ECE 3270 : LAB 4

BIT-PAIR RECODED MULTIPLIER

May 1, 2022

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

ABSTRACT

For this lab we were tasked with implementing a Bit-pair Recoded fixed point multiplier in VHDL. From the material learned in class this is a slightly more complicated version of a Booth Recode algorithm. Instead of identifying every two bits and the operation needed to perform add shift we instead review every three bits and determine the desired operation from a predetermined table. This table can be found in the textbook on page 277; table 12.2 [1]. The multiplier will have a total of five inputs; MULTIPLIER, MULTIPLICAND, clk, reset and start. In addition, the outputs will be finalResult, busy and done. The product of the multiplicand and multiplier will be put in finalResult when the done signal is high. To achieve this desired circuit we will have to develop several smaller circuits to perform desired tasks. We will need four registers (A through D), two multiplexers, an Adder and a Mealy finite state machine.

1 Introduction

For this lab we are designing a fixed point Bit-pair multiplier in VHDL. This circuit will take in two binary values and multiply them together and provide the result as a binary value. As shown in figure 1 we can see the necessary parts to achieve our results. The first requirement is four different registers; A, B, C and D. Each of these registers have different purposes which are described below in their sub section. In addition to these registers we are going to need two different multiplexers. The first multiplexer is shown below register A; this MUX takes in 5 values and outputs only one. The select line comes from the 3 LSB of register B. The next MUX we need is shown above register C. This MUX will either output a STD_LOGIC_VECTOR of all zeros or the value from the adder. Speaking of adders, we will need an Adder component that (surprise!) adds two STD_LOGIC_VECTORs together sending the result to register C. The next component is the finite state machine; the FSM is used to control our next operations. We only have a few operations to perform. The states are wait, load, shift, add and done. The last and most important component is our controller. This piece is the brains to the operation. It connects all of our components and brings functionality to our circuit.

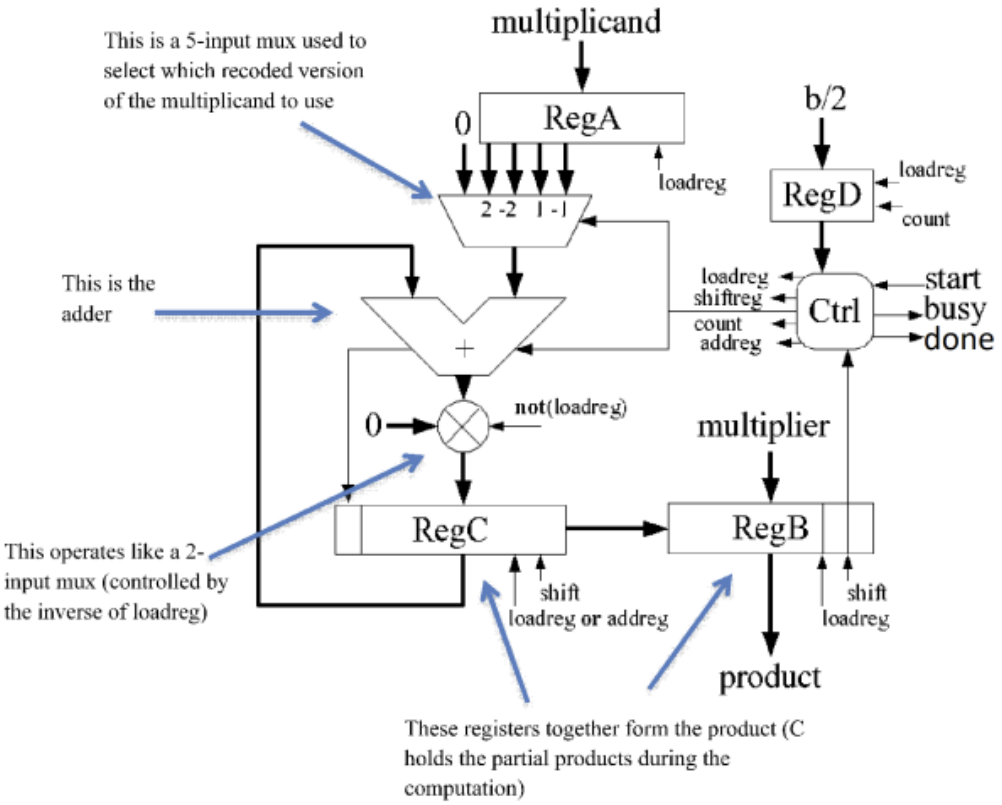

Figure 1: Layout of circuit to be designed. Sourced from lab 4 manual [2]

