ECE3270 Digital Computer Design: Lab 5

BIT-PAIR RECODED MULTIPLIER

April 13, 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 bit-pair recoded fixed point multiplier 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 multiplier (registers, adder, 5-to-1 and 2-to-1 multiplexers, and control unit finite state machine) and (2) implementing the multiplier 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 Slide 3 of Brown [3], multiplying by way of bit-pair guarantees an n-bit multiplier will generate no more than n/2 partial products, and therefore, speeds up the multiplying process by almost 2x over the standard add-shift method. Slide 22 also has a figure showing the datapath circuit for a bit-pair multiplier; the figure, shown in Figure 10.1, has four registers, a decoder, a 5-to-1 MUX, a counter, and a (n+1) bit adder. While this isn't the exact design specified for this lab, it is functionally close to the lab's design, showing how a bit-pair recoded multiplier operates hardware-wise. 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 four registers for the bit-pair multiplier are flip-flops with an input, outputs, some sort of enabler, and a clock. For the bit-pair recoded multiplier, the registers are used to store, shift, and transfer data within the multiplier with other registers. The generic design for the registers reflects this description; because they all needed to be flexible with the bits they can hold, a generic was used and initially set to eight for the bit size for the registers to be used for the multiplier.

2.1 Register A

2.1.1 Design Method

The design of the generic register A was made up an entity and a behavioral architecture for the entity. The entity consists of the multiplicand, a clock, and a load enable as its inputs and four data holders as its outputs.

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 load in new data for its data holders. The process is dependent on the clock and conditioned to the load enable pin; it states that when a rising edge is detected on the clock, if the load enable pin is set to HIGH, then the multiplicand's data will be stored in a temporary holder signal, do some calculations, and then output the four possible versions of the multiplicand that will be needed for the bit-pair recoding - positive one, negative one, positive two, and negative two.

2.1.2 Testing and Simulation

When it came to testing the generic register A with a testbench, the test cases were made to verify that the four possible versions of the multiplicand were calculated correctly and the multiplicand value was loaded at the right time when it was enabled. When simulated in ModelSim, the waveforms, as displayed in Figure 10.3, reflected the correctly expected results from the testbench, making it successful.

2.2 Register B

2.2.1 Design Method

The design of the generic register B was made up an entity and a behavioral architecture for the entity. The entity consists of the multiplier, a clock, load and shift enables, and a shift value holder as its inputs and multiplier selector and partial product value holders as its outputs.

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 load and shift in new data for its data holders. The process is dependent on the clock and conditioned to the load and shift enable pins; it states that when a rising edge is detected on the clock, if the load enable pin is set to HIGH, then the multiplier's data will be stored in a temporary holder signal, and if instead the shift pin is enabled, the temporary holder signal will shift all of its values down two places to make room at the front for the shift bits coming from register C. Outside of the process, the multiplier selector value will read the lower three bits of the temporary value holder while the partial product value holder reads all of the data from the temporary value holder except the lowest bit since it acts as the dummy bit for recoding purposes.

2.2.2 Testing and Simulation

When it came to testing the generic register B with a testbench, the test cases were made to verify that the multiplier value was loaded and the values shifted by two correctly at the right time when they were appropriately enabled. When simulated in ModelSim, the waveforms, as displayed in Figure 10.4, reflected the correctly expected results from the testbench, making it successful.

2.3 Register C

2.3.1 Design Method

The design of the generic register C was made up an entity and a behavioral architecture for the entity. The entity consists of a clock, a carry out bit, an incoming value from the 2-to-1 MUX, and a load, shift, and add enable as its inputs and a sum value to go into the adder, a shift value to feed into register B, and partial product value holder as its outputs.

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 load in and shift out new data for its data holders. The process is dependent on the clock and conditioned to the load, shift, and add enable pins; it states that when a rising edge is detected on the clock, if the load or add enable pin is set to HIGH, then the register will store data in a temporary holder signal from the 2-to-1 MUX, and if the shift enable pin is set to HIGH, then the temporary holder signal will shift all of its values down two places, shift the the lower two bits into register B, and the sign extend in the front. Outside of the process, the

sum input will read in the temporary value holder's data, the shift will read the lowest two bits of the temporary holder, and the partial product value holder reads all of the data from the temporary value holder except the highest bit.

