ECE 3270 : LAB 3

SIMPLE PROCESSOR

April 12, 2022

Aaron Bruner
Clemson University
Department of Electrical and Computer Engineering
ajbrune@clemson.edu

ABSTRACT

For this lab we were tasked with developing a circuit that represents a simple processor. The overall idea is to develop an understanding of how the components work inside of a processor and how to implement it in VHDL. Considering this could become a massive implementation with many modules we are going to limit it to a 4-instruction processor. To ensure that our design is modular we will use VHDL Generics for our components so that they can be easily changed and transported into either different parts of the processor or different labs. However, the top level entity should be a structural design meaning our overall system is comprised of smaller sub-systems that can be reused. The processor will have support for 4-instructions; mvi, mv, add, and sub. mvi will move information from DIN to RX. mv will move data from RY to RX. add will add RX and RY and store the result in RX. Lastly, sub will subtract RY from RX and store the result in RX.

1 Introduction

For this lab we are designing a simple 4-instruction processor. Posted in figure 1 we see the overview for the design that we are wanting to achieve. The blue line represents data-paths and is shown as a 16-bit input. As mentioned in the abstract, this data-width is going to be generic so it is easy to change. That way we can expand or compact our system as needed. The image contains 8 registers labeled R0 through R7. There is an 8-bit instruction register (IR) which is connected to our data input (DIN). We also will have a clock signal to step through each action in our system, this will be controlled by a button. The system has a total of four inputs being DIN, Clock, Run and Resetn. In addition, there are only two output signals being Bus and Done. Considering that the term "Bus" is reserved we will be using Bus0 instead. The rest of the terms are free to use and will be used as such.

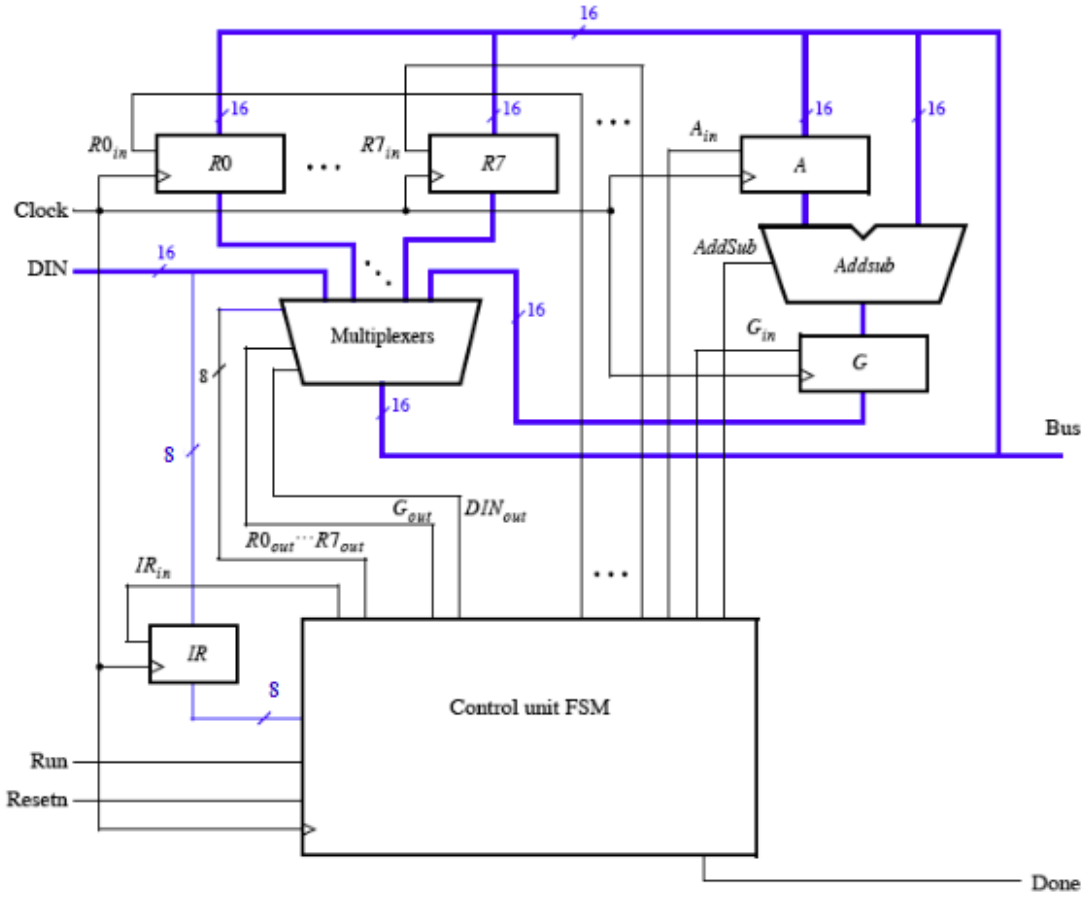

Figure 1: Layout of processor to be designed. Sourced from lab 3 manual [2]