2 Design

For our circuit to function as we need it to we must develop eight components. There are many ways to go about doing the development process; which component you develop first does not have a massive impact on the final result. This Bit-pair multiplier requires four different registers. The first register is REGA, it's job is to house the multiplicand and it's alternate forms. In particular, the operations we are needing are 1x, 2x, -1x and -2x. These are computed on loadreg high and left for future reference. Register B is going to contain the 8 LSB of our final result. To start off we are going to load this register with our multiplier so that we can keep right shifting by two getting our recoder operations from the LSB. In addition to these recoder operations we will be shifting in two bits from register C and putting them in the upper MSB position. Register C is similar to register B; it's job is to hold the partial product as we do our multiplication. We will keep feeding REGC's value back to the adder so we can keep adding it to whatever operation we need. Register D is one of the simplest things about this lab. It's purpose is to track how many operations we have done. Once we have done four add shift operations we are done and register D outputs a 1 to the done signal. The other necessary components are two multiplexers; the first is a 5-to-1 MUX which takes in four 9-bit vectors and the select bit then outputs the desired operation. The second multiplexer is a simple 2-to-1 MUX which either outputs the result from the adder or all zeros. This is used for loadreg and is not very detailed. One other component is the adder; it takes in two 9-bit vectors and adds them together then spits out the 9-bit result. We don't care about overflow or carry out so it's not implemented. The last component is the finite state machine. It is a Mealy FSM which depends on the current state and also the input provided. Below will discuss how the FSM is designed and what states are used.

2.1 Register A

Register A is a pretty simple register. It's main purpose is to load in the multiplicand and to find the 1x, 2x, -1x and -2x values and put them on the output lines. Doing these things is pretty simple considering the not() operation. The only thing we need to note about this register is signed bit extensions. We are provided with a b-1 size vector and need to output a b-bit vector. So, we need to make the b-th bit the correct sign extension since these values are signed. Thus, while we set the output we make the MSB equal to the value on the right of the MSB. Figure 2 shows how when we input the value 86 in binary we get 86 for 1x, 172 for 2x, -86 for -1x and -172 for -2x.

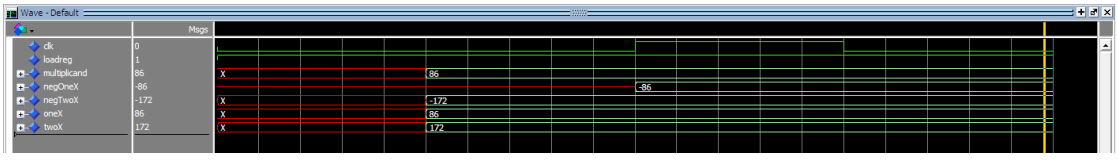

Figure 2: ModelSim demonstration of how register A functions.

2.2 Register B

Register B has a much different purpose compared to register A. Register B is designed to contain the 8 LSB of our final product. Register B stores the multiplier in it's register when loadreg is high. To make our lives easier and reduce the amount of input lines we always will put the value of register B in the final product output. Another functionality of register B is it's shiftreg operation. When shiftreg is high it takes the input from register C and makes that the two MSB of register B. The binary value in register B is right shifted by two. In addition to everything else mentioned, register B outputs the next recoder operation using the 3 LSB of register B. This is passed on to the 5-to-1 MUX to select the desired operation. We can see these values shown in figure 3.

