University of Missouri-Columbia

Final Project: 8 Bit Booth Signed Multiplier using Structural Mode in VHDL

Students name: Jack Burgess

Student Number: 14356387

Email Address: john.burgess@missouribeta.com

Date: May 6th, 2023

Table of Contents:

Introduction and History:

Booth multiplication was created in 1950 by Andrew Donald Booth. He was doing research on Crystallography at Birkbeck College in London. Booth created this process to help increase efficiency. The other processes such as array multiplication or looped multiplication are very slow when compared to Booth's solution.

Booth's motivation was to reduce the total number of multiplicand multiples. This is because the delay is mainly determined by the number of additions performed. Used for multiplying larger signed and unsigned numbers. Can be used for only multiplication. Its main scope is signal processing, cryptography, and arithmetic. We need to study and use Booth's method as it can greatly reduce the number of operations. This in turn also decreases the amount of power needed when compared to other solutions.

Used for multiplying large signed binary numbers. This is used for hardware and software systems. Used in Cryptography algorithms. This includes RSA algorithms and geometric curve cryptography. Also used in digital filtering processes. This includes image processing and Fourier transform. Also used in education was a teaching tool to computer operations and logic.

The goal of this project was to create an 8 bit signed booth multiplier. Then to output a 16 bit answer. This was done by loading the two inputs into a Register called A and B. Then once a End Flag was flipped on the Z reg was loaded with the final answer. The Load flag would then go low.

The report will include the results of each part. The block diagram of each part. The code which is posted at the appendix. A discussion about the algorithm and how 8 bit booth multiplication works from the top down, this will be posted in the back ground part. The design details will include how the different modules will call lower files. This will go in detail about how the gates are built into different parts that are all finally called by the booth multiplier.

Finally the goal of this project was to learn more about structural VHDL coding and to

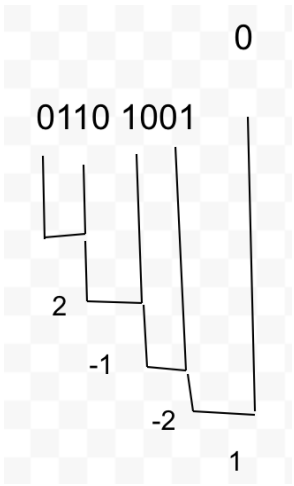
showcase our VHDL knowledge in one big project. This includes our knowledge on truth tables, port mapping and to finally learn how to code in structural mode. From a top down perspective the goal of this project was to create a 8 bit booth multiplier. This includes building a decoder a shifter some gates, a full adder and a solution to twos compliment.

Back-Ground and Algorithms Used:

The first step of performing this type of algorithm is to reduce the number of operations. This is done by separating the bit string into groups of three. I show an example of me dividing the string. We have to add an imaginary zero to the end to make the string work. If the Multiplicand is 2 or -2 then we shift left three times instead of two. If it is -1 then we use 2's complement. If the Multiplicand is 0 then we just shift left 2 digits and move onto the next value. If 1 then shift left two and then write the multicand.

High Bit	Middle	Low Bit	New Multiplier
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

A truth table for what the encoder values will turn into.

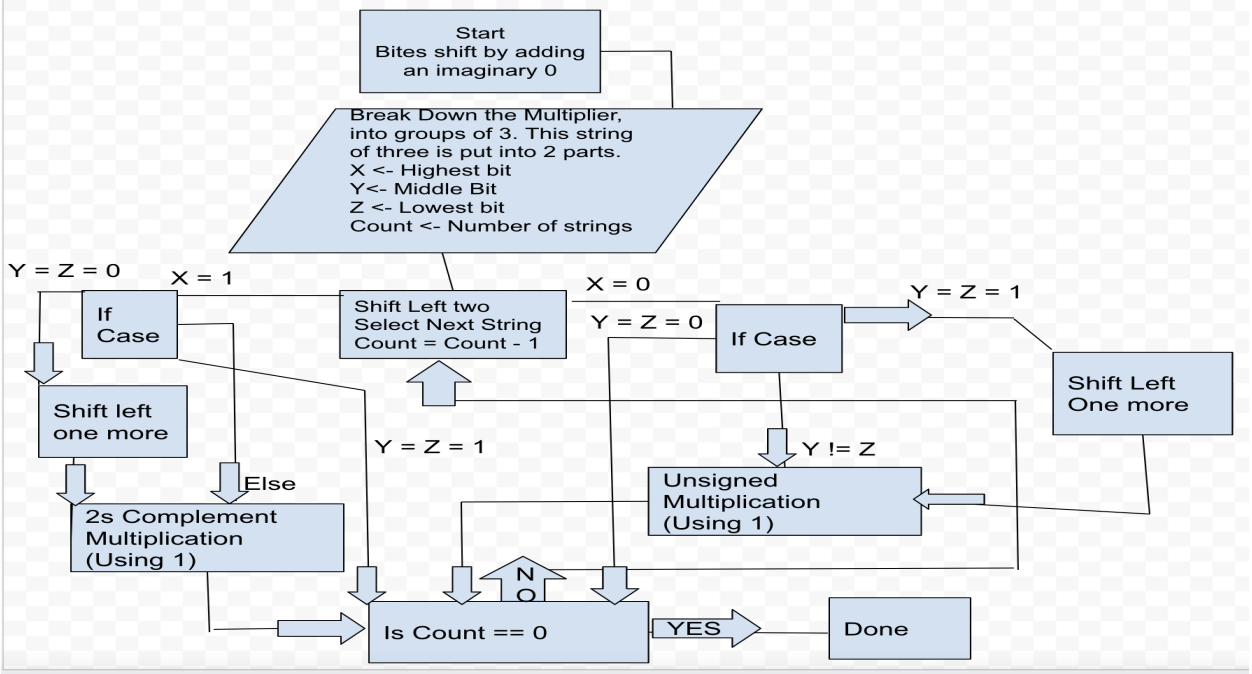


A chart of me manually breaking down the string into the 4 different output values.

Below is a graph that shows what happens after we multiple the terms out. We can see that after each term we must move left two digits. Once we have them in the form of these products we can use a full adder to combine the terms together into one complete string.

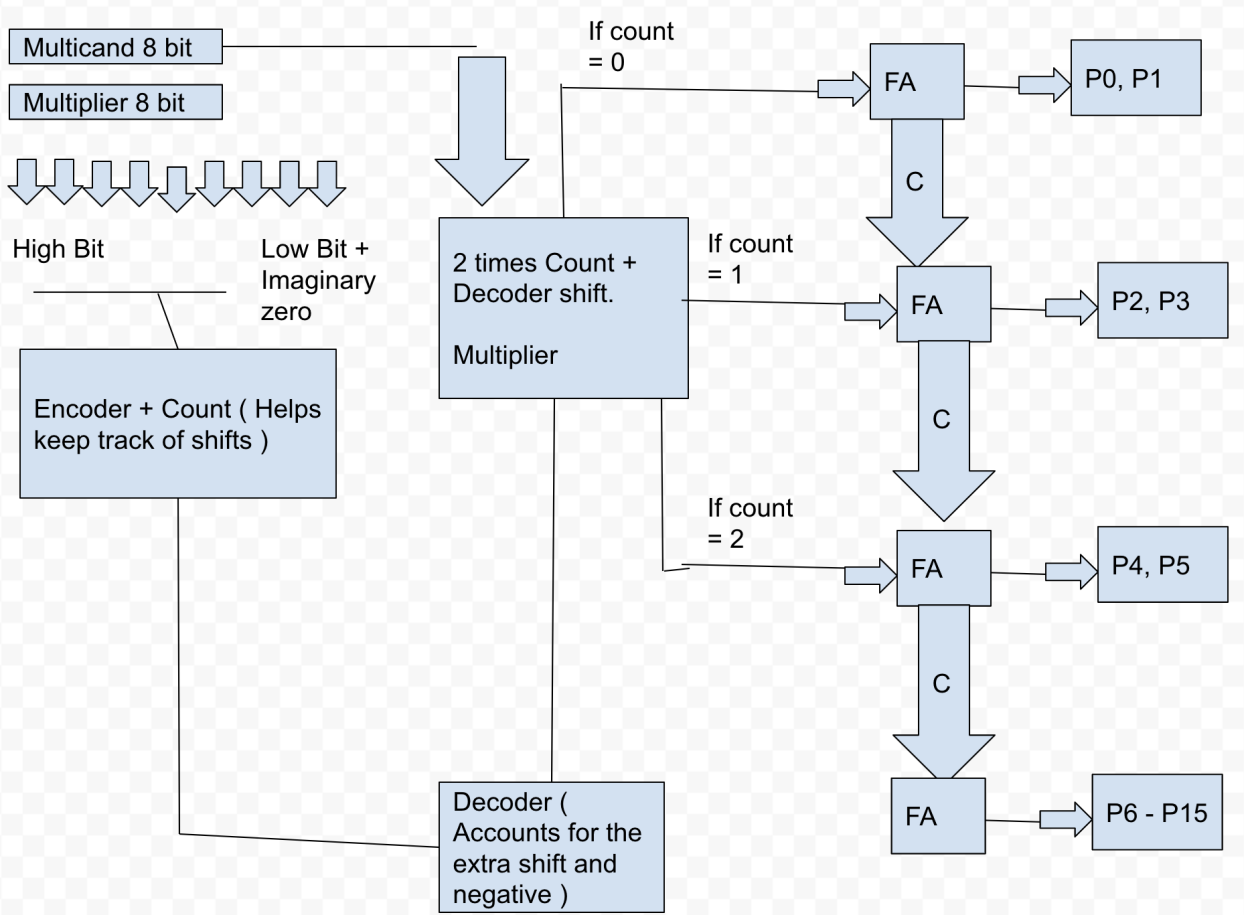
											Multiplicand
										Times	Reduced String (Sn)
						P5	P4	P3	P2	P1	P0
				P7	P6	P5	P4	P3	P2		
		P9	P8	P7	P6	P5	P4				
P11	P10	P9	P8	P7	P6						
P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

A chart showing how each of the partials will add up.



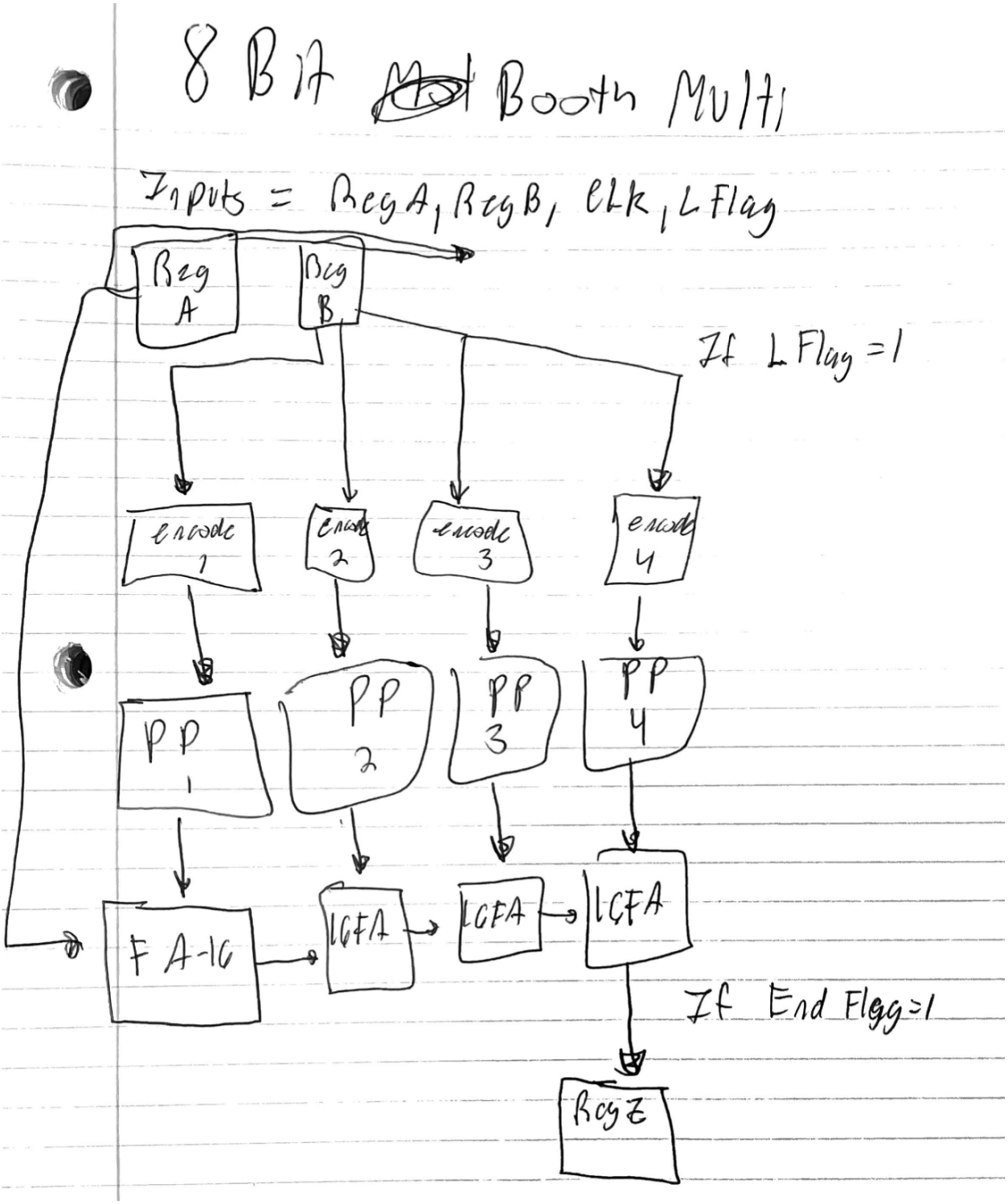
This diagram shows the the different paths that the multiplication can take. This is more

or less a piece of Psuedo code that will show the different path ways to getting the decoder value.



