM next state variable, if you use one)
* 5 => Unused
* 6 => Unused
* 7 => Unused



**Figure 11:** RTL view of Quartus Project

**3. Test Procedures**

In the following sections, we will discuss the testing details of the FSM, the Control Unit, and the Datapath.

**3.1 FSM\_tb**

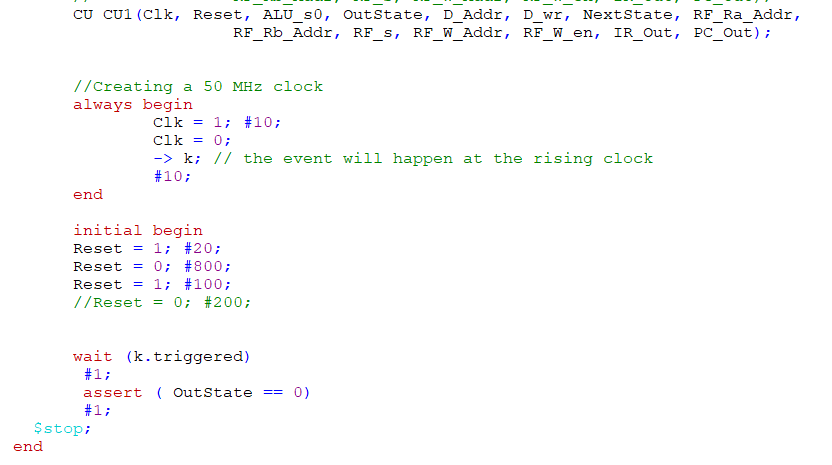
In order to test the FSM module, we built the FSM testbench that simulated the pseudo code provided in the programmable processor guide. At the time of the FSM’s testing, we had not yet built the Control Unit. After translating the pseudo-code into Instruction format, we instantiated the FSM module using the instructions as shown in Figure 12, see transcript in Figure 18 in Section 4.1 of the report.

**Figure 12:**FSM’s Testbench preview

Noted that for the simulation of the FSM we need to use the runrtl.do file, which is different from the usual runlab.do. Also, we had to manually impose a time interval for each instruction allowing them to be completed.

**3.2 CU\_tb**

The CU test bench like the FSM test bench will simulate the provided pseudo code. Because the CU instantiates the FSM, PC, IR, and the instruction memory, we only needed to simulate a clock and reset signal as input to the CU module. In Figure 13, we created a 50 MHz clock using an always block, while setting the reset signal to 1 to clear any unexpected current state and then to 0 for 800 ns, time necessary to run the 8 instructions contain in the Instruction memory file. The test bench ends with the reset signal to 1, in order to clear the FSM state. See transcript of the CU’s testbench in Figure 19 in Part 4.2 of the report.



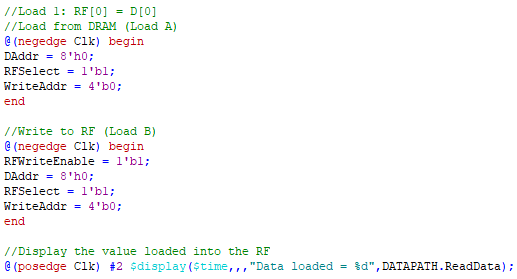
**Figure 13:** CU’s testbench preview

**3.3 Datapath\_tb**

The Datapath testbench was designed to test the following set of instructions:

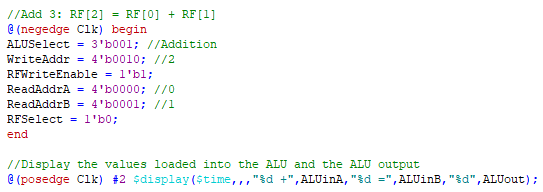
1. Load : RF[0] = D[0]
2. Load : RF[1] = D[1]
3. Add : RF[2] = RF[0] + RF[1]
4. Sub : RF[3] = RF[0] - RF[1]
5. Store : D[9] = RF[2]
6. Store : D[10] = RF[3]

We designed the testbench so that each instruction was triggered on the falling edge and the displayed values would trigger on the next rising edge. This way, the values running through the Datapath had time to load so displaying them after each instruction was not an issue. When testing the load instruction, we had to ensure that we dedicated a full clock cycle to both Load\_A and Load\_B. In Load\_A, we only specified the data memory address, the register file address, and the register file select bit. In Load\_B, we specified the register file write enable. After each load instruction, we displayed the value loaded into the register file.

