

Eight Bit Booth Multiplier
ECE 4250
Final Project
Allison Yaeger
14244528

Table of Contents

Introduction.....	3
Program Breakdown.....	3
Design Details.....	9
Appendix.....	14
References.....	28

Introduction:

The goal of this program was to create an eight-bit signed multiplier through the use of the booth algorithm. Using the knowledge learned throughout the course of the semester we needed to build a program using only structural VHDL code. We were able to implement a lot of the stuff we used during the lab section such as the full Adder, but we also extended this further to create multiplexer as well as a D Flip flop. Both of these components were essential to the structure of our program. The Multiplexer was used to select the correct output of the booth encoding table, and we also used the full adder to implement an 8-bit addition component.

S2	S1	S0	Output	Operation
0	0	0	0	Multiplicand* 0
0	0	1	1	Multiplicand* 1
0	1	0	1	Multiplicand* 1
0	1	1	2	Multiplicand* 2
1	0	0	-2	Multiplicand* -2
1	0	1	-1	Multiplicand* -1
1	1	0	-1	Multiplicand* -1
1	1	1	0	Multiplicand* 0

Table I – Booth Encoding Table

Program Breakdown:

The base level components are the logical AND, OR, XOR, and NOT components. Each of this was implemented with a simple program that calls the logical command provided to you in the VHDL Language. From here the rest of our programs build off of these four gates to build the booth multiplier. You can see the results of the simulations from these four components in Figures I-IV below.

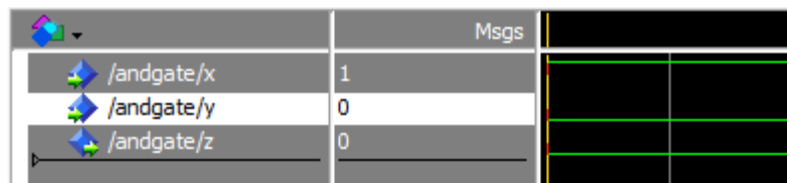
The next level of components that were implemented is the halfAdder, multiplexer, eight-bit inversion, and the d flip flop. The half adder will add the input and carry in bits and output the proper answer through the use of the XOR and AND gates in the base level, and you can see the circuit structure below in figure I. The 2:1 Multiplexer, as seen in Figure II below has three inputs X, Y, and the selection bit if the selection bit is 0 the output will be X and if 1 the output will be Y. This was also implemented through the use of AND, NOT, and OR gates. The Flip flop takes a set, reset, clk and d as inputs and outputs d if set is high during a clock tick event. Using a flip flop ensures that the instruction is happening on the rising edge of the clock. The eight-bit inverter simply calls the not gate 8 times and is implemented later in the code when

taking the two's complement of a number. The Simulation results for these three components can be seen below in Figures V-VII.

From The components listed above many of the main components used to implement the booth multiplier can be built. First the 8:1 Multiplexer was built from the 2:1 multiplexer. Calling the other multiplexer seven times using the different selection bits at the proper time yields the 8 but multiplexer. In Figure XI below we can see that when the selection bits are '000' the first input bit is selected. Another program was implemented that would shift the output to left one bit. This is built using the 2:1 multiplexer also and is used later in the program when getting information from the booth encoding table if we need to multiply our number by 2 or -2 this component is called. An 8-bit D Flip Flop was also created again by calling the original flip flop component, as you can see from the simulation results below in Figure X below we can see that during the clock high signal and when the set bit is high that the results are the same that is in the D input. The Full Adder is also in this level of the program as its implemented by calling the half adder twice. This program will add two bits while taking into account a carry in, this program can then be further used to create an addition function. Results from simulating this component can be seen in Figure XII below.

The next three components added were an eight-bit addition component, an 8:1 multiplexer that takes 8 bit vectors as inputs, and a 16-bit flip flop. The Eight-bit addition function is used for adding the first signal together four times after it has been shifted according to the booth encoding table. By calling the full adder eight times and feeding the carry outs back into the carry in's this program is accomplished, simulation results are below in Figure XIII. The multiplexer that takes in vectors is used for the booth encoding table. The three selection bits correspond to the three bits in our multiplier that we are currently using, and the correct output is selected. As you can see in Figure XIV below when the selection bits are set to 2 the third element inputted becomes the output. The 16-bit flip flop was implemented to use before setting our output product calling this will ensure that the output is set on the rising edge of the clock. This component calls the 8-bit flip flop twice, it could also be accomplished by calling the original flip flop 16 times.

The last level of this program combines all of these together to make the booth multiplier. Using two separate components. The first component takes in the multiplicand and the 3 bits of the multiplier and returns the selected number we need to add to our sum to make the product. This component is named Radix Numbers, the simulation for which can be seen in Figure XVI below. The other program used is called Configure Sum, this program takes the radix numbers and saves the last two bits of it into the respective place in the product term and adds the rest to the current sum. Each of these components is implemented four times in our get product function to create the multiplier. The simulations can be seen in Figures XVII and XVIII. The last level is called multiplier, this program then calls the get product function to do the booth multiplying and then updates the end and load bits through the use of a process.



		Msgs
/andgate/x	1	
/andgate/y	0	
/andgate/z	0	

Figure I -- And Gate Simulation

	Msgs	
/orgate/x	0	
/orgate/y	1	
/orgate/z	1	

Figure II -- Or Gate Simulation

	Msgs	
/xorgate/x	1	
/xorgate/y	1	
/xorgate/z	0	

Figure III -- Xor Gate Simulation

	Msgs	
/notgate/x	1	
/notgate/y	0	

Figure IV -- Inversion Simulation

	Msgs	
/halfadder/X	1	
/halfadder/Y	1	
/halfadder/Sum	0	
/halfadder/Cout	1	

Figure V -- Half Adder Simulation

	Msgs	
/multiplexer/X	1	
/multiplexer/Y	0	
/multiplexer/sel	1	
/multiplexer/Z	0	
/multiplexer/snot	0	
/multiplexer/hold1	0	
/multiplexer/hold2	0	

Figure VI -- Multiplexer Simulation

	Msgs	
/dff/d	1	
/dff/set	1	
/dff/rst	0	
/dff/dk	1	
/dff/q	1	

Figure VII -- D Flip Flop Simulation

Wave - Default		
	Msgs	
+ /not_eight_bit/invec	8'h15	8'h15
+ /not_eight_bit/outvec	8'hEA	8'hEA

