CMPE 160 Laboratory Exercise 3

Arithmetic Logic Unit

Brian Landy
Performed March 25th, 2017
Submitted March 25th, 2017

Lab Section L2
Instructor: Tejaswini Ananthanarayana
TA: Barry Wu

Lecture Section 2
Professor: Richard Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Your Signature: _____

Table of Contents

Abstract

This exercise established understanding in designing and testing an arithmetic logic unit that is scalable with generics. The arithmetic logic unit has many functions which include N bit addition, subtraction, bitwise logic, logical left and right shifts, arithmetic right shifts, and N/2 bit multiplication. The objective was to design the modules and test benches for this logic unit and simulate the results for testing. This ALU was then programmed to an FPGA and tested on hardware. The largest concepts developed in this lab were working with generics to create a scalable ALU and understand that exhaustive testing is important and should be performed. ALUs are an important hardware component that allow for mathematical and logical operations to be performed without and additional software programming. This reduces code size and simplifies programming in many cases. The objectives of this lab were accomplished by building each internal component and then testing each component separately. After these tests provided correct results, a larger component was built with smaller components and was tested and results were observed for correctness. The results were correct and simulation demonstrated proper and ideal functionality of this ALU. The lab was a success.

Design Methodology

An ALU is an arithmetic logic unit which is comprised of 4 smaller components and a select. All 4 operations are performed and the control bet selects the ideal output for the operation. There are 3 inputs, Input 1, Input 2 and Control. There is only one output. With these small design points, an over view of the top level of the ALU was constructed. Figure 1 is the top level block diagram for the ALU.

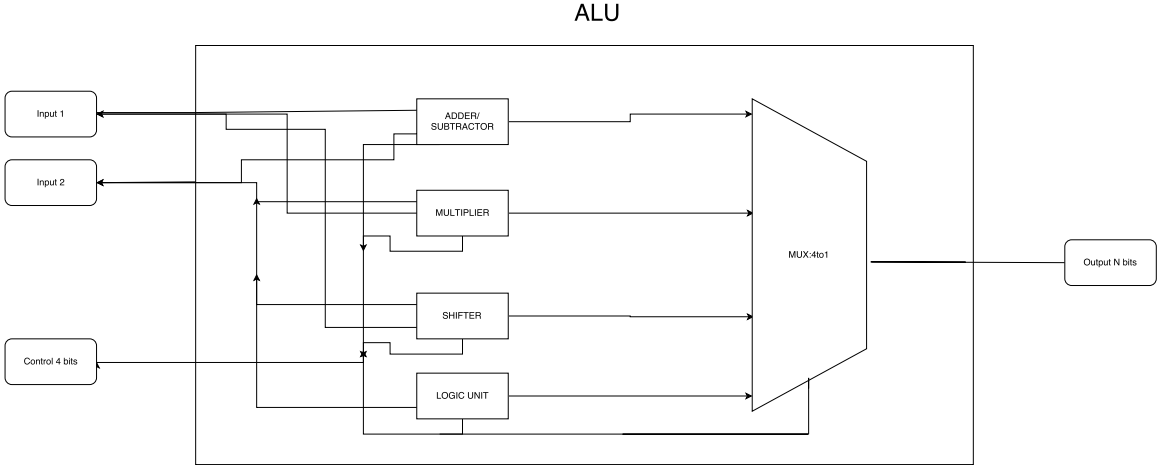

Figure 1 – ALU top level block diagram

In figure 1, all three input bits flow to each component, and there is a mux to select the ideal output for the operation performed. This design is also meant to be scalable. So, each input and output had to be n bits wide except for control. All operations had to take size of a vector into consideration. The control bit acted as a select in the end, but was also used to determine functionality of each component as well. Table 1 demonstrates the control inputs required for a specific process to be performed.

Table 1 – ALU process select with control bit

OPERATION	CONTROL
Add	0100
Subtract	0101
Multiply	0110
OR	1000
NOT	1001
AND	1010
XOR	1011
Left Logical Shift	1100
Right Logical Shift	1101
Right Arithmetic Shift	1110

To create the ALU, each component was created individually and the ALU was assembled using structural architecture in VHDL. The code for this can be found in the appendix. The ripple carry adder and subtracter is one unit that is a series of full adders that are one bit in size. This performs both addition and subtraction by using XOR gates on one input. Each input bit is placed in an XOR with the Carry in, allowing it to behave like a subtracter circuit as well. Each one-bit full adder's carry out flows into the next full adders carry in and the total n bit sum is recorded as the sum out of each nth place adder. The carry out is discarded. The last bit of the control input determined addition or subtraction. The ripple carry source code can be found in the appendix.

To construct the N bit shifter, a dataflow approach was taken. There are three shift operations that can take place. These are regulated by the last two bits of the bit control. If a 00 is detected, a logical left shift will be performed a number of times specified by the input 2 value on input 1. The values of shifts range from 0 to the ceiling of the log base 2 of the 2nd input -1. Thus for a 16-bit value, shifting can happen anywhere from 0 to 15 times. If control is a 01, a logical right shift will be performed. For logical shifts, a certain amount of Input 1 will be kept and moved over Input2 number of times. Then a complimentary portion of a zeroes vector will be appended to either side depending on the direction of the shift. When control is equal to 1110, an arithmetic right shift will be performed. An arithmetic right shift is a shift that preserves the sign bit. The operation is the same as the logical right shift when the first bit is a 0. When the first bit is a 1, the vector appended to the MSB side will be a complimentary portion of the all 1's vector. This way, the sign is preserved. If control is equal to 1111, the output is the original value of Input1 and no shift is performed. The main shift code can be found in the appendix.

For the bitwise logic block, there are 2 N bit inputs and a control which will select one of 4 logical operators. When control is 1000, output is Input1 or Input2. When control is 1001, output is the inversion of Input1 and Input 2 is ignored entirely. When control selects the XOR and the AND, output is that function performed on Input1 and Input2, much like the OR selection. This was created using a dataflow architecture. The code for this logic block can be found in the appendix.