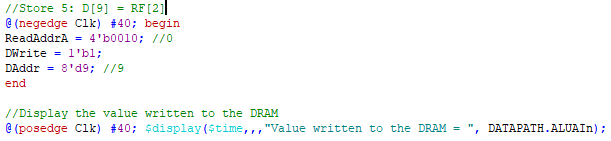


**Figure 14:** SystemVerilog code for a LOAD instruction in the Datapath testbench

Next, we tested the arithmetic instructions, SUB and ADD. The code used to perform the SUB and ADD operations is almost identical. The only differences were the ALU select bit value and the register where the result was stored. For subtraction, the select bit value must be 2, and for addition, the value must be 1. After each arithmetic instruction, we displayed the ALU inputs and output.

****

**Figure 15:** SystemVerilog code for an ADD instruction in the Datapath testbench

****Lastly, we tested the store instruction. We stored the results of the addition and subtraction into the data memory. We specified *ReadAddrA* (the address of the value being taken from the RF), *DWrite* (sets the DRAM to writing mode), and *DAddr* (address of the value being stored in the DRAM). After each store instruction, we displayed the value written to the DRAM.

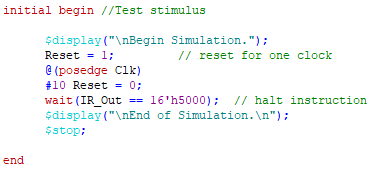
**Figure 16:** SystemVerilog code for a STORE instruction in the Datapath testbench

**3.4 testProcessor**

To test the Processor module, we used the testbench provided in the file *testProcessor*. However, we had to make a slight alteration. Since our counter uses an active-high reset, we had to invert the reset values in the testbench. We converted the following instructions into Hexadecimal and loaded them into the instruction memory:

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

We also preloaded the following values into the data memory:

* D[1A] = 0x10AC
* D[2B] = 0xCC05
* D[3C] = 0x01B5
* D[4D] = 0xA040

The testbench is designed so that the reset signal is initially on for one clock cycle and then turned off, allowing the instructions to cycle through until the IR contains the value 16’h5000 (Halt instruction). The output signals are monitored using a $monitor statement in a separate initial block.

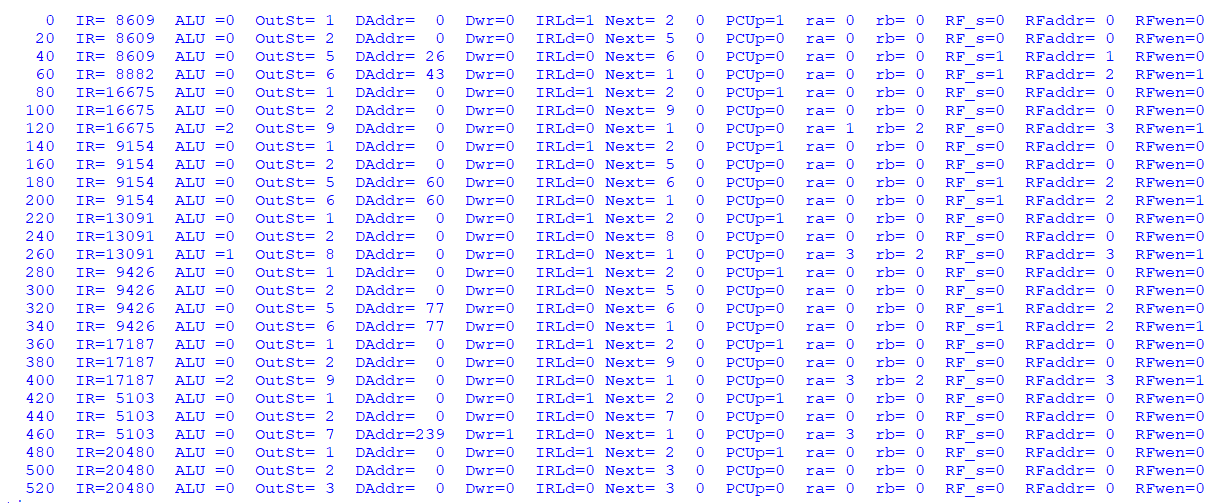
**Figure 17:** SystemVerilog code showing the input sequence of the Processor testbench