2.3.2 Testing and Simulation

When it came to testing the generic register C with a testbench, the test cases were made to verify that the value from the 2-to-1 MUX was loaded at the right time when it was enabled appropriately, the values correctly went into the outputs, and the register was shifting correctly. When simulated in ModelSim, the waveforms, as displayed in Figure 10.5, reflected the correctly expected results from the testbench, making it successful.

2.4 Register D

2.4.1 Design Method

The design of the generic register D was made up an entity and a behavioral architecture for the entity. The entity consists of a clock, and load and count enables as its inputs and counting data value, valueD, as its output.

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 load in new data for its data holders. The process is dependent on the clock and conditioned to the load and count enable pins; it states that when a rising edge is detected on the clock, if the load enable pin is set to HIGH, then a value - with its highest bit set to '1' and the rest to '0' - will be stored in a temporary holder signal, and if the count enable pin is set to HIGH, the temporary holder will shift its value down by one and bring in a '0' along with it. Outside of the process, the temporary value was read into the output value, valueD.

2.4.2 Testing and Simulation

When it came to testing the generic register D with a testbench, the test cases were made to verify that output, valueD, shifts and keeps track of the number of bit-pair recodes that have already been completed correctly. When simulated in ModelSim, the waveforms, as displayed in Figure 10.6, reflected the correctly expected results from the testbench, making it successful.

3 Multiplexer

The multiplier's multiplexers are able to select from the incoming data and determine which of their data values will be able to be the value that will be sent out as the output for the bus. The designs for the multiplexers reflect this description and was based off the multiplexer design found on page 343 of Brown [1]. The two multiplexers made for the multiplier were a 5-to-1 and a 2-to-1 multiplexer; because they needed to be flexible with the bits they can hold, a generic was used and initially set to eight for the bit size for the multiplexers' input signals.

3.1 5-to-1 Multiplexer

3.1.1 Design Method

The design of the multiplexer was made up an entity and a behavioral architecture for the entity. The entity consists of a 3-bit selector signal and four different signals to hold the data values for the incoming data signal as its inputs, an n-bit buffer filled with zeros - since that is another possible version of the multiplicand - and a data holder, mux-out, as its output.

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. The process is dependent on the selector signal and all of the possible values the output value could be - positive one, negative one, positive two, negative two, and zero. Using a case-when statement, the process will set a temporary holder for the output value to one of the five input signals that corresponds to one of the values of selector. The selector's values were made using the multiplier recoding chart from Slide 10 of Brown[3] shown in Figure 10.2. The default selection for the multiplier was set as zero since that is the option that will not do damage to the multiplier if a problem comes about. Outside of the process, the output value will read in the value the temporary holder is storing.

3.1.2 Testing and Simulation

When it came to testing the multiplexer with a testbench, the test cases 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 10.7, reflected the correctly expected results from the testbench, making it successful.

3.2 2-to-1 Multiplexer

3.2.1 Design Method

The design of the multiplexer was made up an entity and a behavioral architecture for the entity. The entity consists of a 1-bit selector signal represented through the load enable pin and the signal, in-value, the incoming data signal as its inputs,- and a data holder, out-value, as its output.

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. The process is dependent on the selector signal and all of the possible values the output value could be - in this case, just in-value - and conditioned to the inverse of the load enable pin. If the inverse of the pin is '0' then the multiplexer will output all zeros, else, it will output the value stored in in-value.

3.2.2 Testing and Simulation

When it came to testing the multiplexer with a testbench, the test cases were made to check the input signal's value or zeros were being appropriately outputted when the selector's value corresponded to the correct input signal. When simulated in ModelSim, the waveforms, as displayed in Figure 10.8, reflected the correctly expected results from the testbench, making it successful.

