ECE3270 Digital Computer Design: Lab 4

SIMPLE PROCESSOR

March 30, 2019

Aayahna Herbert
Clemson University
Department of Electrical and Computer Engineering
aayahnh@g.clemson.edu

Abstract

The goal of this lab was to implement a simple, 4-instruction processor in VHDL where the top level entity is a structural design and all lower level components are kept completely modular and use VHDL generics. This design lab was split into two parts: (1) completing a VHDL design of each part of the processor (registers, adder/subtractor, multiplexer, and control unit finite state machine) and (2) implementing the processor on a DE1-SoC board. In this lab, all parts of the lab were successfully designed, tested, and executed on the DE1-SoC board and through simulation done in ModelSim.

1 Introduction

According to Chapter 8 of Brown [1],a processor is a digital system that performs different types of operations: loading, moving, adding, and subtracting. A processor is usually made up of some number of registers, a multiplexer (or tri-state buffers for each registers), a add-sub unit and two registers associated with it, a finite state machine as it control unit, and another register for storing the processor's data used for the different operations. The next sections will go into details about how each part of the lab was constructed and the reason it was done the way it was.

2 Registers

The registers for the processor are D flip-flops with an input, output, enabler, and clock. For the simple processor, the registers are used to store and transfer data within the processor with other registers. The generic design for the registers reflects this description and based off the register design found in the appendix of Smith [2].

2.1 Design Method

The design of the generic register was made up an entity and a behavioral architecture for the entity. The entity consists of a data holder (d), a clock (clk), and an enable pin (en) as its inputs and a data holder (q) as its output. Because it needed to be flexible with the bits it can hold, a generic was used and initially set to sixteen for the bit size for the registers to be used for the processor.

A behavioral architecture was made for the register in order to make a process to set some conditions for the register and when it can take in new data for its data holder (q). The process is dependent on the clock (clk) and the enable pin (en); it states that if rising edge is detected on the clock and the enable pin is set to HIGH - meaning the register is active and open for business - then the data being kept in the input data holder (d) will be stored in the output data holder (q) so the register will represent that new data.

2.2 Testing and Simulation

When it came to testing the generic register with a testbench, the test cases, shown in Figure 10.1 were made to check how the register would function when enable is HIGH and LOW and a rising edge was and wasn't detected. When simulated in ModelSim, the waveforms, as displayed in Figure 10.2, reflected the correctly expected results from the testbench, making it successful.

3 Multiplexer

The processor's multiplexer is able to select from the incoming data and nine registers which of their data values will be able to be the output and sent out as the output for

the bus. The design for the multiplexer reflects this description and was based off the multiplexer design found on page 343 of Brown [1].

3.1 Design Method

The design of the multiplexer was made up an entity and a behavioral architecture for the entity. The entity consists of a 10-bit selector signal (sel) and ten different signals to hold the data values for the incoming data signal (din) and the nine registers as its inputs and a data holder (bout) as its output. Because it needed to be flexible with the bits it can hold, a generic was used and initially set to sixteen for the bit size for the multiplexer's input signals to be used for the processor.

A behavioral architecture was made for the multiplexer in order make process to choose what the output of the multiplexer will be based off what is selected by the signal (sel). The process is dependent on the selector signal (sel); using a case-when statement, the process will set the output (bout) to one of the ten input signals that corresponds to one of the values of (sel). The selector's values were made using a one-hot approach since only one of the signals can be active at a time; this approach makes the design easier to understand. The default selection for the multiplier was set as the incoming data signal (din) based off the design specifications laid out in this lab's PDF description overview.

3.2 Testing and Simulation

When it came to testing the multiplexer with a testbench, the test cases, shown in Figure 10.3 were made to check each input signal's value was being appropriately outputted when the selector's value corresponded to the correct input signal and the default signal was outputted when the selector's value was one not specifically set in the architecture. When simulated in ModelSim, the waveforms, as displayed in Figure 10.4, reflected the correctly expected results from the testbench, making it successful.

4 Add-Sub Unit

The processor's add-sub unit is able to add or subtract the two values put in it directly and by a register depending on the processor's instructions and then output the sum or difference on a register specifically for the result of the unit. The design for the add-sub unit reflects this description and was based off on a generic version of the ripple carry adder that was design in a previous lab.