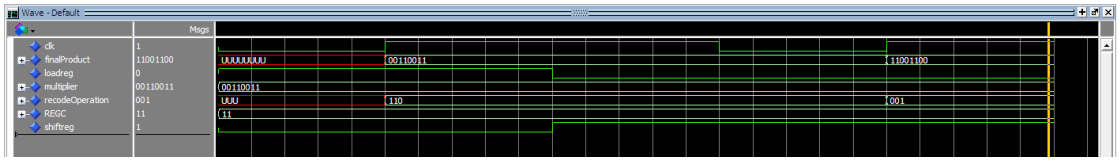

Figure 3: ModelSim demonstration of how register B functions.

2.3 Register C

Register C has a very similar task as register B does. Instead of storing the lower 8 bits of our final result, register C stores the upper 8 bits of the final result. We know that when doing multiplication of b-bit binary numbers our result will be 2*b-bit long. So, storing 8 bits in B and 8 bits in C makes up for our 16-bit result. Register C similarly has the shiftreg option to right shift by two; these two bits shifted out go into the 2 MSB of register B. On the left hand side of register c we need to sign extend the value with either 1s or 0s depending on if it's negative or positive. Figure 4 shows a simple example of various inputs to REGCd (register C data) and how shiftreg and loadreg impact that register.

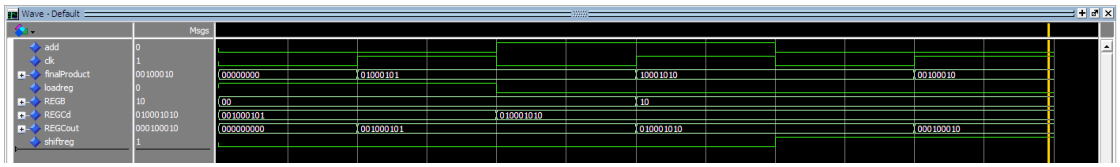

Figure 4: ModelSim demonstration of how register C functions.

2.4 Register D

Register D is a very simple register. It's main purpose is to keep track of add shift operations. Once we have done four add shift operations register D will output a 1 to

the done signal. If we have not done four operations yet then we keep decrementing our counter from b/2 down to 1. If we get the loadreg signal then we reset our counter back to b/2. Figure 5 demonstrates a simple example where we set count high and then see done go high after four clock pulses.

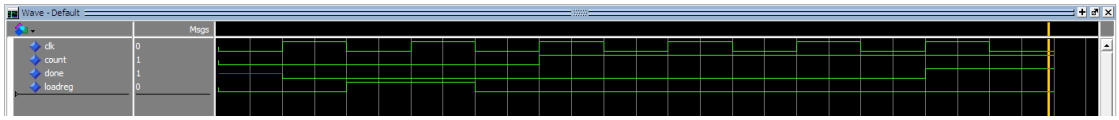

Figure 5: ModelSim demonstration of how register D functions.

2.5 Adder

Similar to register D the adder is pretty simple. All it does is inputs two b+1 vectors, adds them together and puts the result in the output signal. Figure 6 shows a simple example of adding 103 and 17 getting 120 and then -128 and 128 to get 0.

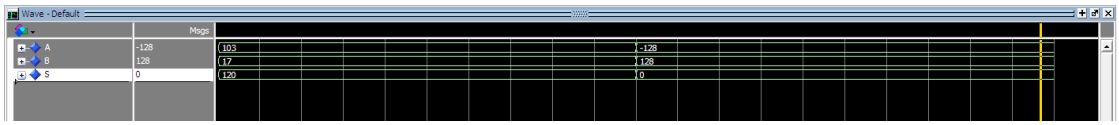

Figure 6: ModelSim demonstration of how the adder functions.

2.6 Multiplexers

The multiplexers section contains both the 5-to-1 multiplexer and the 2-to-1 multiplexer. The 5-to-1 multiplexer has five inputs being 1x, 2x, -1x, -2x and S which is our select line. The select line is a 3-bit value which correlates to table 12.2, our recoder value [1]. The 2-to-1 multiplexer has two inputs being the output from the adder and loadreg signal. If loadreg is high then we output all 0's and if it's 0 then we output the result from our adder. Referencing figures 7 and 8 we can see exactly how these multiplexers work in action. Simple select lines produces the desired result.