4 Adder

The multiplier's adder unit is able to add the two values from the proper version of the multiplicand and the value stored in the sum signal from register C and then output the sum into the input value of the 2-to-1 multiplexer. The design for the adder 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 unit was made up of 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 a carry-in bit (cin) as its inputs and a sum value (s) and a carry-out bit (cout) as its outputs. Because it needed to be flexible with the bits it can hold, a generic was used and initially set to eight for the bit size for the registers to be used for the processor. It's also important to note that this is an n+1 bit adder since two of the possible values of the multiplicand are n+1 bits wide.

A structural architecture had the full adder component, n signals c and 3 instances - the first adder, middle adders via generate statement, and the last adder - of the full adder with their port maps corresponding to their positions in the full adder. Because the ripple carry has n full adders, it would be logical and efficient to structure the architecture as a component and adding instances to it.

4.2 Testing and Simulation

When it came to testing the add-sub unit with a testbench, the test cases were made to make sure the adder was adding correctly since it was modified to become a generic. When simulated in ModelSim, the waveforms, as displayed in Figure 10.9, reflected the correctly expected results from the testbench, making it successful.

5 Control Unit

The control unit for the multiplier controls the steps for bit-pair recoding and multiplication. The control unit can perform four operations: (1) load (2) add (3) shift (4) finish. The unit will then set enable the correct registers so they can do their functions

appropriately. The design for the control unit reflects this description and was based off on a Moore 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 start bit (start), a n/2-bit input to keep count (valueD), and a 3-bit selector input from register B (sel-b) as its inputs and a done bit (done), a 3-bit data holder for the selector (sel-ctrl), a bit for the load, add, and shift enable pins for the registers and multiplexers as its output.

A behavioral architecture was made for the register in order to make four functions to: (1) load in the values, (2) add the values, (3) shift the values, and (4) display the final product once all of the adding and shift is completed.

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 state of the control unit to make sure the right signals are enabled and selected for the other units. When simulated in ModelSim, the waveforms, as displayed in Figure 10.11, reflected the correctly expected results from the testbench, making it successful. The VHDL-generated state diagram for the control unit, as shown in Figure 10.10, reflects the correctness of the design.

6 Bit-Pair Multiplier

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 incoming data signals for the multiplicand and the multiplier, a clock, a resetn pin, and a start pin as its inputs and a done pin, a busy pin, and the n*2-bit product as its outputs. Because it needed to be flexible with the bits it can hold, a generic was used and initially set to eight for the bit size of the multiplier.

A structural architecture was made for the multiplier in order to include all of the components needed for the multiplier, signals for every connection that isn't an essential input or output for the multiplier, and then map everything for functionality. The given diagram for the processor, shown in Figure 10.12, was used as a guide when mapping all of the signals.

6.2 Testing and Simulation

When it came to testing the multiplier with a testbench, the test cases were made to check every state possible to assure it functions correctly and that the answer was correct when tie multiplier said it was done. When simulated in ModelSim, all of the waveforms

from the pins and internal signals, as displayed in Figure 10.13, 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 multiplier on the board, the implementation for the board was done similarly to the processor board implementation from lab 4. Using the board's switches, keys, and ledr lights, three inputs of the processor - start, multiplicand, and multiplier - were mapped to the switches, inputs CLK and RESETN were mapped to keys, while the product, busy, 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 out that because of the size of the board, the generic values need to reduced from 8-bits to 4-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 and functioned correctly. The main issue I faced while working on this lab was was getting the control unit to fully function and get the right values in registers C and B after shifting. When the final part was tested on the board with all the other parts and components combined with it, the board displayed the correct products based on the inputs, meaning it was a 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; I also got a nice review and in-depth look into bit-pair multiplication. If I could go back and change how things were done throughout the lab, I would try to see if there were any other ways to make the hardware more efficient while reducing the amount of coding done, even though it was pretty short with the exception of the bit-pair file dealing the port mapping.

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.

