BIT PAIR RECODED MULTIPLIER

Lab Report for ECE3270
Digital Systems Design

Submitted by

Ryan Barker

Department of Electrical & Computer Engineering
Clemson University

03/16/2015

Abstract

The goal of this experiment was to design, simulate, and test an eighteen-bit bit-pair recoded multiplier (also known as a modified booth algorithm recoded multiplier). Though a basic data flow diagram for the circuit was provided, the design process was still very complex and required careful preparation of a more complete data flow diagram and an algorithmic state machine (ASM) chart for the controller macro. Most of the macros designed took advantage of generic statements, so it would be easy to re-size them for any number of bits. After design was complete, the bit-pair multiplier was simulated with three different test benches containing six total tests to verify proper operation. The project as a whole stressed the importance of designing software packages modularly, as it required multiple macros to function and each had to be validated and tested individually before it was possible to connect them in the overall circuit.

Introduction

This lab only contained one goal of designing the bit pair multiplier, but accomplishing it required multiple steps and vigilant preparation. Because this lab contain a surplus of information, the design process began by breaking the multiplier down into manageable pieces and generating the data flow diagram in Figure I.1. This was used as the framework for the entire design, and made it easy to track the designer's progress as macros were built. Additionally, a modular approach was taken while building the overall machine, so each required macro was contained in its own file, and test code was written for every macro with the exception of the shift registers (Registers B and C) and the controller. For these two cases, two additional VHDL files were constructed that port mapped registers B and C together and the controller and the counter (Register D) together. This way, the behavior of Register C passing Register B its two least significant bits on each shift could be observed and the controller could be simulated with counter input independent of the rest of the circuit. This process created slightly more work for the designer, but paid off in the end, as it drastically simplified the debugging process.

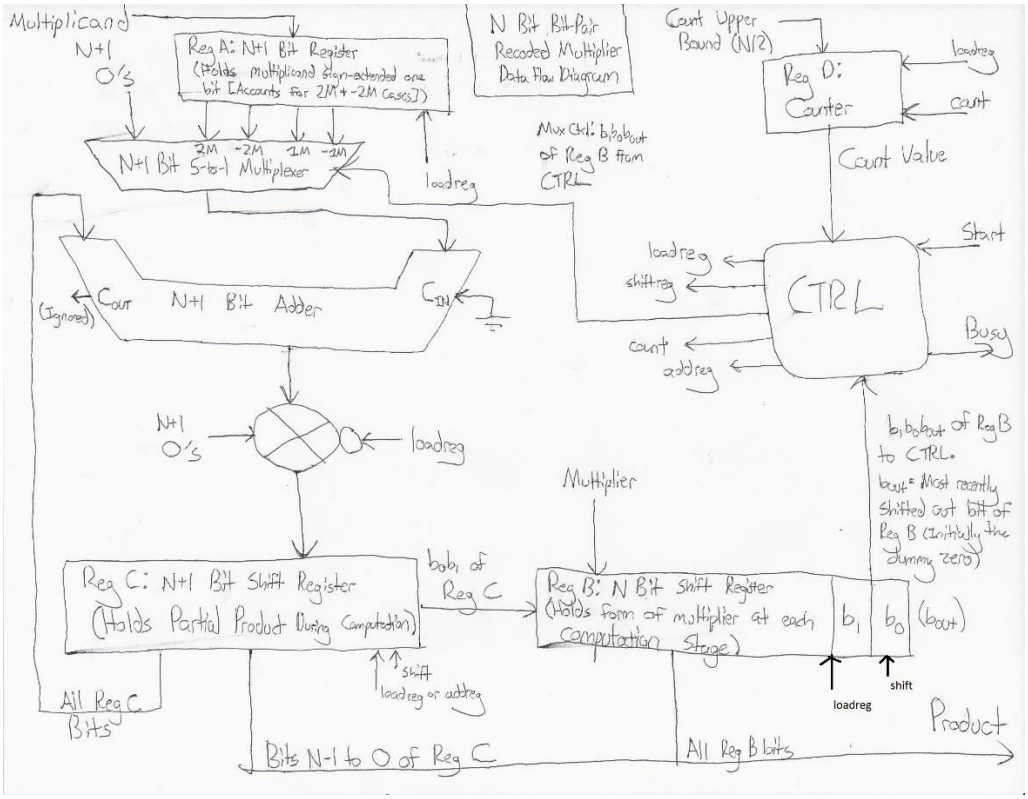


Figure I.1. Data Flow Diagram for Bit-Pair Multiplier Design (Based on Diagram in [1]).

Section 1: Register A and its Components

Section 1.A: Designing Register A

The purpose of the upper left-hand portion of the circuit is to select the proper form of the multiplicand based on the recoding bits and send it to the main adder macro. It is then summed with the partial product at each stage inside of the adder. Accomplishing this behavior requires a register to hold the multiplicand during computation and a 5-to-1 multiplexer, as the bit pair algorithm states that at each stage, either zeroes, the multiplicand, two times the multiplicand, negative one times the multiplicand, or negative two times the multiplicand is added to the partial product [2].

Because of the above information, Register A is nothing more than a falling edge activated, nineteen bit register that loads the multiplicand sign extended one bit when its loadreg signal is high. It is 19 bits rather than 18 bits because the multiplexer is capable of selecting two times the multiplicand and negative two times the multiplicand (Or two times the two's complement of the multiplicand). Note that these values are not computed inside of Register A, and are rather saved for combinational logic in the inputs to the multiplexer to keep the overall circuitry simple. All VHDL code for Register A is seen in appendix A.1.1.

Section 1.B: Testing Register A

Register A was very simply simulated by loading a value on the falling edge of its clock, changing the value with the load signal low and observing that the change was ignored, and bringing the load signal high and watching the changed value become absorbed by the register. Both positive and negative values were used in simulation to verify proper operation of the sign extend. Appendix A.1.2.1 shows the test bench used in simulation and appendix A.1.2.2 shows the output waveform, which exhibits the described behavior.

Section 1.C: Designing the 5-to-1 Multiplexer

The first multiplexer in the design is a nineteen bit, five-to-one multiplexer that recodes the multiplicand according to the least two significant bits of Register B and the most recently shifted out bit of Register B, as per the bit-pair algorithm. It is designed around a case-when statement that goes through all possible values of these control bits, and outputs the appropriate

form of the multiplicand accordingly. When two times the multiplicand is computed, the multiplexer takes the existing multiplicand and shifts it left one bit, which is the reason the multiplexer output is nineteen bits rather than eighteen. The two's complement of the multiplicand is computed in a variable in the architecture and used when the multiplicand is multiplied by a negative value. This computation follows the methodology that the two's complement of a number is one plus the one's complement of that number. It utilizes the unsigned() conversion function from the numeric_std library [3], since Quartus does not allow addition for signals of type std_logic_vector. All VHDL code for the 5-to-1 Multiplexer is seen in appendix A.1.3.

Section 1.D: Testing the 5-to-1 Multiplexer

The 5-to-1 multiplexer was simulated by loading a positive value in for the multiplicand and checking the output for each sequence of control bits. The tests were then repeated for a negative multiplicand. Figure 1.1 shows the output from simulation, and appendix A.1.4 shows the test bench used. Notice that in Figure 1.1 and most of the figures forward of this point, the simulation bits are kept in abbreviated form, as eighteen and nineteen bit numbers make the simulation very hard to read (As seen in appendix A.1.2.2).

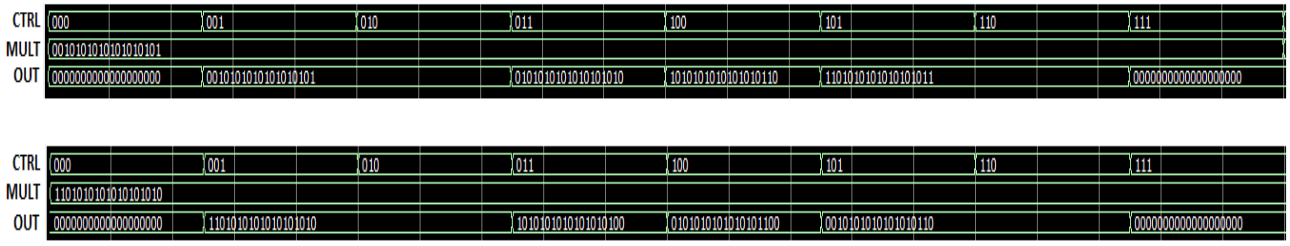


Figure 1.1. 5-to-1 Multiplexer Simulation Waveform (Top = Test 1, Bottom = Test 2).

Section 2: Shift Registers B and C

Section 2.A: Designing Shift Register B

Shift Register B is eighteen bits and holds shifted forms of the multiplier during the multiplication, and the eighteen least significant bits of the product when computation is complete. It is falling edge activated, as all of the registers in the design are. Its least two

significant bits plus its last shifted out bit make up the recoding bits for the bit pair algorithm, which are sent to the FSM controller to be redirected to Register A's multiplexer at each stage of the multiplication. At the beginning of computation, Register B loads the multiplier and outputs its last two bits plus a "dummy zero" bit as the initial recoding bits. It then shifts its contents two bits right during each stage of computation, using the bits shifted out of Register C as its new most significant two bits so no information is lost. This process repeats until the multiplication is complete. All VHDL code for Shift Register B is seen in appendix A.2.1. The code for Register B was not tested until the code for Register C was completed, so all shifting behavior could be observed at once.

Section 2.B: Designing Shift Register C

Shift Register C is nineteen bits and holds the partial product during the multiplication, and the eighteen most significant bits of the product plus an extra bit when the computation is complete. It is nineteen bits because it has to be added with the output from the 5-to-1 multiplexer at each stage, which is nineteen bits. Any time a load or add occurs, Register C loads the output from the main adder's 2-to-1 multiplexer as the partial product and outputs them into the main adder. Then, when a shift occurs, it utilizes variables in its architecture to shift its contents two bits right and send its least two significant bits to Register B, so no information is lost. Its new two most significant bits are sign extends of the old most significant bit. All VHDL code for Shift Register C is seen in appendix A.2.2.

Section 2.C: Testing the Shift Registers

Once both shift registers were built, wrapper VHDL code was written which connected them together in a structural architecture so they could both be simulated at once. This was useful for watching Register C correctly pass Register B its least two significant bits during each shift. The wrapper code is seen in appendix A.2.3.1.