Figure VIII -- Eight Bit Inversion Simulation

	Msgs	
+ /bsl/invec	8'h06	8'h06
+ /bsl/outvec	8'h0C	8'h0C

Figure IX -- Bit Shift Left Simulation

Wave - Default		
	Msgs	
/dff8bit/set	1	
/dff8bit/rst	0	
/dff8bit/clk	1	
+ /dff8bit/d	8'h15	8'h15
+ /dff8bit/q	8'h15	8'h15

Figure X -- 8 Bit D Flip Flop Simulation

	Msgs	
/multiplexer8/x0	1	
/multiplexer8/x1	0	
/multiplexer8/x2	0	
/multiplexer8/x3	0	
/multiplexer8/x4	U	
/multiplexer8/x5	0	
/multiplexer8/x6	0	
/multiplexer8/x7	0	
+ /multiplexer8/sel	3'h0	3'h0
/multiplexer8/out0	1	
/multiplexer8/h1	1	
/multiplexer8/h2	0	
/multiplexer8/h3	U	
/multiplexer8/h4	0	
/multiplexer8/h5	1	
/multiplexer8/h6	U	

Figure XI -- 8:1 Multiplexer

	Msgs	
/fulladder/X	1	
/fulladder/Y	0	
/fulladder/Cin	1	
/fulladder/Cout	1	
/fulladder/Sum	0	
/fulladder/hold1	1	
/fulladder/hold2	0	
/fulladder/hold3	1	

Figure XII -- Full Adder Simulation

	Msgs	
/eight_bit_adder/A	8'h10	8'h10
/eight_bit_adder/B	8'h06	8'h06
/eight_bit_adder/Ci	0	
/eight_bit_adder/S	8'h16	8'h16
/eight_bit_adder/Co	0	
/eight_bit_adder/C	7'h00	7'h00

Figure XIII -- Eight Bit Adder Simulation

	Msgs	
/multiplexerv/y0	8'h00	8'h00
/multiplexerv/y1	8'h01	8'h01
/multiplexerv/y2	8'h02	8'h02
/multiplexerv/y3	8'h03	8'h03
/multiplexerv/y4	8'h04	8'h04
/multiplexerv/y5	8'h05	8'h05
/multiplexerv/y6	8'h06	8'h06
/multiplexerv/y7	8'h07	8'h07
/multiplexerv/sel	3'h2	3'h2
/multiplexerv/output	8'h02	8'h02

Figure XIV -- Vector Multiplexer Simulation

	Msgs	
/dff16bit/set	1	
/dff16bit/rst	0	
/dff16bit/clk	1	
/dff16bit/d	16'h1234	16'h1234
/dff16bit/q	16'h1234	16'h1234

Figure XV -- 16 Bit D Flip Flop Simulation

Wave - Default			Msgs	
+ /raddixnumbers/input	3'h2		3'h2	
+ /raddixnumbers/num	8'h06		8'h06	
+ /raddixnumbers/out...	8'h06		8'h06	
/raddixnumbers/c0	0			
/raddixnumbers/c1	0			
+ /raddixnumbers/shif...	8'h0C		8'h0C	
+ /raddixnumbers/not...	8'hF9		8'hF9	
+ /raddixnumbers/nsn	8'hF3		8'hF3	
+ /raddixnumbers/nu...	8'h00		8'h00	

Figure XVI -- Radix Numbers Simulation

			Msgs	
+ /configsum/invec	8'h06		8'h06	
+ /configsum/cSum	8'h03		8'h03	
/configsum/x0	0			
/configsum/x1	0			
+ /configsum/nSum	8'h02		8'h02	
+ /configsum/product	2'h1		2'h1	
+ /configsum/hold	8'h09		8'h09	
/configsum/Cout1	0			
/configsum/msb	0			

Figure XVII -- Configure Sum Simulation

			Msgs	
+ /getprod/multiplicand	8'h10		8'h10	
+ /getprod/multiplier	8'h06		8'h06	
/getprod/ld	1			
/getprod/dr	0			
/getprod/clk	1			
+ /getprod/prod	16'h0060		16'hXXX	16'h0060
+ /getprod/zero	8'h00		8'h00	
+ /getprod/W	8'h00		8'h00	
+ /getprod/X	8'h00		8'h00	
+ /getprod/Y	8'h01		8'h01	
+ /getprod/hold1	8'h00		8'h00	
+ /getprod/hold2	8'h00		8'h00	
+ /getprod/hold3	8'h06		8'h06	
+ /getprod/hold4	8'h00		8'h00	
+ /getprod/buff	16'h0060		16'h0060	
+ /getprod/h0	8'h10		8'h10	
+ /getprod/h1	8'h06		8'h06	
+ /getprod/suminit	3'h0		3'h0	
+ /getprod/carries	8'h00		8'h00	

Figure XVIII -- Get Product Simulation

Design Details:

The flow of this design can be seen in Figure XIX below, the system uses flip flops to synchronize the timing. When a flip flop is used it ensures that the signal is sent on a clock tick event. Looking at Figure XXI below you can see that during the first clock pulse everything is loaded into the registers and all the operations performed, however the output product is not loaded into the register until the second clock pulse. This is because it goes through the 16-bit flip flop before it is loaded so it has to wait for the second clock pulse. So, during the first clock cycle everything in Figure XX is executed and the output is not stored in the product register until the clock signal goes high again. The output of this program can be seen in Figure XXI - XXIII below. As you can see the output of each of these different cases is as expected, when a negative number is entered we can see that negative output is received and similarly when two negative numbers the positive output is seen. This structural program is built from the AND, OR, XOR and NOT operations. Each of these was used to build smaller components and in turn those became larger components and our circuit was built. RTL schematics of the individual components can be seen in figures XXIV – XXX. This Design could also be implemented on the FPGA, in the previous labs we displayed the outputs using the 7-segment displays provided. Much of this code could be recycled and used for this but it would also require building a test bench. Since there are only 8 switches which we previously used for inputs, they cannot provide enough information to run the program. Building a test bench would allow for you to run multiple multiplication examples and output the different results on the 7-segment display in hexadecimal. Since there are only four 7 segment displays it is not enough room to display the largest possible number in decimal so it must be outputted in hexadecimal. In the top level of this program a process is also used, however all attempts to do this without a process were unsuccessful. While the multiplier does not need load or clear bits to run it is required in the project specifications and to update these a process really is the only way. This project demonstrates the knowledge learned throughout the semester of the VHDL language and working with FPGA's and how to create a signed multiplier using only AND, OR, NOT, and XOR gates.