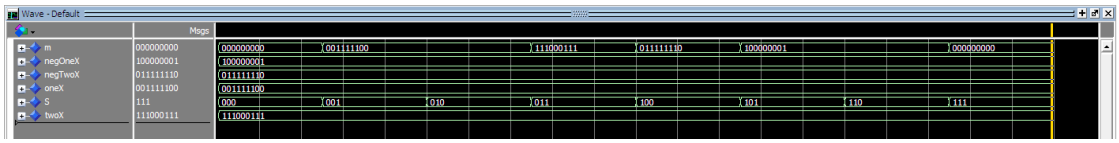

Figure 7: ModelSim demonstration of how the large 5-to-1 multiplexer functions.

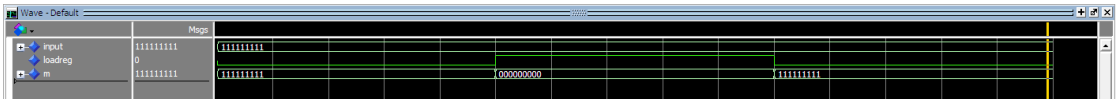

Figure 8: ModelSim demonstration of how the small 2-to-1 multiplexer functions.

2.7 Mealy Finite State Machine

The finite state machine is not as complex as it looks. There are only a hand full of sates that we have to go between. The first state that we begin in is the waiting state. Unless the start sign is read in we will continue to sit here until then. The next state is load which tells our registers to load with the necessary data. For our D register that means our counter is set to b/2, C register is filled with zeros, and the B register is filled with the multiplier. Then the next cycle of states are add and shift in that order. We will continue to add and shift until regD gives us a 1 which is our done signal. This means that we have done four add shift operations and are ready to display our result. Then we transition into the done state and set the done LED high and display our 2*b-bit result. Figure 9 shows a simulation of how the FSM functions and changes whenever we change our input during the states. Figure 10 shows the quartus generated finite state machine.