The test bench for the wrapper code runs two tests that simulate one times a positive number and one times a negative number. Each test loads the initial number as the partial product, one as the multiplier, and shifts for nine cycles. During each shift in the process, the last two bits of Register C move into Register B and Register C sign extends itself based on the sign of the partial product. At the end of both tests, the initial number is returned as the product.

Figure 2.1 shows this behavior, with the bits moving from Register C to Register B circled in red. Appendix A.2.3.2 shows the test bench used in simulation.

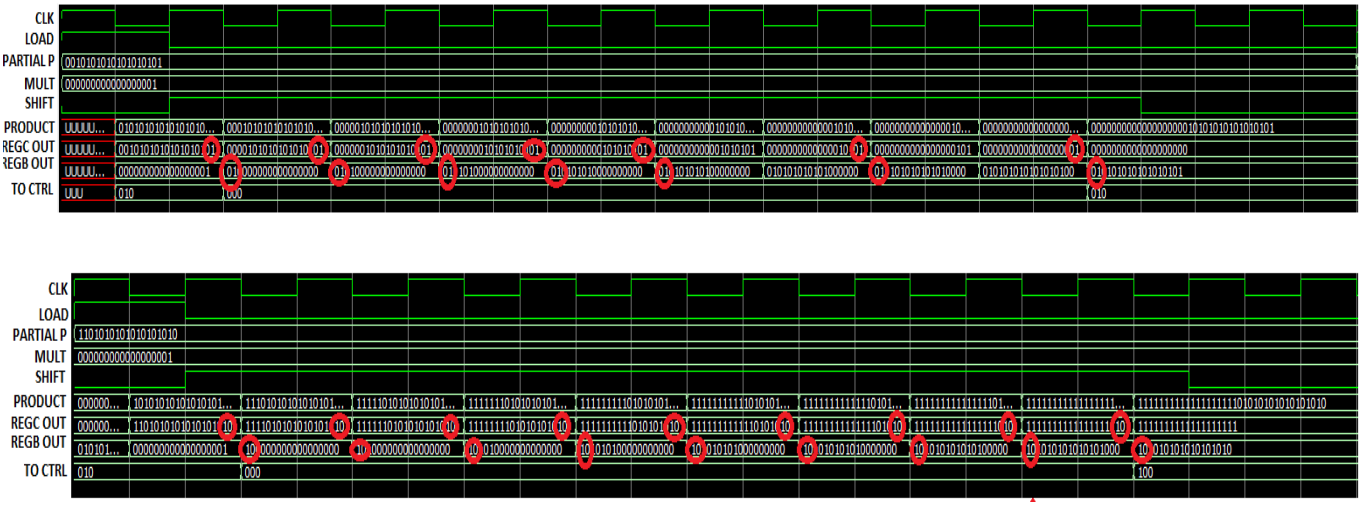


Figure 2.1. 5-to-1 Shift Registers Simulation Waveform (Top = Test 1, Bottom = Test 2).

Section 3: Counter Register D

Section 3.A: Designing Register D

Register D is a four bit counter that counts from zero to its max value input (In this case, nine). It is one of the only parts of the design that is not generic, as the designer had trouble trying to implement floor($\log_2(N/2)$) as the upper bound of its output std_logic_vector. In any case, when load is set to one on a falling edge of the clock signal, register D initializes an internal unsigned counter signal to zero. Then, whenever count is high on a falling edge of the clock, if the current value of the counter is less than the maximum value, Register D increments the counter. Otherwise, it wraps the counter back around to zero. In the final design, Register D is used by the FSM controller to determine when a multiplication is finished, since the bit pair algorithm guarantees that an N bit multiplication will take N / 2 iterations [2]. All VHDL Code for Counter Register D is seen in appendix A.3.1.

Section 3.B: Testing Register D

Register D was tested by loading nine as its max value, toggling the load signal to initialize the internal counter to zero, and setting count high. It was then left alone until it successfully counted to nine, at which point the maximum value input was changed to five and it began counting from zero to five. Figure 3.1 shows this behavior. Appendix A.3.2 shows the test bench used in simulation.

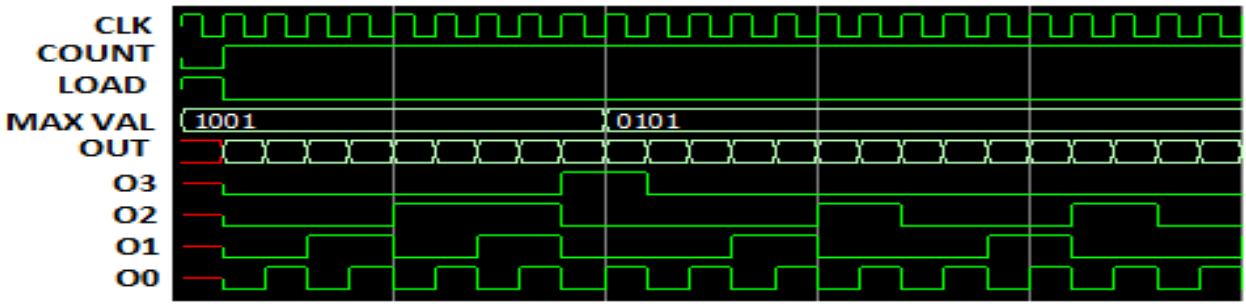


Figure 3.1. Counter Register Simulation Waveform.

Section 4: FSM Controller and the Remaining Components

Section 4.A: Designing the FSM Controller

Before design for the controller began, the ASM chart shown in appendix A.4.1.1 was drawn to obtain an idea of the structure of the code. In the overall multiplication device, the machine will sit idle with its busy signal low until the start signal is received as high, at which point it will load the multiplier and multiplicand by outputting the loadreg signal. Then, on the next clock cycle the controller commands an addition by outputting addreg, followed by a shift on the subsequent cycle by outputting shiftreg. Further, every time an addition occurs, the controller outputs count to increment the value in Register D. This add-shift cycle continues until the count value output by Register D reaches nine (eighteen over two), at which point the machine idles again. Therefore four states and two decisions are necessary in the process, as shown in appendix A.4.1.1. Also, the controller needs to send its input from Register B to the 5-to-1 multiplexer, so it simply ties this input to that multiplexer's control output line. Finally, for synchronization purposes, the controller was set to change states on rising edges of the clock and all registers responding to its input were set to take input on falling edges of the clock. This

allowed output from the controller could be safely processed by each register. All VHDL code for the FSM controller is seen in appendix A.4.1.2.

Section 4.B: Testing the FSM Controller

Because the state of the Controller is dependent on the count output from Register D, VHDL wrapper code was again constructed to map the FSM Controller to Register D for testing as shown in appendix 4.2.1. Beyond this, testing the Controller was as simple as pulsing the start signal and observing the previously discussed behavior, as shown in Figure 4.1. Note because Register B was not connected, the test bench simply holds the multiplexer control line constant. Appendix A.4.2.2 shows the test bench used in simulation.

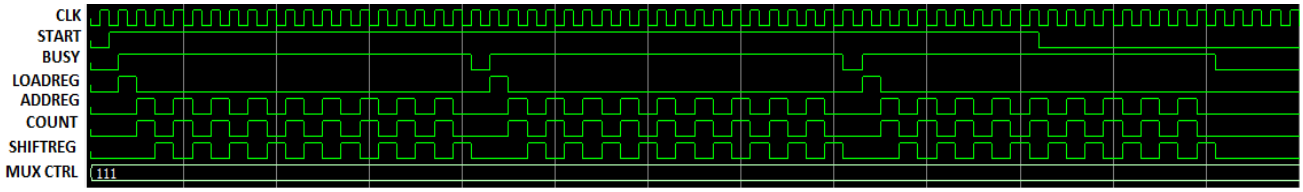


Figure 4.1. Controller Test Simulation Waveform.

Section 4.C: Designing the N + 1 Bit Full Adder

Thankfully, though the rest of the circuit is complex, the remaining two macros are both very simple. The N + 1 Bit Full Adder (Set up to be a 19 bit full adder for this case) is a simple ripple adder made of several cascaded one-bit full adders. Each of these was created with strict Boolean logic, specifically AND, OR, and XOR gates. The carry in for the overall adder was grounded, and the carry out was ignored, since each of the adder's inputs are already sign extended an extra bit. All VHDL code for the N + 1 Bit Full Adder is seen in appendix A.4.3.

Section 4.D: Testing the N + 1 Bit Full Adder

Testing the adder was as straightforward as designing it. The test bench simply adds three different sets of numbers, which run tests on both positive and negative numbers and the carry out bit, as shown in Figure 4.2. Appendix A.4.4 shows the test bench used in simulation.

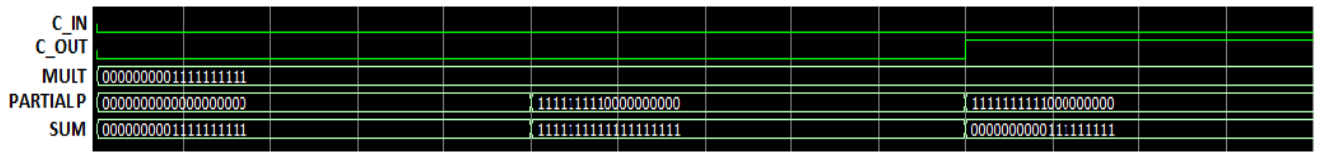


Figure 4.2. Full Adder Simulation Waveform.

Section 4.E: Designing the 2-to-1 Adder Multiplexer

The final macro required for the overall design is also the simplest. The 2-to-1 adder multiplexer is a very forthright 2-to-1 multiplexer circuit that sends zeroes into Register C on a load and otherwise sends the N + 1 Adder's output into Register C. All VHDL code for the 2-to-1 adder multiplexer is seen in appendix A.4.5.

Section 4.F: Testing the 2-to-1 Adder Multiplexer

It is no surprise that the simplest construct in the design also has the simplest test bench. The test here initializes the input of the 2-to-1 Adder Multiplexer to all ones and toggles to loadreg line to ensure correct behavior. Appendix A.4.5 shows the test bench used in simulation.

Figure 4.3. 2-to-1 Adder Multiplexer Simulation Waveform.

