The development of a four instruction processor

LAB 3: SIMPLE PROCESSOR

March 13, 2020

Seth Kantz
Clemson University
Department of Electrical and Computer Engineering
skantz@clemson.edu

Abstract

Assignment 2 consists of 4 different components, all wired together in order to create a simple processor. A register, add/sub module, multiplexer, and control unit FSM. After testing all these modules individually, they can be implemented in the final stage. All of the modules are to be created on the board and wired together, so that the processor has desired functionality. Finally the processor is to port mapped and tested on the DE1-SoC board.

1 Introduction

The goal of this lab was to create a simple processor module, capable of doing 4 operations: mv, mvi, add, and sub. Registers were to be used to hold values at the desired state. The multiplexer will use one hot encoding to select between 10 inputs feeding in from the registers. The add/sub will either add or subtract two values as needed. The finite state machine is the brain of the operation and will operate as a mealy FSM. This will control the other components as needed based on the current state and the inputs. When all wired together, it will be able to store, move, add, and subtract numbers as needed.

2 Design and Testing

2.1 Design of Generic Register

The first step of this lab was to create a register that uses a generic size. A D flip flop was to be created in order to accomplish this, as was done in the past, by using a mixed architecture. The register uses a generic size for the data input / output. The default value for the generic is 16 because 16 bits is the desired size for the main bus. The register will output a new value when the select bit is high and the clock rises, otherwise it will hold its old value.

2.1.1 Testing of a Generic Register

To test the register, the standard generic size of 16 bits was used. First the register was tested with the select bit high, meaning that it should update the values on rising clock edge. The input values were varied as the clock ticked. This can be seen in figure 1. Next the select bit was moved off and then the input values were varied to ensure that it would not update. This worked as desired and can be seen in figure 2. The two tests show that the register is functioning as desired. The output will only update on rising edge clock ticks if the select bit is high. These figures show that the register is operating as desired.

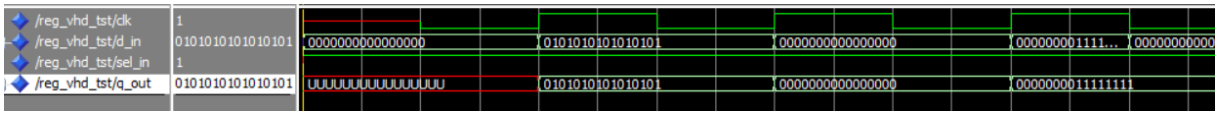

Figure 1: Register Testbench - High Select

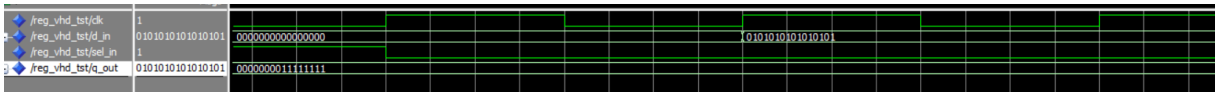

Figure 2: Register Testbench - Low Select

2.2 Design of a Generic Multiplexer

Next, a generic multiplexer was to be designed. The data lines of this mux were to be a generic size, which was chosen to default to 16. The multiplexer is a 1 hot design, meaning that one bit at a time is selected to represent a certain line. Whichever bit is "hot" or high will represent a certain line. Bits 0 - 7 are R0 - R7, bit 8 is the g in from the add/sub module, bit 9 is the normal data line. If more than one bit is hot or if there is an error in the input, the multiplexer will simply default its output to the data in line.