For the multiplier, the output is N bits wide and the inputs are N/2 bits wide. This is because when 2 binary numbers are multiplied, the maximum number of places of the output is double the 2 numbers multiplied. For example, 2 4 bit binary numbers will multiply and create 1 8-bit output. This multiplier is comprised of and gates and full

adders. For a 4 by 4 multiplication, there is a 2D array of and gates that is a 4 by 4 array. There is a 3 by 4 array of full adders, as the first row is not considered because there are only AND gates and no full adders. Figure 2 is a 4 by 4 multiplier circuit. This design was followed for the n bit multiplier in the ALU.

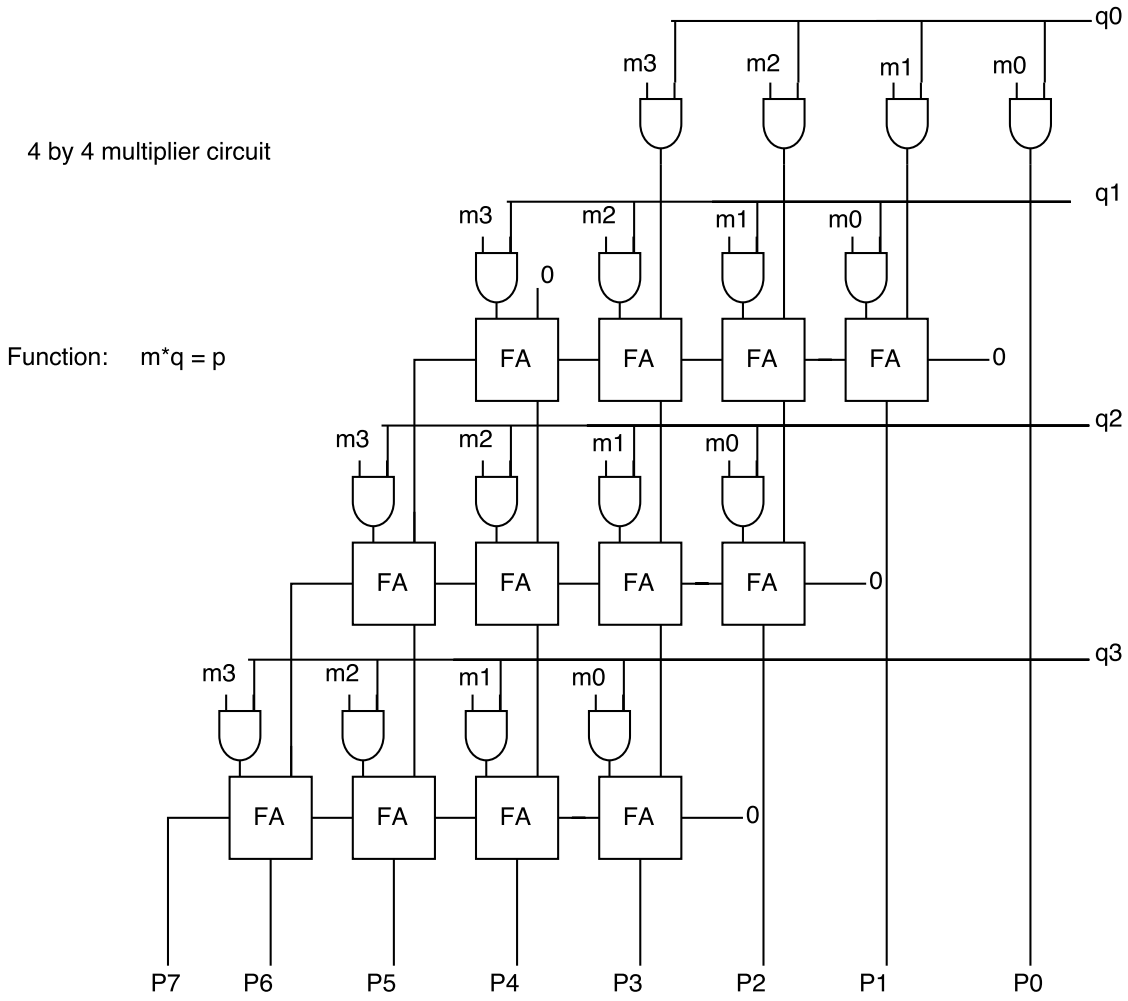

Figure 2 – Multiplier circuit

The multiplier in figure 2 is composed structurally of one bit full adders and AND gates. The multiplier in figure 2 is a 4 by 4 multiplier circuit that results in an 8 bit output. The multiplier was create by using generate statements that ran in nested loops. These loops spanned a width of 4 and depth of 4 for the adder circuit and a width of 4 and depth of 3 for the full adder circuit. The numbers mentioned are defined in terms of a generic number N. In this case, N is assumed to be four. There will always be one less row in the full adder arrays of signals. There are three 2D arrays in total. This can be seen in the code for the multiplier in the appendix. The multiplier code located in the appendix.

A multiplexer was used in this exercise, but was not made structurally. The multiplexer was a series of conditional statements that behave like a multiplexer but allow for more cases than a multiplexer. This is because an ideal 4to1 mux can not be

used. The first 2 bits of control for addition, subtraction and multiplication are all 01. This means that a multiplexer receiving the first 2 bits of control would send out the wrong information because the same control bit references 2 components of the ALU. A conditional structure of if/else was set up and compared the first three values of the control bit to distinguish a difference between multiplication and addition. The multiplexer had 4 inputs, and they were the n bit wide internal signals output by each component. The output of this mux was the output of the ALU. The code for this component can be found in the code for the ALU in the appendix.