Section 5: The Overall Bit-Pair Recoded Multiplier

Section 5.A: Designing the Overall Bit-Pair Recoded Multiplier

The top level entity had a very large structural architecture that mapped all nine previously discussed macros in the manner Figure I.1 shows to build the overall multiplier. The architecture contained fifteen signals for all of the intermediate lines within the circuit. Figure 5.1 shows the overall bit-pair recoded multiplier from the RTL Viewer's point of view. All VHDL code for the overall bit-pair recoded multiplier is seen in appendix A.5.1.

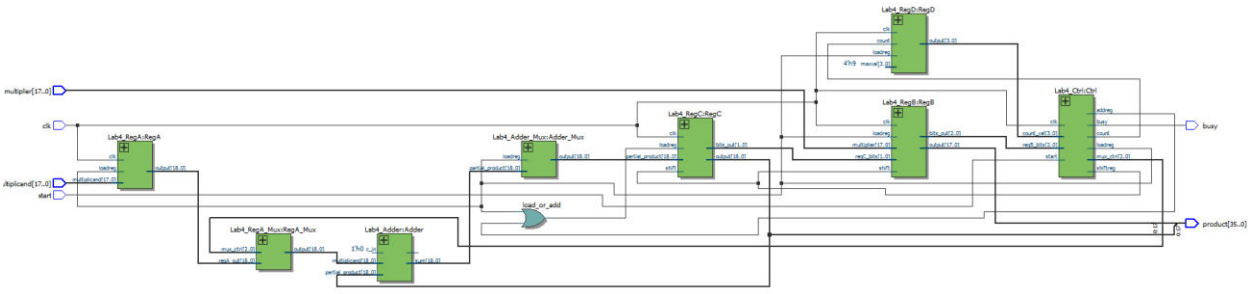


Figure 5.1. The Bit Pair Recoded Multiplier Circuit.

Section 5.B: Testing the Overall Bit-Pair Recoded Multiplier

All of the testing done up to this point was to ensure the bit pair multiplier would function when the numerous macros in the design were hooked together, but to be safe, the multiplier itself was thoroughly tested to observe its behavior with five different test benches. This section describes each, and then examines the output from each in ModelSim.

The first test bench was a multiplication of two positive numbers, specifically a one and alternating zeroes and ones. This multiplication helped determine if the overall circuit was functional, as for the first test, the output was expected to be the multiplicand since the multiplier was a one. It then did the same multiplication in reverse order for the second test, to ensure the commutative property of multiplication was preserved in the circuit. Figure 5.2 shows the simulation waveform. Notice that the busy signal goes high immediately after the start signal pulses, and as soon as the busy signal is low, the correct product is available from the circuit. Also notice that the products for both multiplications are correctly the same. Test bench one is shown in appendix A.5.2.1.

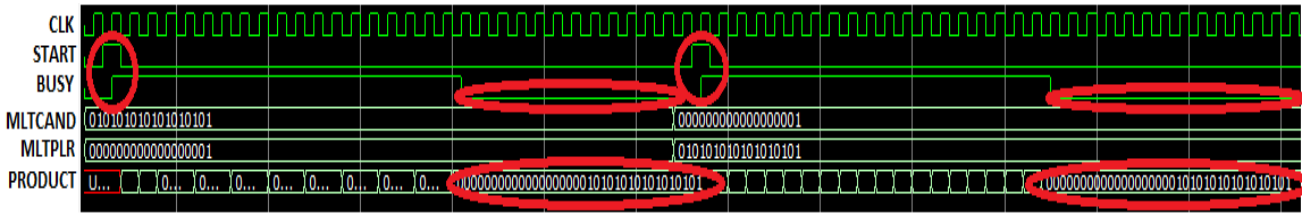


Figure 5.2. Test Bench One Simulation Waveform.

The second test bench ran tests very similar to the first, it just multiplied a negative number by a positive number instead of a positive number by a positive number. The output was

still expected to be the same for both tests. Figure 5.3 shows the simulation waveform, which has the desired behavior. Test bench two is shown in appendix A.5.2.2.

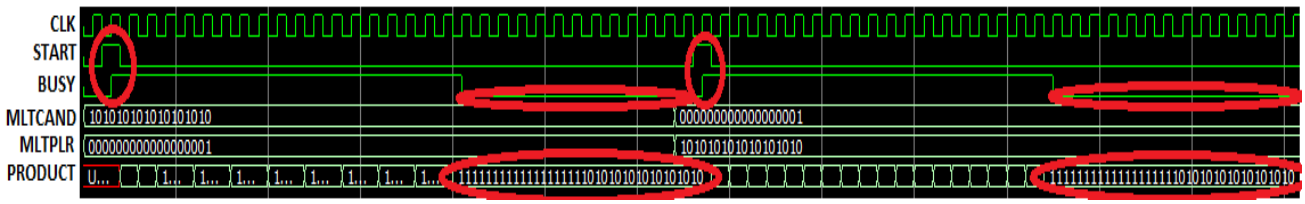


Figure 5.3. Test Bench Two Simulation Waveform.

The third test bench was the final sign-testing test bench, which multiplied a negative number by a negative number, specifically negative one and alternating ones and zeroes. It comes as no surprise that the expect output was the positive variant of the ones and zeroes operator. Figure 5.4 shows the simulation waveform, which has the desired behavior. Test bench three is shown in appendix A.5.2.3.

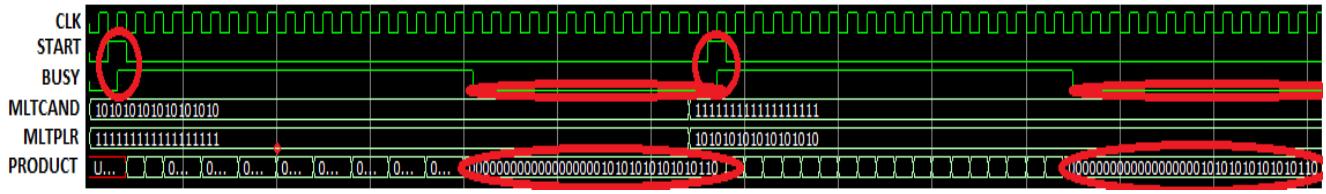
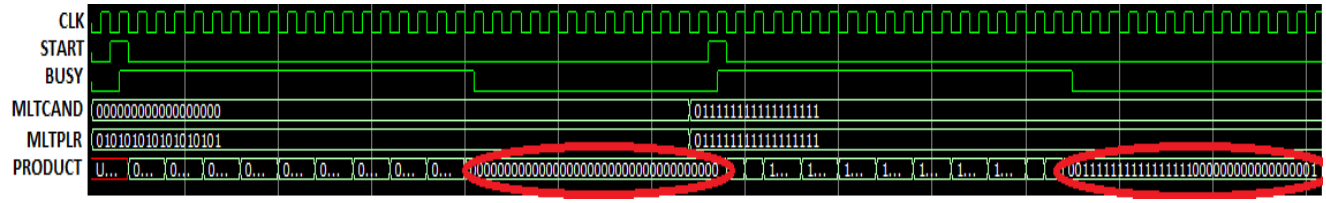


Figure 5.4. Test Bench Three Simulation Waveform.

The last two tests were range tests for positive and negative operands that multiplied the smallest and largest positive and negative multiplicands and multipliers together and showed the output product. Test bench four tested the range of positive numbers, and test bench five tested the range of negative numbers. Each test performed the minimum bound multiplication and then the maximum bound multiplication. The output for both test benches is shown in Figure 5.5. Notice that because of the sign bit in the multiplier and multiplicand, not all thirty-six bit numbers can be represented by the product.



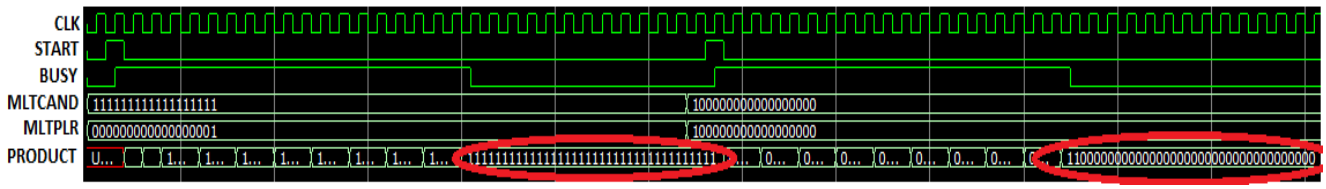


Figure 5.5. Test Bench Four and Five Simulation Waveforms (Top = TB4, Bottom = TB5)

Conclusions

The biggest lesson this lab taught was the importance of careful preparation and modular code design. When the overall multiplier was first constructed, it did not work as expected, due to an error in the shift registers. This error would have been impossible to debug without the Lab4_Shift_tst.vhdl file. Appendix A.C provides some perspective by showing the overall circuit completely expanded in the RTL Viewer: The circuit just contains too many components to debug at the top level. Breaking things into pieces is a much easier, sane approach.

The lab also showed the power of recoding techniques when dealing with multiplication of large bit numbers. Multiplying two eighteen bit numbers together under a traditional add-shift method requires thirty-six clock cycles. The eighteen cycles the bit-pair algorithm provides is a substantial reduction.

Lastly, though the designer could not fully implement the functionality, the lab showed the power of generic statements in code design. These allow circuits to be designed for any number of bit inputs and outputs, which makes porting code from different systems substantially easier. They make code easier to read and understand, as they force the designer to write code for the general case and make the designer avoid coding for specific cases only.