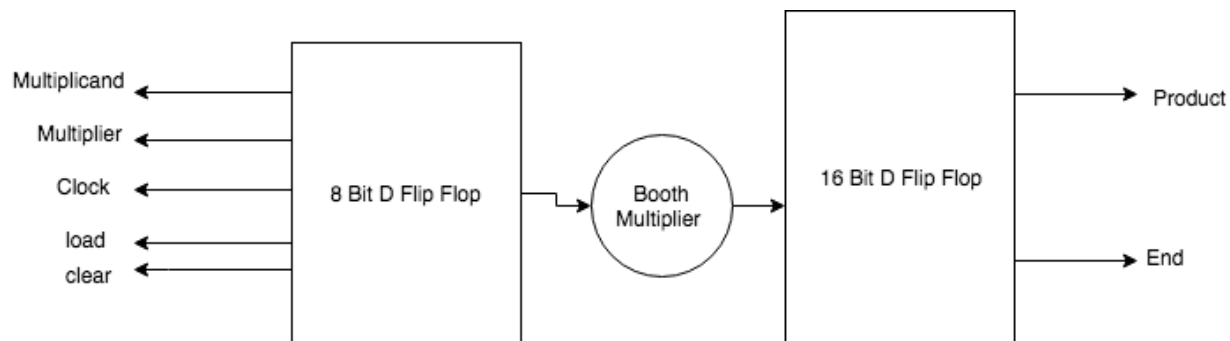


Figure XIX – Overall System Design

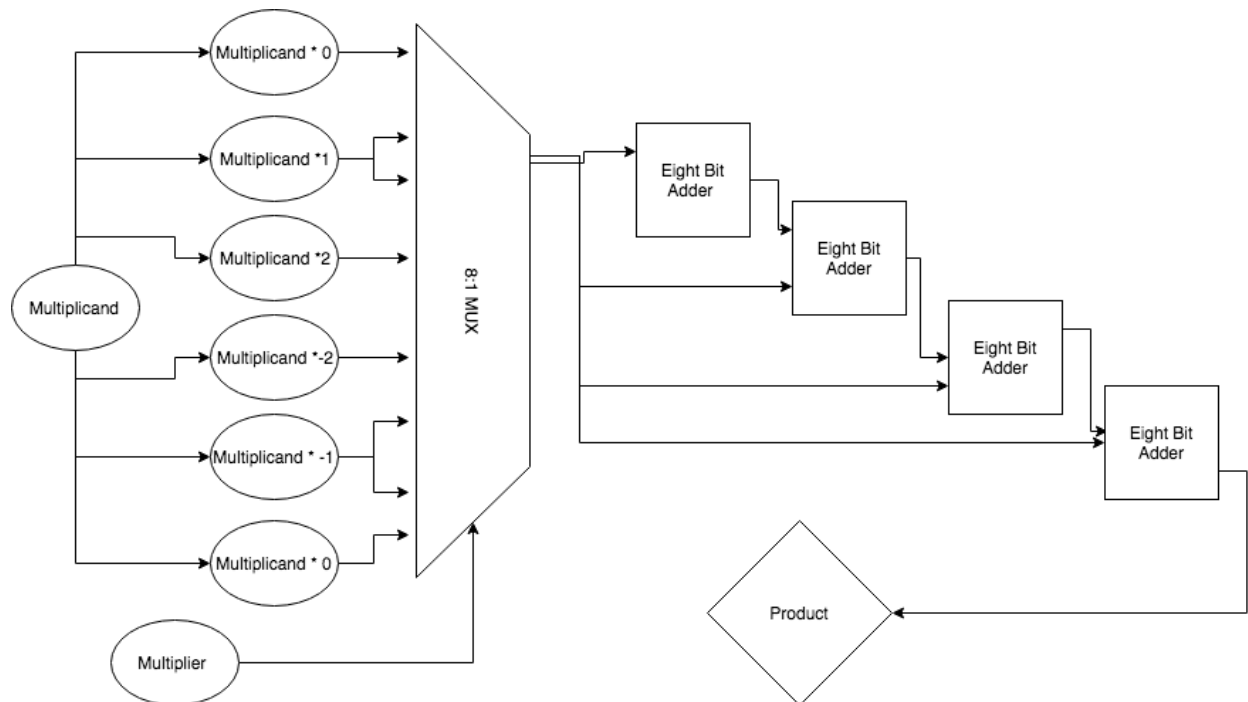


Figure XX – Detailed Booth Multiplier Design

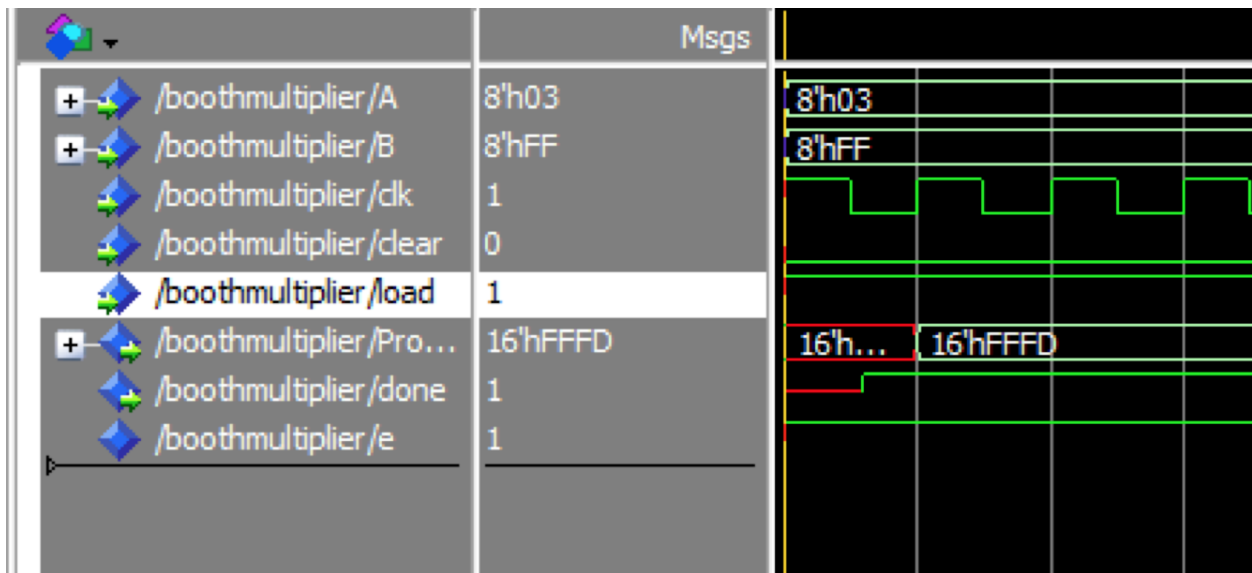


Figure XXI – Output Example (3) * (-1) = (-3)

