INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, NAGPUR.

ECL 303 : HARDWARE DESCRIPTION LANGUAGE

PROJECT REPORT ON –

IMPLEMENTATION OF MULTIPLICATION USING BOOTH'S ALGORITHM AND BIT PAIR RECODING ALGORITHM .

SUBMITTED BY :

NAME : PRATIK R. ADLE

ENROLLMENT NO. : BT17ECE034

CONTENTS

ACKNOWLEDGEMENT

I would like to acknowledge the guidance and mentorship provided by **Dr. Vipin Kamble** during the course of this semester, inspiring and motivating us throughout. He helped us conceive this project and provided me with solutions of various potential problems. Also I am very grateful to all my friends and family members who supported me throughout this venture.

ABSTRACT

Low power consumption and smaller area are some of the most important criteria for the fabrication of DSP systems and high performance systems. Optimizing the speed and area of the multiplier is a major design issue. However, area and speed are usually conflicting constraints so that improving speed results mostly in larger areas. In this project we try to determine the best solution to this problem by comparing a few multipliers.

This project presents an efficient implementation of high speed multiplier using the Booth Algorithm (Radix_2) and Bit Pair Recoding Algorithm (Radix_4) or modified Booth multiplier algorithm. In this project we compare the working of the two multiplier by implementing each of them separately .

The parallel multipliers like radix 2 and radix 4 modified booth multiplier does the computations using lesser adders and lesser iterative steps. As a result of which they occupy lessr space as compared to the serial multiplier. This a very important criteria because in the fabrication of chips and high performance system requires components which are as small as possible.

The low power consumption quality of booth multiplier makes it a preffered choice in designing different circuits . In this project we designed two different type of multipliers using radix 2 and radix 4 modified booth multiplier algorithm. We used different type of adders like sixteen bit full adder in designing those multiplier.

The result of our project helps us to choose a better option between two multiplier in fabricating different systems. Multipliers form one of the most important component of many systems. So by analyzing the working of different multipliers helps to frame a better system with less power consumption and lesser area.

INTRODUCTION

Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc. A system's performance is generally determined by the performance of the multiplier because the multiplier is generally the slowest element in the system. Furthermore, it is generally the most area consuming. Hence, optimizing the speed and area of the multiplier is a major design issue. However, area and speed are usually conflicting constraints so that improving speed results mostly in larger areas. As a result, a whole spectrum of multipliers with different area-speed constraints have been designed with fully parallel.The Booth Multiplier and Bit Pair Recoding Multiplier have better performance in speed and area as compared to normal serial multipliers .

THEORY

The Booth Algorithm :

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation. I will illustrate the booth algorithm with the following example:

Example, $(2)_{10}$ x $(-4)_{10}$

$(0010)_2 * (1100)_2$

Step 1 : Making the Booth table

1) From the two numbers, pick the number with the smallest difference between a series of consecutive numbers, and make it a multiplier .

i.e., 0010 -- From 0 to 0 no change, 0 to 1 one change, 1 to 0 another change ,so there are two changes on this one

1100 -- From 1 to 1 no change, 1 to 0 one change, 0 to 0 no change, so there is only one change on this one.

Therefore, multiplication of 2 x (– 4), where $(2)_{10}$ ($(0010)_2$) is the multiplicand and $(-4)_{10}$ ($(1100)_2$) is the multiplier.

2) Let X = 1100 (multiplier)

Let Y = 0010 (multiplicand)

Take the 2's complement of Y and call it –Y

–Y = 1110

3) Load the X value in the table.

4) Load 0 for X-1 value it should be the previous first least significant bit of X

5) Load 0 in U and V rows which will have the product of X and Y at the end of operation.

6) Make four rows for each cycle; this is because we are multiplying four bits numbers.

Step 2: Booth Algorithm

Booth algorithm requires examination of the multiplier bits, and shifting of the partial product. Prior to the shifting, the multiplicand may be added to partial product, subtracted from the partial product, or left unchanged according to the following rules:

Look at the first least significant bits of the multiplier "X", and the previous least significant bits of the multiplier "X - 1".

1) 0 0 Shift only

1 1 Shift only.