Results

Many tests were performed to verify functionality of the ALU. An exhaustive test bench was constructed for the multiplier that tested every multiplication from 00 by 00 to FF by FF. This code can be found in the appendix. A text file was created and populated with VHDL text IO. This created a file for 8 bit by 8-bit multiplication that was around 65,600 lines long. That was done in the first process in the multiplier test bench. A second process in the multiplier test bench iterated through each line and compared the expected product out with the product out of the multiplier circuit. This test is a good way to test but there needs to be a test for the test. So 2 specific multiplications in the text file were made incorrect so the test bench would catch the 2 values that didn't match up. This worked as expected with assert and report statements. The testbench simulation would output error to the console window if the results did not match. Figure 3 shows the simulation results of the multiplier for a small sample range of values to show correctness. Figure 3 can be found in the appendix.

To make sure the test bench was working correctly, 2 lines were changed. The simulation of this new text file caught 2 expected errors. These reported errors are seen in figure 4 which is the console window after simulation.

Figure 4 – incorrect multiplier results

After the multiplier was determined to be correct, overall ALU functionality was tested for a 16 bit set of operations. The ALU test bench tested operations in order, starting with addition, followed by subtraction, multiplication, logic, then shifts. Figure 5 is a small sample range of tests that show addition is functional for the ALU. Figure 5 can be found in the appendix.

Subtraction testing worked similar to addition testing. Figure 6 is the waveform for the ALU simulation for subtraction. It can be found in the appendix. It shows 16-bit subtraction.

A small sample of multiplication results were captured from the ALU test bench as well. Figure 7 is the ALU multiplication results captured from the simulation of the ALU. Figure 7 is in the appendix.

Testing the bitwise logic component was completed with a series of non trivial tests. Figure 8 is a series of tests that test the entire logic unit. These tests include OR, NOT, AND and XOR, in that order. This is figure 8 in the appendix.

The shifter was tested by looping through the range of the shift amount signal. This allowed for observation of results of all shifts from size 0 to 15 for a 16-bit number. Figure 9, found in the appendix is the left logical shift results for the ALU. Figure 10, found in the appendix is the correct simulation results for the logical shift right. Figure 11, found in the appendix demonstrates functional correctness of the arithmetic right shift when the sign bit is positive. Figure 12, found in the appendix, demonstrates the correct results of an arithmetic right shift for a negative number. The code for the ALU testbench can be found in the appendix.

All tests performed were correct and to further show correct implementation, an RTL schematic was synthesized. Figure 13 is the RTL schematic produced for the top level block diagram for the ALU.

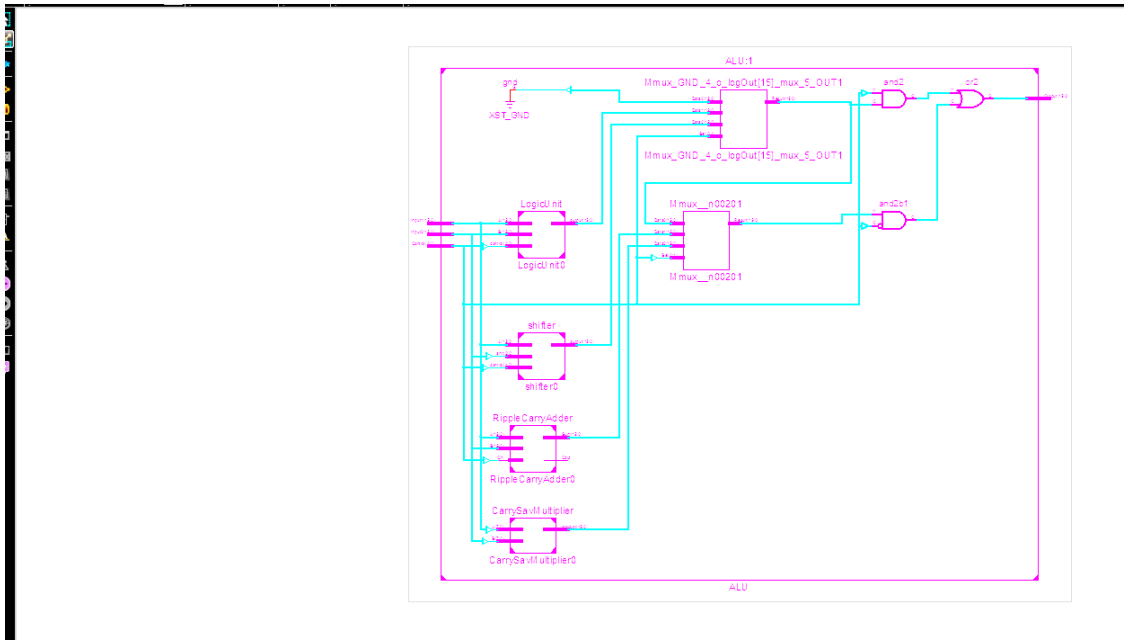

Figure 13 – Top Level ALU schematic

Figure 13 demonstrates the proper implementation of all sub components into the ALU. In figure 13, the extra logic and doubled muxes are due to the way the if else conditions were written.

Timing results were gathered by adding d flip-flops onto the front and back of the design. The timing results for this circuit are shown in figure 14.

```
---------------------------------------------------------------------------+-----------------------------+-------+
(*) This 1 clock signal(s) are generated by combinatorial logic,
and XST is not able to identify which are the primary clock signals.
Please use the CLOCK_SIGNAL constraint to specify the clock signal(s) generated by combinatorial logic.
INFO:Xst:2169 - HDL ADVISOR - Some clock signals were not automatically buffered by XST with BUFG/BUFR resources. Please use the buffer_type constraint in order to

Asynchronous Control Signals Information:
----------------------------------------
No asynchronous control signals found in this design

Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: 7.218ns
   Maximum output required time after clock: 2.170ns
   Maximum combinational path delay: 17.723ns

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'shifter0/GND_9_o_GND_9_o_OR_28_o'
  Total number of paths / destination ports: 919 / 16
-------------------------------------------------------------------------
Offset:              7.218ns (Levels of Logic = 5)
  Source:            dflip1:Q<1> (PAD)
  Destination:       shifter0/outpoop_1 (LATCH)
  Destination Clock: shifter0/GND_9_o_GND_9_o_OR_28_o falling

  Data Path: dflip1:Q<1> to shifter0/outpoop_1
                                Gate     Net
```

Figure 14 – Timing results of the ALU

The minimum input time is 7.218 nanoseconds. The maximum output required time is 2.17nanoseconds. Most importantly the maximum combinational path delay is 17.723 nanoseconds.