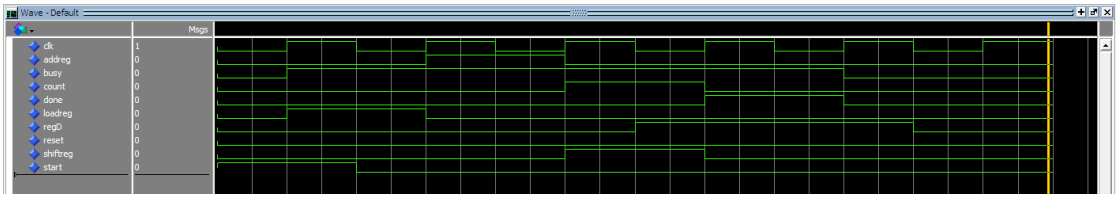

Figure 9: Quartus generated state machine diagram

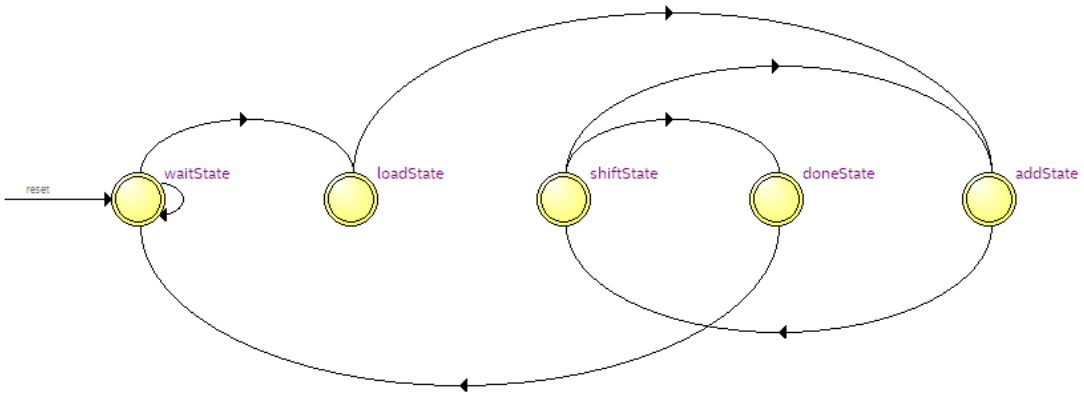

Figure 10: Quartus generated state machine diagram

2.8 Controller

We reference figure 12 and figure 1 to compare our controller design. Using figure 1 we can generate our controller and connect all of our components together. The first thing that we do is list our the inputs and outputs for our system. Using figure 1 we can see that the inputs to our system are clk, start, reset, multiplicand and multiplier. For the outputs we have busy, done and finalResult. After getting these inputs and outputs declared we move down to design our architecture; first we list out all of our components. Doing this is basically coping the ENTITY from each component's file and replacing ENTITY with COMPONENT. Once that is done we list our all of our components and start the PORT MAPs. I personally list out all of the inputs and outputs using their normal component values and then replace them with signals if need be. For this lab I appended a capital S to the beginning of the signal name if it was a signal. For those that have the same names I extended the last letter by one. So, for our small multiplexer output m we renamed it to Smm because Sm is being used by the adder input line. Once we list out all of the signals and connect all of the pieces we generate the Quartus circuit and compare it to what we have been desiring. Looking at figure 12 we can see that everything appears to be there and it looks much more appealing than the lab manual image. Figure 11 demonstrates how the controller works using ModelSim.

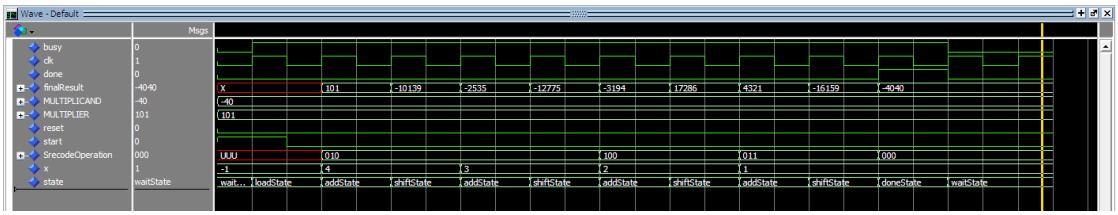

Figure 11: ModelSim demonstrating how the entire circuit works adding two binary values.

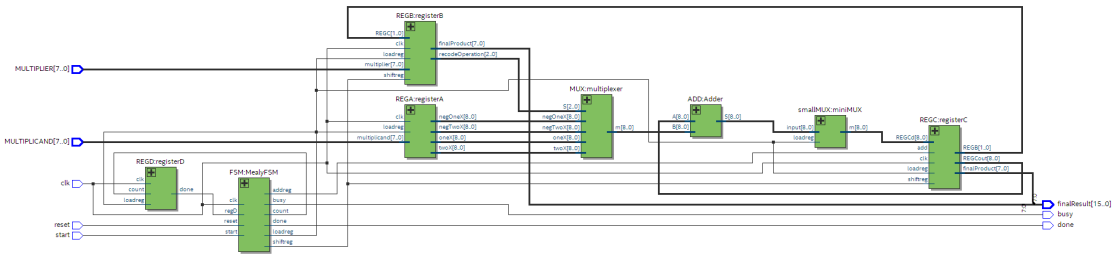

Figure 12: Quartus generated circuit

3 Results

The ensure that everything in our circuit is functioning as expected we generate test benches. For this lab we generated test benches and provide generic data on the inputs and do calculations by hand to ensure the results match. In particular, for our controller test bench we work out the problem by hand and figure out what exactly the values should be in each register to track down errors much faster. Doing so gives a much more enjoyable programming experience. The test benches have confirmed that each component is doing it's job and combining them provides us with good results.

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. One very useful tool that Quartus provides is the RTL viewer which shows us the circuit layout and how our components are tied together. This makes confirming our hand drawn designs with the simulated designs much easier. In addition, being able to utilize ModelSim to ensure that everything is working one clock pulse at a time is extremely helpful. The images demonstrate our multipliers functionality which is very 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. However, for this lab we are not programming on the boards due to the fact that the license on the lab machines has expired. Thus, we are heavily utilizing ModelSim for demos.

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 4: Bit-pair recoded multiplier. PDF, Clemson, 2022.