Here is a diagram showing the flowing chart of a more general booth multiplier. This has 2 inputs each with 8 bit arrays. One array gets encoded and the other gets added with the first partial product. Then the next 3 partial products are added together with this value and the output is created. We can see here that the bits 0-5 are spit out at different stages because of the two bit shifts for each partial product.

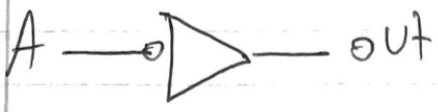
Design Details + Implementation:



8 Bit- Booth Multi Final Block Diagram. The implementation is simple. We have two inputs that are 8 bits long that are loaded into the A and B regs. The B reg is decoded into 1 - 4. These are then sent to the partial product creators. This is then added with the only Reg A using a 16 bit full adder. Then once the EndFlag is high the RegZ is filled with the final answer.

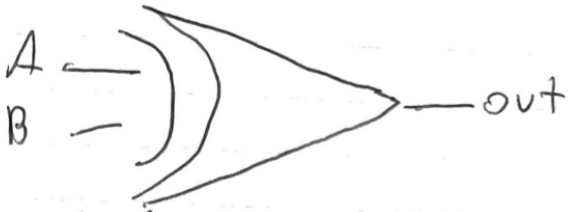
logic Gates

Not 2



out = ! A

XOR 2

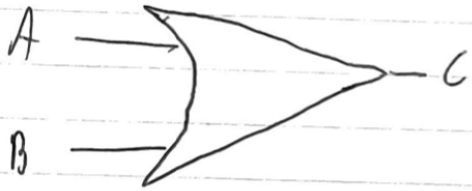


If A and B are different then out = 1 else out = 0

And 2

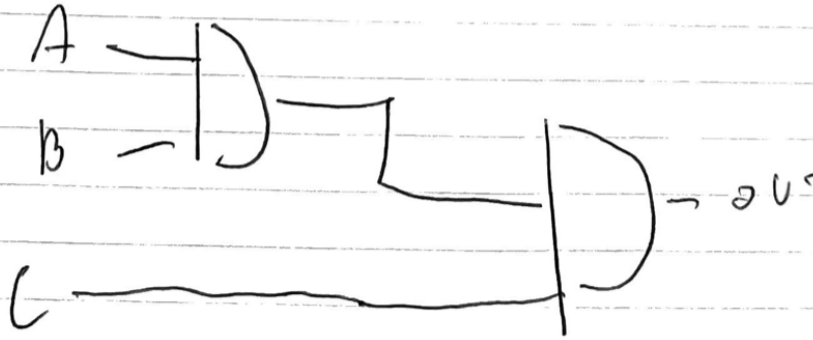
If A = 1 and B = 1 then C → 1

OR 2



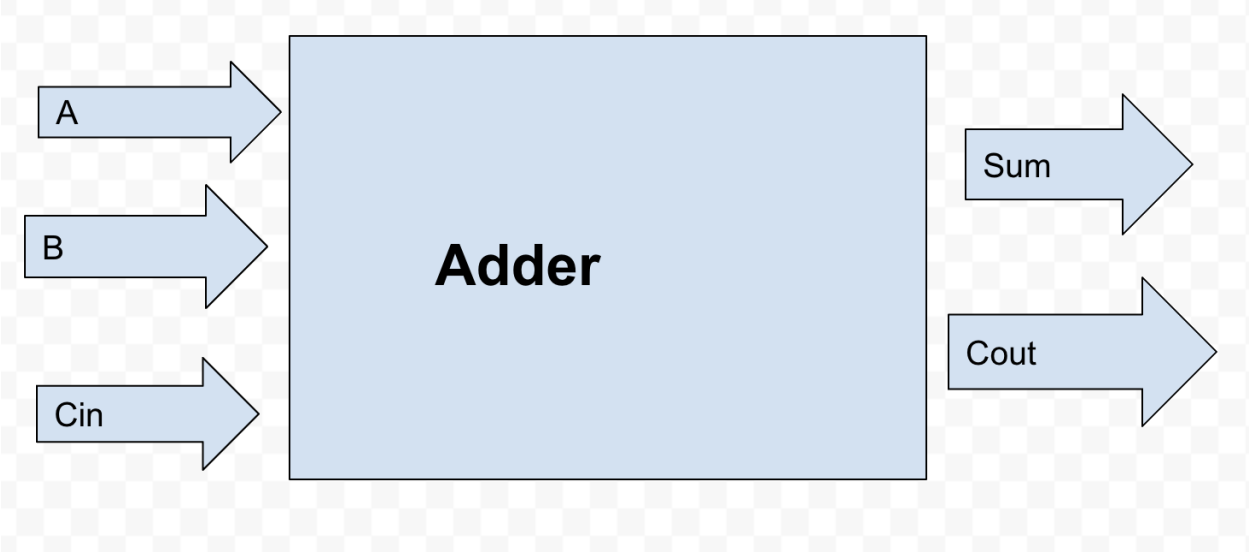
If A = 1 or B = 1 then C = 1

AND 3

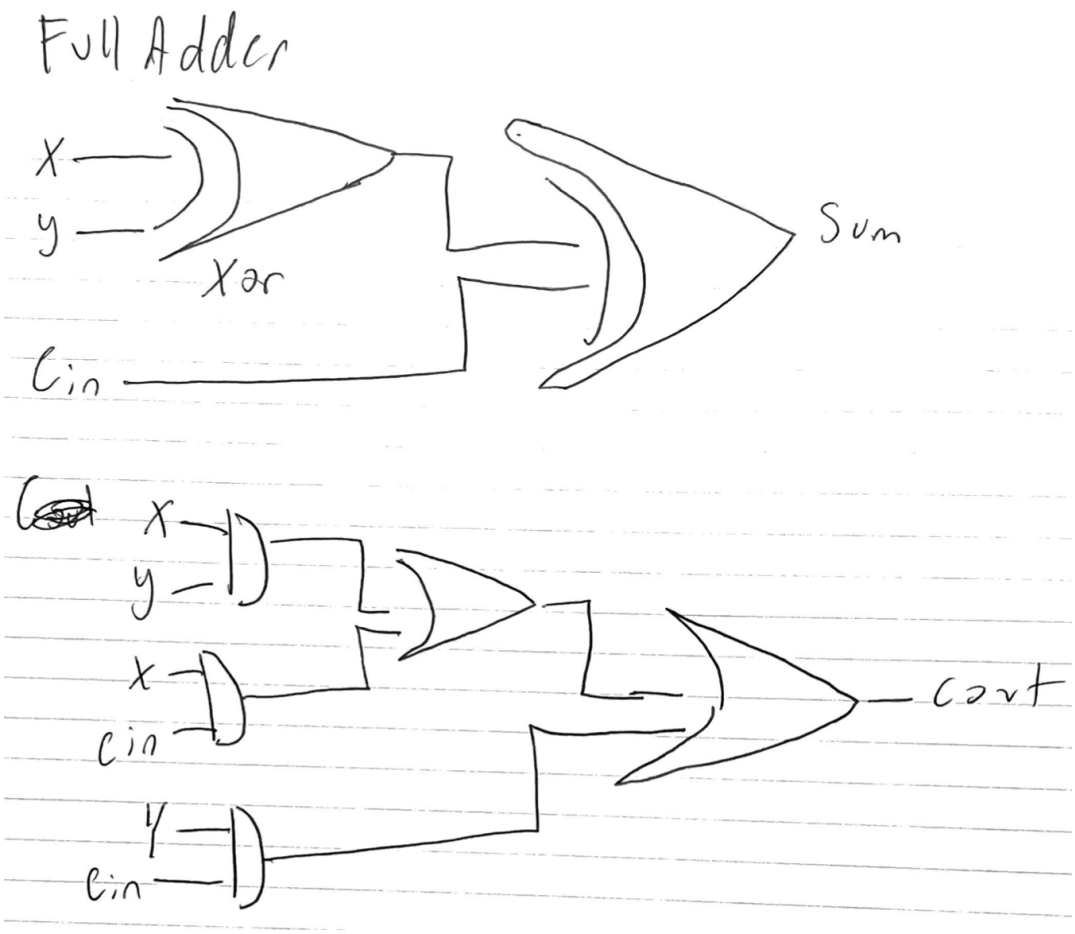


If (A and B) and C then out = C

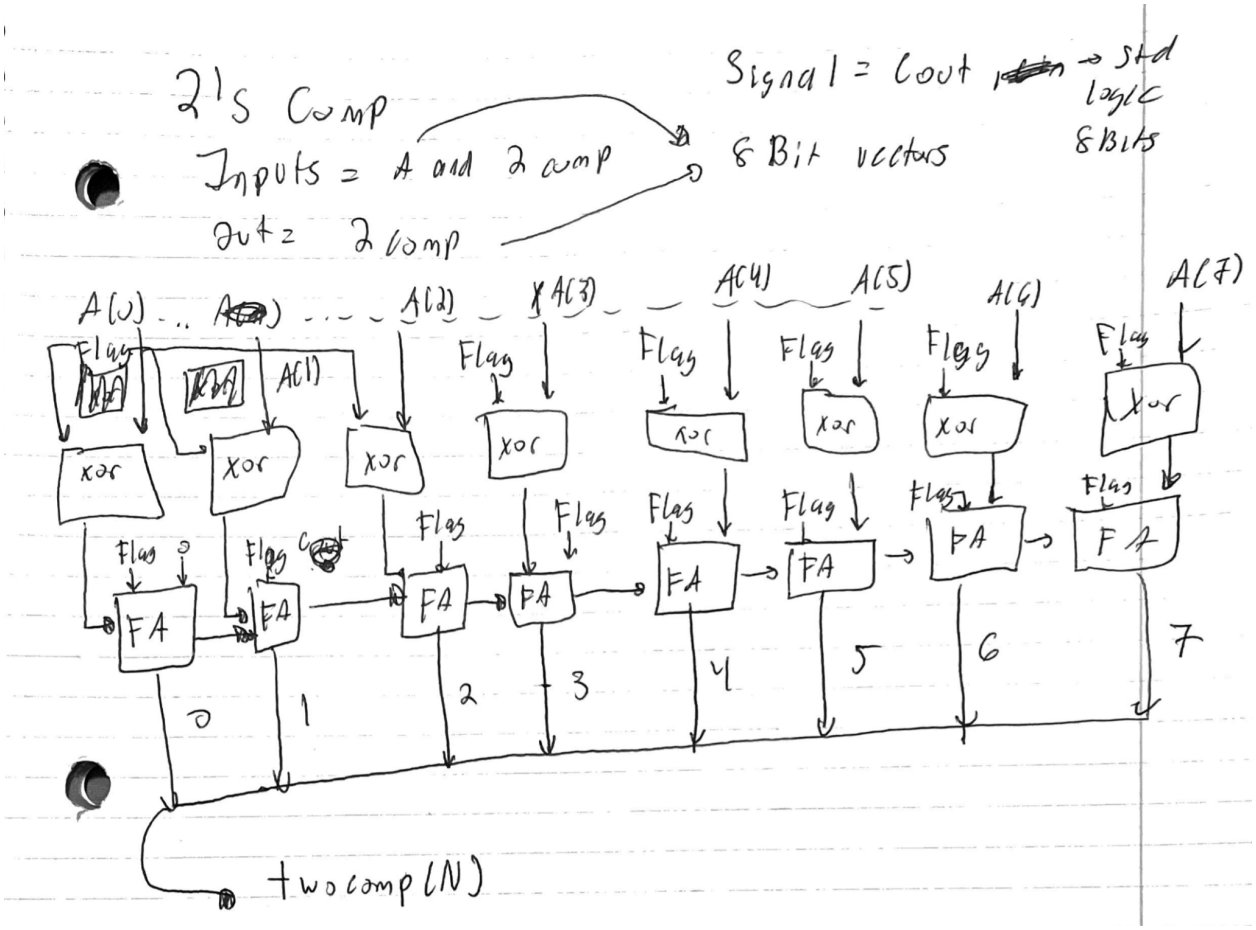
These are the block diagrams for the 5 different logic gates that I used. The code is simple just the inputs and then A (Logic) B -> out.