References

[1] M. Smith. *ECE327 Digital System Design, Lab 4: Bit Pair Recoded Multiplier.* [Online]. Available: https://bb.clemson.edu/bbcswebdav/pid-1898812-dt-content-rid-22057806_2/courses/smithmc-ece-327-DSD/Lab4.bitpair.mult.project%286%29.pdf

[2] M. Smith. *ECE327 L06P3-Bit Pair.* [Online]. Available: https://bb.clemson.edu/bbcswebdav/pid-1598005-dt-content-rid-8493450_2/courses/smithmc-ece-327-DSD/L06P3-sequential.ckts.alu%28bit-pair%29%281%29.pdf

[3] IEEE and UMBC. *Numeric_std.vhdl.* [Online]. Available: http://www.csee.umbc.edu/portal/help/VHDL/numeric_std.vhdl

APPENDIX

Appendix A.1: Register A and its Components

A.1.1: Register A VHDL Code

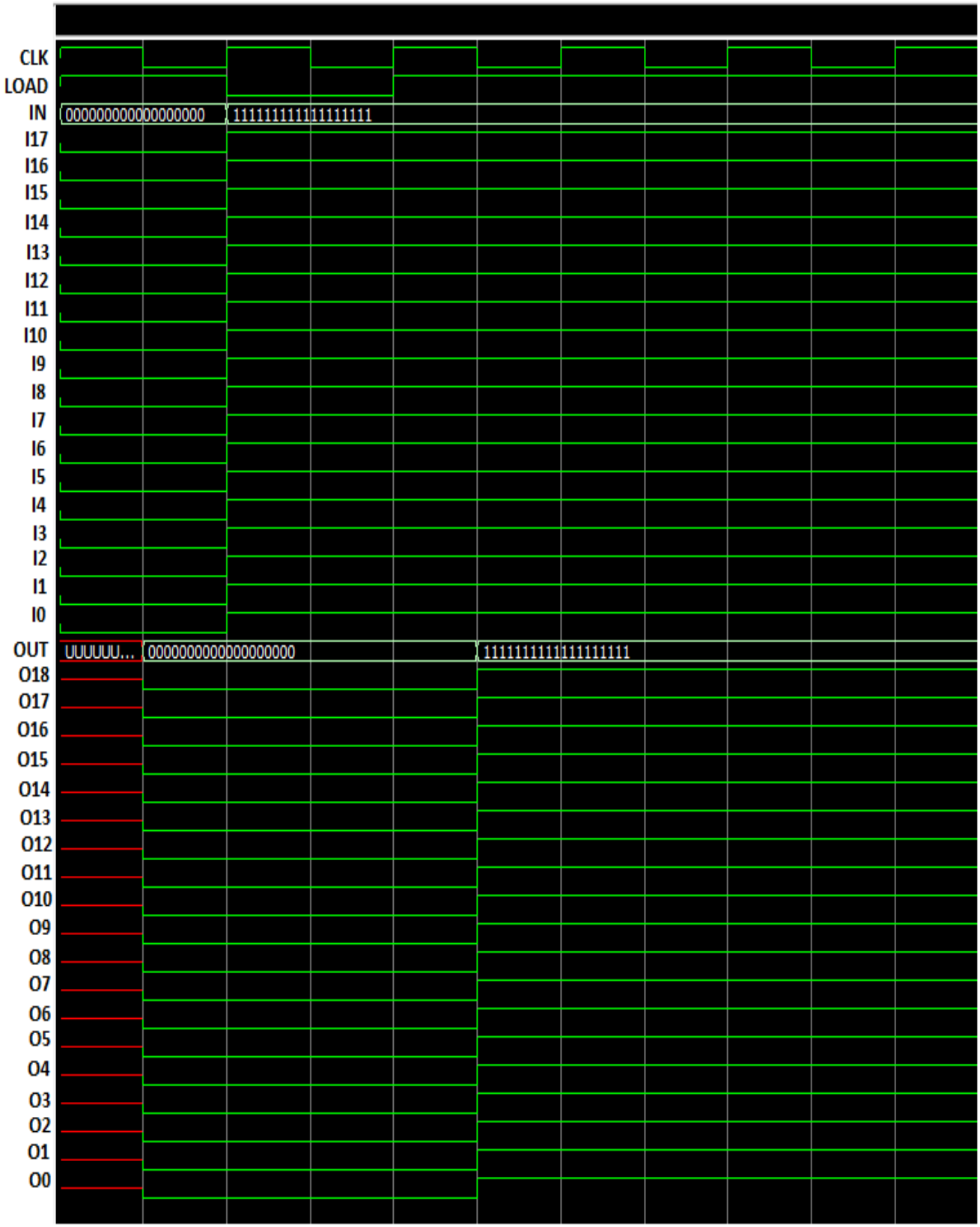
```
1   -- Ryan Barker --
13
14  LIBRARY ieee;
15  USE ieee.std_logic_1164.all;
16
17  -- Declare Register A --
18  ENTITY Lab4_RegA IS
19     GENERIC (N : INTEGER := 18);
20     PORT (multiplicand : IN std_logic_vector(N - 1 DOWNTO 0);
21           loadreg      : IN std_logic;
22              clk          : IN std_logic;
23              output       : OUT std_logic_vector(N DOWNTO 0));
24  END Lab4_RegA;
25
26  -- Architecture of Register A --
27  ARCHITECTURE Lab4_RegA_B OF Lab4_RegA IS
28  BEGIN
29      init: PROCESS (clk, loadreg)
30      BEGIN
31          IF(falling_edge(clk) AND loadreg = '1') THEN
32              -- If loading, set output to multiplicand --
33              output(N) <= multiplicand(N - 1); -- Sign Extend --
34              output(N - 1 DOWNTO 0) <= multiplicand;
35          END IF;
36      END PROCESS init;
37  END Lab4_RegA_B;
```

A.1.2.1: Register A Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab4_RegA_vhd_tst IS
32  END Lab4_RegA_vhd_tst;
```

```
33  ARCHITECTURE Lab4_RegA_arch OF Lab4_RegA_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL clk : STD_LOGIC;
37  SIGNAL loadreg : STD_LOGIC;
38  SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
39  SIGNAL output : STD_LOGIC_VECTOR(18 DOWNTO 0);
40  COMPONENT Lab4_RegA
41      PORT (
42      clk : IN STD_LOGIC;
43      loadreg : IN STD_LOGIC;
44      multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
45      output : OUT STD_LOGIC_VECTOR(18 DOWNTO 0)
46      );
47  END COMPONENT;
48  BEGIN
49      i1 : Lab4_RegA
50      PORT MAP (
51  -- list connections between master ports and signals
52      clk => clk,
53      loadreg => loadreg,
54      multiplicand => multiplicand,
55      output => output
56      );
57  init : PROCESS
58  -- variable declarations
59  BEGIN
60          -- code that executes only once
61          multiplicand <= "0000000000000000";
62          loadreg <= '1'; wait for 10 ps;
63            loadreg <= '0';
64            multiplicand <= "1111111111111111"; wait for 10 ps;
65            loadreg <= '1';
66  WAIT;
67  END PROCESS init;
68  always : PROCESS
69  -- optional sensitivity list
70  -- (        )
71  -- variable declarations
72  BEGIN
73          -- code executes for every event on sensitivity list
74  WAIT;
75  END PROCESS always;
76  falling_clock : PROCESS
77  BEGIN
78      clk <= '1'; wait for 5 ps;
79      clk <= '0'; wait for 5 ps;
80  END PROCESS falling_clock;
81  END Lab4_RegA_arch;
82
```

A.1.2.2: Register A ModelSim Output



A.1.3: 5-to-1 Multiplexer VHDL Code

```
1  -- Ryan Barker --
11 LIBRARY ieee;
12 USE ieee.numeric_std.all;
13 USE ieee.std_logic_1164.all;
14
15 -- Declare Register A multiplexer --
16 ENTITY Lab4_RegA_Mux IS
17     GENERIC (N : INTEGER := 18);v
18     PORT (regA_out     : IN std_logic_vector(N DOWNTO 0);
19           mux_ctrl     : IN std_logic_vector(2 DOWNTO 0);
20              output       : OUT std_logic_vector(N DOWNTO 0));
21 END Lab4_RegA_Mux;
22
23 -- Architecture of Register A multiplexer --
24 ARCHITECTURE Lab4_RegA_Mux_B OF Lab4_RegA_Mux IS
25 BEGIN
26     multiplex: PROCESS (regA_out, mux_ctrl)
27         VARIABLE twos_complement : std_logic_vector(N DOWNTO 0);
28     BEGIN
29         -- Initialize two's complement --
30          twos_complement := NOT(regA_out);
31          twos_complement := std_logic_vector(unsigned(twos_complement) + 1);
32
33          CASE mux_ctrl IS
34              WHEN "000"|"111" =>
35                  -- Output = 0*M --
36                  zeroes: FOR i IN 0 TO N LOOP
37                  output(i) <= '0';
38                END LOOP;
39              WHEN "001"|"010" =>
40                  -- Output = +1*M --
41                  output <= regA_out;
42              WHEN "011" =>
43                  -- Output = +2*M --
44                  output(0) <= '0';
45                  output(N DOWNTO 1) <= regA_out(N - 1 DOWNTO 0);
46              WHEN "100" =>
47                  -- Output = -2*M --
48                  output(0) <= '0';
49                  output(N DOWNTO 1) <= twos_complement(N - 1 DOWNTO 0);
50              WHEN "101"|"110" =>
51                  -- Output = -1*M --
52                  output <= twos_complement;
53              WHEN OTHERS =>
54                  -- Impossible --
55                  error: FOR i IN 0 TO N LOOP
56                  output(i) <= '0';
57              END LOOP;
58        END CASE;
59     END PROCESS multiplex;
60 END Lab4_RegA_Mux_B;
```

A.1.4: 5-to-1 Multiplexer Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab4_RegA_Mux_vhd_tst IS
32  END Lab4_RegA_Mux_vhd_tst;
33  ARCHITECTURE Lab4_RegA_Mux_arch OF Lab4_RegA_Mux_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL mux_ctrl : STD_LOGIC_VECTOR(2 DOWNTO 0);
37  SIGNAL output : STD_LOGIC_VECTOR(18 DOWNTO 0);
38  SIGNAL regA_out : STD_LOGIC_VECTOR(18 DOWNTO 0);
39  COMPONENT Lab4_RegA_Mux
40      PORT (
41      mux_ctrl : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
42      output : OUT STD_LOGIC_VECTOR(18 DOWNTO 0);
43      regA_out : IN STD_LOGIC_VECTOR(18 DOWNTO 0)
44      );
45  END COMPONENT;
46  BEGIN
47      i1 : Lab4_RegA_Mux
48      PORT MAP (
49  -- list connections between master ports and signals
50      mux_ctrl => mux_ctrl,
51      output => output,
52      regA_out => regA_out
53      );
54  init : PROCESS
55  -- variable declarations
56  BEGIN
57          -- check for a positive number --
58          regA_out <= "00101010101010101010";
59          mux_ctrl <= "000"; wait for 25 ps;
60          mux_ctrl <= "001";  wait for 25 ps;
61           mux_ctrl <= "010";     wait for 25 ps;
62           mux_ctrl <= "011"; wait for 25 ps;
63          mux_ctrl <= "100";  wait for 25 ps;
64          mux_ctrl <= "101";  wait for 25 ps;
65          mux_ctrl <= "110";  wait for 25 ps;
66          mux_ctrl <= "111"; wait for 25 ps;
67
68          -- code that executes only once
69          regA_out <= "11010101010101010010";
70
71          mux_ctrl <= "000"; wait for 25 ps;
72          mux_ctrl <= "001";  wait for 25 ps;
73           mux_ctrl <= "010";     wait for 25 ps;
74           mux_ctrl <= "011"; wait for 25 ps;
75          mux_ctrl <= "100";  wait for 25 ps;
76          mux_ctrl <= "101";  wait for 25 ps;
77          mux_ctrl <= "110";  wait for 25 ps;
78          mux_ctrl <= "111";
```

```
79  WAIT;
80  END PROCESS init;
81  always : PROCESS
82  -- optional sensitivity list
83  -- (        )
84  -- variable declarations
85  BEGIN
86          -- code executes for every event on sensitivity list
87  WAIT;
88  END PROCESS always;
89  END Lab4_RegA_Mux_arch;
90
```