**4. Test Results**

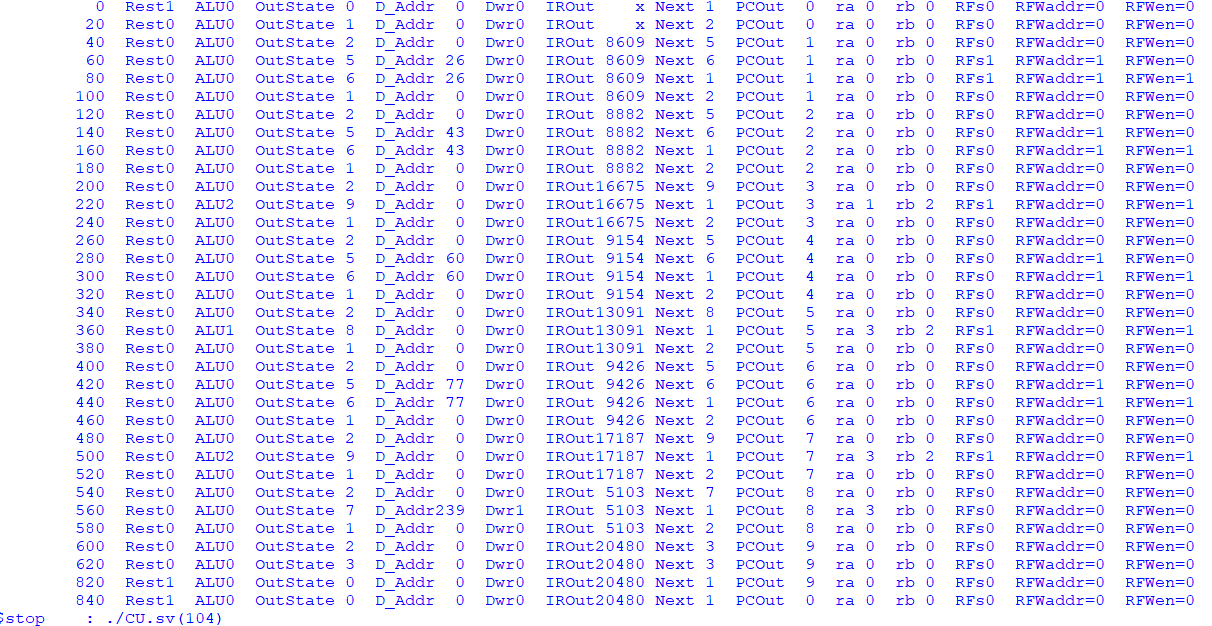
In this section, we will describe the results of the testbench files for the FSM, Control Unit, Datapath, and Processor modules.

**4.1 FSM\_tb**

The FSM testbench result in Figure 18, demonstrate the proper operation of the FSM module. By analyzing the FSM output, we can see that the FSM module operates the instructions as expected. In the first 60 ns, the FSM state change from initial, Fetch, Decode, Load\_A and Load\_B. IR output represents the instruction in decimal number, and because the first instruction is the load (IR\_ld=1 at 0 ns) the data at data memory address 26 (D\_Addr = 26 at 40 ns) to the register file 3 (RF\_Addr = 3 at 40 ns). At the same time, the mux select line (RF\_s= 1) allowed the data from the data memory to access the register file. The testbench transcript shows that all instruction was operated properly.

**4.2 CU.sv (Control Unit)**

**Figure 18:** Transcript of the FSM’s Testbench

As for the FSM testbench, the CU testbench transcript demonstrates the proper operation of the CU module. We observed that both transcripts are identical and show the same output value for all instruction that was processed. However, we noted that the clock signal orchestrated all submodules and completed one instruction after the other. In the other hand, because we manually force the change of state for each instruction in the FSM testbench, we forced 2 load operations in one.