4.1 Design Method

The design of the generic add-sub unit was made up an entity and a behavioral, structural mixed architecture for the entity. The entity consists of two data values (A and B) for the adding and subtraction and a carry-in bit (AorS) that determines whether the unit will add or subtract as its inputs and a data value (ANSWER) for the and carry-out bit (COUT) as its output. Because it needed to be flexible with the bits it can hold, a

generic was used and initially set to sixteen for the bit size for the registers to be used for the processor.

A mixed architecture of behavior and structure was made for the add-sub unit in order to include a generic version of the ripple carry adder to the design as a component along with a signal to map to the unit and to make a process to tell the unit when to add or subtract the inputs (A) and (B) depending on (AorS)'s value. The process is dependent on the carry-int bit (AorS); it states that if the bit is set HIGH then the unit needs to subtract - (B) is inverted and then added to (A) and the carry-in bit - and if the bit is set LOW then the unit needs to add (A), (B), and the carry-in bit. A signal (intB) is used as a middle man for (B) so the entity has time to figure out what to make (B) depending on if the unit is wanting to add or subtract.

4.2 Testing and Simulation

When it came to testing the add-sub unit with a testbench, the test cases, shown in Figure 10.5 were made to check how the unit would function when adding normally and subtracting with (A) being greater than and less than (B) to test if the answer will show up as positive and negative numbers when expected. When simulated in ModelSim, the waveforms, as displayed in Figure 10.6, reflected the correctly expected results from the testbench, making it successful.

5 Control Unit

The control unit for the processor controls the different types of operations that can be performed. The control unit can perform four operations: (1) mv (2) mvi (3) add (4) sub. The unit will then set all signals to latch the registers, select the appropriate output from the multiplexer, and set appropriate done and add-sub signals. The design for the control unit reflects this description and was based off on a mealy finite state machine.

5.1 Design Method

The design of the control unit was made up an entity and a behavioral architecture for the entity. The entity consists of a clock (clk), an inverted reset (resetn), a run bit (run), and an 8-bit register to hold the instruction (IR) as its inputs and a done bit (done), a 10-bit data holder for the selector (muxout), a bit for the add-sub unit's carry-in (addsub), and an array (regin) and two 1-bit signals (Gin and Ain) for the enables of the registers as its output.

A behavioral architecture was made for the register in order to make four processes to: (1) convert the instruction for RX so the multiplexer can read them, (2) convert the instruction for RY so the multiplexer can read them, (3) switch the time states on every rising edge of the clock (clk) as long as (resetn) is not on, and (4) to describe the actions that take place in each state within each of the four operations; the actions that happen in each time state are shown in Figure 10.7.

5.2 Testing and Simulation

When it came to testing the control unit with a testbench, the test cases were made to check the functionality of each time state of the control unit to make sure the right signals are enabled and selected for the multiplexer. When simulated in ModelSim, the waveforms, as displayed in Figure 10.9, reflected the correctly expected results from the testbench, making it successful. The VHDL-generated state diagram for the control unit, as shown in Figure 10.8, reflects the correctness of the design.

6 Processor

6.1 Design Method

The design of the generic register was made up an entity and a structural architecture for the entity. The entity consists of an incoming data signal (DIN), a clock (CLK), a resetn pin (RESETN), and a run pin (RUN) as its inputs and a done pin (DONE) and a data holder (BUSS) as its outputs. Because it needed to be flexible with the bits it can hold, a generic was used and initially set to sixteen for the bit size for the registers to be used for the processor.

A structural architecture was made for the register in order to include all of the components needed for the processor, signals for every connection that isn't an essential input or output for the processor, and then map everything for functionality. The given diagram for the processor, shown in Figure 10.10, was used as a guide when mapping all of the signals. A constant was used to easily convert the entities from different values for the bit size and not worry about dealing with a lot of numbers and avoid confusion. After finishing the mapping, VHDL-generated RTL diagram displayed in Figure 10.11 of the processor was made, and compared to see how accurate it was to the original diagram provided.