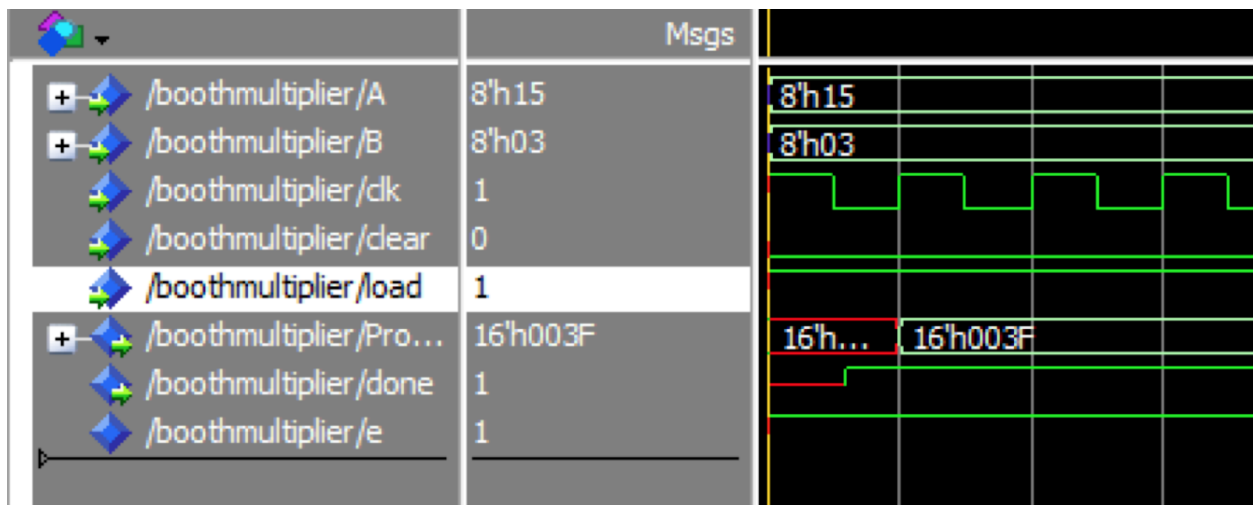


Figure XXII – Output Example $(15) * (3) = (3F)$

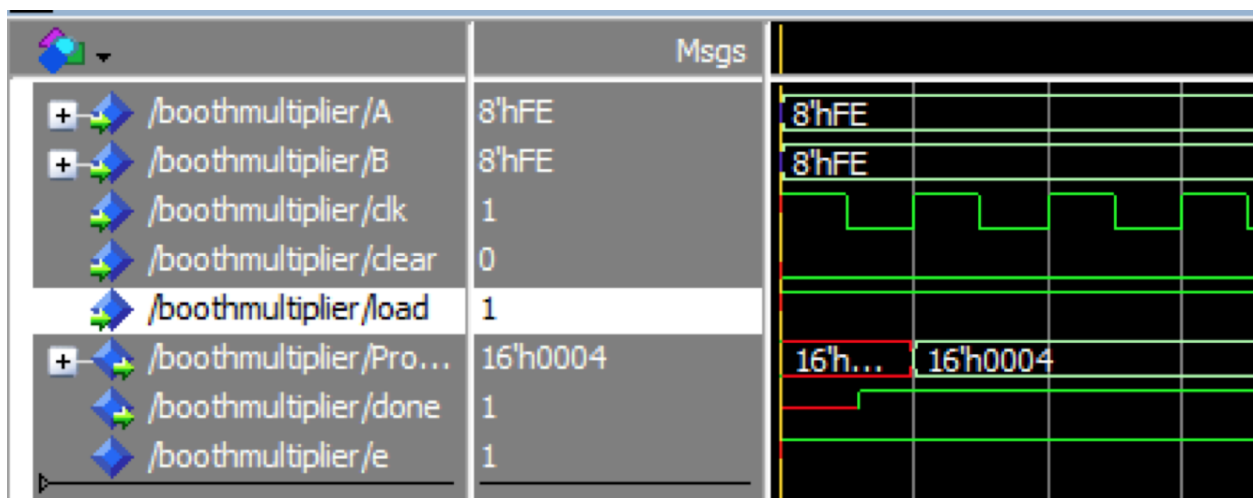


Figure XXIII– Output Example $(-2) * (-2) = (4)$

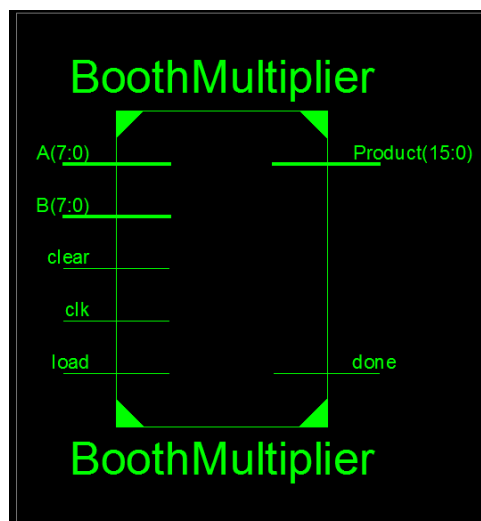


Figure XXIV – Booth Multiplier Circuit

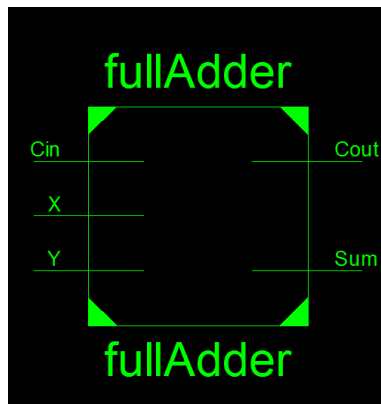


Figure XXV – Full Adder Circuit

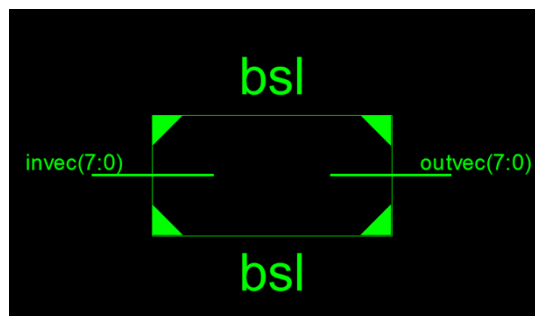


Figure XXVI – Bit Shifting Left Circuit

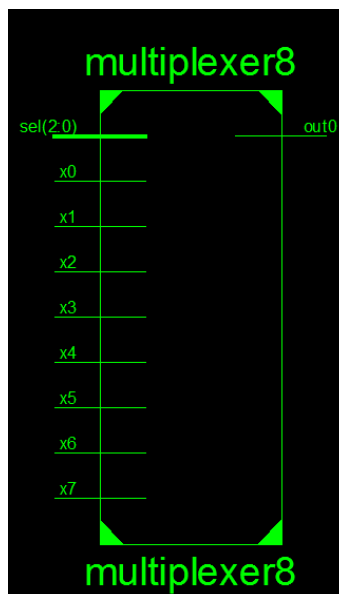


Figure XXVII – 8:1 Multiplexer Circuit

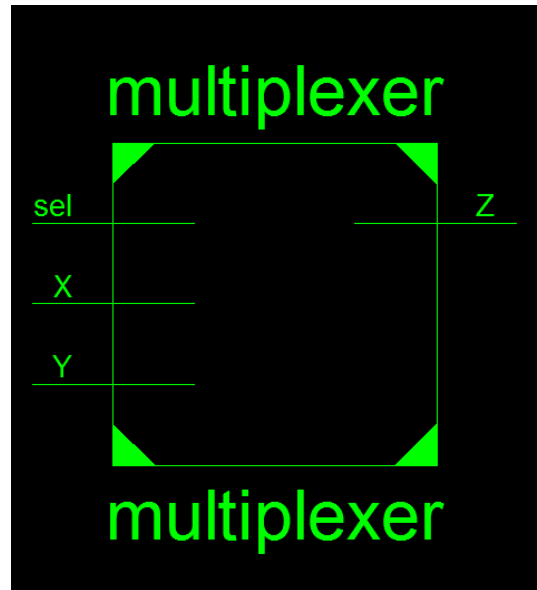


Figure XXVIII – 2:1 Multiplexer Circuit

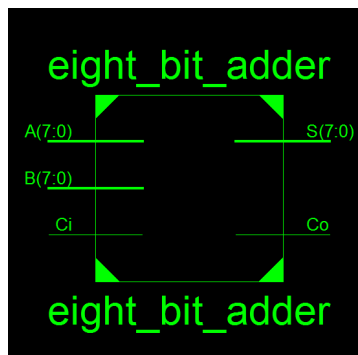


Figure XXIX – Eight Bit Adder Circuit

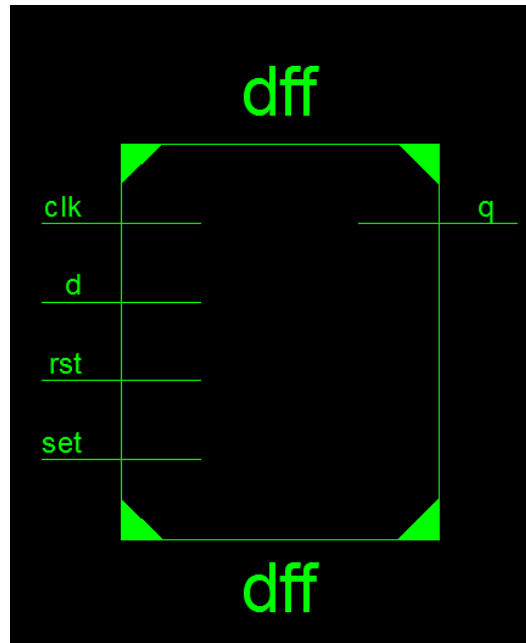


Figure XXX – D Flip Flop Circuit

Appendix:

andGate.vhd –

```
library IEEE;
use IEEE.std_logic_1164.all;

entity andGate is
    port(x, y: in std_logic;
         z: out std_logic);
end andGate;

architecture behavior of andGate is
begin
    z <= x and y;
end behavior;
```

orGate.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;

entity orGate is
    port(x, y: in std_logic;
         z: out std_logic);
end orGate;
```

architecture behavior of orGate is

```
begin
    z <= x or y;
end behavior;
```

xorGate.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity xorGate is
    port( x, y: in std_logic;
          z: out std_logic);
end xorGate;
architecture behavior of xorGate is
begin