[3] M. Smith. Sequential circuits: Alu part iii. Lecture Notes, South Carolina, USA, 2019.

10 Appendix

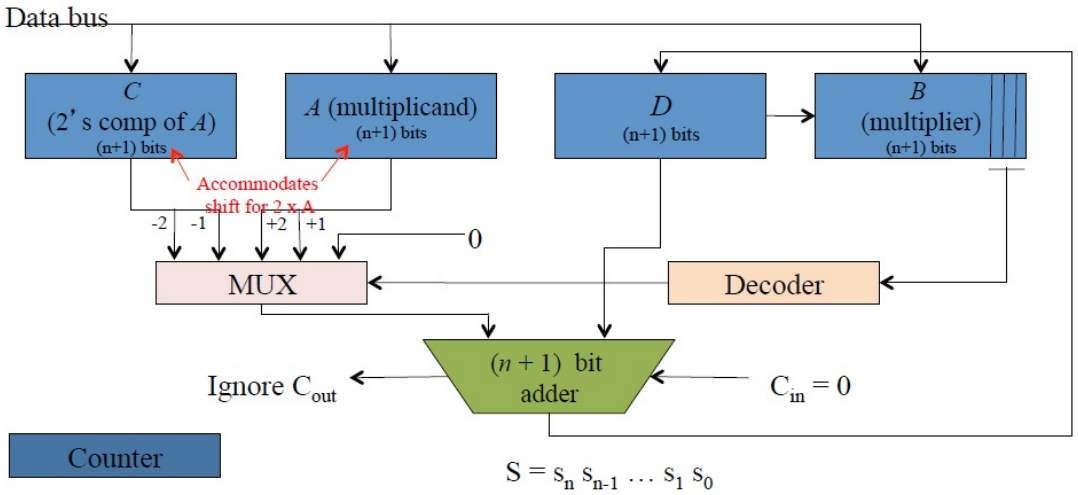

Figure 10.1: Bit-Pair Datapath Circuit from Lecture Notes

Multiplier bit-pair		Multiplier bit on the right	Multiplicand version to be added	Relation to string of 1's
2^1	2^0			
$i+1$	i	$i-1$		
0	0	0	0 x multiplicand	No string
0	0	1	+1 x multiplicand	End of string
0	1	0	+1 x multiplicand	Single 1 (+2 -1)
0	1	1	+2 x multiplicand	End of string
1	0	0	-2 x multiplicand	Beginning of string
1	0	1	-1 x multiplicand	End/beginning of string
1	1	0	-1 x multiplicand	Beginning of string
1	1	1	0 x multiplicand	String of 1's