6.2 Testing and Simulation

When it came to testing the processor with a testbench, the test cases were made to check every time state possible to assure it functions correctly. When simulated in ModelSim, all of the waveforms from the pins and internal signals, as displayed in Figure 10.12, reflected the correctly expected results from the testbench, making it successful.

7 Implementation on the DE1-SoC Board

Since there was freedom to implement the processor on the board, the implementation for the board was done similarly to the state machine board implementation from lab 3. Using the board's switches, keys, and ledr lights, two inputs of the processor - DIN and RUN - were mapped to the ledr lights, inputs CLK and RESETN were mapped to keys, while the outputs BUSS and DONE were mapped to the ledr lights to display the outputs to show the functionality of the processor on the board. It's also important

to point that because of the size of the board, the generic values need to reduced from 16-bits to 8-bits using a generic map.

8 Results

When the processor and its lower level components were modified to be tested on the Altera DE1-SoC using guidance from Smith [2], it compiled functioned correctly. The main issue I faced while working on this lab was was getting the control unit to fully function. While I was able to get the majority of the operations to perform correctly, for the mvi operation, I was unable to store the value for RX that came from the DIN pins by the next active clock edge after storing the instruction in the IR register during state T1; instead, the instruction from DIN is what is stored in RX. With that issue disregarded, when the final part was tested on the board with all the other parts and components combined with it, the board displayed the correct states based on the value of (BUSS), meaning it was a relative success.

9 Conclusion

This lab has given me the chance to test and apply my knowledge about architecture, state machines while also providing a little challenge to help me improve my problem-solving skills and knowledge on VHDL coding, architecture setup, and entity elements and techniques learned from class lectures to make the code as efficient as I could possibly make it. By doing this lab, I have a clearer understanding on the intricacies of port mapping and how each of the components made for this lab work separately and together and what they all need in order to do their jobs correctly. If I could go back and change how things were done throughout the lab, I would manage my time quite differently in order to focus more of my time on the mvi operation issue in order to get a 100-percent, fully functioning simple processor.

References

[1] S. Brown and Z. Vranesic. Fundamental of digital logic with vhdl design. Textbook, New York, USA, 2009.

[2] M. Smith. Digital computer design lab manual. Manual, South Carolina, USA, 2019.

10 Appendix

```
BEGIN
   -- code executes for every event on sensitivity list
   clk <= '0';                  --No rising edge yet
   en <= '1';                   --Register enabled
   d <= "0000111010011001";
   wait for 10ns;               --q should be undefined still

   clk <= '1';                  --Rising edge now available
   wait for 10ns;               --q should now have the same value as d

   clk <= '0';                  --No rising edge yet
   en <= '0';                   --Register disabled
   d <= "1111111100001010";
   wait for 10ns;               --q shouldn't update

   clk <= '1';                  --Rising edge now available
   wait for 10ns;               --q still shouldn't update

   clk <= '0';                  --No rising edge yet
   en <= '1';                   --Register enabled again
   wait for 10ns;               --q still shouldn't update

   clk <= '1';                  --Rising edge now available
   wait for 10ns;               --q should now update to the latest value of d

WAIT;
END PROCESS always;
END regN_arch;
```