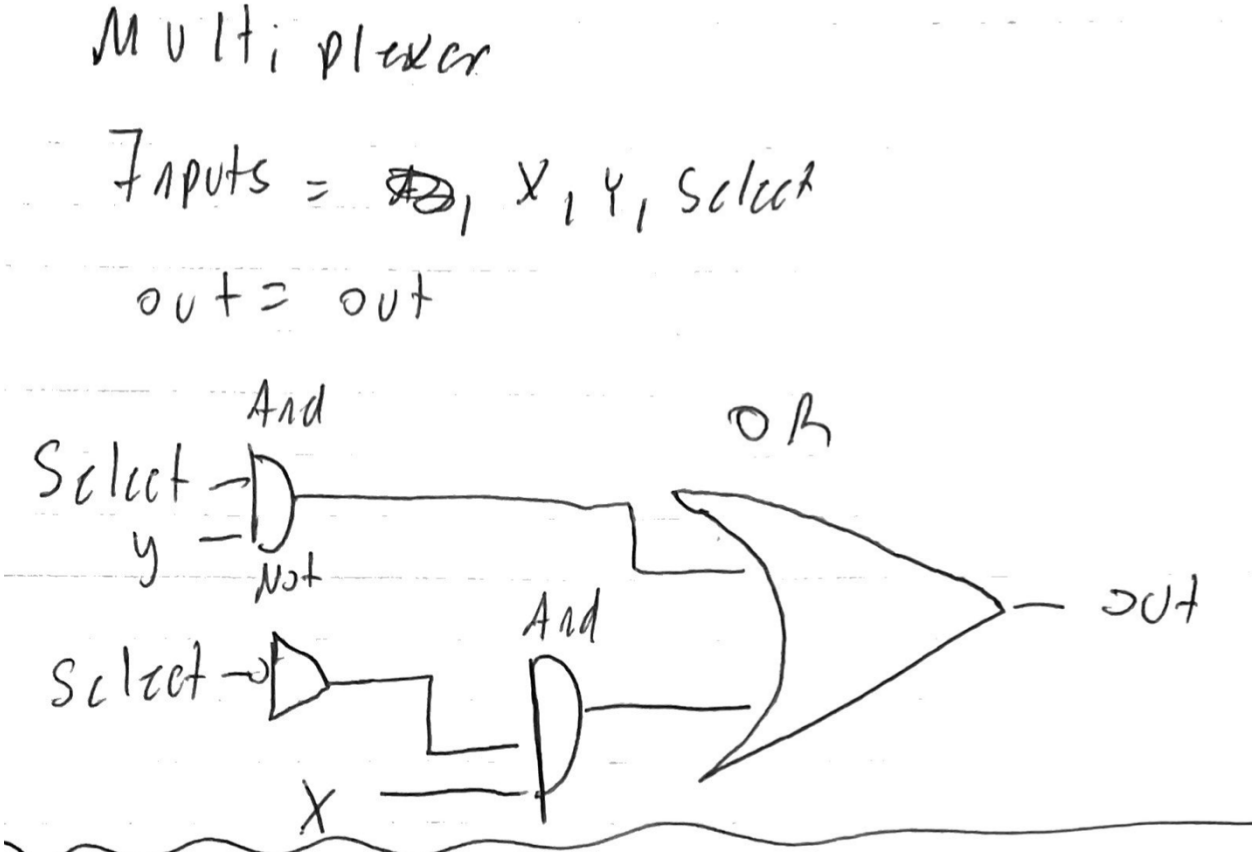
Full Adder block diagram.



Full Adder implementation. Here we can see that the full adder has 2 outputs of Sum and Cout. With the inputs X, Y and Cin. With the following logic lines for each.
Sum == (X Xor Y) XOR C
Cout == (X and Y) XOR (X and Cin) Xor (Y and Cin)

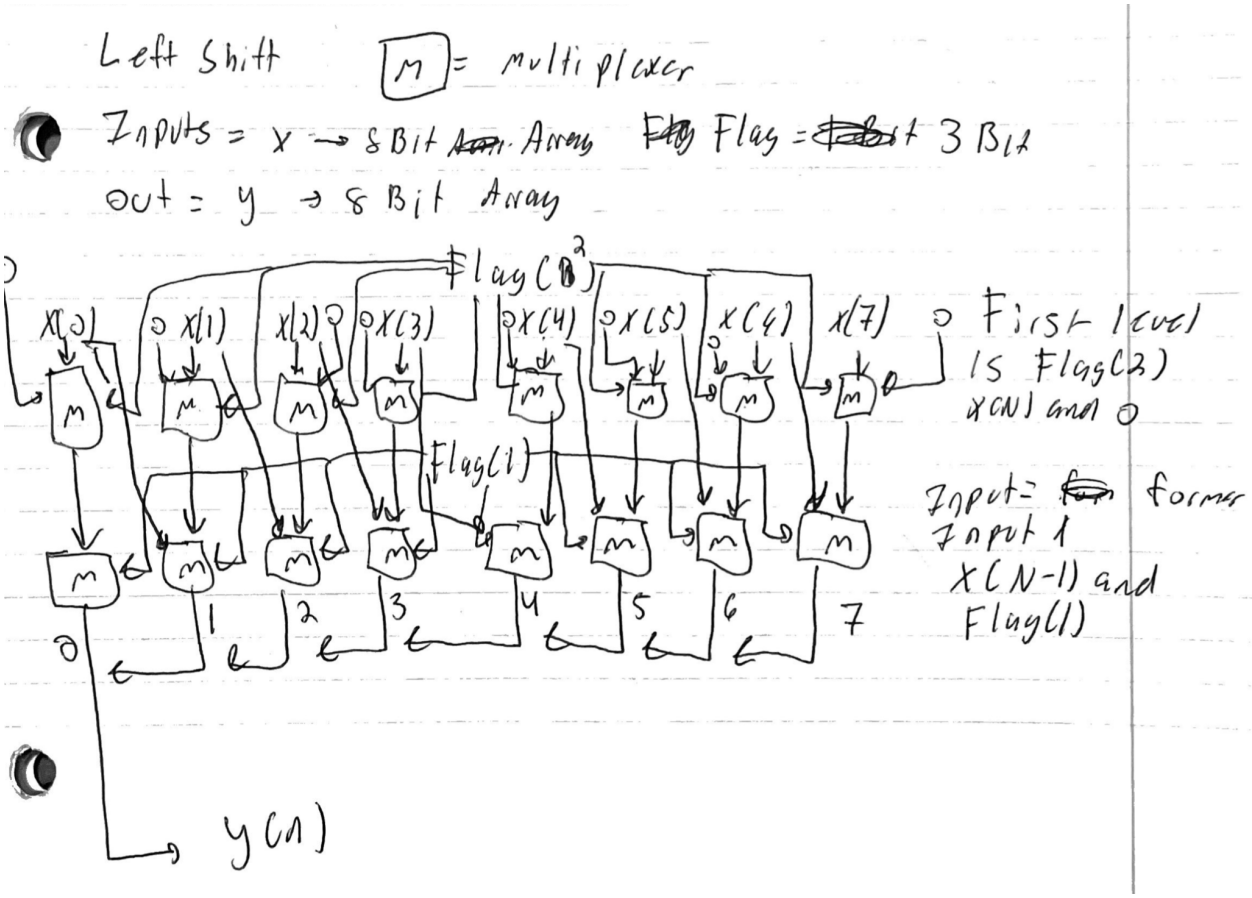


Twos Comp implementation. Here we can see how an array goes into an 8 bit Xor gate with a 2 comp Flag. This output is set into a full adder with the flag. The cout of each FullAdder acts as the Cin for the next full adder. This results is then saved in the Two Comp output array.

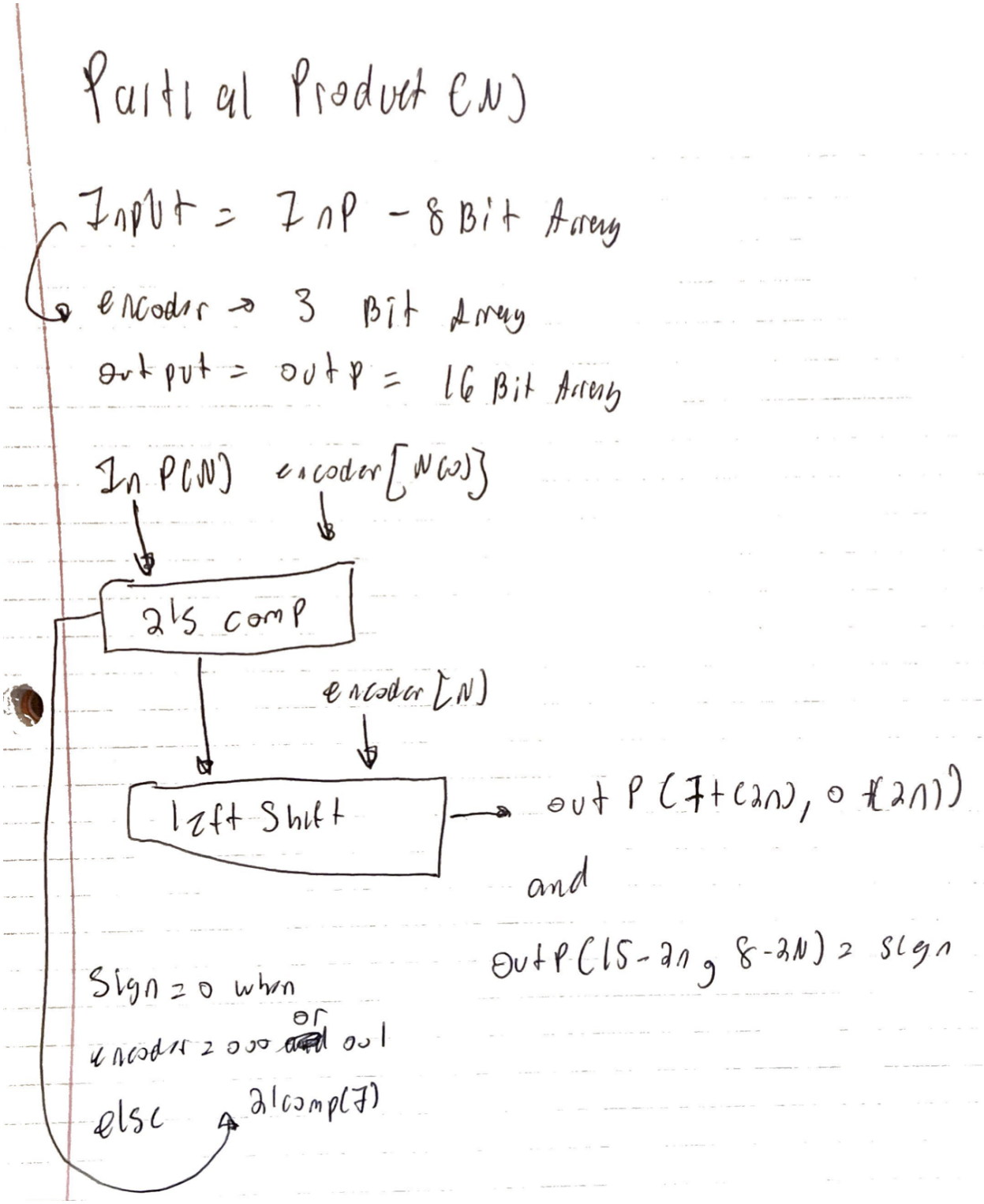


Here is the MultiPlexer. Here we have an input or x y and select with an output of out. This is with the following equation

OUT == (Select and Y) OR [(Not Select) And (X)]