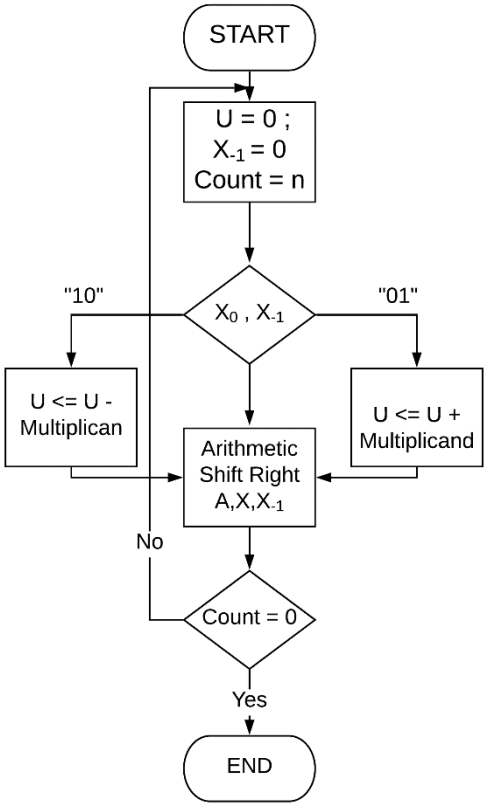
0 1 Add Y to U, and shift

1 0 Subtract Y from U, and shift or add (-Y) to U and shift

2) Take U & V together and shift arithmetic right shift which preserves the sign bit of 2's complement number. Thus a positive number remains positive, and a negative number remains negative.

3) Shift X circular right shift because this will prevent us from using two registers for the X value.

Cycle	U	V	X	X-1	Operation
Initialize	0000	0000	1100	0	-
1 Cycle	0000	0000	0110	0	Shift Only
2 Cycle	0000	0000	0011	0	Shift Only
3 Cycle	1110 1111	0000 0000	0001 0000	1 1	Add -Y and Shift
4 Cycle	1111	1000	0000	0	Shift Only



Multiplication Using Bit-Pair Recoding of Multipliers :

A technique called bit-pair recoding of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following. The pair (+1 −1) is equivalent to the pair (0 +1). That is, instead of adding −1 times the multiplicand M at shift position i to +1 × M at position i + 1, the same result is obtained by adding +1 × M at position i. Other examples are: (+1 0) is equivalent to (0 +2), (−1 +1) is equivalent to (0 −1), and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one

version of the multiplicand to be added to the partial product for each pair of multiplier bits.

One of the solutions of realizing high speed multipliers is to enhance parallelism which helps to decrease the number of subsequent calculation stages. The original version of the Booth algorithm (Radix-2) had two drawbacks. They are:

(i) The number of add subtract operations and the number of shift operations becomes variable and becomes inconvenient in designing parallel multipliers.

(ii) The algorithm becomes inefficient when there are isolated 1's. These problems are overcome by using modified Bit Pair Recoding (Radix4 Booth algorithm) which scan strings of three bits with the algorithm given below:

1) Extend the sign bit 1 position if necessary to ensure that n is even.

2) Append a 0 to the right of the LSB of the multiplier.

3) According to the value of each vector , each Partial Product will he 0, +Y , -Y, +2Y or -2Y.

The negative values of Y are made by taking the 2's complement . The multiplication of Y is done by shifting Y by one bit to the left. Thus, in any case, in designing a n-bit parallel multipliers, only n/2 partial products are generated.

X(i+1)	X(i)	X(i-1)	Y	Operation
0	0	0	0 x Y	Shift Only
0	0	1	+1 x Y	Add and Shift
0	1	0	+1 x Y	Add and Shift
0	1	1	+2 x Y	Shift and Add then Shift
1	0	0	-2 x Y	Complement Y and Shift then Add and Shift
1	0	1	-1 x Y	Complement Y and then Add and Shift
1	1	0	-1 x Y	Complement Y and then Add and Shift
1	1	1	0xY	Shift Only

Table : Bit Pair Recoding or Radix4 Modified Booth algorithm Scheme .