Figure 10.1: Generic Register Testbench

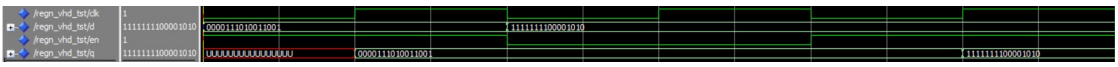

Figure 10.2: Generic Register Simulated Testwaves

```
BEGIN
   -- code executes for every event on sensitivity list
   reg0 <= "0000000011111111";
   reg1 <= "0000000011110000";
   reg2 <= "0000000011001100";
   reg3 <= "0000000010101010";
   reg4 <= "1111111100000000";
   reg5 <= "1111000000000000";
   reg6 <= "1100110000000000";
   reg7 <= "1010101000000000";
   din <= "0101010101010101";
   g <= "1111111111111111";

   sel <= "0000000000";    --not a real selection; default din selected
   wait for 10ns;

   sel <= "0000000001";    --reg0 selected for bout
   wait for 10ns;

   sel <= "0000000010";    --reg1 selected for bout
   wait for 10ns;

   sel <= "0000000100";    --reg2 selected for bout
   wait for 10ns;

   sel <= "0000001000";    --reg3 selected for bout
   wait for 10ns;

   sel <= "0000010000";    --reg4 selected for bout
   wait for 10ns;

   sel <= "0000100000";    --reg5 selected for bout
   wait for 10ns;

   sel <= "0001000000";    --reg6 selected for bout
   wait for 10ns;

   sel <= "0010000000";    --reg7 selected for bout
   wait for 10ns;

   sel <= "0100000000";    --din selected for bout
   wait for 10ns;

   sel <= "1000000000";    --g selected for bout
   wait for 10ns;

   sel <= "1111111111";    --not a real selection; default din selected
   wait for 10ns;

WAIT;
```

Figure 10.3: Multiplexer Testbench

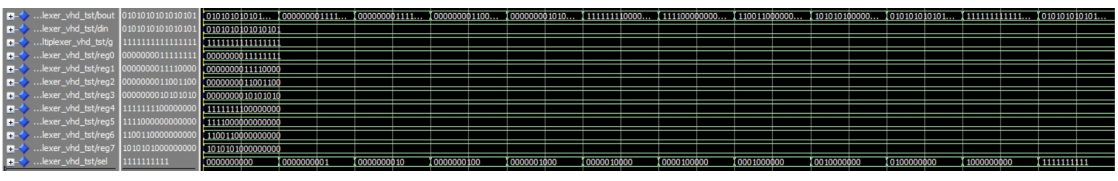

Figure 10.4: Multiplexer Simulated Testwaves

```
BEGIN
    -- code executes for every event on sensitivity list

    A <= "0000000011010001";
    B <= "0000000000010001";
    AorS <= '0';                    --Adding
    wait for 10ns;                  --Answer = 0000000011100010

    AorS <= '1';                    --Subtracting
    wait for 10ns;                  --Answer = 0000000011000000

    A <= "0000000000001101";
    B <= "0000000000000111";
    AorS <= '0';                    --Adding
    wait for 10ns;                  --Answer = 0000000000010100

    AorS <= '1';                    --Subtracting
    wait for 10ns;                  --Answer = 0000000000000110

    A <= "0000000000000111";
    B <= "0000000000001100";
    AorS <= '0';                    --Adding
    wait for 10ns;                  --Answer = 0000000000010011

    AorS <= '1';                    --Subtracting
    wait for 10ns;                  --Answer = 1111111111111011

    A <= "0000000000000000";
    B <= "0000000000000101";
    AorS <= '0';                    --Adding
    wait for 10ns;                  --Answer = 0000000000000101

    AorS <= '1';                    --Subtracting
    wait for 10ns;                  --Answer = 1111111111111011
WAIT;
END PROCESS always;
END AddSub_arch;
```

Figure 10.5: Add-Sub Unit Testbench

Figure 10.6: Add-Sub Unit Simulated Testwaves

	T_1	T_2	T_3
(mv): I_0	RY_{out}, RX_{in}, *Done*		
(mvi): I_1	DIN_{out}, RX_{in}, *Done*		
(add): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(sub): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Figure 10.7: Control Unit State Table