    z <= x xor y;
end behavior;
```

notGate.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity notGate is
    port(x: in std_logic;
          y: out std_logic);
end notGate;
```

```
architecture behavior of notGate is
begin
    y <= not x;
end behavior;
```

halfAdder.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
```

```
entity halfAdder is
    port(X, Y: in std_logic;
          Sum, Cout: out std_logic);
end halfAdder;
```

```

architecture behavior of halfAdder is
--Xor gate component
component xorGate is
    port(X, Y: in std_logic;
          Z: out std_logic);
end component;
--and gate component
component andGate is
    port(X, Y: in std_logic;
          Z: out std_logic);
end component;

begin
    --X xor Y gives the proper Sum bit
xor1:  xorGate port map(X, Y, Sum);
    --X and Y gives the proper Cout bit
and1:  andGate port map(X, Y, Cout);

end behavior;

```

Multiplexer.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity multiplexer is
    port(X, Y: in std_logic;
          sel: in std_logic;
          Z: out std_logic);
end multiplexer;

architecture behavior of multiplexer is
--AND GATE
component andGate is
    port(X, Y: in std_logic;
          Z: out std_logic);
end component;
--OR GATE
component orGate is
    port(X, Y: in std_logic;
          Z: out std_logic);
end component;
--NOT GATE
component notGate is
    port(X: in std_logic;
          Y: out std_logic);

```