VHDL CODING

Multiplication Using Booth Algorithm:

Main Code :

File Name : booth_multiplication.vhd

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

entity booth_multiplication is
      generic (M : INTEGER := 5 ; P : INTEGER := 10) ;

    Port ( multiplier : in  STD_LOGIC_VECTOR(M-1 downto 0);
           multiplicand : in  STD_LOGIC_VECTOR(M-1 downto 0);
           mul_answer : out  STD_LOGIC_VECTOR(P-1 downto 0));
end booth_multiplication;

architecture arch_booth_multiplication of booth_multiplication is

      component bin_adder_Nbit
                           -- Declaring Component 9 bit adder
            generic (N : INTEGER) ;
            Port ( A : in  STD_LOGIC_VECTOR(N-1 downto 0);        -- LSB
is the implied zero
           B : in  STD_LOGIC_VECTOR(N-1 downto 0);
                    C_IN : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR(N-1 downto 0);
           C_OUT : out  STD_LOGIC);
      end component bin_adder_Nbit ;


      signal A,A_COMP,Q: STD_LOGIC_VECTOR(P downto 0) ;

      signal P1,P2,P3,P4,P5,P6,P7,P8,P9,P10 : STD_LOGIC_VECTOR(P downto
0) ;

      signal partial_product1,partial_product2,partial_product3,
partial_product4,partial_product5: STD_LOGIC_VECTOR(P downto 0) ;

      signal partial_product1_shift , partial_product2_shift ,
partial_product3_shift , partial_product4_shift,partial_product5_shift:
STD_LOGIC_VECTOR(P downto 0) ;

      signal Z,complement1_mulpr , complement2_mulpr :
STD_LOGIC_VECTOR(M-1 downto 0) ;
```

```
begin

      A(P downto M+1) <= multiplier ;
      A(M downto 0) <= ( others => '0') ;

      complement1_mulpr <= not(multiplier) ;
      Z <= ( 0 => '1' , others => '0') ;
      A_COMP(P downto M+1) <= complement2_mulpr  ;
      A_COMP(M downto 0) <= ( others => '0') ;

      Q(P downto M+1) <= ( others => '0') ;
      Q(M downto 1) <= multiplicand ;
      Q(0) <= '0' ;

      partial_product1 <= Q when(Q(1 downto 0 ) = "00" or Q(1 downto 0) =
"11") else
                          P1 when Q(1 downto 0) = "01" else
                                  --P1 = Q+A
                          P2 when Q(1 downto 0) = "10" ;
                                  --P2 = Q+A_COMP

      partial_product1_shift (P-1 downto 0) <= partial_product1(P downto
1) ;
      partial_product1_shift (P) <= partial_product1(P) ;

      partial_product2 <= partial_product1_shift
when(partial_product1_shift(1 downto 0 ) = "00" or
partial_product1_shift(1 downto 0) = "11") else
                          P3 when partial_product1_shift(1 downto 0) = "01"
else  --P3 = partial_product1_shift + A
                          P4 when partial_product1_shift(1 downto 0) = "10"
;           --P4 = partial_product1_shift + A_COMP


      partial_product2_shift (P-1 downto 0) <= partial_product2(P downto
1) ;
      partial_product2_shift (P) <= partial_product2(P) ;


      partial_product3 <= partial_product2_shift
when(partial_product2_shift(1 downto 0 ) = "00" or
partial_product2_shift(1 downto 0) = "11") else
                          P5 when partial_product2_shift(1 downto 0) = "01"
else       --P5 = partial_product2_shift + A
                          P6 when partial_product2_shift(1 downto 0) = "10"
;           --P6 = partial_product2_shift + A_COMP


      partial_product3_shift (P-1 downto 0) <= partial_product3(P downto
1) ;
      partial_product3_shift (P) <= partial_product3(P) ;
```

```
      partial_product4 <= partial_product3_shift
when(partial_product3_shift(1 downto 0 ) = "00" or
partial_product3_shift(1 downto 0) = "11") else
                          P7 when partial_product3_shift(1 downto 0) = "01"
else        --P7 = partial_product3_shift + A
                          P8 when partial_product3_shift(1 downto 0) = "10"
;           --P8 = partial_product3_shift + A_COMP

      partial_product4_shift (P-1 downto 0) <= partial_product4(P downto
1) ;
      partial_product4_shift (P) <= partial_product4(P) ;


      partial_product5 <= partial_product4_shift
when(partial_product4_shift(1 downto 0 ) = "00" or
partial_product4_shift(1 downto 0) = "11") else
                          P9 when partial_product4_shift(1 downto 0) = "01"
else        --P9 = partial_product4_shift + A
                          P10 when partial_product4_shift(1 downto 0) =
"10" ;            --P10 = partial_product4_shift + A_COMP

      partial_product5_shift (P-1 downto 0) <= partial_product5(P downto
1) ;
      partial_product5_shift (P) <= partial_product5(P) ;


      BA4_1 : bin_adder_Nbit generic map (M) port map
(complement1_mulpr,Z,'0',complement2_mulpr) ;