2 Design

To design our simple processor we need to consider all of the components necessary to achieve this goal. Referencing the figure in the lab manual we can view all components necessary to design this processor [2]. Each subsection will discuss the details of the component and how it functions overall. Then, the processor will combine all of the components to create our simple 4-instruction processor.

2.1 Multiplexer

The first and simplest component for us to design will be the multiplexer. As we know from lab 0 a multiplexer takes in many data lines and outputs only one of the inputs dependent on a select line. From referencing figure 1 we can see that twe have a total of 10 inputs to the multiplexer. The inputs are as follows; register 0 through 7 (represented by R0-R7), data input (DIN), multiplexer G's output (G), select (S). All we need to do is provide a select option and the resulting output will be one of our inputs. The method in which this was designed was using the least significant bit (LSB) to represent R0 and the most significant bit (MSB) to represent DIN. The bit to the right of the MSB will be G. Thus, if we wanted to select R6 we would input "0001000000" into the select line. Similarly, to represent DIN we would input "1000000000" which would output DIN. Another thing to note is by design we will always output DIN for any input that contains more than one HIGH bit or no HIGH bits.

2.2 Adder

This section and AddSub are rather short for a few different reasons. The first reason being that the code written for these sections comes directly from the textbook. Practically every line of both sections comes straight from the text and is explained in detail through the text. Thus, for a much more in-depth explanation as to how the Adder and AddSub components work please reference the text [1].

From page 229, figure 10.7, we can see the implementation of an Adder written in VHDL. We use the same example for our system without changing anything. We declare the term 'size' to be the implemented size of our adder. For this system we are using a 16-bit implementation. Below is the in-depth explanation sources from the textbook; it does a phenomenal job at explaining the adder we are designing.

> Our n-bit adder accepts two n-bit inputs and produces an (n + 1)-bit output. This ensures that we have enough bits to represent the largest possible sum. For example with a three-bit adder, adding binary 111 to 111 gives a four-bit result, 1110. In many applications, however, we need an n-bit output. For example, we may want to use the output as a later input. In these cases we need to discard the carry out and retain just the n-bit sum. Restricting ourselves to an n-bit output raises the specter of *overflow* - a condition that occurs when we compute an output that is too large to be represented as n bits. Source [1]

For our simple processor we are not going to handle overflow. Instead, we will leave the conditions in our code and deal with that assuming this lab pans out into something larger.

2.3 AddSub

AddSub is another important feature of our processor. This component is responsible for all subtraction and addition. Thus, this component requires the existence of the prior component, Adder. As previously mentioned, we are going to reference page 235, figure 10.13, for the code for AddSub since the book was so kind to provide that for us [1]. Below is another phenomenal explanation of the AddSub component that we are implementing.

> Now that we can add negative numbers, we can build a circuit to subtract. A subtractor accepts two 2's complement numbers, a and b, as input and outputs q = a - b. A circuit to both add and subtract is shown in Figure 10.10 [Figure 12]. In add mode, the *sub* input is low, so the XORs pass the b input unchanged and the adder generates a + b. When the *sub* input is high, the XORs complement the b input and the carry into the adder is high, so the adder generates
>
> $$a + \bar{b} + 1 = a - b \tag{1}$$

Applying what we have learned from the textbook we are able to setup the same AddSub system using our Adders from the previous section. The only difference in our code from the textbook is that we removed the XOR statement from the PORT MAP and put it in its own signal. This resolves errors that Quartus Prime generates.

2.4 Rising Edge D Flip-Flop

The D Flip-Flop that we needed to implement was very simple and almost identical to the one developed for lab 0. We are still detecting on the rising edge of the clock signal but are now taking into account the select signal that is passed in from the processor code. The reason we have this D Flip-Flop is to help keep track of data in our processor. We are able to store data and retrieve data by using the select line.