```

end component;

signal snot, hold1, hold2: std_logic;
begin

invert1:      notGate port map(sel, snot);      --get select not
and1:         andGate port map(snot, X, hold1); --and this and X our first bit
and2:         andGate port map(sel, Y, hold2);  --and sel and Y our second bit
or1:          orGate port map(hold1, hold2, Z);  --or both of these and store in Z

end behavior;

```

dff.vhd ---

```

library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
    port(d, set, rst, clk: in std_logic;
         q: out std_logic);
end dff;

architecture behavior of dff is

begin

    process(clk)
    begin
        if clk'event and clk = '1' then --on rising edge of the clock
            if set = '1' then --if the set variable is high
                if d = '1' then q <= '1'; --set q equal to d
                elsif d = '0' then q <= '0';
                end if;
            elsif rst = '1' then q <= '0'; --else q is zero if we want to reset
            end if;
        end if;
    end process;
end behavior;

```

FullAdder.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;

entity fullAdder is

```

```

        port(X, Y: in std_logic;
              Cin: in std_logic; --inputs
              Cout: out std_logic;
              Sum: out std_logic); --outputs
end fullAdder;

architecture behavior of fullAdder is
--OR component
component orGate is
    port(x, y: in std_logic;
          z: out std_logic);
end component;
--Half Adder Component
component halfAdder is
    port(X, Y: in std_logic;
          Sum, Cout: out std_logic);
end component;

signal hold1, hold2, hold3, hold4: std_logic;
begin
    --implement the half adder twice to make the full adder
    HA1: halfAdder port map(X, Y, hold1, hold2);
    HA2: halfAdder port map(hold1, Cin, Sum, hold3);
    OR1: orGate port map(hold2, hold3, Cout);

end behavior;

```

Dff8bit.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity dff8bit is
    port(set, rst, clk: std_logic;
          d: in std_logic_vector(7 downto 0);
          q: out std_logic_vector(7 downto 0));
end dff8bit;

architecture behavior of dff8bit is
--ADD REGULAR FLIP FLOP COMPONENT
component dff is
    port(d, set, rst, clk: std_logic;
          q: out std_logic);
end component;

begin

```

```

--use the one bit flip flop 8 times to implement the 8 bit flip flop
d0:    dff port map(d(0), set, rst, clk, q(0));
d1:    dff port map(d(1), set, rst, clk, q(1));
d2:    dff port map(d(2), set, rst, clk, q(2));
d3:    dff port map(d(3), set, rst, clk, q(3));
d4:    dff port map(d(4), set, rst, clk, q(4));
d5:    dff port map(d(5), set, rst, clk, q(5));
d6:    dff port map(d(6), set, rst, clk, q(6));
d7:    dff port map(d(7), set, rst, clk, q(7));

```

```
end behavior;
```

Multiplexer8.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

```

```

entity multiplexer8 is
    port(x0, x1, x2, x3, x4, x5, x6, x7: in std_logic;
          sel: in std_logic_vector(2 downto 0);
          out0: out std_logic);
end multiplexer8;

```

```

architecture behavior of multiplexer8 is
--2 by 1 mux component
component multiplexer is

```

```

    port(X, Y: in std_logic;
          sel: in std_logic;
          Z: out std_logic);
end component;

```

```
end component;
```

```
signal h1, h2, h3, h4, h5, h6: std_logic;
```

```
begin
```

```

--implement the 2:1 multiplexer seven times to get the 8:1
m0:    multiplexer port map(x0, x1, sel(2), h1);    --use sel(2) for the
m1:    multiplexer port map(x2, x3, sel(2), h2);    --top level
m2:    multiplexer port map(x4, x5, sel(2), h3);
m3:    multiplexer port map(x6, x7, sel(2), h4);
m4:    multiplexer port map(h1, h2, sel(1), h5);    --then using sel(1) with
m5:    multiplexer port map(h3, h4, sel(1), h6);    --the hold variables
m6:    multiplexer port map(h5, h6, sel(0), out0);  --finally use sel(0) for the last 2
holds

```

```
end behavior;
```

Bsl.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;

entity bsl is
    port(invec: in std_logic_vector(7 downto 0);
          outvec: out std_logic_vector(7 downto 0));
end bsl;

architecture behavior of bsl is
    --use multiplexer to shift bits
    component multiplexer is
        port(X, Y: in std_logic;
              sel: in std_logic;
              Z: out std_logic);
    end component;
    begin

        --use the 2:1 multiplexer to shift left by sending 0 as the select and the bit we want
        --in the X option then storing in the output one to the left
        mux0: multiplexer port map('0', invec(0), '0', outvec(0));
        mux1: multiplexer port map(invec(0), invec(1), '0', outvec(1));
        mux2: multiplexer port map(invec(1), invec(2), '0', outvec(2));
        mux3: multiplexer port map(invec(2), invec(3), '0', outvec(3));
        mux4: multiplexer port map(invec(3), invec(4), '0', outvec(4));
        mux5: multiplexer port map(invec(4), invec(5), '0', outvec(5));
        mux6: multiplexer port map(invec(5), invec(6), '0', outvec(6));
        mux7: multiplexer port map(invec(6), invec(7), '0', outvec(7));

    end behavior;
```

Not_eight_bit.vhd --

```
library IEEE;
use IEEE.std_logic_1164.all;

entity not_eight_bit is
    port(invec: in std_logic_vector(7 downto 0); --input vector to be inverted
          outvec: out std_logic_vector(7 downto 0)); -- inversion of the input
end not_eight_bit;

architecture behavior of not_eight_bit is
    --NOT GATE COMPONENT TO CALL
    component notGate is
        port( x: in std_logic;
```