      BAN_1 : bin_adder_Nbit generic map (P+1) port map (Q,A,'0',P1) ;
      BAN_2 : bin_adder_Nbit generic map (P+1) port map (Q,A_COMP,'0',P2)
;
      BAN_3 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,A,'0',P3) ;
      BAN_4 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,A_COMP,'0',P4) ;
      BAN_5 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,A,'0',P5) ;
      BAN_6 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,A_COMP,'0',P6) ;
      BAN_7 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,A,'0',P7) ;
      BAN_8 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,A_COMP,'0',P8) ;
      BAN_9 : bin_adder_Nbit generic map (P+1) port map
(partial_product4_shift,A,'0',P9) ;
      BAN_10 : bin_adder_Nbit generic map (P+1) port map
(partial_product4_shift,A_COMP,'0',P10) ;                              --
Association of Component ports with Entity Ports

      mul_answer <= partial_product5_shift(P downto 1) ;
end arch_booth_multiplication;
```

Code for N Bit Binary Adder :

File Name : bin_adder_Nbit.vhd

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

entity bin_adder_Nbit is
                  generic (N : INTEGER := 9) ;
                  Port ( A : in  STD_LOGIC_VECTOR(N-1 downto 0);        -
- LSB is the implied zero
           B : in  STD_LOGIC_VECTOR(N-1 downto 0);
                    C_IN : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR(N-1 downto 0);
           C_OUT : out  STD_LOGIC);
end bin_adder_Nbit;

architecture arch_bin_adder_Nbit of bin_adder_Nbit is
      component full_add is
            port(a,b,c_in : in STD_LOGIC ;
                        sum,c_out : out STD_LOGIC);
      end component full_add ;

      signal CARRY : STD_LOGIC_VECTOR(N downto 0) ;
      begin
            CARRY(0) <= C_IN ;
            GK : for K in N-1 downto 0 generate
                  FA : full_add port map
(CARRY(K),A(K),B(K),S(K),CARRY(K+1)) ;                                --
Association of Component ports with Entity Ports .
            end generate GK ;
            C_OUT <= CARRY(N) ;
end arch_bin_adder_Nbit;
```

Code for Full Adder :

File Name : full_add.vhd

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

entity full_add is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           c_in : in  STD_LOGIC;
           sum : out  STD_LOGIC;
           c_out : out  STD_LOGIC);
end full_add;
```