2.2.1 Testing of a Generic Multiplexer

The multiplexer was ran in a testbench to ensure desired outputs. Various select bits were chosen and the mux transitioned between inputs as desired. When more than one bit is hot, it defaults to the d in line. This output is as expected. 3

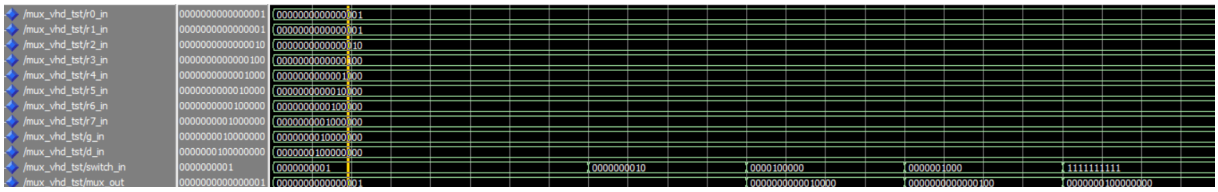

Figure 3: Testbench for the multiplexer

2.3 Design of the AddSub Module

The AddSub module is a basic calculator that either adds or subtracts two data lines of generic size based on the select input. It accomplishes this by making use of the ieee.NUMERIC STD.all library. This allows the addition of signed integers by casting a std logic vector to a signed value and then performing an addition or subtraction. This makes the process a lot easier than bitwise math and allows easier scalability, due to the generic nature of the inputs.

2.3.1 Testing of the AddSub Module

There are many possible combinations for the addsub module. It was tested with addition of two positive numbers, subtraction of two positive numbers, addition overflow, and subtraction to obtain a negative number, as seen in figure 4 . This performs as desired, and handles edge cases as expected. We want a positive number to be able to transition into a negative number by subtraction.

Figure 5 shows more cases being tested such as the addition of a two negative numbers,the addition of a negative and positive number, the subtraction of two negative numbers, and the subtraction of a positive and negative number.

This shows that the calculator performs as expected and handles signs correctly.

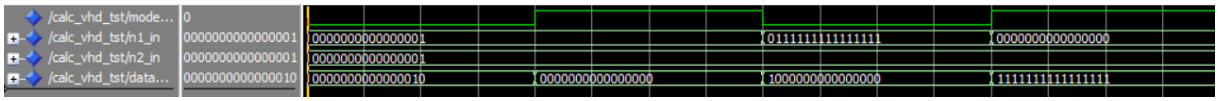

Figure 4: Add-Sub Testbench

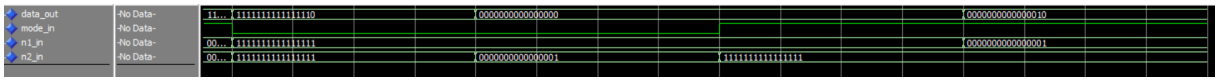

Figure 5: Add-Sub Testbench part 2

2.4 Design of the Control Unit FSM

The figure 6 from the lab manual, shows the desired output of the FSM. It is to be a four state mealy FSM, that will control the operation of the entire processor. The 8-bit input from the input register is the main control and tells the FSM how to operate / determines what to output. The machine should only startup when the run in signal is given. The FSM will step through the steps and move data as needed.

	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 6: FSM Chart

2.4.1 Testing of the Control Unit FSM

Quartus generated the following state machine from my code, as seen in figure 7 As seen