**Figure 19:** Transcript of the CU’s Testbench

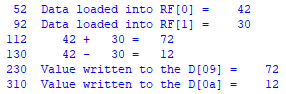
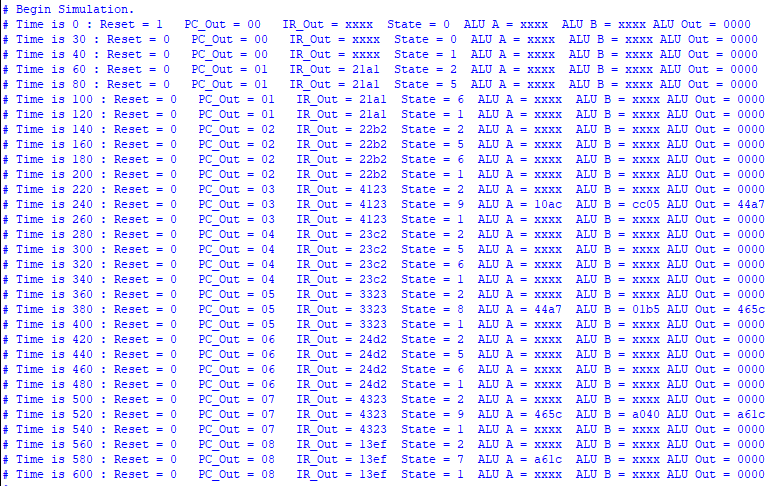
**4.3 Datapath\_tb**

Figure 20 shows the results of the Datapath testbench in the ModelSim transcript. We can see that the value 42 is loaded into RF[0] and 30 is loaded into RF[1]. Next, the addition and subtraction take place giving us 72 and 12. Lastly, the values are written to the data memory in slots 9 and 10.

**Figure 20:** Transcript of the Datapath testbench

**4.4 testProcessor**

**Figure 21:** Transcript of the Processor testbench

Figure 21 shows the results of the Processor testbench in the ModelSim transcript. We can see that after the reset goes to 0, the PC begins counting and the current state begins to change in accordance with the instructions in the instruction memory. The first instruction to go through is a load instruction with a value of 21A1 in the IR, which goes from state 1 => 2 => 5 => 6. The next instruction is also a load instruction, with a value of 22B2 in the IR. Next, the subtraction takes place with a value of 4123 in the IR, which goes from state 1 => 2 => 9. At this point, we can also see that ALU\_A and ALU\_B are showing the loaded values from the data memory (10AC and CC05) and the output is showing the result of the subtraction (44A7). Next, we load another value from data memory address 3C. After loading that value, we add it to the result of the earlier subtraction, getting a value of 465C Once more, we load a value, this time from data memory address 4D. We then subtract this value from the result of the earlier addition to get an output of A61C. Next, this output is stored into the data memory at address 3 using a store instruction with a value of 13EF in the IR.

**5. Observations**

In this section, we share our observations for each submodule and upper-level module.

**5.1 regfile16x16a.sv**

The design of the register file module was very straightforward. We were given a template where we only needed to adjust the number of registers. When testing the register file, we had to find a way to read and write data to it. It worked best when we wrote data to the register file first and then read it afterwards.

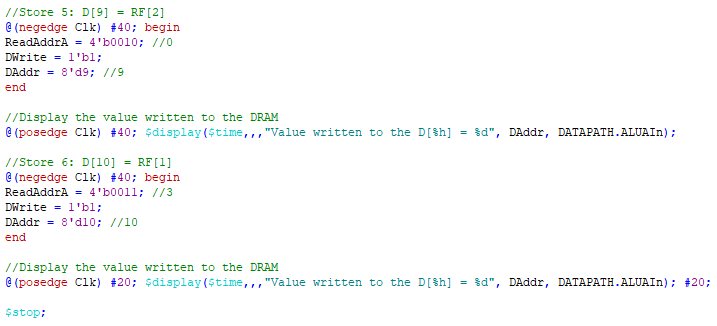
**5.2 Mux\_n\_2to1.sv**

The design of the 2-to-1 Mux module was also very straightforward. We only needed to modify a 2-to-1 Mux we have made previously to use 16-bit inputs and outputs. Testing the Mux was the simple and not worth commenting on.