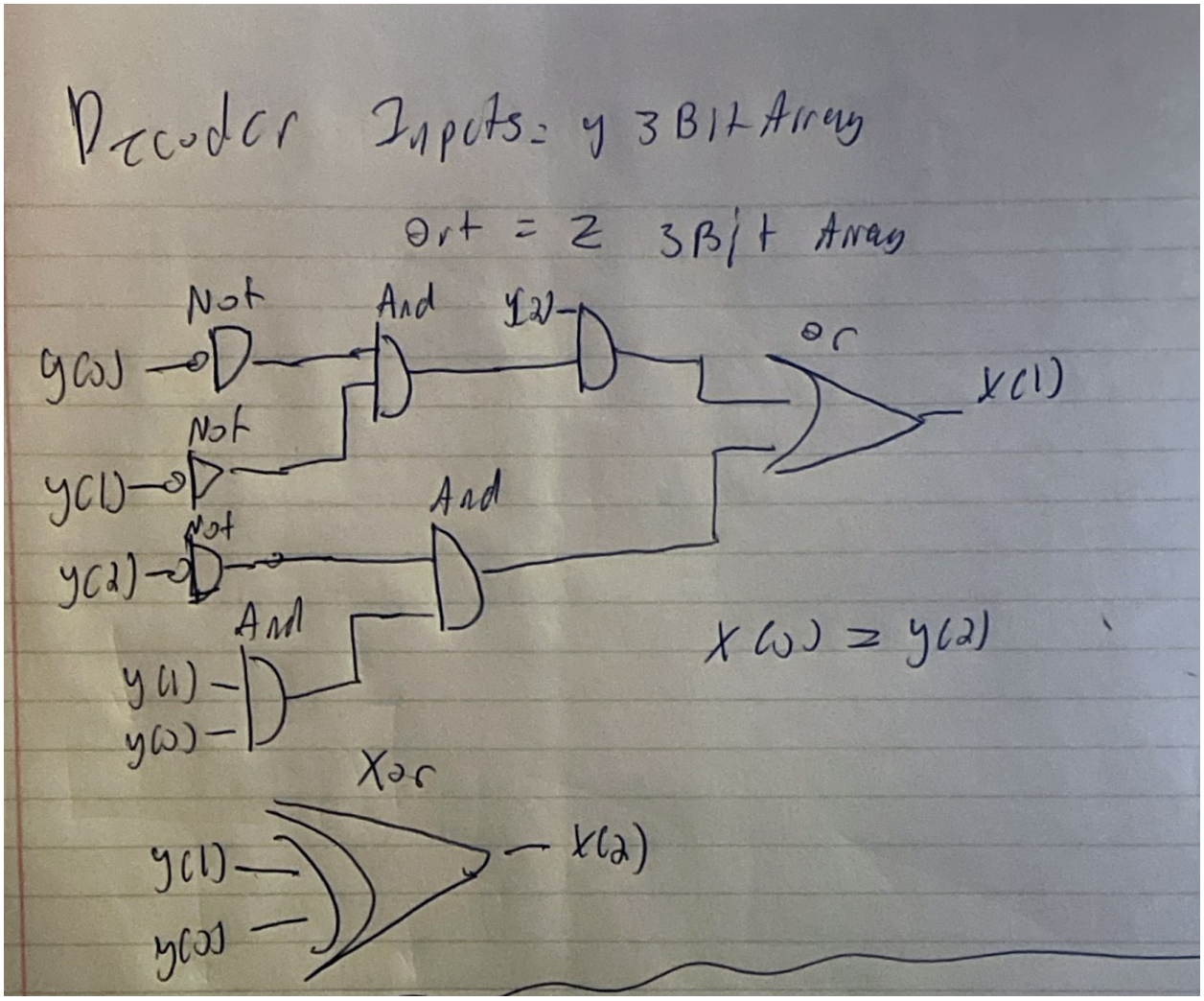


Here is the Left shift diagram and implementation. Here we can see that X(N) and 0 and the highest bit of the Flag is put into a multiplexer. With this output being put sent to another multiplexer with the first having an input of 0 and the following multiplexers having the input X(n-1). The third input is the middle bit of the flag. These outputs are then saved into the y output 8 bit binary array.



This is the partial product creator. Since PP1 - PP4 have the same diagram I have only shown one. It works by sending the inP and encoder(0) value into a 2s comp and then sending this output and the encoder value(all digits) into the left shift. This is then saved into outP based on this equation.

(7 +2n , 0 +2n) and the rest being outP(15 - 2n , 8 - 2n). The digits that are missed are filled in as 0.



This is the Booth Decoder. Here we can see the logic gates that are used to create X(1). With X(0) being y(2). So if the value is negative it will be stored here. We can also see that y(1) XOR y(0) is X(2). The output of this will be used in the partial product and left shift stage to tell what the multicand value is.

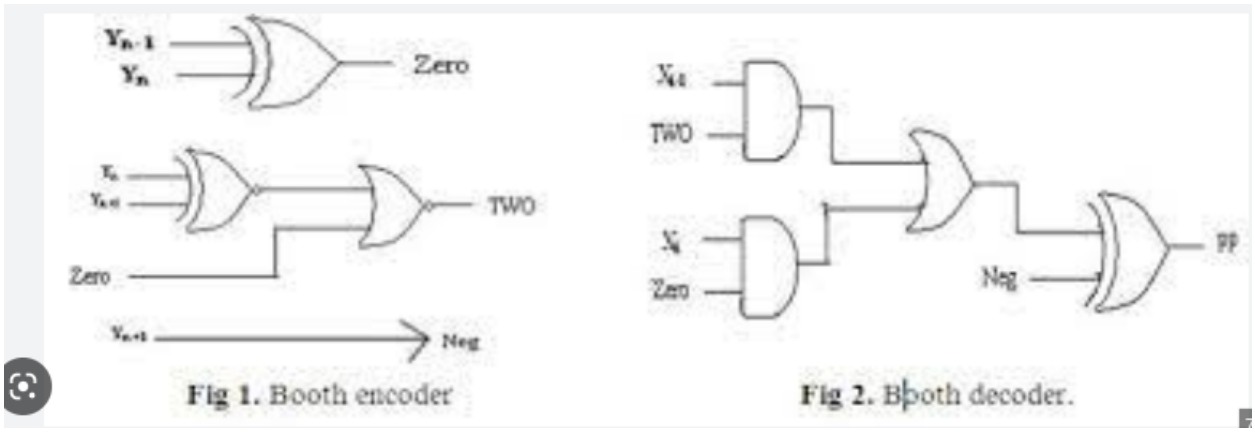


Fig 1. Booth encoder

Fig 2. Booth decoder.

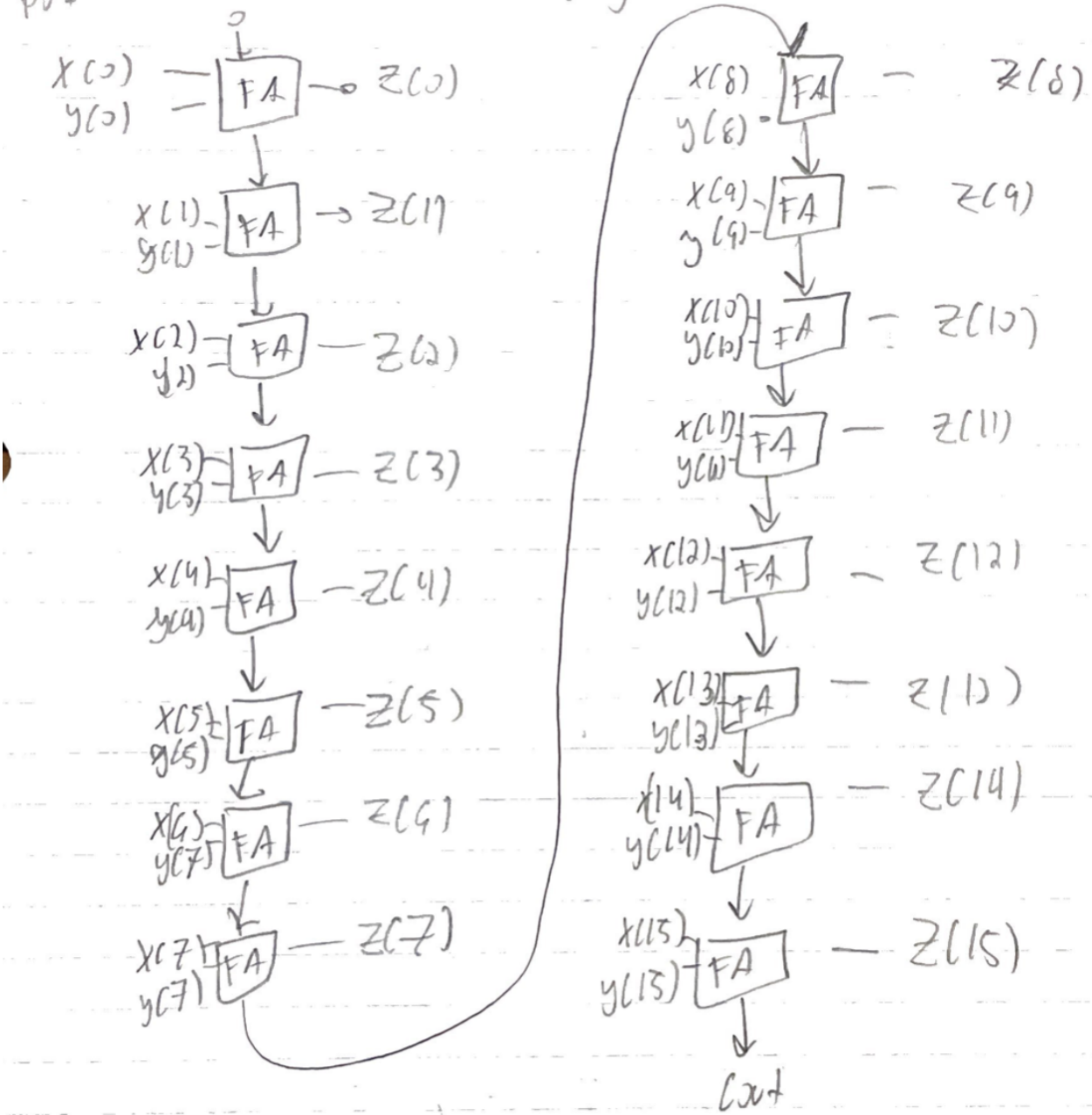
Booth example from the Text book

16 Bit Full Adder

Inputs = X, Y → 16 Bit Arrays

Cin = 1 Bit

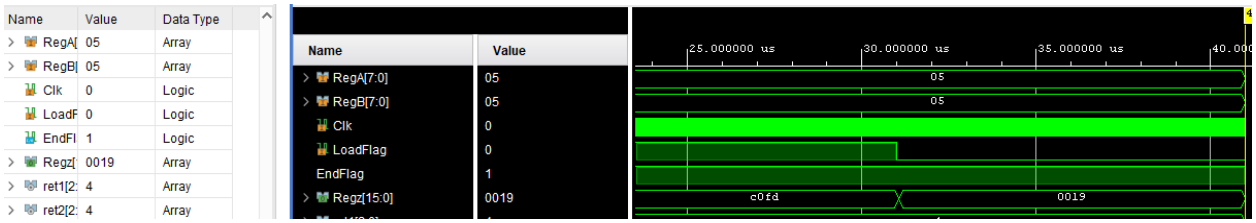
Output = Z → 16 Bit Array, Cout = 1 Bit



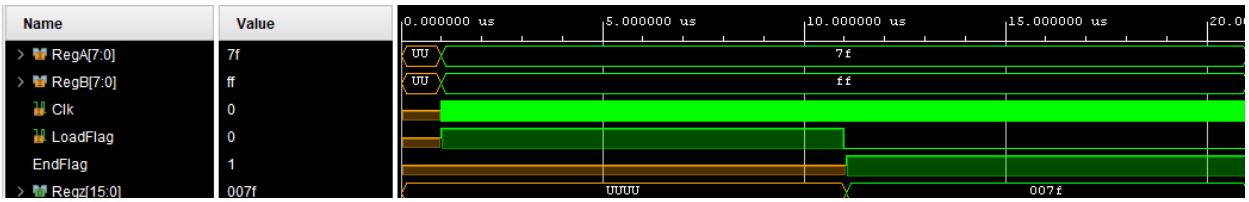
This is the 16 bit fuller adder. We can basically say that this works by combining 16 Full Adders with the Cout of the previous being loaded into the next FAs Cin. Each FA sum is saved into Z(N).

Results:

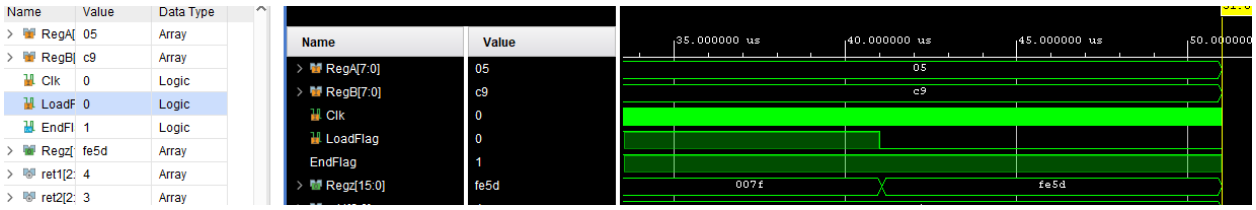
Below are some results for the final Booth multiplier. Results 1 - 5 show the final product with given inputs. The Reg A and Reg B are the inputs and the output is in Reg Z. All of these numbers are shown in Hexadecimal format.



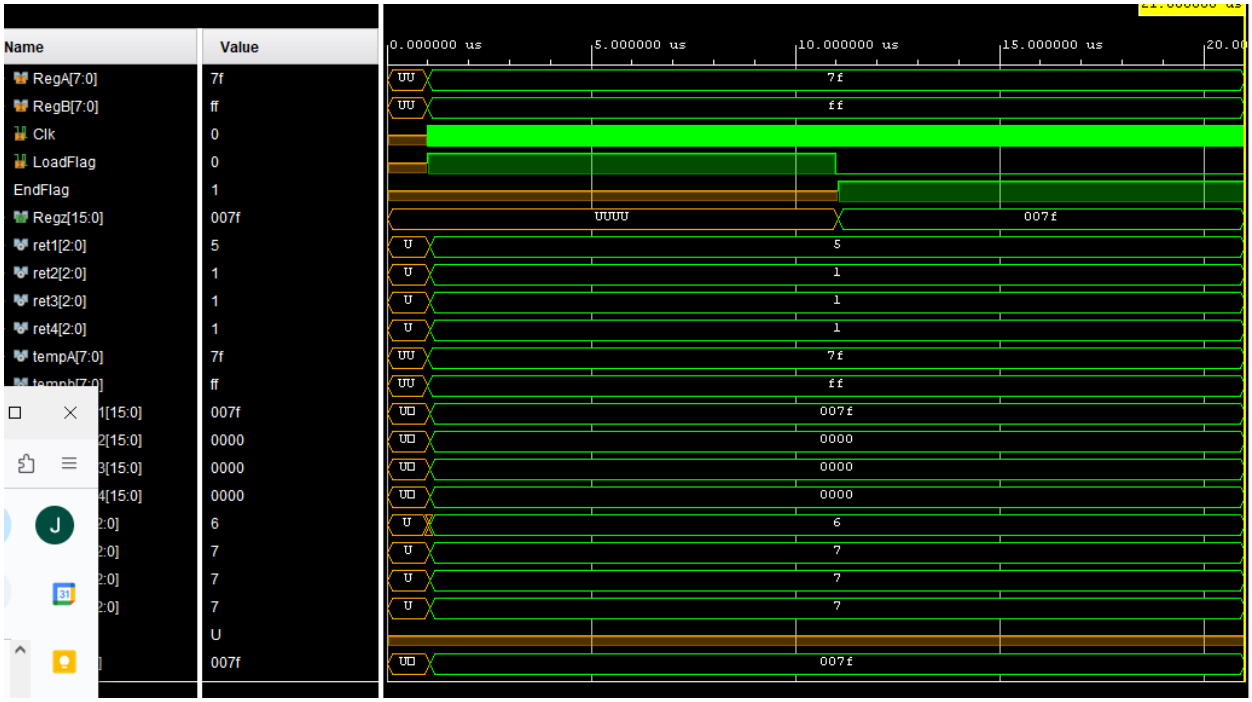
Results 1) Final with inputs 5 and 5. Final is 19 hex or 25 decimal.