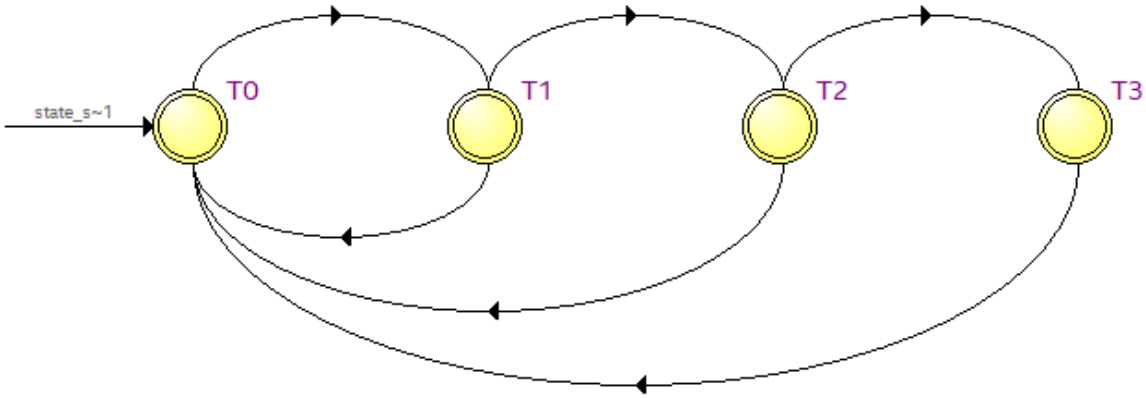

Figure 7: Quartus Generated State Machine

in figure 8, the mealy FSM operates correctly under normal circumstances. When given mv, (reset) then mvi (reset), then add (reset), and finally subtract, it can be seen that the fsm generates the correct outputs, and sets the correct things needed. Figure 9 shows

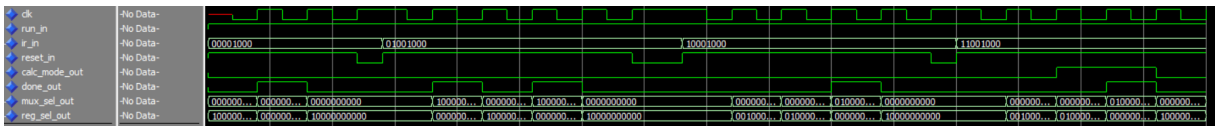

Figure 8: normal FSM testbench

that the fsm will not run without run being high and that it will constantly reset when reset is low.

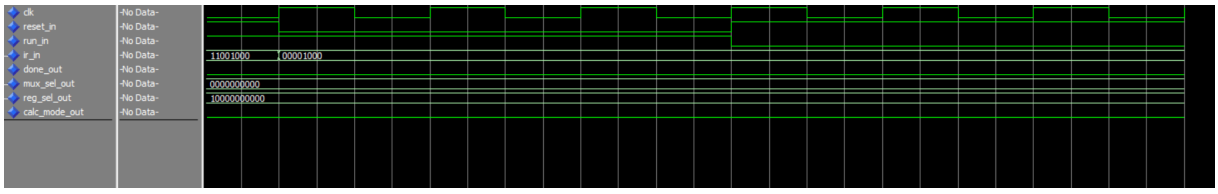

Figure 9: FSM testbench with no run and reset

This shows that when instruction mv is received, it will move data from register Y into register X then output done, when mvi is received, it will move data from data in to register X then output done, when add or sub is received it will move register X into A,

select G, high and then perform the correct math. Finally the add sub will move G into register X and output done

2.5 Design of the Processor

The last logistical step was to create a structural file wiring all of the previously created components together in order to create a processor. 11 Registers were created: 8 data registers wired to the bus R0 - R7, the 8 bit input register IR, the register holding one line before the addsub A, and one register holding the output of the addsub G.

2.5.1 Testing of the Processor

Now that we have the simple processor created, it is time to test it. Quartus generated a wiring diagram, which is visible in figure 10 .