**5.3 ALU.sv**

The design of the ALU module was fairly straightforward. There was a truncation warning that we encountered only when running the top-level Project module. In the case where the value A is incremented by 1, we changed it from simply “A + 1” to “A + 16’h1.” Another thing to note is that there is no carry bit so when the numbers are very large, there will likely be an incorrect result.

**5.4 Datapath.sv**

The design of the Datapath had some hurdles. The main module was fairly straightforward except for an issue caused by connecting the outputs of the Datapath to the inputs of a submodule inside the Datapath. This problem was solved by creating wires and assigning the outputs to the wire values. The largest struggle in the entire project was in designing the Datapath testbench. For a long time, we couldn’t see the values in the data memory change when testing a store instruction. This was fixed by adding more time between the instructions right after the @(negedge) and @(posedge) statements as shown below in Figure 22.

**Figure 22:** SystemVerilog code the store instructions in the Datapath testbench

(Note the extra time after each @ statement)

**5.5 FSM.sv**

The FSM design was not the most challenging but time consuming. By following the slides “Processor2022” provided by the instructor, the functionality of the FSM was easy to understand. The use of the state machine template simplifies the design. We encountered some bumps during the testbench due to the omission of setting the output to the default value in the always loop before evaluating the different state case. It was clear which output should change when changing the state statue. We had to experiment multiple time intervals for each instruction we tested in the testbench before we could see all the instructions completely processed.

**5.6 PC.sv**

The Program counter design was straightforward using the procedural statement. Inside the always block, we used the non-blocking assignment to evaluate the output signal at each rising edge of the clock cycle. We reused the CounterSixteen from Lab 5 and modify the output size to 8-bit in order to count to 127 and used an active high reset.

**5.7 IR.sv**

The IR module is a simple shift register that latches the 16-bit data input when the input load is high. Instead of instantiating 16 flip flops, we used an always block with non-blocking assignment that evaluates the output data at each rising edge of the clock when load is high.

**5.8 CU.sv (Control Unit)**

The CU module was very straightforward to design and did not present any issue. The instantiation of the submodule FSM, PC, IR, and the instruction memory was correct at the first trial. As for testing the FSM, we used the runrtl.do file for its simulation.

**5.9 Processor.sv**

The processor module showed some issues when we were testing it. One issue was that was previously mentioned in section 3.4 was that provided testProcessor file used an active-low clear when our counter used an active-high clear. We fixed this by inverting the reset values in the testbench file. Another issue we found was that the output *PC\_Out* of the Control Unit was 7 bits but the *PC\_Out* of the Processor module was set to be 8 bits. We solved this by creating a 7-bit wire called *PC\_Out7* that acted as the output of the CU and then concatenated it with a 0 to make an 8-bit *PC\_Out*. This was also mentioned in Section 2.9.

**5.10 Project.sv**

The top-level module had a couple small issues. We accidentally did not instantiate the KeyFilter with the correct clock input. We were using the original clock signal name *Clk* when we should have used *CLOCK\_50*. Another warning that was also mentioned in the section 5.3, was the truncation warning for the ALU.

**6. Conclusion**

In this Project, we successfully built a Programmable Processor that interfaces with the DE2 FPGA board. In the design and testing of the Processor, its submodules, and the Quartus Project, we learned several key concepts: how to make RAM/ROM LPM modules, how to write testbenches for more complex, higher-level modules, how to organize files efficiently, and how to work effectively in a group of two. The most difficult part of this project was creating the testbench for each module, namely the Control Unit and Datapath. The Datapath testbench had several roadblocks that we had to overcome. One of these was the timing of these instructions. We needed to effectively space out the instructions in time to ensure that each one had time to finish processing before the next instruction was loaded. This was especially evident with the store instruction. We had to add significantly more time delay after the store instructions compared to the load and arithmetic instructions. Otherwise, the Data Memory would not properly update the results. This took much of our time in the project, but it was critical to our understanding. We also learned that keeping every module file in the same folder was more efficient, even though it doesn’t look very organized. Since we only had one folder, we only had one version for each module. This meant that if we needed to make a change to a module or a .mif file, we would only have to change it in one location. When helping other groups, we noticed that when they had separate folders for different modules, the changes didn’t transfer over unless they deliberately changed the files in both locations. This caused them to have a lot of errors. Overall, one of the most important things we learned was effective communication. We both replied to messages in a timely manner and made time to meet in person to discuss things when we got stuck. Because of this, I believe we had great teamwork that helped us complete the project in a timely manner.