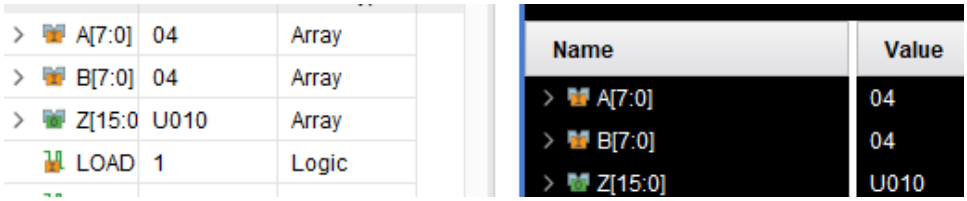
Results 2) Here is with -1 and 127. Which gives us the correct answer of 7f in Hex or -127



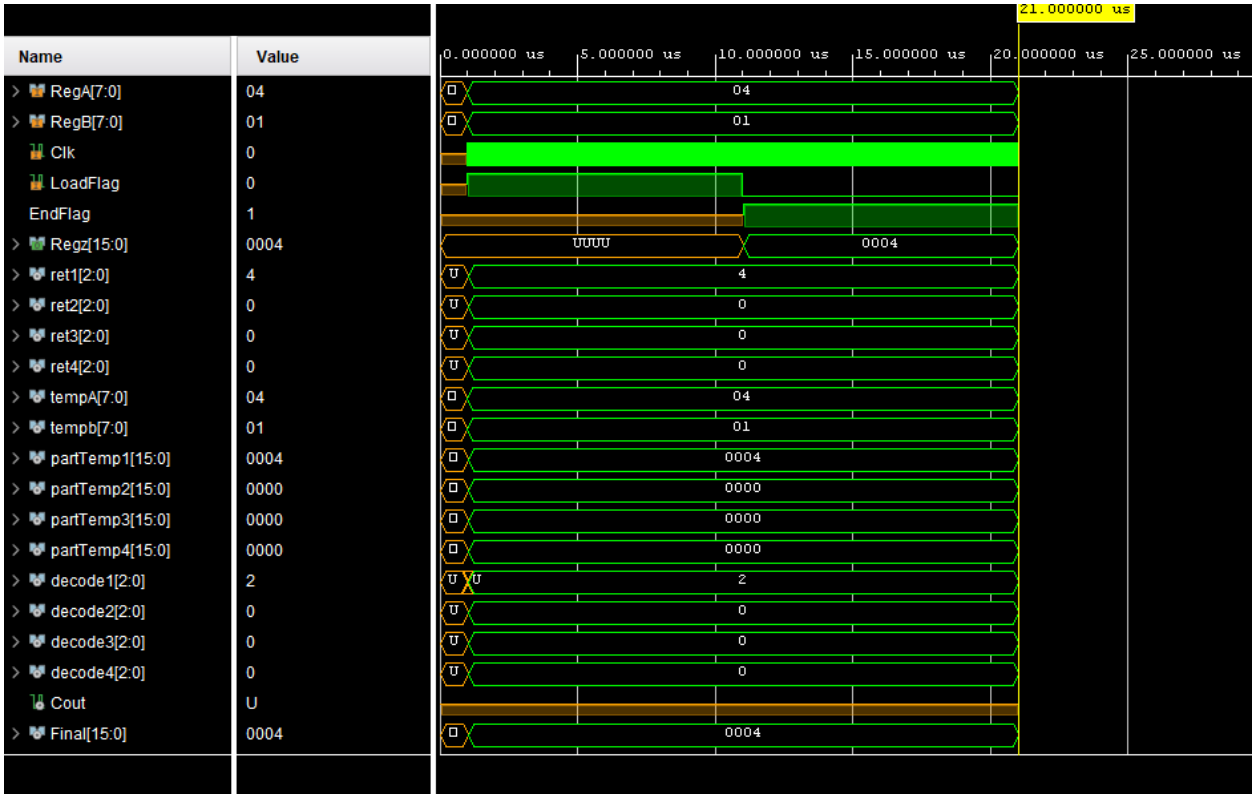
Results 3) Here is 5 times -55, Which gives us an answer of FE4D or -275.



Results 4) Here is a timing diagram. We can see that whenever the load flag changes from high to low that the Z Reg is filled in. We can also see that the A and B regs get filled when Load is turned on. Furthermore we can see the temp signals that are running below with the final output being loaded into Z when End flag goes high.



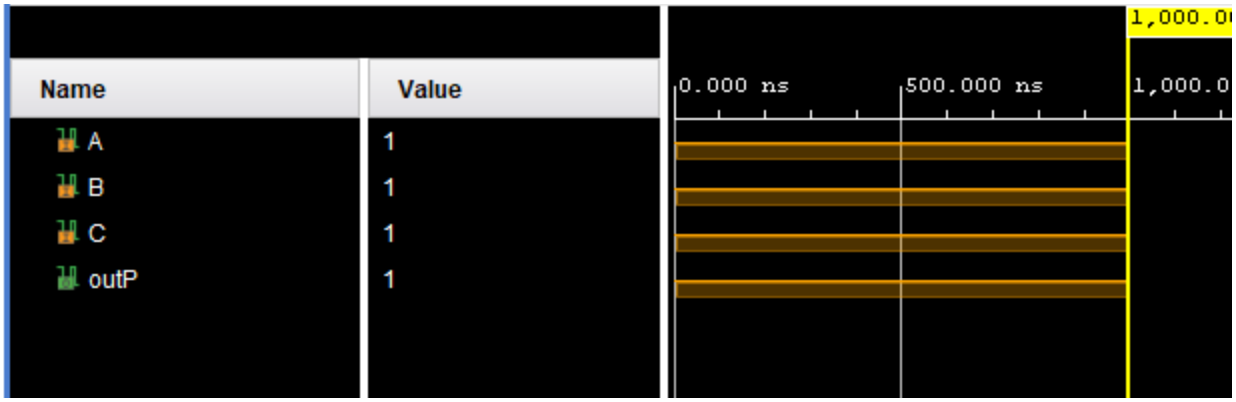
Results 5) With an input of 4 and 4 which gives an output of 010 or 16 in decimal.



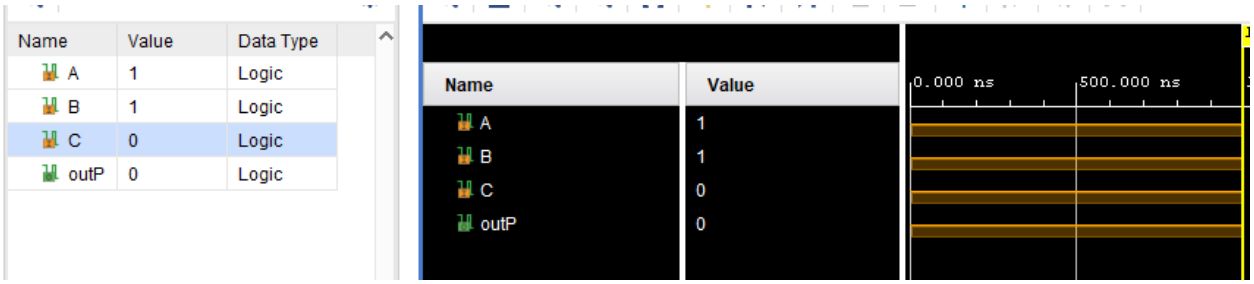
Results 6) Here is a timing diagram. We can see that at 10 ns after the first clock system the Reg A and Reg B are loaded. Then after 10ns once the load flag is switched off the code will produce the output and save it into Reg Z.. This timing diagram is per the assignment requires and shows how each of the waves and signals work based on the clock.

Component Results:

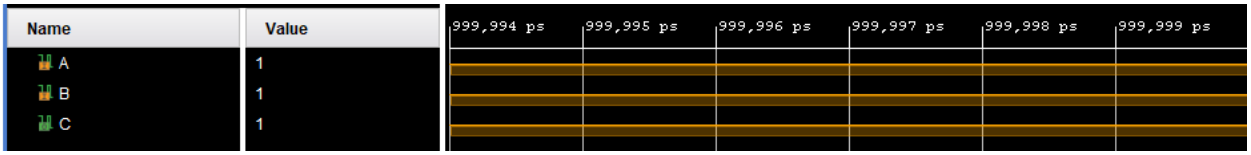
Below is the results for each module. Each will have a screen shot and I will discuss the input and how it got that output.



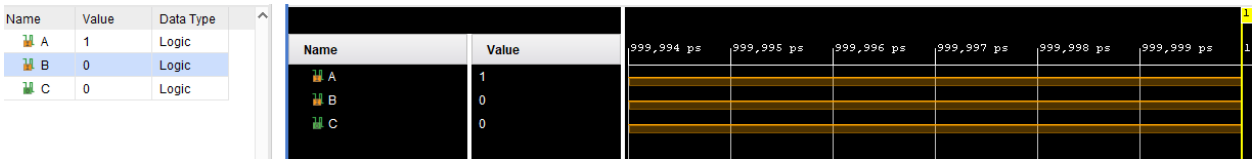
AndIn3) In this module we use an and gate with 3 inputs. A,B, C were all set to 1 and the output is 1.



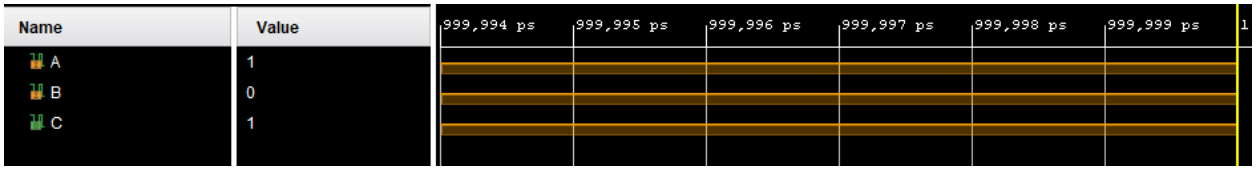
AndIns3) Here is the same module but with the inputs A and B = 1 and C = 0. Which gives C = 0.



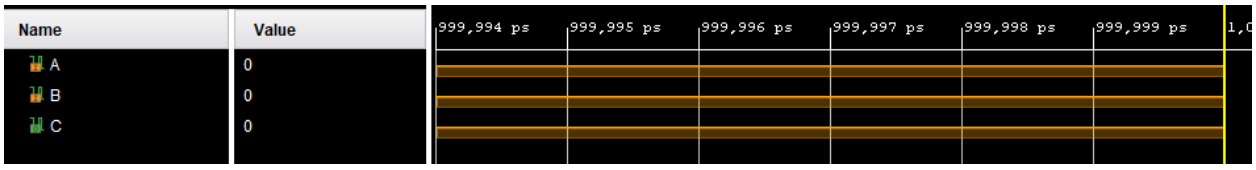
And2) Here is the results from the And gate with the inputs A and B = 1. This gives an output C = 1.



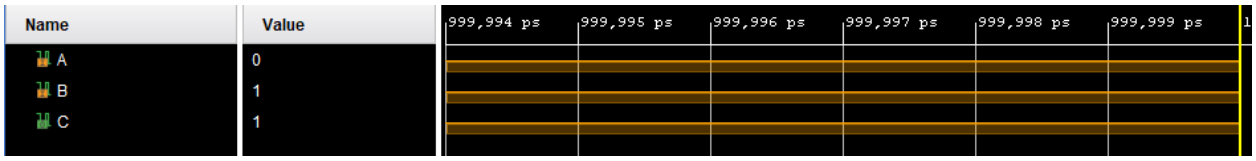
And2) Here is the results from the And gate with the inputs A = 1 , B = 0. Which gives an out of C = 0.