To verify that the vhdl code would work properly when it was burnt to the fpga, a place and route simulation was simulated. A place and route simulation will simulate actual propagation delays that are unavoidable in any logic gate or component. Figure 15 is a waveform simulation that highlights exactly why a place and route is run. All tests were identical in output when compared with the behavioral simulation. The place and route has a propagation delay. This simulation can be found in the appendix in figure 15. At first, output is not shown because of propagation delay. The point in time in which the output starts to be displayed is 17.723 nanoseconds after input has been entered. This further shows that the maximum path delay is 17.723 nanoseconds.

This design uses a total of 15 Flip Flop and Lut pairs and a total of 263 Luts used. This is demonstrated by figure 16 which shows the slice utilization.

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	16	18,224	1%	
Number used as Flip Flops	0			
Number used as Latches	16			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	263	9,112	2%	
Number used as logic	263	9,112	2%	
Number using O6 output only	238			
Number using O5 output only	0			
Number using O5 and O6	25			
Number used as ROM	0			
Number used as Memory	0	2,176	0%	
Number of occupied Slices	104	2,278	4%	
Number of MUXCYs used	0	4,556	0%	
Number of LUT Flip Flop pairs used	264			
Number with an unused Flip Flop	248	264	93%	
Number with an unused LUT	1	264	1%	
Number of fully used LUT-FF pairs	15	264	5%	
Number of unique control sets	1			
Number of slice register sites lost to control set restrictions	0	18,224	0%	
Number of bonded IOBs	52	232	22%	
Number of LOCed IOBs	16	52	30%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	0	64	0%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	0	16	0%	
Number of DCM/DCM_CLKGENs	0	4	0%	

ALU_tb.vhd | CarrySavMultiplier.vhd | LogicUnit.vhd | RippleCarryAdder.vhd | multiplier_tb.vhd | Design Summary

Figure 16 – slice utilization

This design simulated successfully in all ways, and was run on hardware to further show the correct implementation of the design. All results returned by the circuit were correct. This exercise was successful.

Conclusions

All expected results were output by the circuit designed for exercise three. The ALU has been designed properly and is functionally correct. This exercise was a success as the circuit required for completion is correct. This exercise promoted an understanding of arithmetic logic units and scalable design in vhdl using generics and generates. All tests and integrations of components were successful including the exhaustive testing of the multiplier circuit. This lab was successful.