**7. References**

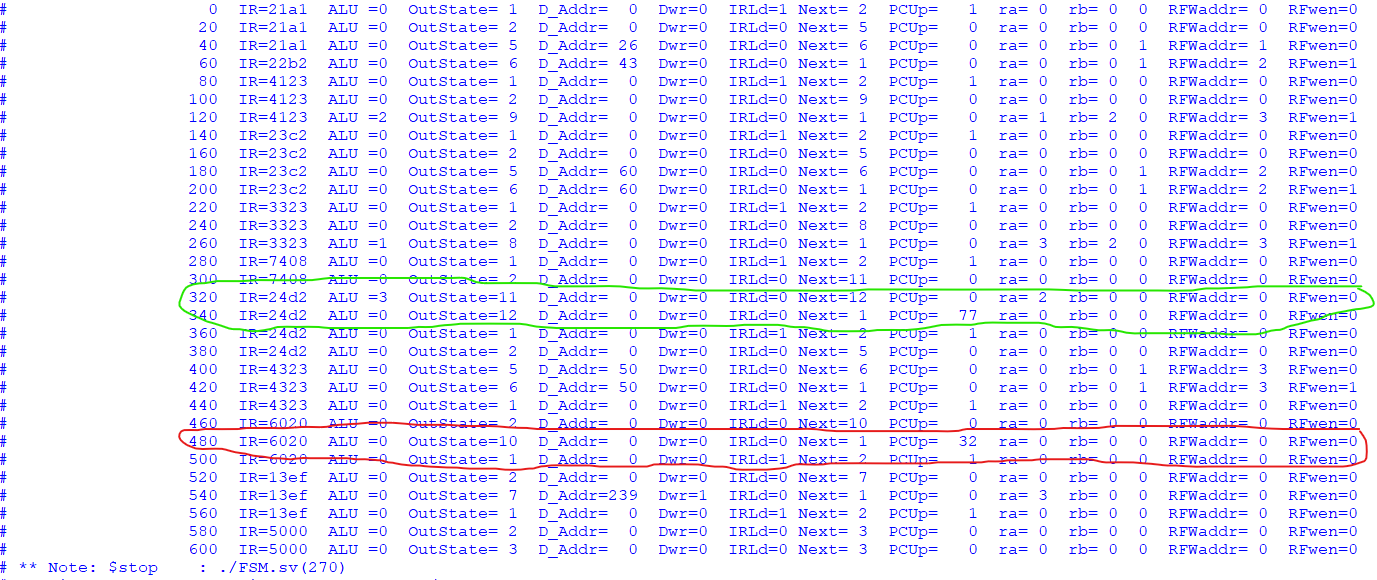
**Course Documents:**

* Processor2021\_StableClkByButton.pdf
* AFirstLookAtProcessor2022.pdf
* Processor2022\_cont.pdf
* Processor2022\_start.pdf
* Project\_SimpleProcessor.pdf