2.5 Finite State Machine

The next component needing to be implemented is the Finite-State Machine. This is where we begin to implement the 4 simple instructions for our simple processor. The first operation that we need to implement is the mv operation. This operation will take two inputs, register X (RX) and register Y (RY). The mv operation will move the data from RY to RX. The next operation is mvi which also takes in two inputs. The first input represents the register we are wanting to select, the second argument is required but has no meaning. Once this operation is loaded into the IR the value to be stored in RX must be on the DIN pins by the next active clock edge. The next operation needed is add

which takes in two arguments RX and RY. For this operation we will add RX to RY and store the result in RX. The last operation is sub which requires two inputs RX and RY. The result of this operation is subtraction of RY from RX and the result is stored in RX.

Our instructions are encoded as IIXXXYYY, where II represents the instruction, XXX represents the X register, and YYY represents the Y register. The table in figure 2 demonstrates the setup for our Mealy state machine to control the internal signals. This table shows the signals asserted in successive time steps for each instruction. After drawing out the FSM we get the result in Figure 3. We cross reference this with what Quartus generates in figure 11 to view that we have a very similar system.

	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 2: Reference table from lab manual

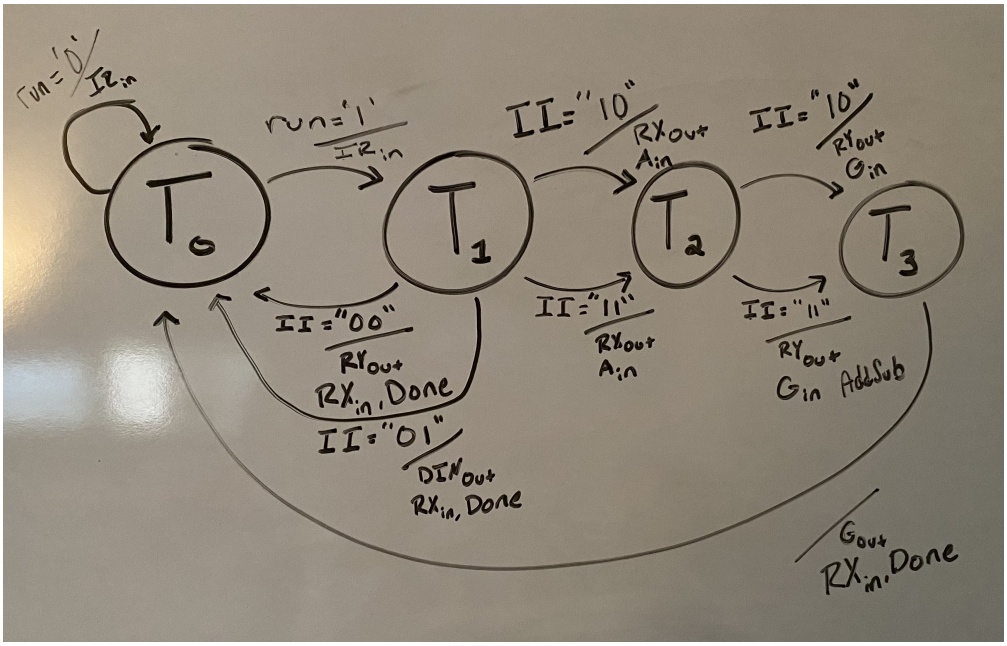

Figure 3: Drawing of the FSM

2.6 Processor

The processor was basically a compilation of everything up to this point. Referencing figure 1 it is easy to setup our processor. All we have to do is follow each of the signal paths and connect everything as shown. Once we do this we can begin to test everything using our test bench.

The first thing to test is the mv operation. When we perform the mv operation we want to input two register options RX and RY. mv will move what is in RY into RX. Shown in figure 4 we can see R4 gets the value from R2 put into it's register. A side note is that we disabled run and pulsed the clock to ensure we could read the image better.

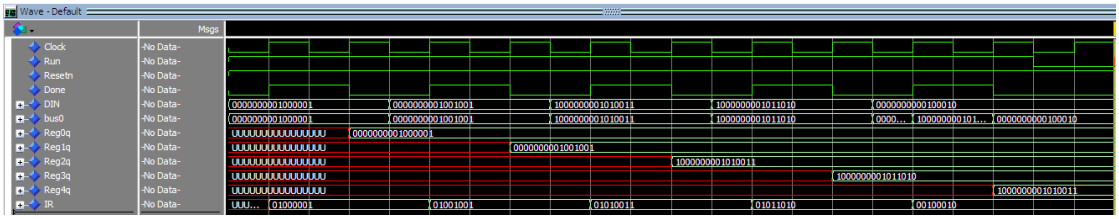

Figure 4: Simulation of VHDL code to ensure functionality of mv operation

The next thing to test is the mvi operation. When performing the mvi operation we will be moving information from DIN to RX. The RY still requires an input value but it does not matter what that value is. For this example we will set the value of R0, R1, R2 and R3. For each register, we will set it's value to the command we used to set the value. Also, putting random values for RY so that when doing operations later we don't have very similar values. Figure 5 demonstrates the registers taking on the values mentioned above.