Appendix

Waveforms

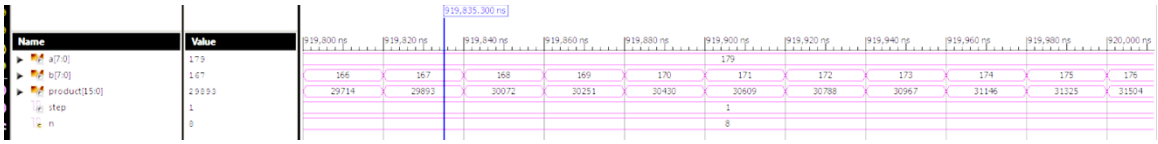

Figure 3 – multiplier results

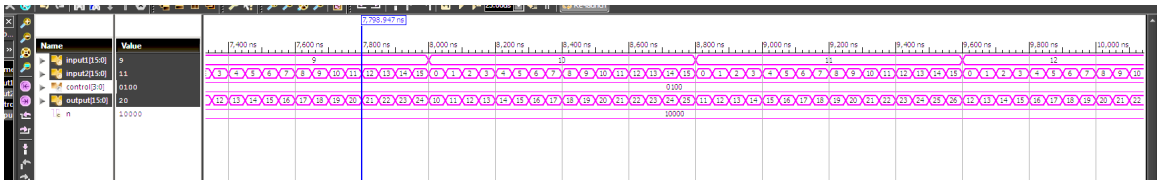

Figure 5 – correct 16 bit ALU addition

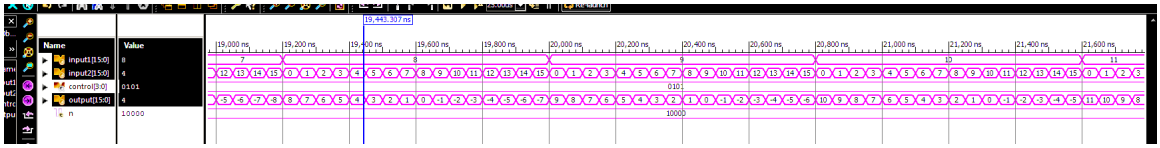

Figure 6 – correct 16 bit ALU Subtraction

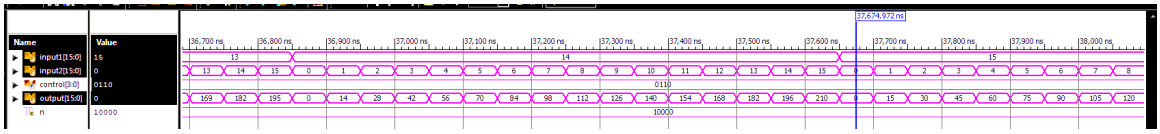

Figure 7 – correct 16 bit ALU Multiplication

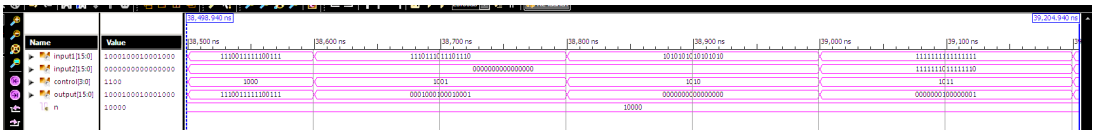

Figure 8 – correct 16 bit ALU Bitwise logic tests

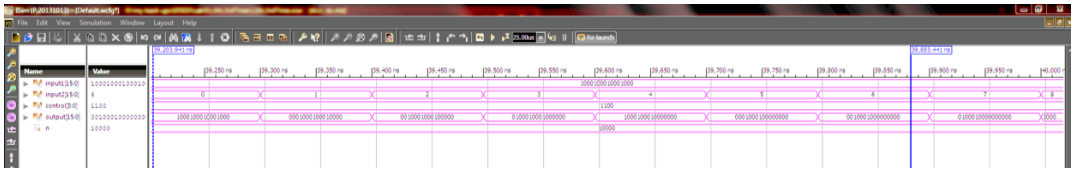

Figure 9 – correct 16 bit ALU Logical Shift Left

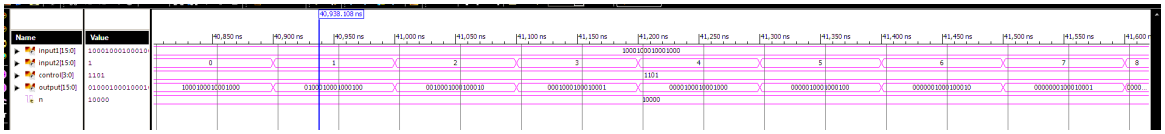

Figure 10 – correct 16 bit ALU Logical Shift Right

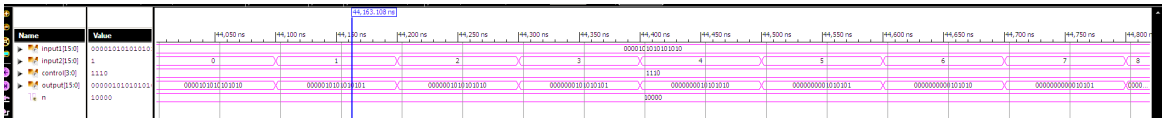

Figure 11 – correct 16 bit ALU Arithmetic Shift Right – positive number

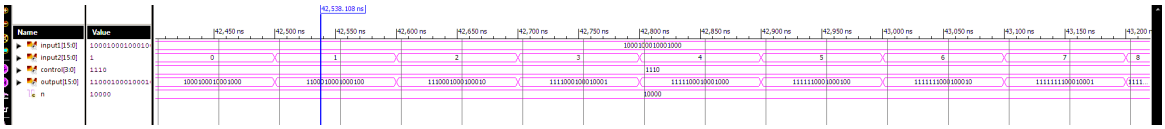

Figure 12 – correct 16 bit ALU Arithmetic Shift Right – negative number

Figure 15 – Place and post route with time delays

Code Segments

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.math_real."ceil";
23  use IEEE.math_real."log2";
24
25  entity ALU is
26
27  generic( N : Integer :=16);
28
29      Port ( Input1 : in  STD_LOGIC_VECTOR (N-1 downto 0);
30             Input2 : in  STD_LOGIC_VECTOR (N-1 downto 0);
31             Control : in  STD_LOGIC_VECTOR (3 downto 0);
32               clk : in std_logic;
33             Output : out  STD_LOGIC_VECTOR (N-1 downto 0));
34  end ALU;
35
36  architecture Behavioral of ALU is
37
38  signal logOut :STD_LOGIC_VECTOR (N-1 downto 0);
39  signal addOut :STD_LOGIC_VECTOR (N-1 downto 0);
40  signal multOut :STD_LOGIC_VECTOR (N-1 downto 0);
41  signal shiftOut :STD_LOGIC_VECTOR (N-1 downto 0);
42
43  signal dflipin1 : STD_LOGIC_VECTOR (N-1 downto 0);
44  signal dflipin2 :STD_LOGIC_VECTOR (N-1 downto 0);
45  signal dflipout:STD_LOGIC_VECTOR (N-1 downto 0);
46  signal throwaway : std_logic;
47
48  --here are my component declarations for all of the poarts used in the alu
49  component dflipflop
50  port(CLK: in std_logic;
51      D : in std_logic_vector(N-1 downto 0);
52      Q : out std_logic_vector(N-1 downto 0 ));
53      end component;
54  COMPONENT LogicUnit
55      PORT(
56          A : IN  std_logic_vector(N-1 downto 0);
57          B : IN  std_logic_vector(N-1 downto 0);
58          control : IN  std_logic_vector(1 downto 0);
59          output : OUT  std_logic_vector(N-1 downto 0)
60         );
61      END COMPONENT;
62
63        COMPONENT RippleCarryAdder
64       Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
65              B : in  STD_LOGIC_VECTOR (N-1 downto 0);
66              Cin : in  STD_LOGIC;
67              Cout : out  STD_LOGIC;
68              Sum : out  STD_LOGIC_VECTOR (N-1 downto 0));
69      END COMPONENT;
70
71       COMPONENT CarrySavMultiplier
72      Port ( A : in  STD_LOGIC_VECTOR (N/2-1 downto 0);
73              B : in  STD_LOGIC_VECTOR (N/2-1 downto 0);
74              product : out  STD_LOGIC_VECTOR (N-1 downto 0));
75      END COMPONENT;
76
77   COMPONENT shifter
78     Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
79              amt : in  STD_LOGIC_VECTOR (integer(ceil(log2(real(N))))-1 downto 0);--alter to be ceiling  integer(ceil(log2(real(a))))
80              control : in  STD_LOGIC_VECTOR (1 downto 0);-- first 2 bits - "11"
81              output : out  STD_LOGIC_VECTOR (N-1 downto 0));
82                end component;
83  begin
84  --here i instantiate the components of the alu and match up the ports with the internal signals
85
86  dflip0 : dflipflop
87  port map(clk=>clk,D=>input1,Q=>dflipin1);
88
89  dflip1 : dflipflop
90  port map(clk=>clk,D=>input2,Q=>dflipin2);
91
92  LogicUnit0 : LogicUnit
93  port map(A=> dflipin1,B=>dflipin2,control=>Control(1 downto 0),output=>logOut);
94
95  RippleCarryAdder0 : RippleCarryAdder
96  port map(A=> dflipin1,B=>dflipin2,Cin=>Control(0),Cout=>throwaway,Sum=>addOut);
97
98  shifter0 : shifter
99  port map(A=>dflipin1, amt=>dflipin2(integer(ceil(log2(real(N))))-1 downto 0),control=>control(1 downto 0),output=>shiftOut);
100
101 CarrySavMultiplier0 : CarrySavMultiplier
102 port map(A=>dflipin1(N/2-1 downto 0), B=>dflipin2(N/2-1 downto 0), product=> multOut);
103
104 --this is my output process in which i determine which component of the alu will be output to the final output of the alu
105 outproc : process(Input1, Input2,Control,addOut,multOut,logOut,shiftOut,dflipout,dflipin1,dflipin2) begin
106 if control(3 downto 1)="010" then
107 dflipout<=addOut;
108 elsif control(3 downto 1)="011" then
109 dflipout<=multOut;
110 elsif control(3 downto 2)="10" then
111 dflipout<=logOut;
112 elsif control(3 downto 2)="11" then
113 dflipout<=shiftOut;
114 else
115 dflipout<= (others=>'0');
116 end if;
117 end process;
118 dflip3 : dflipflop
119 port map(clk=>clk,D=>dflipout,Q=>Output);
120 end Behavioral;
121
122
```

Figure 17 –ALU top level source code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RippleCarryAdder is
Generic(N : integer := 16);
    Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
           B : in  STD_LOGIC_VECTOR (N-1 downto 0);
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Sum : out  STD_LOGIC_VECTOR (N-1 downto 0));
end RippleCarryAdder;

architecture Behavioral of RippleCarryAdder is
 signal int_cout : std_logic_vector(N-1 downto 0);
 --this is the one bitt adder that my structural adder is built of
component fadder is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Sum : out  STD_LOGIC);
              end component;
begin
--create full adders from the one- this is the first with unique inputs
 fulladders: entity work.fadder port map (
    A => A(0), B => B(0) XOR Cin, Cin => Cin, Sum => Sum(0), Cout => int_cout(0));
  --this is the generate for the rest of the adders
  adders: for i in 1 to N-1 generate
    fulladders: entity work.fadder port map (
      A => A(i), B => B(i) XOR Cin, Cin => int_cout(i-1), Sum => Sum(i), Cout => int_cout(i));
  end generate;
  Cout <= int_cout(N-1);
end Behavioral;
```