```
architecture arch_full_add of full_add is
      begin
      sum <= a xor b xor c_in ;
      c_out <= ( a and b ) or ( b and c_in ) or ( c_in and a ) ;
end arch_full_add ;
```

Multiplication Using Bit Pair Recoding Algorithm:

Main Code :

File Name : bit_pair_multiplication.vhd

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

entity bit_pair_multiplication is
      generic (M : INTEGER := 8 ; P : INTEGER := 16) ;
          Port ( multiplier : in  STD_LOGIC_VECTOR(M-1 downto 0);
                  multiplicand : in  STD_LOGIC_VECTOR(M-1 downto 0);
                  mul_answer : out  STD_LOGIC_VECTOR(P-1 downto 0));
end bit_pair_multiplication;

architecture arch_bit_pair of bit_pair_multiplication is
      component bin_adder_Nbit
               -- Declaring Component 9 bit adder
            generic (N : INTEGER) ;
            Port ( A : in  STD_LOGIC_VECTOR(N-1 downto 0);        -- LSB
is the implied zero
           B : in  STD_LOGIC_VECTOR(N-1 downto 0);
                    C_IN : in  STD_LOGIC;
           S : out  STD_LOGIC_VECTOR(N-1 downto 0);
           C_OUT : out  STD_LOGIC);
      end component bin_adder_Nbit ;

      signal A,A_COMP,S,S_COMP,Q: STD_LOGIC_VECTOR(P downto 0) ;

      signal P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16 :
STD_LOGIC_VECTOR(P downto 0) ;

      signal partial_product1 , partial_product2 ,
                  partial_product3 , partial_product4  :
STD_LOGIC_VECTOR(P downto 0) ;

      signal partial_product1_shift , partial_product2_shift ,
                partial_product3_shift , partial_product4_shift :
STD_LOGIC_VECTOR(P downto 0) ;
```

```
      signal Z , complement1_mulpr , complement2_mulpr :
STD_LOGIC_VECTOR(M-1 downto 0) ;

begin
      A(P downto M+1) <= multiplier ;
      A(M downto 0) <= ( others => '0') ;

      complement1_mulpr <= not(multiplier) ;
      Z <= ( 0 => '1' , others => '0') ;
      A_COMP(P downto M+1) <= complement2_mulpr  ;
      A_COMP(M downto 0) <= ( others => '0') ;

      S(P downto 1) <= A(P-1 downto 0) ;
      S(0) <= '0' ;

      S_COMP(P downto 1) <= A_COMP(P-1 downto 0) ;
      S_COMP(0) <= '0' ;

      Q(P downto M+1) <= ( others => '0') ;
      Q(M downto 1) <= multiplicand ;
      Q(0) <= '0' ;

      partial_product1 <= Q when(Q(2 downto 0 ) = "000" or Q(2 downto 0)
= "111") else
                                          P1 when (Q(2 downto 0) =
"001" or Q(2 downto 0) = "010") else
-- P1 = Q+A
                                          P2 when (Q(2 downto 0) =
"101" or Q(2 downto 0) = "110") else                              --
P2 = Q+A_COMP
                                          P3 when Q(2 downto 0) =
"011" else                                                         -
- P3 = Q+S
                                          P4 when Q(2 downto 0) =
"100";                                                             -
- P4 = Q+S_COMP

      partial_product1_shift (P-2 downto 0) <= partial_product1(P downto
2) ;
      partial_product1_shift (P) <= partial_product1(P) ;
      partial_product1_shift (P-1) <= partial_product1(P) ;

      partial_product2 <= partial_product1_shift
when(partial_product1_shift(2 downto 0 ) = "000" or
                                          partial_product1_shift(2
downto 0) = "111") else
                                          P5 when
(partial_product1_shift(2 downto 0) = "001" or

      partial_product1_shift(2 downto 0) = "010") else
-- P5 = partial_product1_shift+A
                                          P6 when
```

```
(partial_product1_shift(2 downto 0) = "101" or

      partial_product1_shift(2 downto 0) = "110") else
-- P6 = partial_product1_shift+A_COMP
                                                P7 when
partial_product1_shift(2 downto 0) = "011" else                    -- P7 =
partial_product1_shift+S
                                                P8 when
partial_product1_shift(2 downto 0) = "100";                                -
- P8 = partial_product1_shift+S_COMP

      partial_product2_shift (P-2 downto 0) <= partial_product2(P downto
2) ;
      partial_product2_shift (P) <= partial_product2(P) ;
      partial_product2_shift (P-1) <= partial_product2(P) ;

      partial_product3 <= partial_product2_shift
when(partial_product2_shift(2 downto 0 ) = "000" or
                                          partial_product1_shift(2
downto 0) = "111") else
                                                P9 when
(partial_product2_shift(2 downto 0) = "001" or

      partial_product2_shift(2 downto 0) = "010") else
-- P9 = partial_product2_shift+A
                                                P10 when
(partial_product2_shift(2 downto 0) = "101" or

      partial_product2_shift(2 downto 0) = "110") else
-- P10 = partial_product2_shift+A_COMP
                                                P11 when
partial_product2_shift(2 downto 0) = "011" else                    -- P11
= partial_product2_shift+S
                                                P12 when
partial_product2_shift(2 downto 0) = "100";                                -
- P12 = partial_product2_shift+S_COMP

      partial_product3_shift (P-2 downto 0) <= partial_product3(P downto
2) ;
      partial_product3_shift (P) <= partial_product3(P) ;
      partial_product3_shift (P-1) <= partial_product3(P) ;

      partial_product4 <= partial_product3_shift
when(partial_product3_shift(2 downto 0 ) = "000" or
                                          partial_product3_shift(2
downto 0) = "111") else
                                                P13 when
(partial_product3_shift(2 downto 0) = "001" or

      partial_product3_shift(2 downto 0) = "010") else
-- P13 = partial_product3_shift+A
                                                P14 when
```