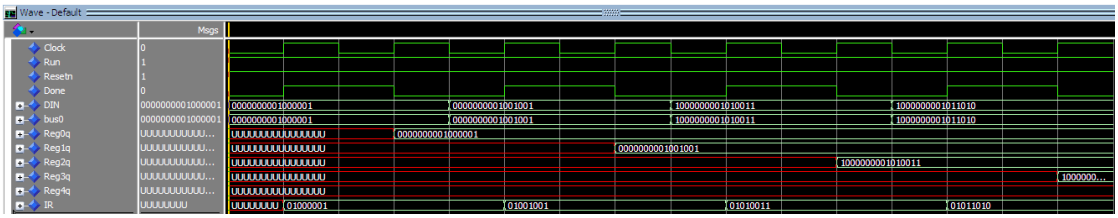

Figure 5: Simulation of VHDL code to ensure functionality of mvi operation

The next operation to test is add. For the add option we are going to take in two registers RX and RY. The addition will add RX and RY then store the result in RX. For our example shown in figure 6 we will add R0 to R3 and view the result put in R0.

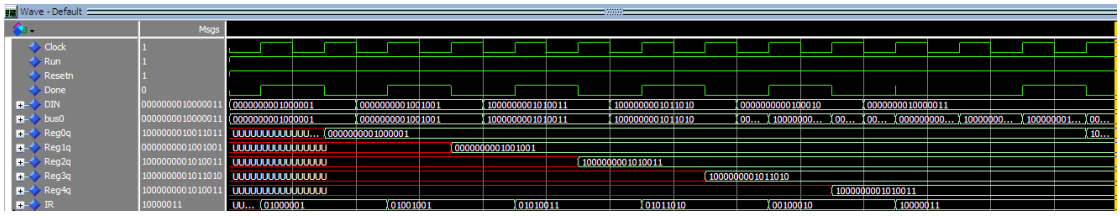

Figure 6: Simulation of VHDL code to ensure functionality of add operation

To further confirm our results we use a simple programmer calculator to do the addition for us and provide us with the result. The value shown in figure 7 adds our R0 value shown in the previous figure to R3 also shown in the previous figure. Then, we observe R0 change to the result of our calculator.

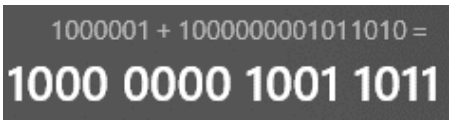

Figure 7: Demonstration using programmer calculator to add binary confirming results

The next operation to check is sub. This is one of the more complex operations but not too difficult for us. Basically we are going to input RX and RY like the prior operations. The processor will subtract RY from RX and then store the result in RX. We view this in figure 8. For our example we are subtracting R4 from R1 and storing the result in R4. Once again, we include a pause to view all of the results by disabling run and pulsing the clock.

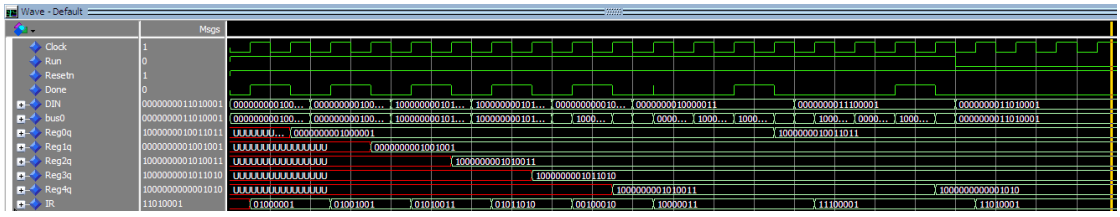

Figure 8: Simulation of VHDL code to ensure functionality of sub operation

Similar to the add operation we will confirm our results with our programmer calculator. The value on the left being R4 and the right being R1, and finally the result being stored in R4.