Figure 18 –Ripple carry source code

```
-----------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use Math_real.all;
use IEEE.math_real."ceil";
use IEEE.math_real."log2";

use IEEE.NUMERIC_STD.ALL;

entity shifter is

generic (N : integer := 16);
    Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
           amt : in  STD_LOGIC_VECTOR (integer(ceil(log2(real(N))))-1 downto 0);--alter to be ceiling  integer(ceil(log2(real(a))))
           control : in  STD_LOGIC_VECTOR (1 downto 0);-- first 2 bits - "11"
           output : out  STD_LOGIC_VECTOR (N-1 downto 0));
end shifter;

architecture Behavioral of shifter is
--ones and zeroes are vectors of n size to be placed into an output. their size is altered by amt.
signal ones :  std_logic_vector(N-1 downto 0) := (others=>'1');
signal zeroes: std_logic_vector(N-1 downto 0) := (others=>'0');
signal num : Integer := to_integer(unsigned(amt));
signal msb: std_logic := A(N-1);

signal outputsig : std_logic_vector(N-1 downto 0);
begin

--this process is used for determining which shift to perform
shift : process(A,amt,control,msb,outputsig)
begin
if amt = zeroes(integer(ceil(log2(real(N))))-1 downto 0) then
--output the original value
    outputsig<=A;
else
--left logical
    if control = "00" then
        outputsig <= A((N-to_integer(unsigned(amt))-1) downto 0) & zeroes(to_integer(unsigned(amt))-1 downto 0);
        --right logical
    elsif control =   "01" then
        outputsig <= zeroes(to_integer(unsigned(amt))-1 downto 0) & A((N-1) downto to_integer(unsigned(amt)));
    elsif control = "10" then
    --right arithmetic
        if  A(N-1) = '0' then
            outputsig <= zeroes(to_integer(unsigned(amt))-1 downto 0) & A((N-1) downto to_integer(unsigned(amt)));
            else
            outputsig <= ones(to_integer(unsigned(amt))-1 downto 0) & A((N-1) downto to_integer(unsigned(amt)));
    end if;
end if;
end if;
output<=outputsig;
end process;
end Behavioral;
```

Figure 19 –Shifter source code

```
10  ----------------------------------------------------------------------------------------------------
11  library IEEE;
12  use IEEE.STD_LOGIC_1164.ALL;
13
14  entity LogicUnit is
15  generic (N :integer :=16);
16      Port ( A : in  STD_LOGIC_VECTOR (N-1 downto 0);
17             B : in  STD_LOGIC_VECTOR (N-1 downto 0);
18             control : in  STD_LOGIC_VECTOR (1 downto 0);
19             output : out  STD_LOGIC_VECTOR (N-1 downto 0));
20  end LogicUnit;
21    architecture Behavioral of LogicUnit is
22  begin
23  process(control,A,B) is--this uses loops to do the bitwise comparisons and
24  --the last 2 control bits to determine the operation
25  begin
26          if control = "00" then
27              output <=A OR B;
28          elsif control = "01" then
29              output <=NOT A;
30          elsif control = "10" then
31              output <=A AND B;
32          else
33              output <=A XOR B;
34          end if;
35  end process;
36  end Behavioral;
37
38
```