Appendix A.2: Shift Registers B and C

A.2.1: Register B VHDL Code

```
1   -- Ryan Barker --
13
14  LIBRARY ieee;
15  USE ieee.std_logic_1164.all;
16
17  -- Declare Register B --
18  ENTITY Lab4_RegB IS
19      GENERIC (N : INTEGER := 18);
20      PORT (multiplier : IN std_logic_vector(N - 1 DOWNTO 0);
21            regC_bits  : IN std_logic_vector(1 DOWNTO 0);
22            loadreg    : IN std_logic;
23            shift      : IN std_logic;
24            clk        : IN std_logic;
25            bits_out   : OUT std_logic_vector(2 DOWNTO 0);
26            output     : BUFFER std_logic_vector(N - 1 DOWNTO 0));
27  END Lab4_RegB;
28
29  -- Architecture of Register B --
30  ARCHITECTURE Lab4_RegB_B OF Lab4_RegB IS
31      SIGNAL last_bit : std_logic;
32  BEGIN
33      main : PROCESS (clk, loadreg, shift)
34      BEGIN
35          IF(falling_edge(clk) AND loadreg = '1') THEN
36              -- Initially set last_bit to the "dummy zero" --
37              last_bit <= '0';
38
39              -- Initially set output to multiplier --
40              output <= multiplier;
41          END IF;
42
43          IF (falling_edge(clk) AND shift = '1') THEN
44              -- If shifting, shift right two bits --
45              last_bit <= output(1); -- Saves output(1) before it is lost --
46              output(N - 3 DOWNTO 0) <= output(N - 1 DOWNTO 2);
47              output(N - 1 DOWNTO N - 2) <= regC_bits;
48          END IF;
49      END PROCESS main;
50
51      -- Set bits_out to proper bits --
52      bits_out(2 DOWNTO 1) <= output(1 DOWNTO 0);
53      bits_out(0) <= last_bit;
54  END Lab4_RegB_B;
```

A.2.2: Register C VHDL Code

```
1   -- Ryan Barker --
12
13  LIBRARY ieee;
14  USE ieee.std_logic_1164.all;
15
16  -- Declare Register C --
17  ENTITY Lab4_RegC IS
18      GENERIC (N : INTEGER := 18);
19      PORT (partial_product : IN std_logic_vector(N DOWNTO 0);
20            loadreg         : IN std_logic;
21            shift           : IN std_logic;
22            clk             : IN std_logic;
23            bits_out        : OUT std_logic_vector(1 DOWNTO 0);
24            output          : BUFFER std_logic_vector(N DOWNTO 0));
25  END Lab4_RegC;
26
27  -- Architecture of Register C --
28  ARCHITECTURE Lab4_RegC_B OF Lab4_RegC IS
29  BEGIN
30      main: PROCESS (clk, loadreg, shift)
31          VARIABLE shifted_out : std_logic_vector(1 DOWNTO 0);
32          VARIABLE msb : std_logic;
33      BEGIN
34          IF(falling_edge(clk) AND loadreg = '1') THEN
35              -- If loading or adding, set output to input from adder --
36              output(N DOWNTO 0) <= partial_product;
37
38                 -- Not needed for load, but set to last two bits of partial product --
39              bits_out <= partial_product(1 DOWNTO 0);
40          END IF;
41
42          IF(falling_edge(clk) AND shift = '1') THEN
43              -- Save bits coming out and most signficant bit before shift --
44              shifted_out := output(1 DOWNTO 0);
45              msb := output(N);
46
47              -- Shift right two bits --
48              output(N - 2 DOWNTO 0) <= output(N DOWNTO 2);
49
50              -- Sign extend output after shift --
51              output(N - 1) <= msb;
52              output(N) <= msb;
53
54              -- Set bits out to proper value --
55              bits_out <= shifted_out;
56          END IF;
57      END PROCESS main;
58  END Lab4_RegC_B;
```

A.2.3.1: Shift Registers Wrapper VHDL Code

```
1  -- Ryan Barker --
8
9  LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11
12 -- Declare Bit Pair Multiplier --
13 ENTITY Lab4_Shift_tst IS
14    GENERIC (N : INTEGER := 18);
15    PORT (partial_product : IN std_logic_vector(N DOWNTO 0);
16          multiplier      : IN std_logic_vector(N - 1 DOWNTO 0);
17          loadreg         : IN std_logic;
18          shift           : IN std_logic;
19          clk             : IN std_logic;
20          regB_to_ctrl    : OUT std_logic_vector(2 DOWNTO 0);
21          regB_out        : BUFFER std_logic_vector(N - 1 DOWNTO 0);
22          regC_out        : BUFFER std_logic_vector(N DOWNTO 0);
23          product         : OUT std_logic_vector((2 * N) - 1 DOWNTO 0));
24 END Lab4_Shift_tst;
25
26 -- Architecture of Bit Pair Multiplier --
27 ARCHITECTURE Lab4_Shift_tst_B OF Lab4_Shift_tst IS
28     SIGNAL regC_to_regB  : std_logic_vector(1 DOWNTO 0);
29     COMPONENT Lab4_RegB
30        PORT (multiplier : IN std_logic_vector(N - 1 DOWNTO 0);
37        END COMPONENT;
38     COMPONENT Lab4_RegC
39        PORT (partial_product : IN std_logic_vector(N DOWNTO 0);
45        END COMPONENT;
46 BEGIN
47    RegB: Lab4_RegB
48        PORT MAP ( multiplier => multiplier,
49                   regC_bits => regC_to_regB,
50                   loadreg => loadreg,
51                   shift => shift,
52                   clk => clk,
53                   bits_out => regB_to_ctrl,
54                   output => regB_out );
55
56     RegC: Lab4_RegC
57        PORT MAP ( partial_product => partial_product,
58                   loadreg => loadreg,
59                   shift => shift,
60                   clk => clk,
61                   bits_out => regC_to_regB,
62                   output => regC_out );
63
64    -- Product --
65    product((2*N) - 1 DOWNTO N) <= regC_out(N - 1 DOWNTO 0);
66    product(N - 1 DOWNTO 0) <= regB_out;
67 END Lab4_Shift_tst_B;
```

A.2.3.2: Shift Registers Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab4_Shift_tst_vhd_tst IS
32  END Lab4_Shift_tst_vhd_tst;
33  ARCHITECTURE Lab4_Shift_tst_arch OF Lab4_Shift_tst_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL clk : STD_LOGIC;
37  SIGNAL loadreg : STD_LOGIC;
38  SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
39  SIGNAL partial_product : STD_LOGIC_VECTOR(18 DOWNTO 0);
40  SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
41  SIGNAL regB_out : STD_LOGIC_VECTOR(17 DOWNTO 0);
42  SIGNAL regB_to_ctrl : STD_LOGIC_VECTOR(2 DOWNTO 0);
43  SIGNAL regC_out : STD_LOGIC_VECTOR(18 DOWNTO 0);
44  SIGNAL shift : STD_LOGIC;
45  COMPONENT Lab4_Shift_tst
46      PORT (
47      clk : IN STD_LOGIC;
48      loadreg : IN STD_LOGIC;
49      multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
50      partial_product : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
51      product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
52      regB_out : BUFFER STD_LOGIC_VECTOR(17 DOWNTO 0);
53      regB_to_ctrl : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
54      regC_out : BUFFER STD_LOGIC_VECTOR(18 DOWNTO 0);
55      shift : IN STD_LOGIC
56      );
57  END COMPONENT;
58  BEGIN
59      i1 : Lab4_Shift_tst
60      PORT MAP (
61  -- list connections between master ports and signals
62      clk => clk,
63      loadreg => loadreg,
64      multiplier => multiplier,
65      partial_product => partial_product,
66      product => product,
67      regB_out => regB_out,
68      regB_to_ctrl => regB_to_ctrl,
69      regC_out => regC_out,
70      shift => shift
71      );
72  init : PROCESS
73  -- variable declarations
```

```
74  BEGIN
75          -- Test with positive PP --
76          partial_product <= "0010101010101010101";
77           multiplier <=       "000000000000000001";
78            shift <= '0';
79            loadreg <= '1'; wait for 10 ps;
80            loadreg <= '0';
81
82            -- Shift for 9 cycles --
83            shift <= '1'; wait for 90 ps;
84          shift <= '0';
85
86            wait for 20 ps;
87
88            -- Test with negative PP --
89          partial_product <= "1101010101010101010";
90           multiplier <=       "000000000000000001";
91            shift <= '0';
92            loadreg <= '1'; wait for 10 ps;
93            loadreg <= '0';
94
95            -- Shift for 9 cycles --
96            shift <= '1'; wait for 90 ps;
97          shift <= '0';
98  WAIT;
99  END PROCESS init;
100 always : PROCESS
101 -- optional sensitivity list
102 -- (        )
103 -- variable declarations
104 BEGIN
105         -- code executes for every event on sensitivity list
106 WAIT;
107 END PROCESS always;
108 falling_clock : PROCESS
109 BEGIN
110     clk <= '1'; wait for 5 ps;
111     clk <= '0'; wait for 5 ps;
112 END PROCESS falling_clock;
113 END Lab4_Shift_tst_arch;
114
```

Appendix A.3: Counter Register D

A.3.1: Register D VHDL Code