Figure 10.2: Possible Versions of Multiplicand from Lecture Notes

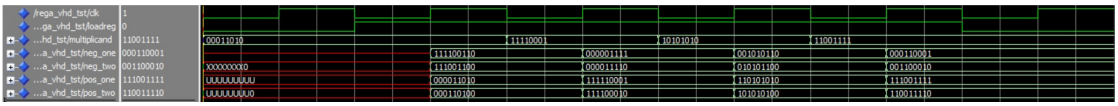

Figure 10.3: Register A Simulated Testwaves

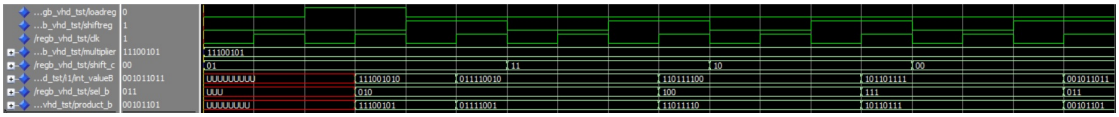

Figure 10.4: Register B Simulated Testwaves

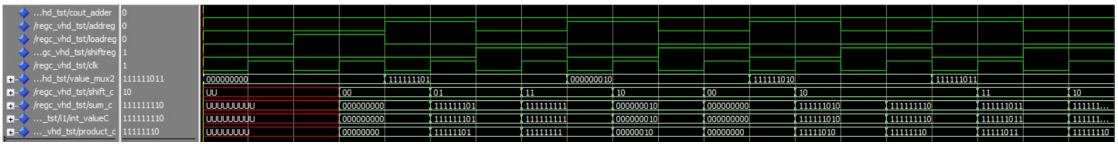

Figure 10.5: Register C Simulated Testwaves

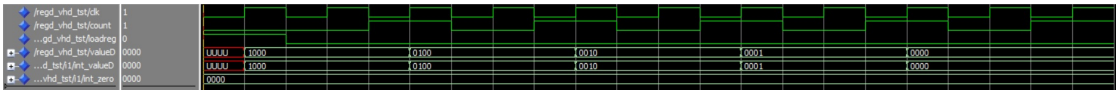

Figure 10.6: Register D Simulated Testwaves

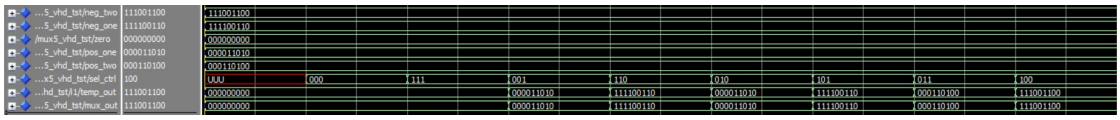

Figure 10.7: 5-to-1 Multiplexer Simulated Testwaves

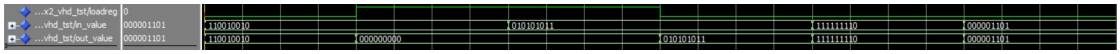