```

        y: out std_logic);
end component;

begin
    --Call Not gate for each bit and store in the output
n0:    notGate port map(invec(0), outvec(0));
n1:    notGate port map(invec(1), outvec(1));
n2:    notGate port map(invec(2), outvec(2));
n3:    notGate port map(invec(3), outvec(3));
n4:    notGate port map(invec(4), outvec(4));
n5:    notGate port map(invec(5), outvec(5));
n6:    notGate port map(invec(6), outvec(6));
n7:    notGate port map(invec(7), outvec(7));

end behavior;

multiplexerV.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity multiplexerV is
    port(y0, y1, y2, y3, y4, y5, y6, y7: in std_logic_vector(7 downto 0);
         sel: in std_logic_vector(2 downto 0);
         output: out std_logic_vector(7 downto 0));
end multiplexerV;

architecture behavior of multiplexerV is
    --declare component of the 8:1 multiplexer
    component multiplexer8 is
        port(x0, x1, x2, x3, x4, x5, x6, x7: in std_logic;
             sel: in std_logic_vector(2 downto 0);
             out0: out std_logic);
    end component;

begin
    --call the 8:1 multiplexer 8 times for each index of the vectors
m80:    multiplexer8 port map(y0(0), y1(0), y2(0), y3(0), y4(0), y5(0), y6(0), y7(0), sel,
    output(0));
m81:    multiplexer8 port map(y0(1), y1(1), y2(1), y3(1), y4(1), y5(1), y6(1), y7(1), sel,
    output(1));
m82:    multiplexer8 port map(y0(2), y1(2), y2(2), y3(2), y4(2), y5(2), y6(2), y7(2), sel,
    output(2));
m83:    multiplexer8 port map(y0(3), y1(3), y2(3), y3(3), y4(3), y5(3), y6(3), y7(3), sel,
    output(3));

```

```

m84: multiplexer8 port map(y0(4), y1(4), y2(4), y3(4), y4(4), y5(4), y6(4), y7(4), sel,
output(4));
m85: multiplexer8 port map(y0(5), y1(5), y2(5), y3(5), y4(5), y5(5), y6(5), y7(5), sel,
output(5));
m86: multiplexer8 port map(y0(6), y1(6), y2(6), y3(6), y4(6), y5(6), y6(6), y7(6), sel,
output(6));
m87: multiplexer8 port map(y0(7), y1(7), y2(7), y3(7), y4(7), y5(7), y6(7), y7(7), sel,
output(7));

end behavior;

```

Eight_bit_adder.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity eight_bit_adder is                                --entity of our 8bit adder
    port(A, B: in std_logic_vector(7 downto 0);          --inputs
          Ci: in std_logic;                               --carry in
          S: out std_logic_vector(7 downto 0);           --sum
          Co: inout std_logic);                           --Carry out
end eight_bit_adder;                                    --end the entity

architecture behavior of eight_bit_adder is              --architecture
    --Full Adder component
    component fullAdder                                  --add fullAdder component
        port(X, Y: in std_logic;
              Cin: in std_logic;
              Cout, Sum: out std_logic);
    end component;

    signal C: std_logic_vector(7 downto 1);              --create signal for carry in and outs
    begin                                                  --begin the adding

        FA1: fullAdder port map(A(0), B(0), Ci, C(1), S(0));
        FA2: fullAdder port map(A(1), B(1), C(1), C(2), S(1));
        FA3: fullAdder port map(A(2), B(2), C(2), C(3), S(2));
        FA4: fullAdder port map(A(3), B(3), C(3), C(4), S(3));
        FA5: fullAdder port map(A(4), B(4), C(4), C(5), S(4));
        FA6: fullAdder port map(A(5), B(5), C(5), C(6), S(5));
        FA7: fullAdder port map(A(6), B(6), C(6), C(7), S(6));
        FA8: fullAdder port map(A(7), B(7), C(7), Co, S(7));

    end behavior;

```

Dff16bit.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity dff16bit is
    port(set, rst, clk: in std_logic;
          d: in std_logic_vector(15 downto 0);
          q: out std_logic_vector(15 downto 0));
end dff16bit;

architecture behavior of dff16bit is
    --ADD D 8 BIT FLIP FLOP COMP
    component dff8bit is
        port(set, rst, clk: std_logic;
              d: in std_logic_vector(7 downto 0);
              q: out std_logic_vector(7 downto 0));
    end component;

    begin
        --implement the 8 bit flip flop twice for the 16 bit
        dff0: dff8bit port map(set, rst, clk, d(7 downto 0), q(7 downto 0));
        dff1: dff8bit port map(set, rst, clk, d(15 downto 8), q(15 downto 8));

    end behavior;

```

raddixNumbers.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity raddixNumbers is
    port(input: in std_logic_vector(2 downto 0);          --three numbers to encode from table
          num: in std_logic_vector(7 downto 0);          --number that we want to add
          according to the table
          output: out std_logic_vector(7 downto 0);--store output number here
          c0, c1: out std_logic);          --output bits
end raddixNumbers;

architecture behavior of raddixNumbers is
    --BSL COMPONENT
    component bsl is
        port(invec: in std_logic_vector(7 downto 0);
              outvec: out std_logic_vector(7 downto 0));
    end component;

    --8 BIT INVERTER COMPONENT
    component not_eight_bit is
        port(invec: in std_logic_vector(7 downto 0);

```

```

        outvec: out std_logic_vector(7 downto 0));
end component;
--8:1 multiplexer
component multiplexer8 is
    port(x0, x1, x2, x3, x4, x5, x6, x7: in std_logic;
          sel: in std_logic_vector(2 downto 0);
          out0: out std_logic);
end component;
--MULTIPLEXER WITH VECTORS
component multiplexerV is
    port(y0, y1, y2, y3, y4, y5, y6, y7: in std_logic_vector(7 downto 0);
          sel: in std_logic_vector(2 downto 0);
          output: out std_logic_vector(7 downto 0));
end component;
--declare all signals we will need to use
signal shiftedNum, notNum, nsn, out0, sum1: std_logic_vector(7 downto 0);
signal numZero: std_logic_vector(7 downto 0);

    begin
        numZero <= "00000000";           --store all zeros in this number
step0:    bsl port map(num, shiftedNum);  --shift num left to get 2*num
step1:    not_eight_bit port map(num, notNum);  --invert number for 2's compliment
step2:    not_eight_bit port map(shiftedNum, nsn);--invert 2*num for 2's compliment

        --use the multiplexer to select the proper options using the input vector as the
selection bits
step3:    multiplexerV port map(numZero, nsn, num, notNum, num, notNum, shiftedNum,
numZero, input, output);
step4:    multiplexer8 port map('0', '1', '0', '1', '0', '1', '0', '0', input, c0);
step5:    multiplexer8 port map('0', notNum(7), num(7), notNum(7), num(7), notNum(7),
num(7), '0', input, c1);

end behavior;

```

configSum.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;

entity configSum is
    port(invec, cSum: in std_logic_vector(7 downto 0); --input and current Sum
          x0, x1: in std_logic;                       --carry in bits
          nSum: out std_logic_vector(7 downto 0);       --new sum output
          product: out std_logic_vector(1 downto 0)); --product output
end configSum;

```