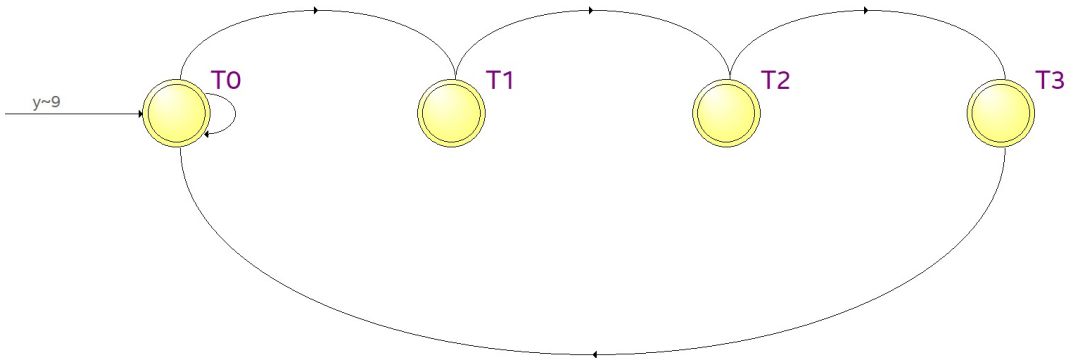

Figure 10.8: Control Unit VHDL-Generated State Diagram

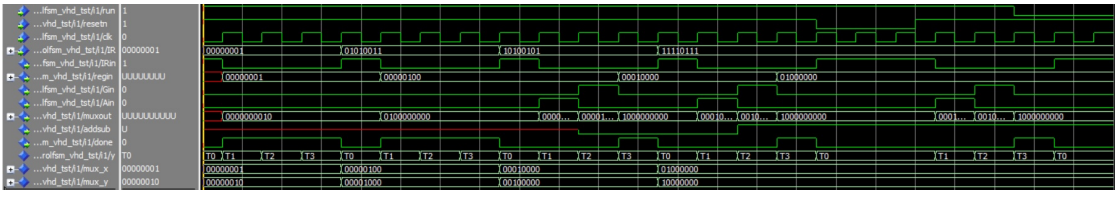

Figure 10.9: Control Unit Simulated Testwaves

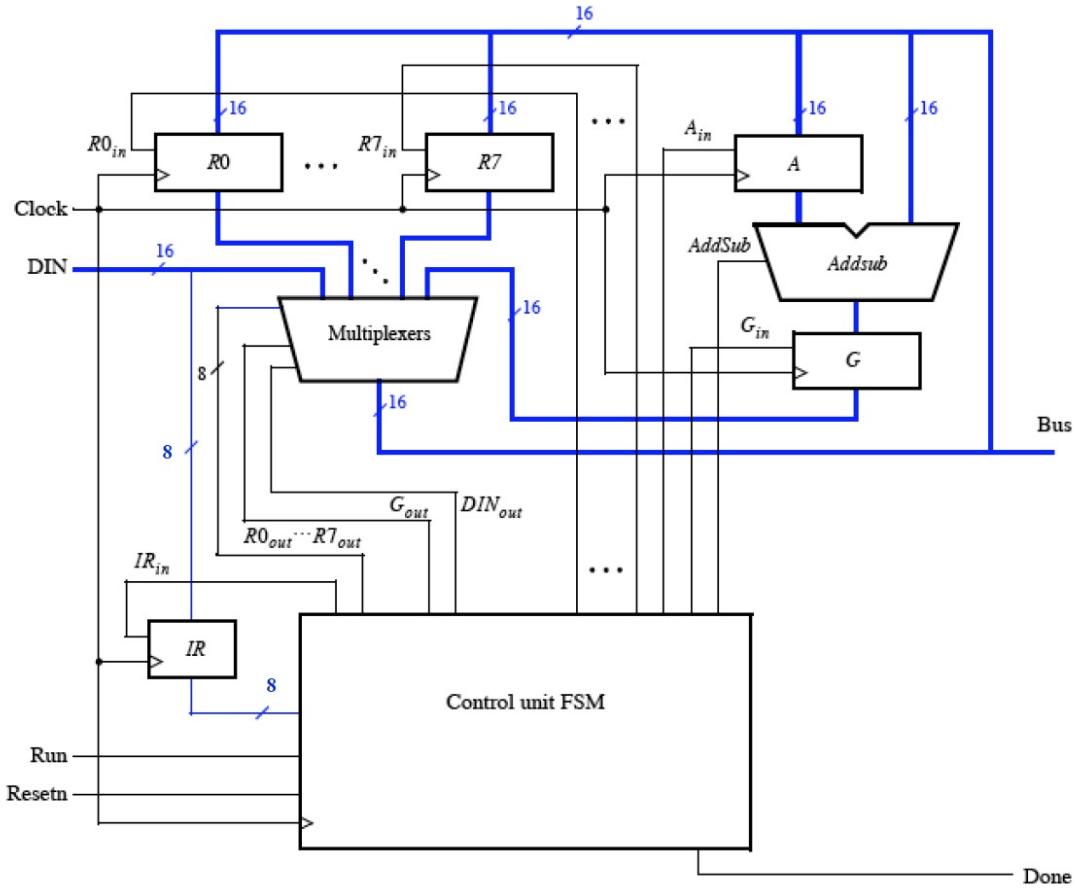