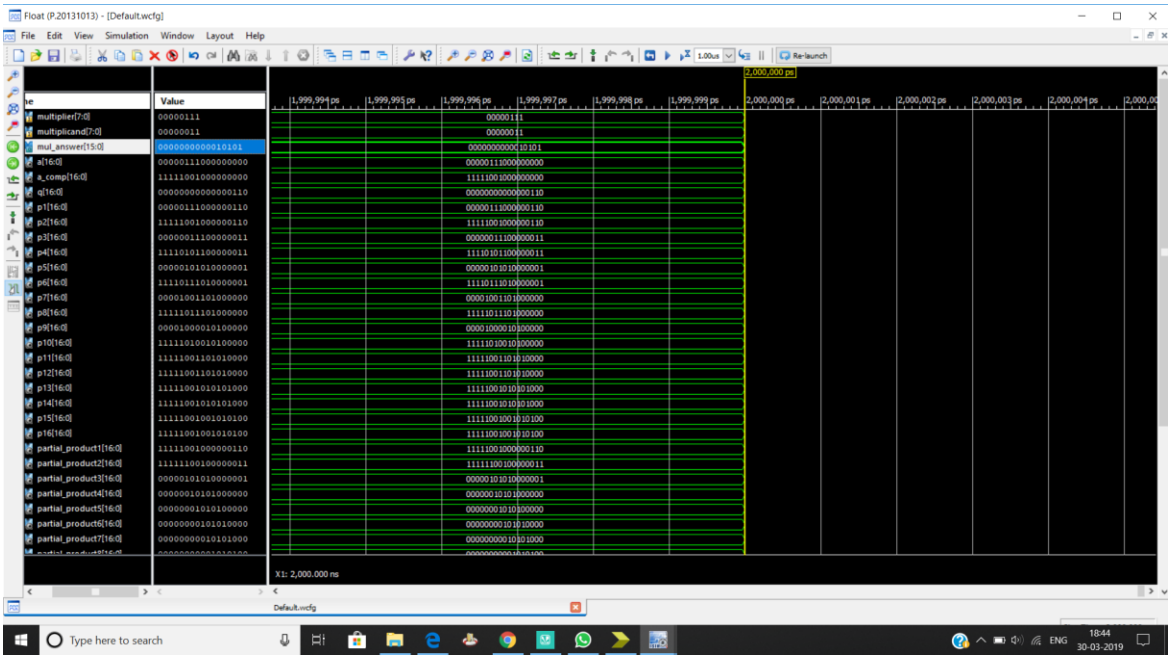
```
(partial_product3_shift(2 downto 0) = "101" or

      partial_product3_shift(2 downto 0) = "110") else
-- P14 = partial_product3_shift+A_COMP
                                                P15 when
partial_product3_shift(2 downto 0) = "011" else                    -- P15
= partial_product3_shift+S
                                                P16 when
partial_product3_shift(2 downto 0) = "100";                                -
- P16 = partial_product3_shift+S_COMP

      partial_product4_shift (P-2 downto 0) <= partial_product4(P downto
2) ;
      partial_product4_shift (P) <= partial_product4(P) ;
      partial_product4_shift (P-1) <= partial_product4(P) ;

      BA4_1 : bin_adder_Nbit generic map (M) port map
(complement1_mulpr,Z,'0',complement2_mulpr) ;

      BAN_1 : bin_adder_Nbit generic map (P+1) port map (Q,A,'0',P1) ;
      BAN_2 : bin_adder_Nbit generic map (P+1) port map (Q,A_COMP,'0',P2)
;
      BAN_3 : bin_adder_Nbit generic map (P+1) port map (Q,S,'0',P3) ;
      BAN_4 : bin_adder_Nbit generic map (P+1) port map (Q,S_COMP,'0',P4)
;
      BAN_5 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,A,'0',P5) ;
      BAN_6 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,A_COMP,'0',P6) ;
      BAN_7 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,S,'0',P7) ;
      BAN_8 : bin_adder_Nbit generic map (P+1) port map
(partial_product1_shift,S_COMP,'0',P8) ;
      BAN_9 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,A,'0',P9) ;
      BAN_10 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,A_COMP,'0',P10) ;
      BAN_11 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,S,'0',P11) ;
      BAN_12 : bin_adder_Nbit generic map (P+1) port map
(partial_product2_shift,S_COMP,'0',P12) ;
      BAN_13 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,A,'0',P13) ;
      BAN_14 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,A_COMP,'0',P14) ;
      BAN_15 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,S,'0',P15) ;
      BAN_16 : bin_adder_Nbit generic map (P+1) port map
(partial_product3_shift,S_COMP,'0',P16) ;                          --
Association of Component ports with Entity Ports
      mul_answer <= partial_product4_shift(P downto 1) ;
end arch_bit_pair;
```