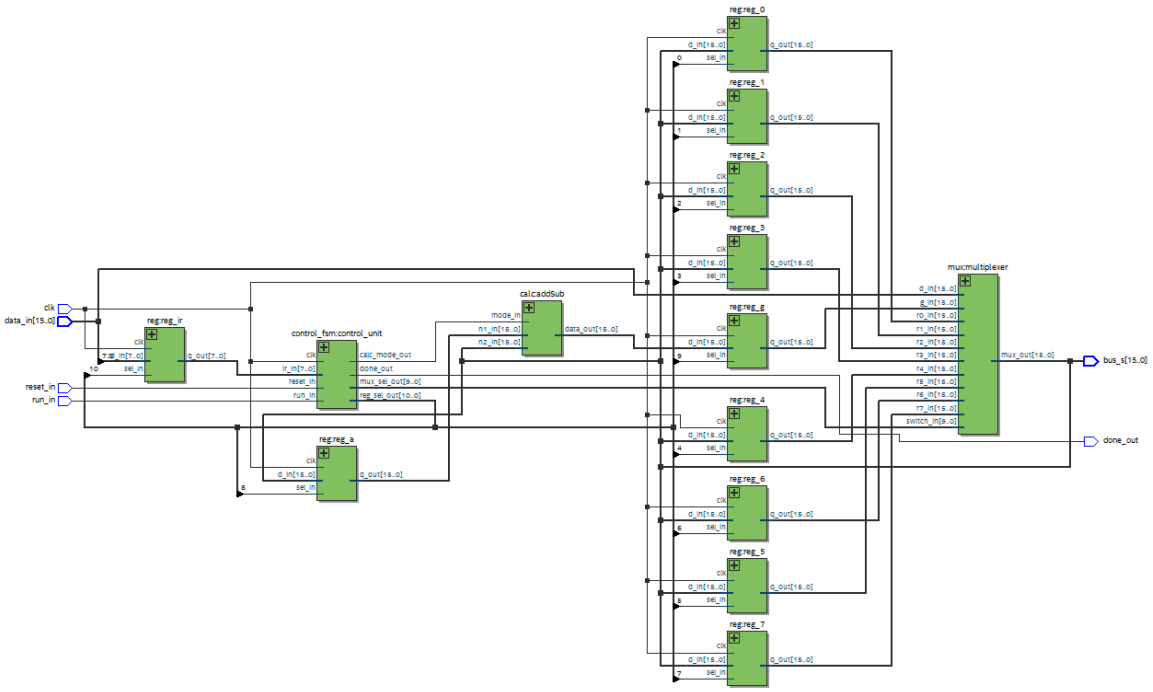

Figure 10: Quartus Generated Wiring Diagram

This was then tested on a testbench. The data line was moved into register 1, then the data from register 1 was moved into register 0, finally those two numbers were added, then those two numbers were subtracted. Lastly the mvi instruction was tested with reset low and then again with run as low. The computations occured as desired under normal operation. When reset or run was low, no actions occurred. These actions can be verified in the testbench in figure 11.

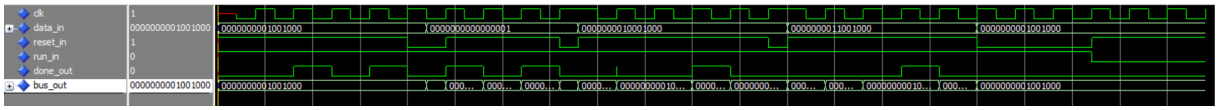

Figure 11: Processor Testbench

2.6 Design of the Board Mapping

The processor is then mapped to the board. KEY0 is mapped to the clk, data in is SW(7 - 0), run is SW(8), and reset is SW(9). The done output is displayed on LEDR(9) and the bus out is displayed from LEDR(7 to 0).

Generics were used to make the processor work with this system. In this case we designed the system for 16 bit, and adapted it to 8bit for the DE1-SoC board to accommodate the number of I/O that exists.

2.6.1 Testing of the Board Mapping

The code is now able to be uploaded to the board and tested on hardware.

3 Conclusion

In conclusion, it is possible to create very intricate designs by using smaller components. The use of a mealy FSM allows for the use of only a few states yet still contains many possible outputs, because each state's output is dependent on its input. The use of registers allows for one input line to set and then output different data to a register. It was discovered that generics are very important when designing systems, because it allows easy scalability to different platforms. The important of good signal names was also discovered during this lab, because it can become confusing fast when wiring up a structural design.

4 Bibliography

[1] Dally, William J. Print. Cambridge University. 2016.