```

architecture behavior of configSum is
--FULL ADDER COMPONENT
component fullAdder is
    port(X, Y: in std_logic;
          Cin: in std_logic; --inputs
          Cout: out std_logic;
          Sum: out std_logic); --outputs
end component;
--EIGHT BIT ADDER COMPONENT
component eight_bit_adder is
    port(A, B: in std_logic_vector(7 downto 0);          --inputs
          Ci: in std_logic;                             --carry in
          S: out std_logic_vector(7 downto 0); --sum
          Co: inout std_logic); --carry out
end component;

signal hold: std_logic_vector(7 downto 0);
signal Cout1, Cout2, msb: std_logic;

begin
    --first add our current sum and input vector
nSum1:      eight_bit_adder port map(cSum, invec, x0, hold, Cout1);
    --then use the full adder to to get the most significant bit of our vector
ncarry1:    fullAdder port map(cSum(7), x1, Cout1, Cout2, msb);
    --configure the new sum
    nSum(5 downto 0) <= hold(7 downto 2);
    nSum(7) <= msb;          --store msb
    nSum(6) <= msb;
    product <= hold(1 downto 0); --store the product

end behavior;

getProd.vhd --

use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity getProd is
    port(multiplicand, multiplier: in std_logic_vector(7 downto 0);
          ld, clr, clk: in std_logic;
          prod: out std_logic_vector(15 downto 0));
end getProd;

architecture behavior of getProd is
--D flip flop for 8 bit

```

```

component dff8bit is
    port(set, rst, clk: std_logic;
          d: in std_logic_vector(7 downto 0);
          q: out std_logic_vector(7 downto 0));
end component;
--D flip flop for 16 bit
component dff16bit is
    port(set, rst, clk: in std_logic;
          d: in std_logic_vector(15 downto 0);
          q: out std_logic_vector(15 downto 0));
end component;
--raddix numbers component
component raddixNumbers is
    port(input: in std_logic_vector(2 downto 0);
          num: in std_logic_vector(7 downto 0);
          -- Sumin: in std_logic_vector(7 downto 0);
          output: out std_logic_vector(7 downto 0);
          c0, c1: out std_logic;
          -- product: out std_logic_vector(1 downto 0));
end component;
--function to add new number to current sum and get product numbers
component configSum is
    port(invec, cSum: in std_logic_vector(7 downto 0);
          x0, x1: in std_logic;
          nSum: out std_logic_vector(7 downto 0);
          product: out std_logic_vector(1 downto 0));
end component;

--declare signals to use
signal zero, W, X, Y, hold1, hold2, hold3, hold4: std_logic_vector(7 downto 0);
signal buff: std_logic_vector(15 downto 0);
signal h0, h1: std_logic_vector(7 downto 0);
signal suminit: std_logic_vector(2 downto 0);
signal carries: std_logic_vector(7 downto 0);

begin
    --call 8 bit flip flop for both the multiplier and multiplicand
ff0: dff8bit port map(ld, clr, clk, multiplicand, h0);
ff1: dff8bit port map(ld, clr, clk, multiplier, h1);

    suminit <= h0(1 downto 0) & '0';    --store the first raddix number to encode
    zero <= "00000000";                --store zero here fpr the current sum

sadd0: raddixNumbers port map(suminit, h1, hold1, carries(0), carries(1));    --call raddix
numbers to get number to add to the current sum

```

```

cs0:    configSum port map( hold1, zero, carries(0), carries(1), W, buff(1 downto 0));    --call
configure the sum to add the two and begin storing product

sadd1: raddixNumbers port map(h0(3 downto 1), h1, hold2, carries(2), carries(3));
cs1:    configSum port map( hold2, W, carries(2), carries(3), X, buff(3 downto 2));

sadd2: raddixNumbers port map(h0(5 downto 3), h1, hold3, carries(4), carries(5));
cs2:    configSum port map( hold3, X, carries(4), carries(5), Y, buff(5 downto 4));

sadd3: raddixNumbers port map(h0(7 downto 5), h1, hold4, carries(6), carries(7));
cs3:    configSum port map( hold4, Y, carries(6), carries(7), buff(15 downto 8), buff(7 downto
6));
        --call the 16 bit flip flop on the buffer and store this in product
ff2:    dff16bit port map('1', clr, clk, buff, prod);

end behavior;

```

Boothmultiplier.vhd --

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity BoothMultiplier is
    port ( A, B: in std_logic_vector(7 downto 0); --input numbers to be multiplied
          clk, clear, load: in std_logic;          -- input bits
          Product: out std_logic_vector(15 downto 0); --output product
          done: out std_logic);                    --done flag goes to one when finished
end BoothMultiplier;

architecture Behavior of BoothMultiplier is
    --component that will get the final product
    component getProd is
        port(multiplicand, multiplier: in std_logic_vector(7 downto 0);
              ld, clr, clk: in std_logic;
              prod: out std_logic_vector(15 downto 0));
    end component;

    signal e: std_logic;    --signal to use to set the cone bit

begin
    --call function that will return the product into the appropriate vector
    GP1: getProd port map(A, B, load, clear, clk, product);

    process(load, clear, clk)

```

```

begin
    if clk'event and clk = '1' then --check for rising edge of clock
        if(load = '1' and clear = '0') then
            e <= '1';          --set e to one if we are loading
        else
            e <= '0';          --if we arent loading or are clearing set e to
0
        end if;
    end if;
end process;

process
begin
    wait for 60 ns;          --wait and then set done to e
    done <= e;
end process;
end Behavior;

```

References:

Booth Encoding. Geoff Knagge, 27, July 2010, <https://www.geoffknagge.com/fyp/booth.shtml>, Accessed May 7, 2018.

Half Adder Module in VHDL and Verilog. Nand Land, <https://www.nandland.com/vhdl/modules/module-half-adder.html>, Accessed May 7, 2018.

Multiplexers in VHDL. Starting Electronics, 24, December 2012. <https://startingelectronics.org/software/VHDL-CPLD-course/tut4-multiplexers/>, Accessed May 7, 2018.

D Flip Flops. Learn About Digital Electronics, 3, September 2017. <http://www.learnabout-electronics.org/Digital/dig53.php>, Accessed May 7, 2018