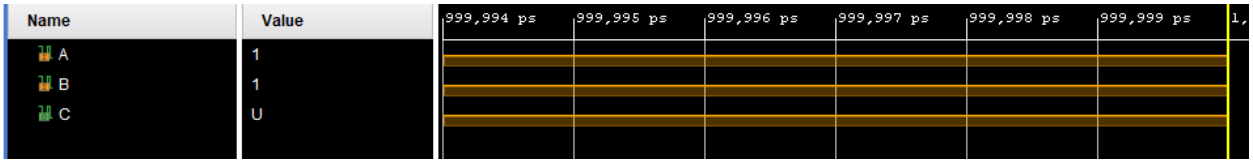
Or2) Here is the output of the Or gate with the inputs A =1 and B = 0. The output C is therefore 1.



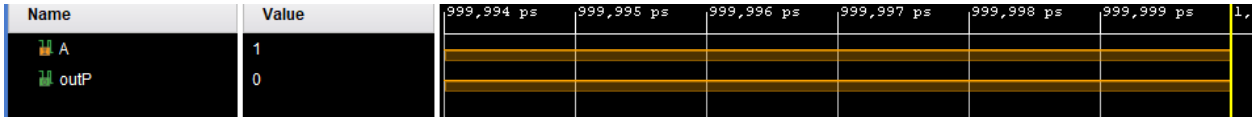
Or2) Here is the output of the Or Gate with the inputs A = 0 and B = 0. Which gives the output of C = 0.



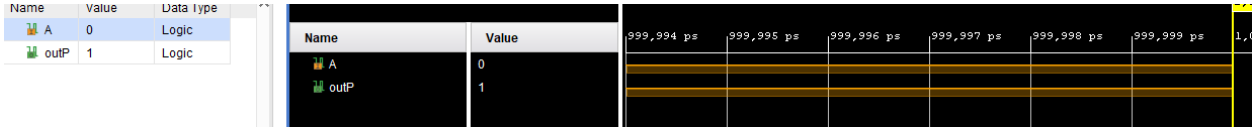
Xor2) Here is the output of the Xor Gate with the inputs A = 0 and B = 1. Which gives the output of C = 1.



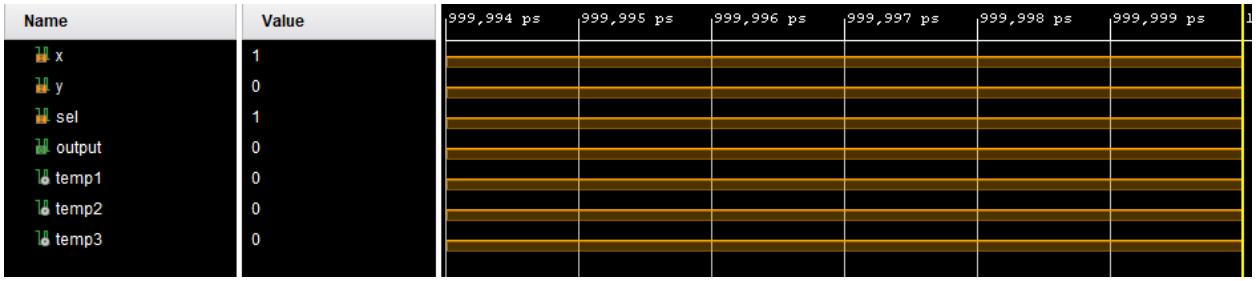
Xor2) Here is the output of the Xor Gate with the inputs A = 1 and B =1 which gives the output C = 0.



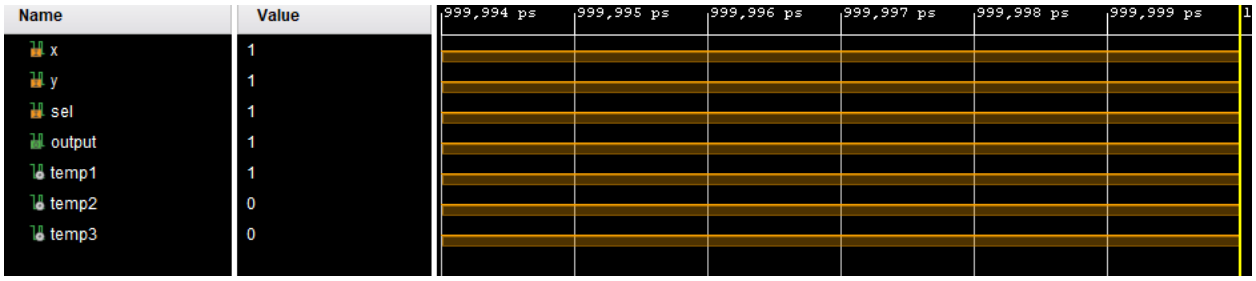
Not 2) Here are the results from the Not 2 gate. With the input A = 1. This gives an output of outP = 0.



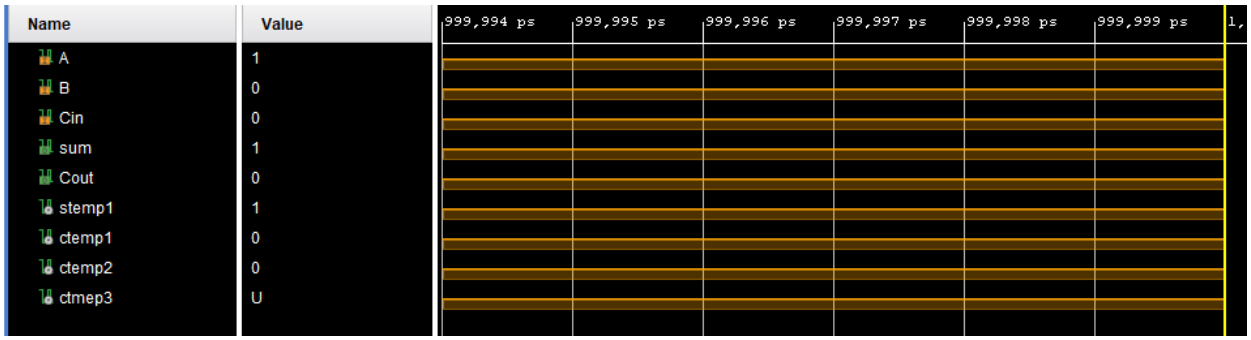
Not 2) Here are the results from the Not 2 Gate with the input A = 0. The outP is therefore 1.



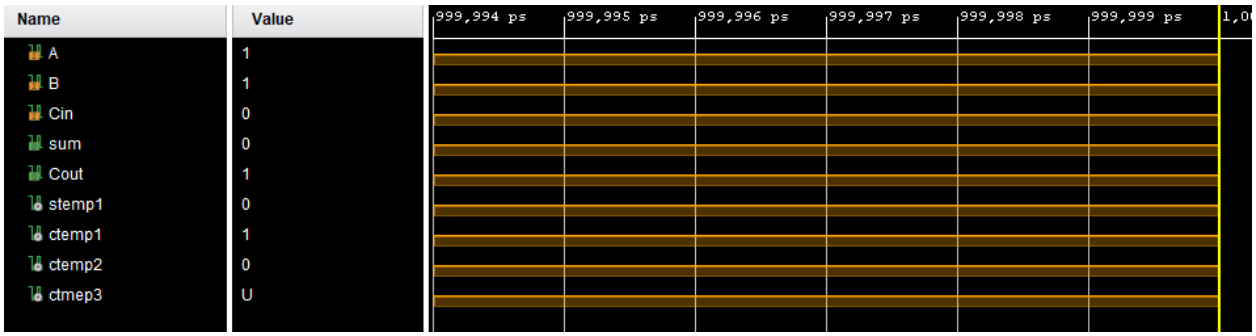
MultiPlexer) Here is the multiplexer with the inputs of X = 1 . Y = 0, Select = 1. Which gives an output of 0.



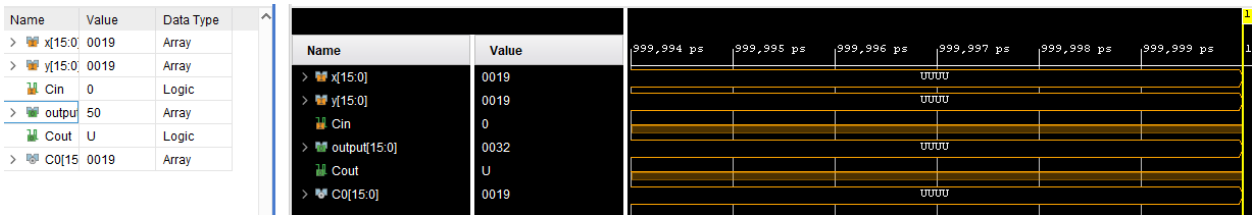
MultiPlexer) Here is the multiplexer with the inputs of X = 1 . Y = 1, Select = 1. Which gives an output of 1.



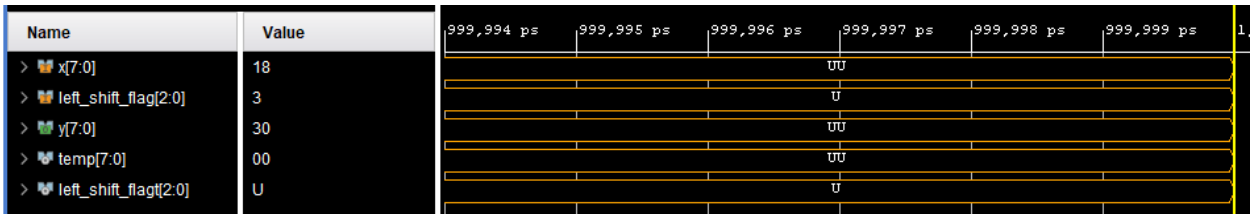
FullAdder) This is with the Inputs A =1 B = 0 and Cin = 0. Which gives the outputs of Sum = 1 and Cout = 0.



FullAdder) This is with the Inputs A =1 B = 1 and Cin = 0. Which gives the outputs of Sum = 0 and Cout = 1.

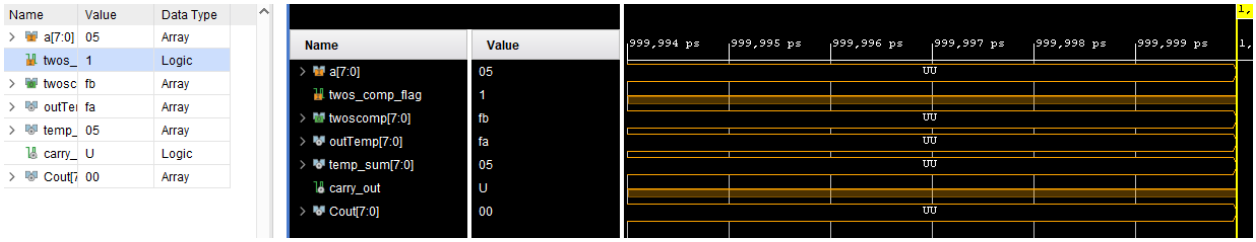


16 Bit FullAdder) Here is the results from my 16 bit Full Adder. Here we can see my X and Y inputs are both 19 (hex) or 25 Decimal. My output is 32 (Hex) or 50 decimal. On the left most side you can see me changing the output to read in decimal which is 50.

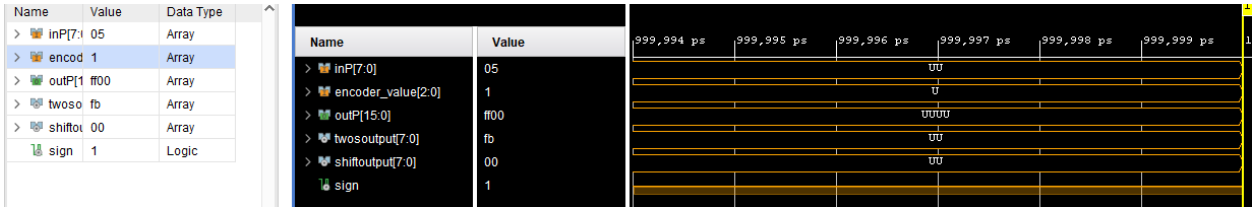


Left Shift Results) Here is the results from the left shift. Here we can see that in input X = 18 Hex. The Left Shift Flag is also 3. The output is then 30 in Hex.

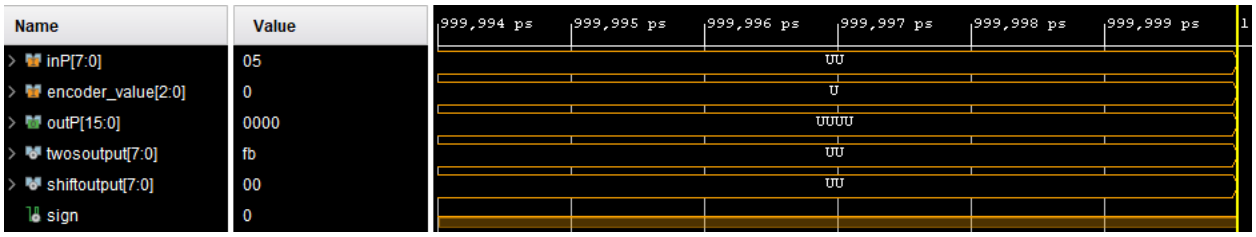
Left Shift Results) Here are some more results if the Left shift input X is equal to 18 Hex and the Left shift is 1. This will give an output Y as 00.