Figure 20 –Logic block source code

```
--
-----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CarrySavMultiplier is
generic( N : Integer :=16);
    Port ( A : in  STD_LOGIC_VECTOR (N/2-1 downto 0);
           B : in  STD_LOGIC_VECTOR (N/2-1 downto 0);
           product : out  STD_LOGIC_VECTOR (N-1 downto 0));
end CarrySavMultiplier;

architecture Behavioral of CarrySavMultiplier is
type out_in_array2D is array(N/2-2 downto 0) of std_logic_vector(N/2-1 downto 0);
type and_array2D is array(N/2 -1 downto 0) of std_logic_vector(N/2-1 downto 0);

--3 signal arrays
signal carry_array : out_in_array2D; -- in/out of the FAS
signal Sum_array : out_in_array2D; --FA sum out
signal and_array: and_array2D; --and outputs

component fadder is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Cin : in  STD_LOGIC;
           Cout : out  STD_LOGIC;
           Sum : out  STD_LOGIC);
             end component;

component gate_and is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           OUTPUTprt : out  STD_LOGIC);
end component;

begin--start multiplying

--populate and array
--  ands: entity work.gate_and port map (
  --  A => A(0), B => B(0), OUTPUTprt => and_array(0)(0));
anddepth : for r in 0 to N/2-1 generate
andwidth : for c in 0 to N/2-1 generate
        ands: entity work.gate_and port map (
        A => A(r), B => B(c), OUTPUTprt => and_array(r)(c));
    end generate;
end generate;

adddepth : for r in 0 to N/2-2 generate
addwidth : for c in 0 to N/2-1 generate
        --start finding exceptions to the rules for generating adders

        --r1c1 corner
        r1c1 : if r=0 AND c=0 generate
            fulladders: entity work.fadder port map (
            --A => and_array(r)(c), B => and_array(r-1)(c+1), Cin =>'0', Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
            A => and_array(r+1)(c), B => and_array(r)(c+1), Cin =>'0', Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r1c2,3 mid
        r1cmid : if r=0 AND (c>0  AND c<N/2-1) generate
            fulladders: entity work.fadder port map (
            --A => and_array(r)(c), B => and_array(r-1)(c+1), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
            A => and_array(r+1)(c), B => and_array(r)(c+1), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r1c4 corner
        r1c4 : if r=0 AND c=N/2-1 generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => '0', Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;
    --fixed up to this point

        --r2c2 edge
        r2c2 : if (r>0  AND r<N/2-1) AND c=0 generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => Sum_array(r-1)(c+1), Cin =>'0', Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r2c3,4 mid
        r2c34 : if (r>0  AND r<N/2-1) AND (c>0  AND c<N/2-1) generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => Sum_array(r-1)(c+1), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r2c5 edge
        r2c5 : if (r>0  AND r<N/2-1) AND c=N/2-1  generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => carry_array(r-1)(c), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

                --need last set of cases (corner,mids,corner
        --r3c3 edge
        r3c3 : if r=N/2-1 AND c=0 generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => Sum_array(r-1)(c+1), Cin =>'0', Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r3c4,5 mid
        r3c45 : if r=N/2-1 AND (c>0  AND c<N/2-1) generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => Sum_array(r-1)(c+1), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;

        --r3c6 edge
        r3c6 : if r=N/2-1 AND c=N/2-1  generate
            fulladders: entity work.fadder port map (
            A => and_array(r+1)(c), B => carry_array(r-1)(c), Cin =>carry_array(r)(c-1), Sum => Sum_array(r)(c), Cout => carry_array(r)(c));
        end generate;
end generate;
end generate;

--HERE I ASSIGN PRODUCT OUTPUTS
    --productproc : process(A,B) begin
    product(N-1)<=carry_array(N/2-2)(N/2-1);
    loop1: for i in 1 to N/2-1 generate
    product(N-1-i)<=Sum_array(N/2-2)(N/2-i);
    end generate;
    loop2 : for j in 1 to N/2-1 generate
    product(N/2-j)<=Sum_array(N/2-1-j)(0);
    end generate;
    product(0)<=and_array(0)(0);
    --end process;
    end Behavioral;
```

Figure 21 –Multiplier source code

```
-- Additional Comments:

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

 use ieee.NUMERIC_std.all;
 use ieee.std_logic_textio.all;
 library std;
 use std.textio.all;
ENTITY multiplier_tb IS
generic( N : Integer :=8);
END multiplier_tb;

ARCHITECTURE behavior OF multiplier_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT CarrySavMultiplier
    PORT(
         A : IN  std_logic_vector(N-1 downto 0);
         B : IN  std_logic_vector(N-1 downto 0);
         product : OUT  std_logic_vector(2*N-1 downto 0)
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic_vector(N-1 downto 0) := (others => '0');
   signal B : std_logic_vector(N-1 downto 0) := (others => '0');

    --Outputs
   signal product : std_logic_vector(2*N-1 downto 0);
    signal step : integer :=0;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
   uut: CarrySavMultiplier PORT MAP (
          A => A,
          B => B,
          product => product
        );

process variable vline:line;
    file output_vectors: text;
  variable p: integer;
  --write file
  begin
  if step = 0 then
    file_open(output_vectors,"D:\my back ups\DSD2\Lab3\L3\text.txt", write_mode);
    for i in 0 to 2**(N)-1 loop
        for j in 0 to 2**(N)-1 loop
            hwrite(vline,std_logic_vector(to_unsigned(i,N)));
            write(vline,string'(" "));
            hwrite(vline,std_logic_vector(to_unsigned(j,N)));
            write(vline,string'(" "));
            p:=i*j;
            hwrite(vline,std_logic_vector(to_unsigned(p,2*N)));

            writeline(output_vectors,vline);
            end loop;
    end loop;

    file_close(output_vectors);
    step<=1;
    end if;
    wait;
    end process;

    process is

    variable vline:line;
    file input_vectors : text;
    variable x_in,y_in : std_logic_vector(((N+3)/4)*4-1 downto 0);
    variable p_in : std_logic_vector(2*((N+3)/4)*4-1 downto 0);
    variable filler : String(1 to 1);
    begin
    --step<=1;
    wait until step =1;
    file_open(input_vectors,"D:\my back ups\DSD2\Lab3\L3\text.txt",read_mode);
    --reaad from file
    while not endfile(input_vectors) loop
        readline(input_vectors,vline);
        hread(vline,x_in);
        read(vline,filler);
        hread(vline,y_in);
        read(vline,filler);
        hread(vline,p_in);
        A<=x_in;
        B<=y_in;
        --feed into components
        wait for 20ns;
        --asserts
        assert(p_in=product)
        report"error";
    end loop;
    end process;


END;
```