Figure 10.8: 2-to-1 Multiplexer Simulated Testwaves

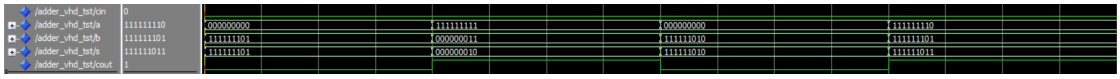

Figure 10.9: Adder Simulated Testwaves

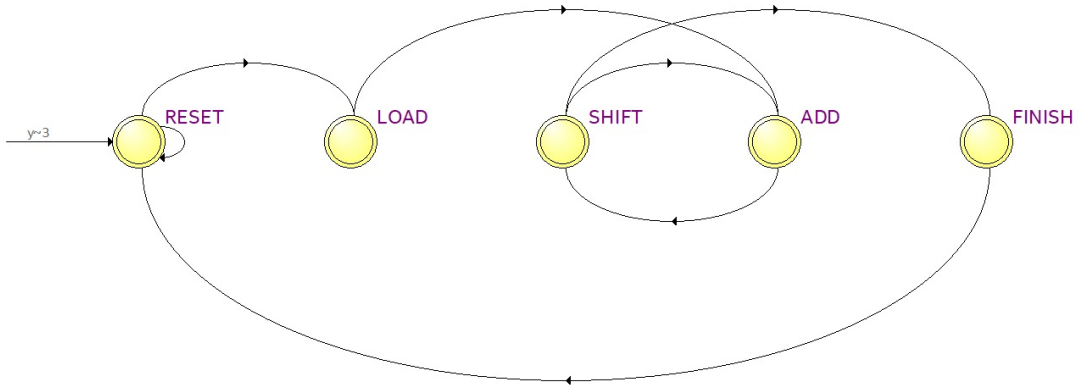

Figure 10.10: Control Unit VHDL-Generate State Machine Diagram

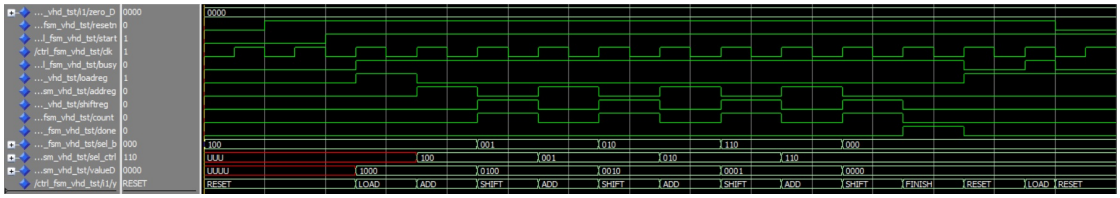

Figure 10.11: Control Unit Simulated Testwaves

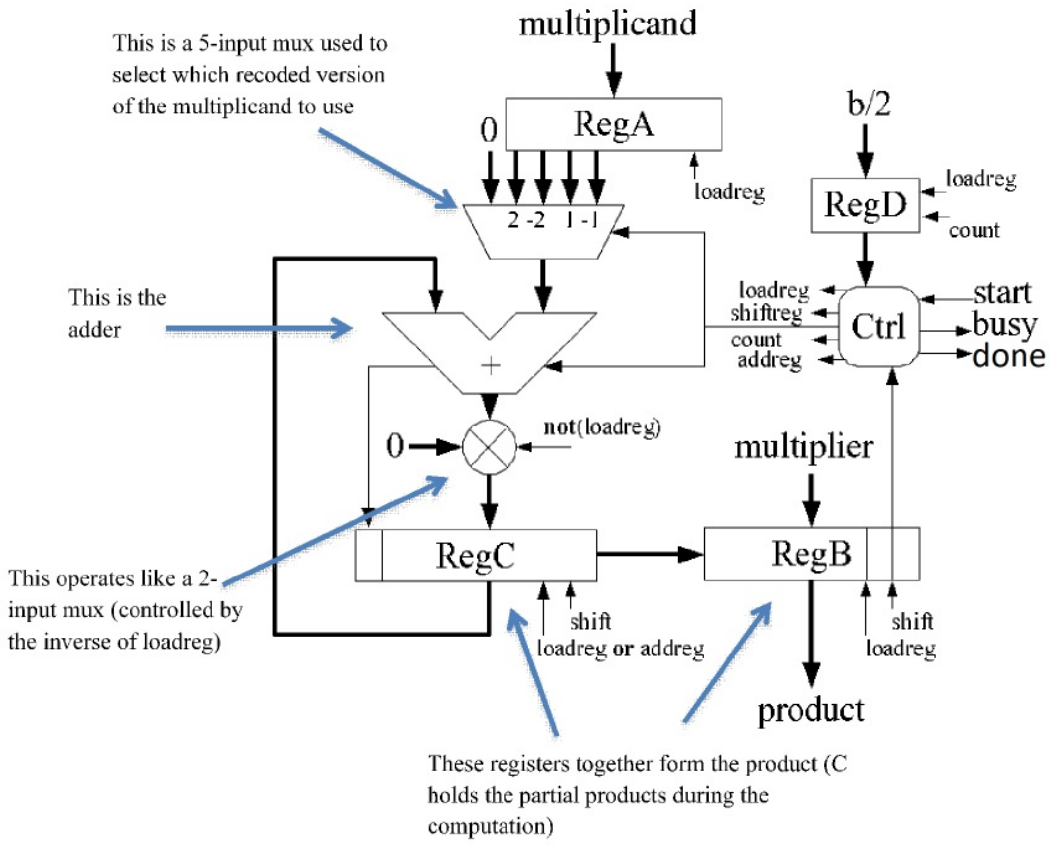

Figure 10.12: Bit-Pair Multiplier Diagram

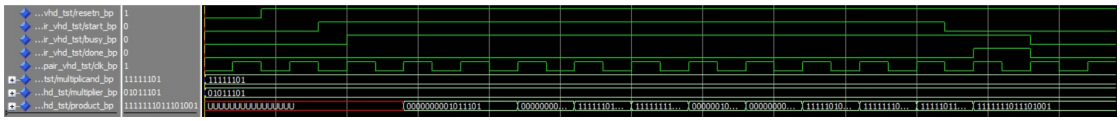

Figure 10.13: Bit-Pair Multiplier Simulated Testwaves