**Instructor-Provided Files:**

* ButtonSync.sv (and relevant .do files)
* KeyFilter.sv
* testProcessor.sv

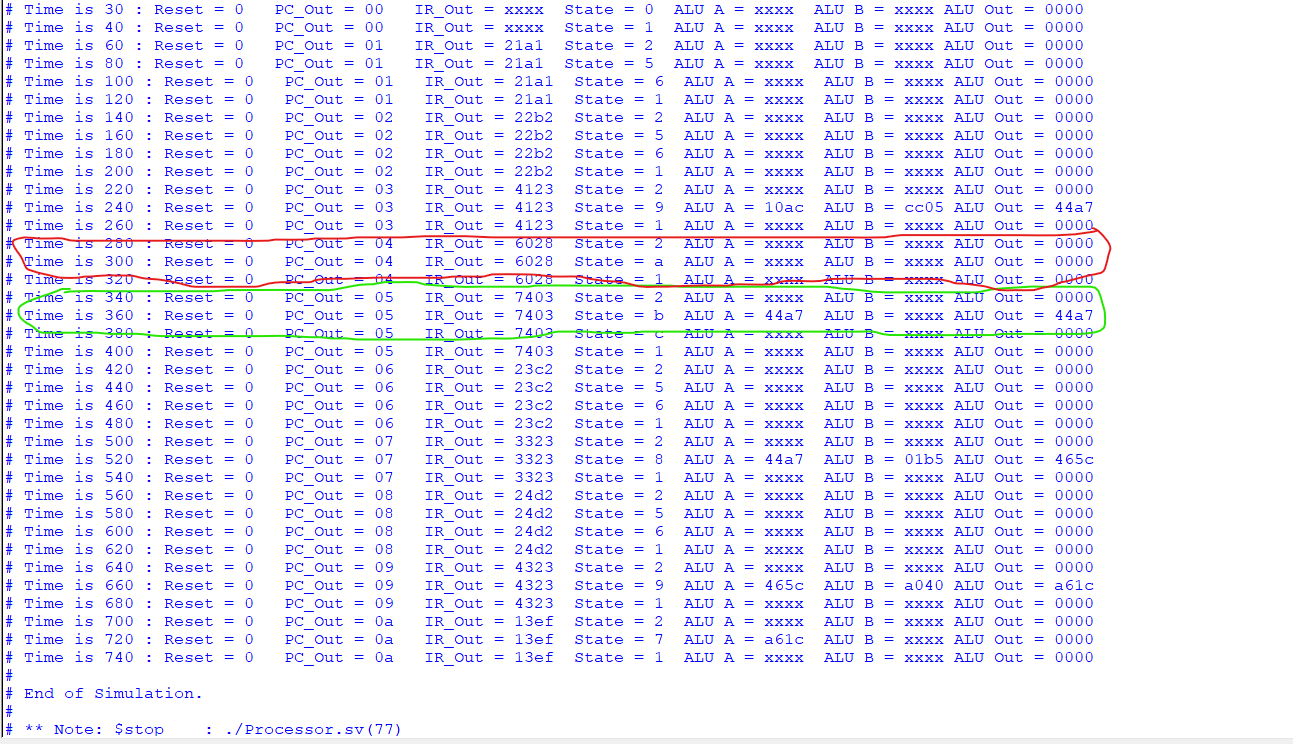
**Appendix**

We implemented the simple JUMP and the JUMP if true as extra credits. In Figure 23, we took a screen shot of the transcript of the FSM simulation with a simple Jump and a JUMP if non-zero in the same simulation. We circled in red the simple JUMP, the hexadecimal Instruction processed was 6020, which instructed the FSM to assign 32 to PC\_Up output. Once the Program counter received the PC\_Up input it added 32 to the current count. We also observed the current state value (Outstate = 10) which represents the simple Jump State. In green, we circled the JUMP if nonzero. The hexadecimal instruction processed was 24d2, which instructed the FSM to assign 77 to PC\_Up output. We observed that the register A value was equal to 2 (ra = 2), which indicates that the ALU\_A has access to register A.

**Figure 23:**Transcript of the FSM simulation for simple JUMP and JUMP if nonzero

In Figure 24, we took a screen shot of the transcript of the Processor simulation with a simple Jump and a JUMP if non-zero in the same simulation.

We circled in red the simple JUMP, the hexadecimal Instruction processed was 6020, which instructed the processor to execute a JUMP of 32, with no other operations. We observed the current state value = 10 (State = a) which represents the simple Jump State.

In green we circle the JUMP if non-zero, the hexadecimal Instruction processed was 7403, which instructed the Processor to execute a JUMP of 64 and to give the ALU\_A access to the register 3 to read and assign its value to ALU-Out.

**Figure 24:**Transcript of the Processor simulation for simple JUMP and JUMP if non-zero