Figure 10.10: Processor Diagram

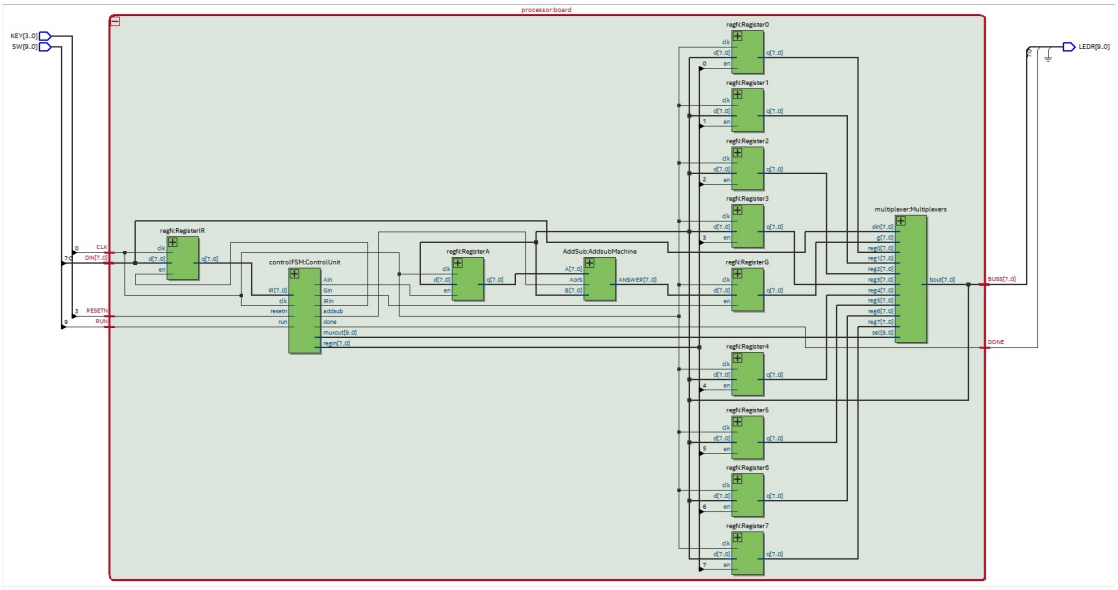

Figure 10.11: Processor VHDL-Generated Diagram

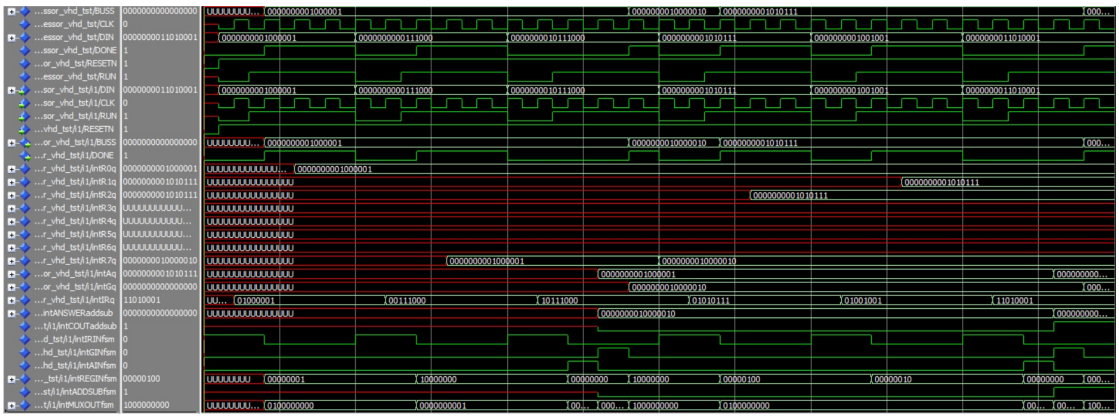

Figure 10.12: Processor Simulated Testwaves