Figure 22 –Multiplier test bench source code

```
1   -- TestBench Template
2
3     LIBRARY ieee;
4     USE ieee.std_logic_1164.ALL;
5     USE ieee.numeric_std.ALL;
6     ENTITY testbench IS
7     generic( N : Integer :=16);
8     END testbench;
9     ARCHITECTURE behavior OF testbench IS
10    -- Component Declaration
11            COMPONENT ALU
12             Port ( Input1 : in  STD_LOGIC_VECTOR (N-1 downto 0);
13             Input2 : in  STD_LOGIC_VECTOR (N-1 downto 0);
14             Control : in  STD_LOGIC_VECTOR (3 downto 0);
15               clk : in std_logic;
16             Output : out  STD_LOGIC_VECTOR (N-1 downto 0));
17            END COMPONENT;
18            Signal Input1 :  STD_LOGIC_VECTOR (N-1 downto 0);
19            Signal Input2 :  STD_LOGIC_VECTOR (N-1 downto 0);
20             Signal Control : STD_LOGIC_VECTOR (3 downto 0);
21            Signal Output :  STD_LOGIC_VECTOR (N-1 downto 0);
22              signal clk : std_logic;
23    BEGIN
24  process
25  begin
26    clk<='1';
27    wait for 50 ns;
28    clk <='0';
29    wait for 50 ns;
30  end process;
31    -- Component Instantiation
32            uut: ALU PORT MAP(
33                  Input1 => Input1,
34             Input2 =>Input2,
35               Control=>Control,
36               clk=>clk,
37             Output=>Output
38            );
39    --  Test Bench Statements
40       tb : PROCESS
41       BEGIN
42          --loop by control and run each smaller test bench
43          --control is 0100 or 0101 run add sub tb
44        for i in 0 to 15 loop
45          for j in 0 to 15 loop
46          Input1 <= std_logic_vector(to_unsigned(i,N));
47          Input2 <= std_logic_vector(to_unsigned(j,N));
48          --addition
49          Control<="0100";
50          wait for 50ns;
51          end loop;
52          end loop;
53          for i in 0 to 15 loop
54          for j in 0 to 15 loop
55          Input1 <= std_logic_vector(to_unsigned(i,N));
56          Input2 <= std_logic_vector(to_unsigned(j,N));
57          --sub
58          Control<="0101";
59          wait for 50ns;
60          end loop;
61          end loop;
62          --test some small multiplications
63           for i in 0 to 15 loop
64          for j in 0 to 15 loop
65          wait for 50ns;
66          Input1<= std_logic_vector(to_unsigned(i,N));
67          Input2 <= std_logic_vector(to_unsigned(j,N));
68          Control<="0110";
69          end loop;
70          end loop;
71          -------test logic block here
72            wait for 100 ns;
73            --or
74          Input1 <="1110011111100111";
75          Input2 <="0000000000000000";
76  --      Input1 <="1110";
77     --   Input2 <="0000";
78          Control<="1000";
79          wait for 100 ns;
```

Figure 23 –ALU test bench source code part 1

```
--      Input1 <="1110";
    --  Input2 <="0000";
        Control<="1000";
        wait for 100 ns;
        --not
        Input1 <="1110111011101110";
        Input2 <="0000000000000000";
        --Input1 <="1110";
        --Input2 <="0000";
        Control<="1001";
        wait for 200ns;
        --and
        Input1 <="1010101010101010";
        Input2 <="0000000000000000";
--      Input1 <="1010";
    --  Input2 <="0000";
        Control<="1010";
        wait for 200ns;
        --xor
        Input1 <="1111111111111111";
        Input2 <="1111111011111110";
    --  Input1 <="1111";
        --Input2 <="1110";
        Control<="1011";
        wait for 200ns;
        --now time for shifting
        for i in 0 to 15 loop
        Input2 <= std_logic_vector(to_unsigned(i,N));
        Control <="1100";
        Input1<="1000100010001000";
        --Input1<="1010";
        wait for 100ns;
        end loop;

        for i in 0 to 15 loop
        Input2 <= std_logic_vector(to_unsigned(i,N));
        Control <="1101";
        Input1<="1000100010001000";
        --Input1<="1010";
        wait for 100ns;
        end loop;

        for i in 0 to 15 loop
        Input2 <= std_logic_vector(to_unsigned(i,N));
        Control <="1110";
        Input1<="1000100010001000";
        --Input1<="1010";
        wait for 100ns;
        end loop;

        for i in 0 to 15 loop
        Input2 <= std_logic_vector(to_unsigned(i,N));
        Control <="1110";
        Input1<="0000101010101010";
        --Input1<="0101";
        wait for 100ns;
        end loop;
        wait for 100 ns; -- wait until global set/reset completes
        -- Add user defined stimulus here
        wait; -- will wait forever
    END PROCESS tb;
  --  End Test Bench
  END;
```

Figure 24 –ALU test bench source code part 2

Lab Report Demo Sheet

Name: Brian Landy L3

Laboratory Sign Off	
Section	**Signature**
Pre-Lab	3/2
Simulation	3/24
Code Critique	3/25
Post – Route Simulation	3/24
Working Board	3/25