```
1   -- Ryan Barker --
11
12  LIBRARY ieee;
13  USE ieee.numeric_std.all;
14  USE ieee.std_logic_1164.all;
15
16  -- Declare Register D --
17  ENTITY Lab4_RegD IS
18      GENERIC (N : INTEGER := 18);
19      PORT (maxval  : IN std_logic_vector(3 DOWNTO 0);
20            loadreg : IN std_logic;
21            count   : IN std_logic;
22            clk     : IN std_logic;
23            output  : OUT std_logic_vector(3 DOWNTO 0));
24  END Lab4_RegD;
25
26  -- Architecture of Register D --
27  ARCHITECTURE Lab4_RegD_B OF Lab4_RegD IS
28      SIGNAL counter : unsigned (3 DOWNTO 0);
29  BEGIN
30      cntr: PROCESS(clk, loadreg, count)
31      BEGIN
32          IF(falling_edge(clk) AND loadreg = '1') THEN
33              -- Initialize counter to zero --
34              load: FOR i IN 0 TO 3 LOOP
35                  counter(i) <= '0';
36              END LOOP;
37          END IF;
38
39          IF(falling_edge(clk) AND count = '1') THEN
40              IF(counter < unsigned(maxval)) THEN
41                  -- Increment counter --
42                  counter <= counter + 1;
43              ELSE
44                  -- Reset Counter --
45                  reset: FOR i IN 0 TO 3 LOOP
46                      counter(i) <= '0';
47                  END LOOP;
48              END IF;
49          END IF;
50
51          -- Set output to counter value --
52          output <= std_logic_vector(counter);
53      END PROCESS cntr;
54  END Lab4_RegD_B;
```

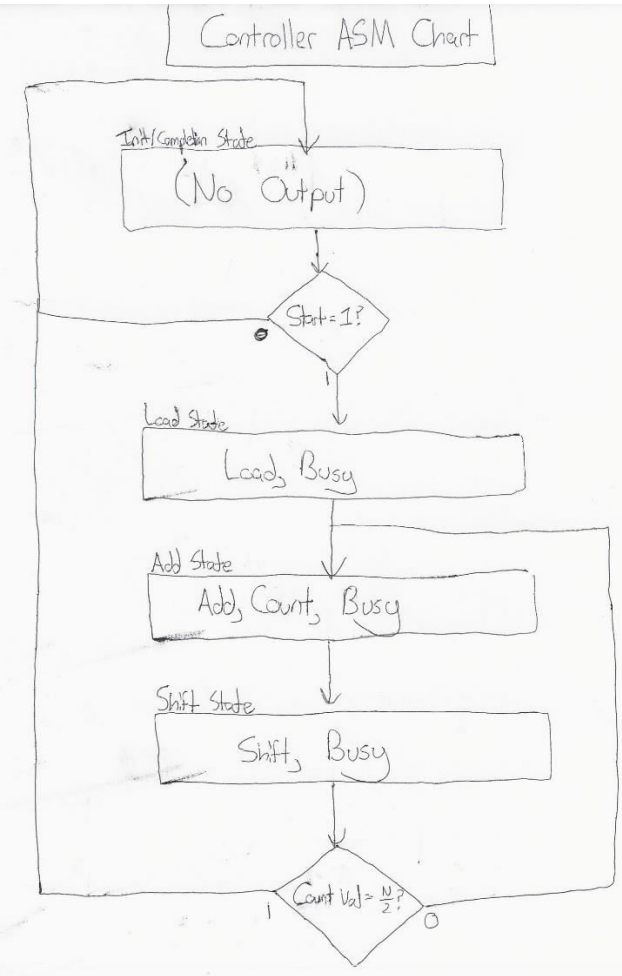
A.3.2: Register D Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab4_RegD_vhd_tst IS
32  END Lab4_RegD_vhd_tst;
33  ARCHITECTURE Lab4_RegD_arch OF Lab4_RegD_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL clk : STD_LOGIC;
37  SIGNAL count : STD_LOGIC;
38  SIGNAL loadreg : STD_LOGIC;
39  SIGNAL maxval : STD_LOGIC_VECTOR(3 DOWNTO 0);
40  SIGNAL output : STD_LOGIC_VECTOR(3 DOWNTO 0);
41  COMPONENT Lab4_RegD
42      PORT (
43      clk : IN STD_LOGIC;
44      count : IN STD_LOGIC;
45      loadreg : IN STD_LOGIC;
46      maxval : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
47      output : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
48      );
49  END COMPONENT;
50  BEGIN
51      i1 : Lab4_RegD
52      PORT MAP (
53  -- list connections between master ports and signals
54      clk => clk,
55      count => count,
56      loadreg => loadreg,
57      maxval => maxval,
58      output => output
59      );
60  init : PROCESS
61  -- variable declarations
62  BEGIN
63          -- Initialize Register D inputs --
64            maxval <= "1001";
65            count <= '0';
66          loadreg <= '1'; wait for 10 ps;
67            loadreg <= '0';
68            count <= '1'; wait for 90 ps;
69          maxval <= "0101";
70  WAIT;
71  END PROCESS init;
72  always : PROCESS
73  -- optional sensitivity list
74  -- (        )
75  -- variable declarations
76  BEGIN
77          -- code executes for every event on sensitivity list
78  WAIT;
```

```
79  END PROCESS always;
80  falling_clock : PROCESS
81  BEGIN
82      clk <= '1'; wait for 5 ps;
83      clk <= '0'; wait for 5 ps;
84  END PROCESS falling_clock;
85  END Lab4_RegD_arch;
86
```

Appendix A.4: FSM Controller and Remaining Components

A.4.1.1: Controller ASM Chart



A.4.1.2: Controller VHDL Code

```
1   -- Ryan Barker --
11
12  LIBRARY ieee;
13  USE ieee.math_real.all;
14  USE ieee.numeric_std.all;
15  USE ieee.std_logic_1164.all;
16
17  -- Declare controller (state machine) --
18  ENTITY Lab4_Ctrl IS
19      GENERIC (N : INTEGER := 18);
20      PORT (start      : IN std_logic;
21            regB_bits  : IN std_logic_vector(2 DOWNTO 0);
22            count_val  : IN std_logic_vector(3 DOWNTO 0);
23            clk        : IN std_logic;
24            mux_ctrl   : OUT std_logic_vector(2 DOWNTO 0);
25            loadreg    : OUT std_logic;
26            shiftreg   : OUT std_logic;
27            count      : OUT std_logic;
28            addreg     : OUT std_logic;
29            busy       : OUT std_logic);
30  END Lab4_Ctrl;
```

```
-- Architecture of controller (state machine) --
ARCHITECTURE Lab4_Ctrl_B OF Lab4_Ctrl IS
    TYPE fsm_state IS (A, B, C, D);
    SIGNAL state : fsm_state;
BEGIN
    -- Process for state machine --
    next_state: PROCESS (clk)
    BEGIN
        IF(rising_edge(clk)) THEN
            CASE state IS
                WHEN A =>
                    -- Pre-processing/Completion state --
                    IF(start = '1') THEN state <= B;
                    ELSE state <= A;
                    END IF;
                WHEN B =>
                    -- Load state --
                    state <= C;
                WHEN C =>
                    -- Add state --
                    state <= D;
                WHEN D =>
                    -- Shift state --
                    IF(count_val = std_logic_vector(to_unsigned((N / 2), 4))) THEN state <= .
                    ELSE state <= C;
                    END IF;
            END CASE;
        END IF;
    END PROCESS next_state;

    outputs: PROCESS (state)
    BEGIN
        CASE state IS
            WHEN A =>
                -- Pre-processing/Completion state outputs --
                busy <= '0';
                loadreg <= '0';
                shiftreg <= '0';
                addreg <= '0';
                count <= '0';
            WHEN B =>
                -- Load state outputs --
                busy <= '1';
                loadreg <= '1';
                shiftreg <= '0';
                addreg <= '0';
                count <= '0';
            WHEN C =>
                -- Add state outputs --
                busy <= '1';
                loadreg <= '0';
```

```
                shiftreg <= '0';
                addreg <= '1';
                count <= '1';
            WHEN D =>
                -- Shift state outputs --
                busy <= '1';
                loadreg <= '0';
                shiftreg <= '1';
                addreg <= '0';
                count <= '0';
        END CASE;
    END PROCESS outputs;

    -- Connect reg B bits to multiplexer --
    mux_ctrl <= regB_bits;
END Lab4_Ctrl_B;
```