Figure 9: Demonstration using programmer calculator to subtract binary confirming results

The only thing left to check is our Run option and we can simply do so by disabling Run and putting an operation on DIN. As shown in figure 10 we see that nothing changes when we disable the Run value which is expected.

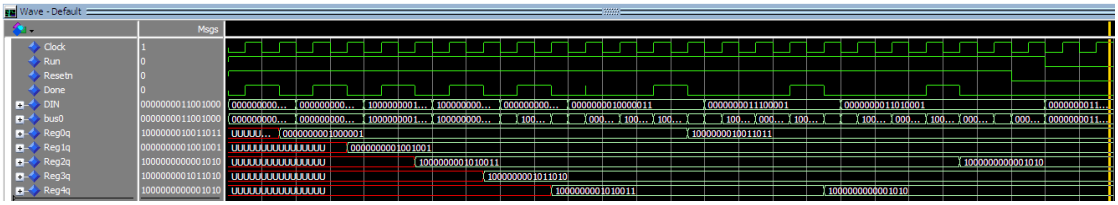

Figure 10: Simulation of VHDL code to ensure functionality of RUN

3 Results

To confirm that our program logic was functioning as expected we are able to develop test benches which can simulate specific inputs and show us the output in wave forms. This is exceptionally beneficial for us because we do not have to try and write out every single state and guess what our DE10-Lite board is doing behind the scenes. Once we created our test bench to try mvi, mv, add and sub operations we tracked the wave forms and each, important, signal to check functionality. As a result we determined functionality of each operation.

The next task is to program the DE10-Lite board and ensure functionality that way. Doing so limits us to only 8-bit input considering we must use the two left most switches for the Run and Resetn inputs. As a result, we designed our board code to only use an 8-bit DIN size. After programming the board we can check to ensure that simple operations listed above work as expected. Doing so proved functionality in both the simulation and on the actual board.

4 Conclusion

To conclude, this lab taught us many things about Quartus, ModelSim and our DE10-Lite. For starters, Quartus Prime has many functionalities that are extremely helpful for us to develop our VHDL code. In addition, being able to utilize ModelSim to ensure that everything is working one clock pulse at a time is extremely helpful. The images demonstrating our processor functionality is so helpful to check the state of every component in the system. Once we have ensured that everything is functional we transfer all code over to the Linux machine in the lab. The biggies issue with doing this task is that the Quartus Prime version in the lab is not only a Linux build but also a much older version than ours. Thus, when we transfer over all of the code we must ensure that no steps are missed and that no files are transferred over, only the text inside of them. Once this has been completed we can compile our code and program the board. After doing so we can instruct the processor to perform tasks for us such as loading data into registers, moving data between registers, adding registers together and subtracting registers. One last note, we are using the button for our clock mainly because it has built in debouncing. Lastly, generating the Quartus state diagram image which helped verify that our VHDL code was functioning as expected. This is a big added benefit to only using ModelSim to verify.

References

[1] W. Dally. Digital design using vhdl a systems approach. PDF, University Printing House, Cambridge CB2 8BS, United Kingdom, 2016.

[2] B. Green. Lab 3: Simple processor. PDF, Clemson, 2022.

5 Appendix

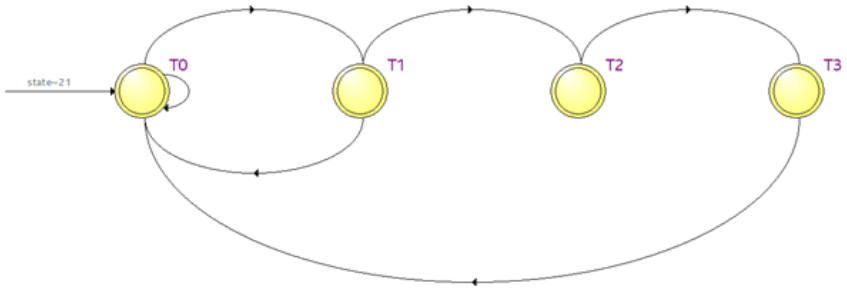

Figure 11: Quartus generated state machine diagram

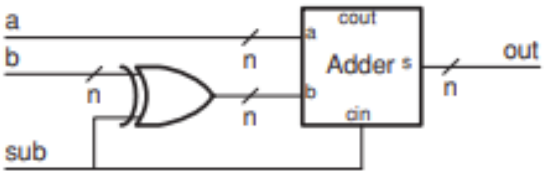

Figure 12: Reference figure from textbook