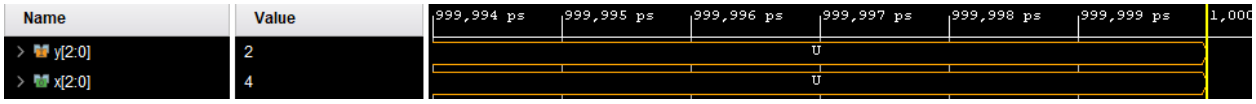
Twos Comp Result) Here is the Results from my 2s comp code. With an input of 5 and the flag of 1 the output is FB which is correct.



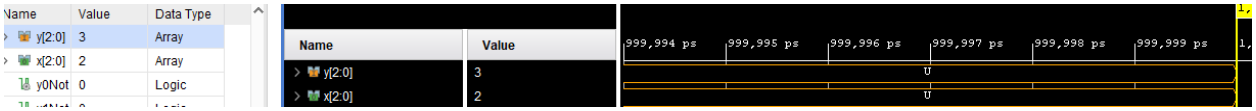
Partial Product 1 Results) I will only be showing 1 PP as the rest are the same minus some basic shifting. Here we can see that with the input of 5 and with the encoder value being negative -1 we get an output of Fb. The outP says FF but this is updated within the booth multiplier.



Partial Product 1 Results) Here is another example if the encoder value is 0 then the partial product will also be zero. This is with the inputs 5 and 0 with the output being 0.



Booth Decoder) Here we have an input of 2, and an output of 4. Or -2.



Booth Decoder) Here we have an input of 3 with an output of 2. 0r 1.

Appendix (Code):

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

entity sixteenFullAdder is -- Takes in 2 16 bit vectors
  Port (x : in std_logic_vector(15 downto 0);
        y : in std_logic_vector(15 downto 0);
        Cin : in std_logic; -- Cin
        output : out std_logic_vector(15 downto 0); -- Output
        Cout : out std_logic); -- Carry Out
end sixteenFullAdder;

architecture Behavioral of sixteenFullAdder is -- It calls the full adder
component FullAdder is
  Port (A, B, Cin : in std_logic;
        sum, Cout : out std_logic);
end component;

signal C0 : std_logic_vector(15 downto 0); -- Temp Signal for the Carry out
begin

U1:  FullAdder port map(x(0), y(0), '0', output(0), C0(0));
U2:  FullAdder port map(x(1), y(1), C0(0), output(1), C0(1));
U3:  FullAdder port map(x(2), y(2), C0(1), output(2), C0(2));
U4:  FullAdder port map(x(3), y(3), C0(2), output(3), C0(3));
U5: FullAdder port map(x(4), y(4), C0(3), output(4), C0(4));
U6:  FullAdder port map(x(5), y(5), C0(4), output(5), C0(5)); -- the port mapping This gets called 16 times
U7:  FullAdder port map(x(6), y(6), C0(5), output(6), C0(6)); -- It is saved to the output vector
U8:  FullAdder port map(x(7), y(7), C0(6), output(7), C0(7));
U9:  FullAdder port map(x(8), y(8), C0(7), output(8), C0(8));
U10: FullAdder port map(x(9), y(9), C0(8), output(9), C0(9));
U11: FullAdder port map(x(10), y(10), C0(9), output(10), C0(10));
U12: FullAdder port map(x(11), y(11), C0(10), output(11), C0(11));
U13: FullAdder port map(x(12), y(12), C0(11), output(12), C0(12));
U14: FullAdder port map(x(13), y(13), C0(12), output(13), C0(13));
U15: FullAdder port map(x(14), y(14), C0(13), output(14), C0(14));
U16: FullAdder port map(x(15), y(15), C0(14), output(15), C0(15));

end Behavioral;
```

Full Adder 16 bit Code

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

entity AND2 is
  Port (A, B : in std_logic; -- This has a input of A and B
        C : out std_logic); -- Output C
end AND2;

architecture Behavioral of AND2 is -- Used in FullAdder and other Future modules

begin

C <= A and B; -- All the logic needed
end Behavioral;
```

And 2 Code

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


entity AND3ins is -- This is an And gate with 3 inputs
    Port(A, B, C : in std_logic;
         outP : out std_logic); -- Output
end AND3ins;

architecture Behavioral of AND3ins is
begin
    outP <= (a and (b and c)); -- A and B and C then OutP
end Behavioral;
```

And3 code

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


entity Encoder is
Port ( y : in STD_LOGIC_VECTOR (2 downto 0); -- Takes in a 3 bit in
           x : out STD_LOGIC_VECTOR (2 downto 0));  -- out is a 3 bit out
end Encoder;

architecture Behavioral of Encoder is
component NOT2 -- calls the not gate
    Port (A : in std_logic;
    outP : out std_logic);
end component;

component AND3ins -- Calls 3 ins And
    Port(A, B, C : in std_logic;
     outP : out std_logic);
end component;

component OR2 -- Calls Or
  Port (A, B : in std_logic;
  C : out std_logic);
end component;

component XOR2 -- Calls Xor
  Port (A, B : in std_logic;
  C : out std_logic);
end component;

signal y0Not, y1Not, y2Not : std_logic; -- Temp signals for port mapping
signal temp1, temp2 : std_logic;
begin
x(0) <= y(2);  -- This is the negative Flag
U1 : NOT2 port map(y(0), y0Not);
U2 : NOT2 port map(y(1), y1Not);
U3 : NOT2 port map(y(2), y2Not);
U4 : AND3ins port map(y2Not, y(1), y(0), temp1);
U5 : AND3ins port map(y(2), y1Not, y0Not, temp2);
U6 : OR2 port map(temp1, temp2, x(1));
U7 : XOR2 port map(y(1), y(0), x(2));

-- This truth Table was found in the Slides and online
-- The truth table was found in my research and is shown above
end Behavioral;
```

Encoder code

```
entity twosComplement is
  port (
    a        : in  std_logic_vector(7 downto 0); -- Has inputs A
    twoscomp : out std_logic_vector(7 downto 0)
  );
end entity twosComplement;

architecture structural of twosComplement is

component NOT2 is -- Basic not Gate
  Port (A : in std_logic;
  outP  : out std_logic);
end component;

component FullAdder is
  Port (A, B, Cin : in std_logic;
        sum, Cout : out std_logic);
end component;


  signal outTemp      : std_logic_vector(7 downto 0); -- Used as temp sums
  signal Cout : std_logic_vector ( 7 downto 0);

begin

  U1: NOT2 port map(a(0) , outTemp(0));
   U2: NOT2 port map(a(1) , outTemp(1));
    U3: NOT2 port map(a(2) , outTemp(2)); -- 2s Comp is 8 nots with 1 added to the end
     U4: NOT2 port map(a(3) , outTemp(3));
      U5: NOT2 port map(a(4) , outTemp(4));
       U6: NOT2 port map(a(5) , outTemp(5));
        U7: NOT2 port map(a(6) , outTemp(6));
         U8: NOT2 port map(a(7) , outTemp(7));

U9 : FullAdder port map ( outTemp(0) , '1' , '0' , twoscomp(0) , Cout(0));
U10 : FullAdder port map ( outTemp(1) ,'0', Cout(0) , twoscomp(1) , Cout(1)); -- Nots with FAdder
U11 : FullAdder port map ( outTemp(2) , '0' , Cout(1) , twoscomp(2) , Cout(2));
U12 : FullAdder port map ( outTemp(3) , '0' , Cout(2) , twoscomp(3) , Cout(3));
U13 : FullAdder port map ( outTemp(4) , '0' , Cout(3) , twoscomp(4) , Cout(4));
U14 : FullAdder port map ( outTemp(5) , '0' , Cout(4) , twoscomp(5) , Cout(5));
U15 : FullAdder port map ( outTemp(6) , '0' , Cout(5) , twoscomp(6) , Cout(6));
U16 : FullAdder port map ( outTemp(7) , '0', Cout(6) , twoscomp(7) , Cout(7));

end architecture structural;
```

Twos Comp Code

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

entity Booth_Multiplier is
  Port (RegA : in std_logic_vector(7 downto 0); -- Reg A
        RegB : in std_logic_vector(7 downto 0); -- Reg B
        Clk: in std_logic; -- Clock for process
        LoadFlag : in std_logic; -- L flag
        EndFlag :inout std_logic; -- E Flag
        Regz : out std_logic_vector(15 downto 0)); -- output
end Booth_Multiplier;

architecture Behavioral of Booth_Multiplier is -- Calls the encoder
component Encoder
Port ( y : in std_logic_vector (2 downto 0);
           x : out std_logic_vector (2 downto 0));
end component;

component PP1 -- Calls the PP1 - PP4
Port (inP : in std_logic_vector(7 downto 0);
        encoder_value : in std_logic_vector(2 downto 0);
        outP : out std_logic_vector(15 downto 0));
end component;

component PP2
Port (inP : in std_logic_vector(7 downto 0);
        encoder_value : in std_logic_vector(2 downto 0);
        outP : out std_logic_vector(15 downto 0));
end component;

component PP3
Port  (inP : in std_logic_vector(7 downto 0);
        encoder_value : in std_logic_vector(2 downto 0);
        outP : out std_logic_vector(15 downto 0));
end component;

component PP4
Port (inP : in std_logic_vector(7 downto 0);
        encoder_value : in std_logic_vector(2 downto 0);
        outP : out std_logic_vector(15 downto 0));
end component;
```

Booth Multi Code 1)

```
component sixteenFullAdder -- Calls 16FullAdder to add together the 4 PPs to A
Port (x : in std_logic_vector(15 downto 0);
        y : in std_logic_vector(15 downto 0);
        Cin : in std_logic;
        output : out std_logic_vector(15 downto 0);
        Cout : out std_logic);
end component;


signal ret1, ret2, ret3, ret4 : std_logic_vector(2 downto 0); -- My returns of these decoder values
signal tempA, tempb : std_logic_vector(7 downto 0); -- Temps for reg a and b
signal partTemp1, partTemp2, partTemp3, partTemp4 : std_logic_vector(15 downto 0); -- more temps for port map
signal decode1, decode2, decode3, decode4: std_logic_vector(2 downto 0); -- My variables to pass into the decoder
signal Cout: std_logic; -- cout
signal Final : std_logic_vector(15 downto 0); -- This is the final output which is saved into ZReg
signal FTemp1, Ftemp2 : std_logic_vector(15 downto 0); -- Temps for the adder


begin

process(Clk , LoadFlag) -- If clk then load regs
begin

    if(Clk'event and LoadFlag = '1') then
    TempB <= RegB;
    TempA <= RegA;

decode1(2 downto 1) <= TempB(1 downto 0);
decode1(0) <= '0';

decode2 <= TempB(3 downto 1);
decode3 <= TempB(5 downto 3);
decode4 <= TempB(7 downto 5);
end if;

end process;

U1: Encoder port map(decode1, ret1);
U2: Encoder port map(decode2, ret2); -- encoders the PP
U3: Encoder port map(decode3, ret3);
U4: Encoder port map(decode4, ret4);
```

Booth Multi Code 2)

```
begin

process(Clk , LoadFlag) -- If clk then load regs
begin

    if(Clk'event and LoadFlag = '1') then
    TempB <= RegB;
    TempA <= RegA;

decode1(2 downto 1) <= TempB(1 downto 0);
decode1(0) <= '0';

decode2 <= TempB(3 downto 1);
decode3 <= TempB(5 downto 3);
decode4 <= TempB(7 downto 5);
end if;

end process;

U1: Encoder port map(decode1, ret1);
U2: Encoder port map(decode2, ret2); -- encoders the PP
U3: Encoder port map(decode3, ret3);
U4: Encoder port map(decode4, ret4);

U5: PP1 port map(TempA, ret1, partTemp1);
U6:  PP2 port map(TempA, ret2, partTemp2); -- Creates the PPs
U7:  PP3 port map(TempA, ret3, partTemp3);
U8:  PP4 port map(TempA, ret4, partTemp4);

U9: sixteenFullAdder port map(partTemp1, partTemp2, '0', Ftemp1, Cout); -- Adds together the 4 PPs
U10: sixteenFullAdder port map(Ftemp1, partTemp3, '0', Ftemp2, Cout);
U11: sixteenFullAdder port map(Ftemp2, partTemp4, '0', Final, Cout);

process(Clk) -- If load == 0 then fill in RegZ
begin
    if(LoadFlag = '0') then
    EndFlag <= '1' ;
RegZ <= Final;
end if;
end process;
end Behavioral;
```

Booth Multi Code 3