A.4.2.1: Controller and Counter Wrapper VHDL Code

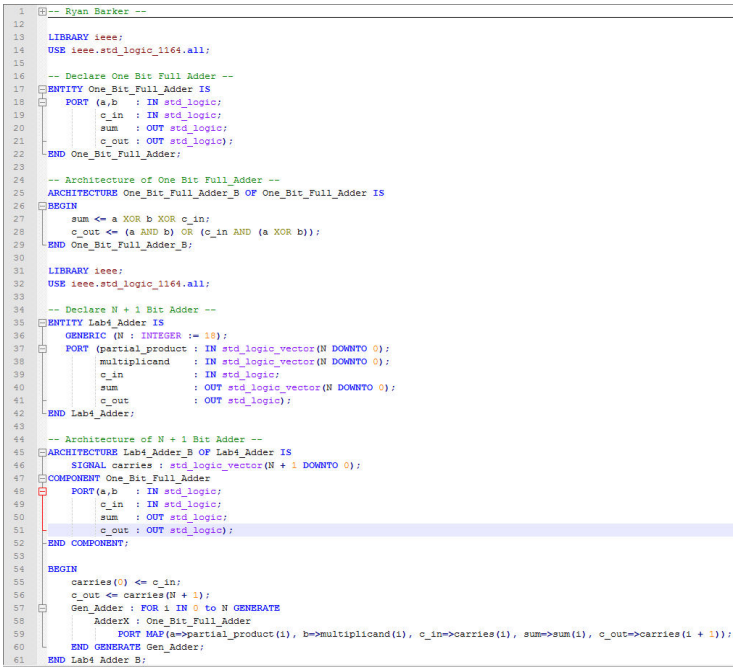
```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Controller Test --
ENTITY Lab4_Ctrl_tst IS
    GENERIC ( N : INTEGER := 18);
    PORT (start     : IN std_logic;
          clk       : IN std_logic;
          mux_ctrl  : OUT std_logic_vector(2 DOWNTO 0);
          loadreg   : BUFFER std_logic;
          shiftreg  : OUT std_logic;
          count     : BUFFER std_logic;
          addreg    : OUT std_logic;
          busy      : OUT std_logic);
END Lab4_Ctrl_tst;

-- Architecture of Controller Test --
ARCHITECTURE Lab4_Ctrl_tst_B OF Lab4_Ctrl_tst IS
    -- Signal for Counter --
     SIGNAL regD_out      : std_logic_vector(3 DOWNTO 0);

     -- Map in CTRL and Reg D:
    COMPONENT Lab4_Ctrl
        PORT (start     : IN std_logic;
              regB_bits : IN std_logic_vector(2 DOWNTO 0);
                count_val : IN std_logic_vector(3 DOWNTO 0);
                clk       : IN std_logic;
                mux_ctrl  : OUT std_logic_vector(2 DOWNTO 0);
                loadreg   : OUT std_logic;
                shiftreg  : OUT std_logic;
               count     : OUT std_logic;
                addreg    : OUT std_logic;
                busy      : OUT std_logic);
        END COMPONENT;

    COMPONENT Lab4_RegD
        PORT (maxval  : IN std_logic_vector(3 DOWNTO 0);
              loadreg : IN std_logic;
                count   : IN std_logic;
                clk     : IN std_logic;
                output  : OUT std_logic_vector(3 DOWNTO 0));
        END COMPONENT;
BEGIN
    Ctrl: Lab4_Ctrl
        PORT MAP ( start => start,
                   regB_bits => "111",
                       count_val => regD_out,
                       clk => clk,
                       mux_ctrl => mux_ctrl,
                       loadreg => loadreg,
                       shiftreg => shiftreg,
                       count => count,
                       addreg => addreg,
                       busy => busy );

    RegD: Lab4_RegD
         PORT MAP ( maxval => "1001",
                    loadreg => loadreg,
                       count => count,
                       clk => clk,
                       output => RegD_out );
END Lab4_Ctrl_tst_B;
```

A.4.2.2: Controller and Counter Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_Ctrl_tst_vhd_tst IS
END Lab4_Ctrl_tst_vhd_tst;
ARCHITECTURE Lab4_Ctrl_tst_arch OF Lab4_Ctrl_tst_vhd_tst
-- constants
-- signals
SIGNAL addreg : STD_LOGIC;
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL count : STD_LOGIC;
SIGNAL loadreg : STD_LOGIC;
SIGNAL mux_ctrl : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL shiftreg : STD_LOGIC;
SIGNAL start : STD_LOGIC;
COMPONENT Lab4_Ctrl_tst
    PORT (
    addreg : OUT STD_LOGIC;
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    count : BUFFER STD_LOGIC;
    loadreg : BUFFER STD_LOGIC;
    mux_ctrl : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
    shiftreg : OUT STD_LOGIC;
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4_Ctrl_tst
    PORT MAP (
-- list connections between master ports and signals
    addreg => addreg,
    busy => busy,
    clk => clk,
    count => count,
    loadreg => loadreg,
    mux_ctrl => mux_ctrl,
    shiftreg => shiftreg,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
```

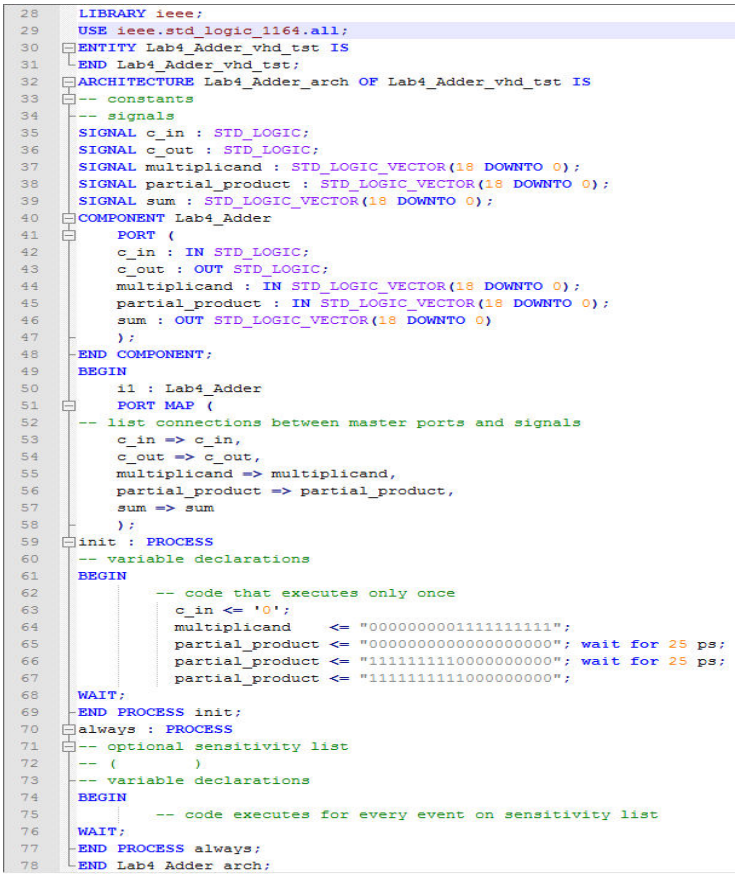
```
BEGIN
        -- Pulse start signal to observe required behavior --
          start <= '0'; wait for 10 ps;
          start <= '1'; wait for 500 ps;
          start <= '0';
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_Ctrl_tst_arch;
```

A.4.3: N + 1 Bit Adder VHDL Code



```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare One Bit Full Adder --
ENTITY One_Bit_Full_Adder IS
    PORT (a,b   : IN std_logic;
          c_in  : IN std_logic;
          sum   : OUT std_logic;
          c_out : OUT std_logic);
END One_Bit_Full_Adder;

-- Architecture of One Bit Full_Adder --
ARCHITECTURE One_Bit_Full_Adder_B OF One_Bit_Full_Adder IS
BEGIN
    sum <= a XOR b XOR c_in;
    c_out <= (a AND b) OR (c_in AND (a XOR b));
END One_Bit_Full_Adder_B;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare N + 1 Bit Adder --
ENTITY Lab4_Adder IS
    GENERIC (N : INTEGER := 18);
    PORT (partial_product : IN std_logic_vector(N DOWNTO 0);
          multiplicand    : IN std_logic_vector(N DOWNTO 0);
          c_in            : IN std_logic;
          sum             : OUT std_logic_vector(N DOWNTO 0);
          c_out           : OUT std_logic);
END Lab4_Adder;

-- Architecture of N + 1 Bit Adder --
ARCHITECTURE Lab4_Adder_B OF Lab4_Adder IS
    SIGNAL carries : std_logic_vector(N + 1 DOWNTO 0);
COMPONENT One_Bit_Full_Adder
    PORT(a,b   : IN std_logic;
         c_in  : IN std_logic;
         sum   : OUT std_logic;
         c_out : OUT std_logic);
END COMPONENT;

BEGIN
    carries(0) <= c_in;
    c_out <= carries(N + 1);
    Gen_Adder : FOR i IN 0 to N GENERATE
        AdderX : One_Bit_Full_Adder
            PORT MAP(a=>partial_product(i), b=>multiplicand(i), c_in=>carries(i), sum=>sum(i), c_out=>carries(i + 1));
    END GENERATE Gen_Adder;
END Lab4 Adder B;
```

A.4.4: N + 1 Bit Adder Test Bench



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY Lab4_Adder_vhd_tst IS
END Lab4_Adder_vhd_tst;
ARCHITECTURE Lab4_Adder_arch OF Lab4_Adder_vhd_tst IS
-- constants
-- signals
SIGNAL c_in : STD_LOGIC;
SIGNAL c_out : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(18 DOWNTO 0);
SIGNAL partial_product : STD_LOGIC_VECTOR(18 DOWNTO 0);
SIGNAL sum : STD_LOGIC_VECTOR(18 DOWNTO 0);
COMPONENT Lab4_Adder
    PORT (
    c_in : IN STD_LOGIC;
    c_out : OUT STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
    partial_product : IN STD_LOGIC_VECTOR(18 DOWNTO 0);
    sum : OUT STD_LOGIC_VECTOR(18 DOWNTO 0)
    );
END COMPONENT;
BEGIN
    i1 : Lab4_Adder
    PORT MAP (
-- list connections between master ports and signals
    c_in => c_in,
    c_out => c_out,
    multiplicand => multiplicand,
    partial_product => partial_product,
    sum => sum
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
        c_in <= '0';
        multiplicand    <= "0000000001111111111";
        partial_product <= "0000000000000000000"; wait for 25 ps;
        partial_product <= "1111111110000000000"; wait for 25 ps;
        partial_product <= "1111111111000000000";
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
END Lab4_Adder_arch;
```

A.4.5: 2-to-1 Multiplexer VHDL Code

```
-- Ryan Barker --
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare adder multiplexer --
ENTITY Lab4_Adder_Mux IS
    GENERIC (N : INTEGER := 18);
    PORT (partial_product : IN std_logic_vector(N DOWNTO 0);
          loadreg         : IN std_logic;
          output          : OUT std_logic_vector(N DOWNTO 0));
END Lab4_Adder_Mux;

-- Architecture of adder multiplexer --
ARCHITECTURE Lab4_Adder_Mux_B OF Lab4_Adder_Mux IS
BEGIN
    multiplex: PROCESS (partial_product, loadreg)
    BEGIN
        CASE loadreg IS
            WHEN '0' =>
                output <= partial_product;
            WHEN '1' =>
                zeroes: FOR i IN 0 TO N LOOP
                    output(i) <= '0';
                END LOOP;
            WHEN OTHERS =>
                -- Impossible --
                error: FOR i IN 0 TO N LOOP
                    output(i) <= '0';
                END LOOP;
        END CASE;
    END PROCESS multiplex;
END Lab4_Adder_Mux_B;
```