```
entity Left_Shift is
  Port (x : in std_logic_vector(7 downto 0); -- Takes in a 8 bit x and a 3 bit y for the # of shifts
        left_shift_flag : in std_logic_vector(2 downto 0);
        y : out std_logic_vector(7 downto 0)); -- Outputs a 8 bit
end Left_Shift;

architecture Behavioral of Left_Shift is
component multiplexer -- Calls a multiplxer for high and low
Port (x, y, sel : in std_logic;
        output : out std_logic);
end component;

signal temp : std_logic_vector(7 downto 0); -- Temp signals
signal flagTemp : std_logic_vector(2 downto 0); -- Temp Flag

begin

-- Calls the Multi based on the FlagTemp
-- the MultiPlexer truth table is
  -- (0,0,0) And (0,0,1) and (0,1,1) and (1,0,0) -> 0
  -- ( 0,1,0 ) and (1,0,1 ) and ( 1,1,0) and (1,1,1) ->1
U1: multiplexer port map('0', x(0), flagTemp(2), temp(0));
U2: multiplexer port map('0', x(1), flagTemp(2), temp(1));
U3: multiplexer port map('0', x(2), flagTemp(2), temp(2));
U4: multiplexer port map('0', x(3), flagTemp(2), temp(3));
U5: multiplexer port map('0', x(4), flagTemp(2), temp(4));
U6: multiplexer port map('0', x(5), flagTemp(2), temp(5));
U7: multiplexer port map('0', x(6), flagTemp(2), temp(6));
U8: multiplexer port map('0', x(7), flagTemp(2), temp(7));

--Calls the Multi based on the FlagTemp
  -- Follows the truth table from above
U9: multiplexer port map( temp(0), '0', flagTemp(1), y(0));
U10: multiplexer port map(temp(1), x(0), flagTemp(1), y(1));
U11: multiplexer port map(temp(2), x(1), flagTemp(1), y(2));
U12: multiplexer port map(temp(3), x(2), flagTemp(1), y(3));
U13: multiplexer port map(temp(4), x(3), flagTemp(1), y(4));
U14: multiplexer port map(temp(5), x(4), flagTemp(1), y(5));
U15: multiplexer port map(temp(6), x(5), flagTemp(1), y(6));
U16: multiplexer port map(temp(7), x(6), flagTemp(1), y(7));

end Behavioral;
```

Left Shift Code

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

entity NOT2 is -- Basic not Gate
  Port (A : in std_logic;
  outP  : out std_logic);
end NOT2;

architecture Behavioral of NOT2 is
begin
    outP <= not(a);
end Behavioral;
```

Not 2 Code

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

entity OR2 is
  Port (A, B : in std_logic;
  C : out std_logic);
end OR2;

architecture structural of OR2 is
begin
    C <= a or b ;
end structural;
```

Or2 Code

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

entity PP1 is
  Port (inP : in std_logic_vector(7 downto 0); -- takes in a 8 bit binary array
        encoder : in std_logic_vector(2 downto 0); -- decoder value
        outP : out std_logic_vector(15 downto 0)); -- spits out a shifter
end PP1;

architecture Behavioral of PP1 is
component leftShift
Port (x : in std_logic_vector(7 downto 0);-- takes in a 8 bit binary array
       flag : in std_logic_vector(2 downto 0); -- Shifter value
        y : out std_logic_vector(7 downto 0));-- Spits out a 8 bit binary array
end component;

component twosComplement -- Calls these Functions
port (
    a : in  std_logic_vector(7 downto 0);
    twoscomp : out std_logic_vector(7 downto 0)
  );
end component;

signal twosoutput : std_logic_vector(7 downto 0); -- Temps
signal shiftoutput : std_logic_vector(7 downto 0);
signal sign : std_logic;

begin

U1: twosComplement port map(inP, twosoutput); -- Calls 2 comp
sign <= encoder(2); -- Sign is the high bit which then fills the outP
U2: leftShift port map(twosoutput, encoder, shiftoutput); -- Then shifts


outP(7 downto 0) <= shiftoutput;
outP(15 downto 8) <= (sign, sign, sign, sign, sign, sign, sign, sign);

end Behavioral;
```

PP1 code

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

entity PP2 is
  Port (inP : in std_logic_vector(7 downto 0);
        encoder : in std_logic_vector(2 downto 0); -- Gets an 8 bit input and a 3 bit in with a 16 bit out
        outP : out std_logic_vector(15 downto 0));
end PP2;

architecture Behavioral of PP2 is
component leftShift -- These are the functions used
Port (x : in std_logic_vector(7 downto 0);
        flag : in std_logic_vector(2 downto 0);
        y : out std_logic_vector(7 downto 0));
end component;

component twosComplement
port (
    a        : in  std_logic_vector(7 downto 0);
    twoscomp : out std_logic_vector(7 downto 0)
  );
end component;

signal twosoutput : std_logic_vector(7 downto 0);
signal shiftoutput : std_logic_vector(7 downto 0);
signal sign : std_logic;

begin

U1: twosComplement port map(inP, twosoutput); -- First it 2 comps then it shifts
sign <= encoder(2);
U2: leftShift port map(twosoutput, encoder, shiftoutput);


outP(1 downto 0) <= ('0', '0'); -- First 2 always 0
outP(9 downto 2) <= shiftoutput;
outP(15 downto 10) <= (sign, sign, sign, sign, sign, sign); -- Then it fills based on sign
end Behavioral;
```

PP2 Code

```vhdl

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity PP3 is
  Port (inP : in std_logic_vector(7 downto 0); -- has a 8 bit input and a 3 bit in
        encoder : in std_logic_vector(2 downto 0);
        outP : out std_logic_vector(15 downto 0)); -- 16 bit out
end PP3;

architecture Behavioral of PP3 is
component leftShift -- Functions it will call
Port (x : in std_logic_vector(7 downto 0);
        flag : in std_logic_vector(2 downto 0);
        y : out std_logic_vector(7 downto 0));
end component;


component twosComplement
port (
    a        : in  std_logic_vector(7 downto 0);
    twoscomp : out std_logic_vector(7 downto 0)
  );
end component;

signal twosoutput : std_logic_vector(7 downto 0);
signal shiftoutput : std_logic_vector(7 downto 0);
signal sign : std_logic;

begin
U1: twosComplement port map(inP,  twosoutput); -- First 2 Comp then Shift
sign <= encoder(2);
U2: leftShift port map(twosoutput, encoder, shiftoutput);



outP(3 downto 0) <= ('0', '0','0', '0'); -- First 4 always zero because of shift left twice
outP(11 downto 4) <= shiftoutput;
outp(15 downto 12) <= (sign, sign, sign, sign); -- Fills based on sign

end Behavioral;
```

PP3 Code

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


entity PP4 is
  Port (inP : in std_logic_vector(7 downto 0);
        encoder: in std_logic_vector(2 downto 0);
        outp : out std_logic_vector(15 downto 0));
end PP4;

architecture Behavioral of PP4 is
component leftShift
Port (x : in std_logic_vector(7 downto 0); -- has a 8 bit in
        flag : in std_logic_vector(2 downto 0); -- 3 bit in
        y : out std_logic_vector(7 downto 0)); -- 16 bit out
end component;

component twosComplement -- functinos it will Call
port (
    a        : in  std_logic_vector(7 downto 0);
    twoscomp : out std_logic_vector(7 downto 0)
  );
end component;

signal twosoutput : std_logic_vector(7 downto 0);
signal shiftoutput : std_logic_vector(7 downto 0);
signal sign : std_logic;

begin
U1: twosComplement port map(inP,  twosoutput); -- first 2 comp then shift
sign <= encoder(2);
U2: leftShift port map(twosoutput, encoder, shiftoutput);



outP(5 downto 0) <= ('0', '0','0', '0','0', '0'); -- First 6 always zero

outP(13 downto 6) <= shiftoutput;
outP(15 downto 14) <= (sign, sign); -- Fill with Sign

end Behavioral;
```

PP4 Code

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

entity XOR2 is
  Port (A, B : in std_logic;
  C : out std_logic); -- Basic Xor Gate
end XOR2;

architecture structural of XOR2 is
begin
  C <= a xor b;
end structural;
```

Xor2 Code

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

use IEEE.NUMERIC_STD.ALL;


entity FullAdder is
  Port (A, B, Cin : in std_logic; -- 3 inputs with 2 outs being Sum and carry
        sum, Cout : out std_logic); -- Basic full Adder from lab 1
end FullAdder;

architecture Behavioral of FullAdder is

component XOR2 -- Gates it will call
  Port (A, B : in std_logic;
  C : out std_logic);
end component;

component OR2
  Port (A , B : in std_logic;
  C : out std_logic);
end component;

component AND2
    Port(A , B : in std_logic;
         C : out std_logic);
end component;

  signal stemp1, ctemp1, ctemp2, ctmep3 : std_logic; -- Temps

begin
U1 : XOR2 port map(a, b, stemp1); -- This equation is from Lab 1
U2 : XOR2 port map(stemp1, cin, sum);
U3 : AND2 port map(a, b, ctemp1);
U4 : AND2 port map(stemp1, cin, ctemp2);
U5 : XOR2 port map(ctemp1, ctemp2, cout);

end Behavioral;
```

Full Adder Code

```
entity multiplexer is
  Port (x, y, sel : in std_logic;
        output : out std_logic);
end multiplexer;

architecture Behavioral of multiplexer is

component AND2
    Port(A, B : in std_logic;
         C : out std_logic);
end component;

component OR2
  Port (A, B: in std_logic;
  C : out std_logic);
end component;

component NOT2
    Port (A : in std_logic;
    outP : out std_logic);
end component;

signal temp1, temp2, temp3 : std_logic;
begin

U1: AND2 port map(sel, y, temp1);
U2: NOT2 port map(sel, temp2);
U3: AND2 port map(temp2, x, temp3);
U4: OR2 port map(temp1, temp3, output);

end Behavioral;
```

MultiPlexer Code

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

use IEEE.NUMERIC_STD.ALL;

entity twosComplement is
  port (
    a        : in  std_logic_vector(7 downto 0); -- Has inputs A
    twoscomp : out std_logic_vector(7 downto 0)
  );
end entity twosComplement;

architecture structural of twosComplement is

component NOT2 is -- Basic not Gate
  Port (A : in std_logic;
  outP  : out std_logic);
end component;

component FullAdder is
  Port (A, B, Cin : in std_logic;
        sum, Cout : out std_logic);
end component;


  signal outTemp      : std_logic_vector(7 downto 0); -- Used as temp sums
  signal Cout : std_logic_vector ( 7 downto 0);


begin

  U1: NOT2 port map(a(0) , outTemp(0));
   U2: NOT2 port map(a(1) , outTemp(1));
    U3: NOT2 port map(a(2) , outTemp(2)); -- 2s Comp is 8 nots with 1 added to the end
     U4: NOT2 port map(a(3) , outTemp(3));
      U5: NOT2 port map(a(4) , outTemp(4));
       U6: NOT2 port map(a(5) , outTemp(5));
        U7: NOT2 port map(a(6) , outTemp(6));
         U8: NOT2 port map(a(7) , outTemp(7));
```

2 comp part 1

```
begin

  U1: NOT2 port map(a(0) , outTemp(0));
   U2: NOT2 port map(a(1) , outTemp(1));
    U3: NOT2 port map(a(2) , outTemp(2)); -- 2s Comp is 8 nots with 1 added to the end
     U4: NOT2 port map(a(3) , outTemp(3));
      U5: NOT2 port map(a(4) , outTemp(4));
       U6: NOT2 port map(a(5) , outTemp(5));
        U7: NOT2 port map(a(6) , outTemp(6));
         U8: NOT2 port map(a(7) , outTemp(7));

U9 : FullAdder port map ( outTemp(0) , '1' , '0' , twoscomp(0) , Cout(0));
U10 : FullAdder port map ( outTemp(1) ,'0', Cout(0) , twoscomp(1) , Cout(1)); -- Nots with FAdder
U11 : FullAdder port map ( outTemp(2) , '0' , Cout(1) , twoscomp(2) , Cout(2));
U12 : FullAdder port map ( outTemp(3) , '0' , Cout(2) , twoscomp(3) , Cout(3));
U13 : FullAdder port map ( outTemp(4) , '0' , Cout(3) , twoscomp(4) , Cout(4));
U14 : FullAdder port map ( outTemp(5) , '0' , Cout(4) , twoscomp(5) , Cout(5));
U15 : FullAdder port map ( outTemp(6) , '0' , Cout(5) , twoscomp(6) , Cout(6));
U16 : FullAdder port map ( outTemp(7) , '0', Cout(6) , twoscomp(7) , Cout(7));

end architecture structural;
```

2 comp Part 2

References:

https://en-academic.com/dic.nsf/enwiki/786401

https://www.ijert.org/design-and-implementation-of-256256-booth-multiplier-and-its-applications#:~:text=The%20algorithm%20of%20booth%20multiplier,subtractor%20multiplier%20are%20more%20complex

https://www.javatpoint.com/booths-multiplication-algorithm-in-coa

https://go.gale.com/ps/i.do?id=GALE%7CA693137206&sid=googleScholar&v=2.1&it=r&linkaccess=abs&issn=2210142X&p=AONE&sw=w&userGroupName=anon%7E6ca95463#:~:text=Booth%20multiplication%20algorithm%20treats%20both,%2D4%20and%20radix%2D8

https://www.tutorialspoint.com/what-is-booth-multiplication-algorithm-in-computer-architecture#:~:text=The%20Booth%20multiplication%20algorithm%20defines,the%20study%20of%20computer%20architecture.

Multipliers PDF on Canvas given by Doctor Hoque.

https://www.geeksforgeeks.org/booths-multiplication-algorithm/

John, L. K. (2016). *Digital Systems design using Vhdl*. Cengage Learning.