A.4.6: 2-to-1 Multiplexer Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_Adder_Mux_vhd_tst IS
END Lab4_Adder_Mux_vhd_tst;
ARCHITECTURE Lab4_Adder_Mux_arch OF Lab4_Adder_Mux_vhd_tst IS
-- constants
-- signals
SIGNAL loadreg : STD_LOGIC;
SIGNAL output : STD_LOGIC_VECTOR(18 DOWNTO 0);
SIGNAL partial_product : STD_LOGIC_VECTOR(18 DOWNTO 0);
COMPONENT Lab4_Adder_Mux
    PORT (
    loadreg : IN STD_LOGIC;
    output : OUT STD_LOGIC_VECTOR(18 DOWNTO 0);
    partial_product : IN STD_LOGIC_VECTOR(18 DOWNTO 0)
    );
END COMPONENT;
BEGIN
    i1 : Lab4_Adder_Mux
    PORT MAP (
-- list connections between master ports and signals
    loadreg => loadreg,
    output => output,
    partial_product => partial_product
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
        partial_product <= "1111111111111111111";
        loadreg <= '1'; wait for 25 ps;
        loadreg <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
END Lab4_Adder_Mux_arch;

```

Appendix A.5: Overall Circuit

A.5.1: Bit Pair Multiplier VHDL Code

```
1   -- Ryan Barker --
8
9   LIBRARY ieee;
10  USE ieee.math_real.all;
11  USE ieee.numeric_std.all;
12  USE ieee.std_logic_1164.all;
13
14  -- Declare Bit Pair Multiplier --
15  ENTITY Lab4 IS
16      GENERIC (N : INTEGER := 18);
17      PORT (start        : IN std_logic;
18            multiplicand : IN std_logic_vector(N - 1 DOWNTO 0);
19            multiplier   : IN std_logic_vector(N - 1 DOWNTO 0);
20            clk          : IN std_logic;
21            busy         : OUT std_logic;
22            product      : OUT std_logic_vector((2 * N) - 1 DOWNTO 0));
23  END Lab4;
24
25  -- Architecture of Bit Pair Multiplier --
26  ARCHITECTURE Lab4_B OF Lab4 IS
27      -- Signals for Registers and Adder --
28      SIGNAL regA_out      : std_logic_vector(N DOWNTO 0);
29      SIGNAL regA_mux_out  : std_logic_vector(N DOWNTO 0);
30      SIGNAL adder_out     : std_logic_vector(N DOWNTO 0);
31      SIGNAL adder_mux_out : std_logic_vector(N DOWNTO 0);
32      SIGNAL regC_out      : std_logic_vector(N DOWNTO 0);
33      SIGNAL regC_to_regB  : std_logic_vector(1 DOWNTO 0);
34      SIGNAL regB_to_ctrl  : std_logic_vector(2 DOWNTO 0);
35      SIGNAL regB_out      : std_logic_vector(N - 1 DOWNTO 0);
36      SIGNAL regD_out      : std_logic_vector(3 DOWNTO 0);
37
38      -- Signals from Controller --
39      SIGNAL loadreg       : std_logic;
40      SIGNAL addreg        : std_logic;
41      SIGNAL load_or_add   : std_logic;
42      SIGNAL shift         : std_logic;
43      SIGNAL count         : std_logic;
44      SIGNAL regA_mux_ctrl : std_logic_vector(2 DOWNTO 0);
45
46      -- Begin Adding Everything:
47      COMPONENT Lab4_Ctrl
59
60      COMPONENT Lab4_RegA
66
67      COMPONENT Lab4_RegB
76
77      COMPONENT Lab4_RegC
85
86      COMPONENT Lab4_RegD
93
94      COMPONENT Lab4_RegA_Mux
```

```
99
100     COMPONENT Lab4_Adder_Mux
105
106     COMPONENT Lab4_Adder
113   BEGIN
114       load_or_add <= loadreg OR addreg;
115
116       Ctrl: Lab4_Ctrl
117           PORT MAP ( start => start,
118                      regB_bits => regB_to_ctrl,
119                          count_val => regD_out,
120                          clk => clk,
121                          mux_ctrl => regA_mux_ctrl,
122                          loadreg => loadreg,
123                          shiftreg => shift,
124                          count => count,
125                          addreg => addreg,
126                          busy => busy );
127
128        RegA: Lab4_RegA
129            PORT MAP ( multiplicand => multiplicand,
130                       loadreg => loadreg,
131                           clk => clk,
132                           output => regA_out );
133
134       RegB: Lab4_RegB
135            PORT MAP ( multiplier => multiplier,
136                       regC_bits => regC_to_regB,
137                           loadreg => loadreg,
138                           shift => shift,
139                           clk => clk,
140                           bits_out => regB_to_ctrl,
141                           output => regB_out );
142
143        RegC: Lab4_RegC
144            PORT MAP ( partial_product => adder_mux_out,
145                           loadreg => load_or_add,
146                           shift => shift,
147                           clk => clk,
148                           bits_out => regC_to_regB,
149                           output => regC_out );
150
151       RegD: Lab4_RegD
152            PORT MAP ( maxval => std_logic_vector(to_unsigned((N / 2), 4)),
153                       loadreg => loadreg,
154                           count => count,
155                           clk => clk,
156                           output => RegD_out );
157
```

```
158        RegA_Mux: Lab4_RegA_Mux
159            PORT MAP ( regA_out => regA_out,
160                       mux_ctrl => regA_mux_ctrl,
161                           output => regA_mux_out );
162
163        Adder_Mux: Lab4_Adder_Mux
164            PORT MAP ( partial_product => adder_out,
165                       loadreg => loadreg,
166                           output => adder_mux_out );
167
168        Adder: Lab4_Adder
169            PORT MAP ( partial_product => regC_out,
170                         multiplicand => regA_mux_out,
171                           c_in => '0',
172                           sum => adder_out );
173
174       -- Product --
175       product((2*N) - 1 DOWNTO N) <= regC_out(N - 1 DOWNTO 0);
176       product(N - 1 DOWNTO 0) <= regB_out;
177   END Lab4_B;
```

A.5.2.1: Multiplier Test Bench 1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_vhd_tst IS
END Lab4_vhd_tst;
ARCHITECTURE Lab4_arch OF Lab4_vhd_tst IS
-- constants
-- signals
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
SIGNAL start : STD_LOGIC;
COMPONENT Lab4
    PORT (
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4
    PORT MAP (
-- list connections between master ports and signals
    busy => busy,
    clk => clk,
    multiplicand => multiplicand,
    multiplier => multiplier,
    product => product,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test a positive times a positive --
        start <= '0';
        multiplicand <= "0101010101010101";
        multiplier   <=    "0000000000000001"; wait for 10 ps;
         start <= '1'; wait for 10 ps;
         start <= '0';

         wait for 300 ps;

         -- Try opposite order --
        multiplicand <= "0000000000000001";
        multiplier   <=    "0101010101010101"; wait for 10 ps;
         start <= '1'; wait for 10 ps;
        start <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_arch;

```

A.5.2.2: Multiplier Test Bench 2

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_2_vhd_tst IS
END Lab4_2_vhd_tst;
ARCHITECTURE Lab4_arch2 OF Lab4_2_vhd_tst IS
-- constants
-- signals
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
SIGNAL start : STD_LOGIC;
COMPONENT Lab4
    PORT (
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4
    PORT MAP (
-- list connections between master ports and signals
    busy => busy,
    clk => clk,
    multiplicand => multiplicand,
    multiplier => multiplier,
    product => product,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test a negative times a positive --
        start <= '0';
         multiplicand <= "1010101010101010";
         multiplier   <=    "0000000000000001"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';

          wait for 300 ps;

          -- Try opposite order --
         multiplicand <= "0000000000000001";
         multiplier   <=    "1010101010101010"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_arch2;

```

A.5.2.3: Multiplier Test Bench 3

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_3_vhd_tst IS
END Lab4_3_vhd_tst;
ARCHITECTURE Lab4_arch3 OF Lab4_3_vhd_tst IS
-- constants
-- signals
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
SIGNAL start : STD_LOGIC;
COMPONENT Lab4
    PORT (
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4
    PORT MAP (
-- list connections between master ports and signals
    busy => busy,
    clk => clk,
    multiplicand => multiplicand,
    multiplier => multiplier,
    product => product,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test a negative times a negative --
        start <= '0';
         multiplicand <= "1010101010101010";
         multiplier   <=    "1111111111111111"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';

          wait for 300 ps;

          -- Try opposite order --
         multiplicand <= "1111111111111111";
         multiplier   <=    "1010101010101010"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_arch3;

```

A.5.2.4: Multiplier Test Bench 4

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_4_vhd_tst IS
END Lab4_4_vhd_tst;
ARCHITECTURE Lab4_arch4 OF Lab4_4_vhd_tst IS
-- constants
-- signals
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
SIGNAL start : STD_LOGIC;
COMPONENT Lab4
    PORT (
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4
    PORT MAP (
-- list connections between master ports and signals
    busy => busy,
    clk => clk,
    multiplicand => multiplicand,
    multiplier => multiplier,
    product => product,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test positive number lower bound
        start <= '0';
         multiplicand <= "00000000000000000";
         multiplier   <=    "0101010101010101"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';

          wait for 300 ps;

          -- Test positive number upper bound --
         multiplicand <= "01111111111111111";
         multiplier   <=    "01111111111111111"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_arch4;

```

A.5.2.5: Multiplier Test Bench 5

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab4_5_vhd_tst IS
END Lab4_5_vhd_tst;
ARCHITECTURE Lab4_arch5 OF Lab4_5_vhd_tst IS
-- constants
-- signals
SIGNAL busy : STD_LOGIC;
SIGNAL clk : STD_LOGIC;
SIGNAL multiplicand : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL multiplier : STD_LOGIC_VECTOR(17 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(35 DOWNTO 0);
SIGNAL start : STD_LOGIC;
COMPONENT Lab4
    PORT (
    busy : OUT STD_LOGIC;
    clk : IN STD_LOGIC;
    multiplicand : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    multiplier : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(35 DOWNTO 0);
    start : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab4
    PORT MAP (
-- list connections between master ports and signals
    busy => busy,
    clk => clk,
    multiplicand => multiplicand,
    multiplier => multiplier,
    product => product,
    start => start
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test negative number lower bound --
        start <= '0';
         multiplicand <= "1111111111111111";
         multiplier   <=    "0000000000000001"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';

          wait for 300 ps;

          -- Test negative number upper bound --
         multiplicand <= "1000000000000000";
         multiplier   <=    "1000000000000000"; wait for 10 ps;
          start <= '1'; wait for 10 ps;
          start <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    clk <= '0'; wait for 5 ps;
    clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab4_arch5;

```

Appendix A.C: The Fully Expanded Circuit in the RTL Viewer