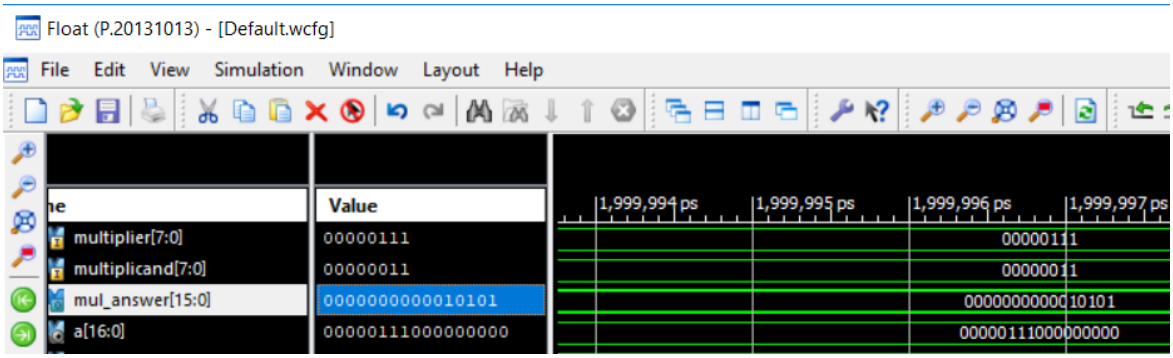
OUTPUT

Multiplication Using Booth Algorithm:

1) $(07)_{10} \times (03)_{10} = (21)_{10}$

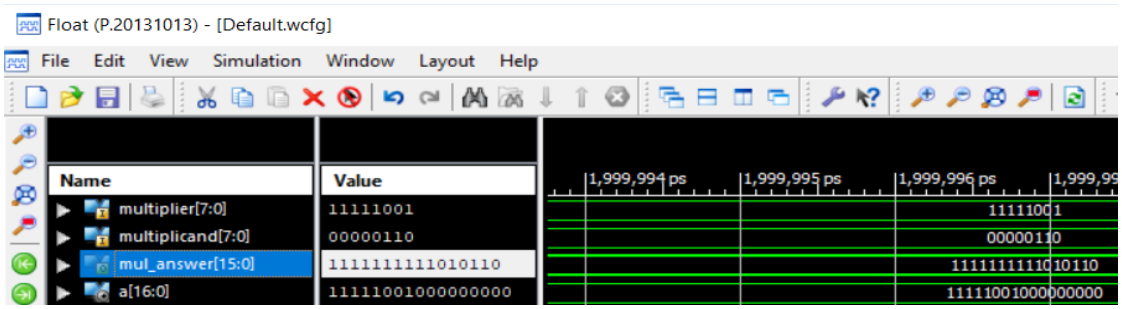
$(0000\ 0111)_2 \times (0000\ 0011)_2 = (0000\ 0000\ 0001\ 0101)_2$





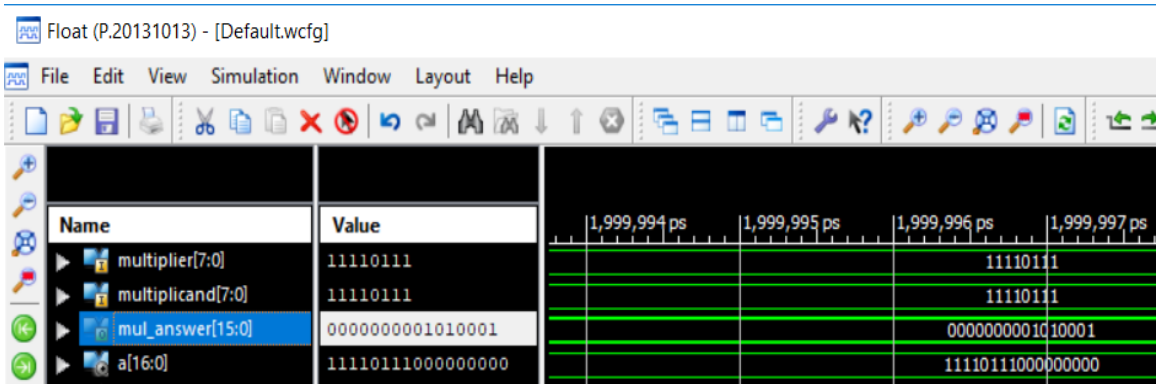
2) $(-07)_{10} \times (06)_{10} = (-42)_{10}$

$(1111\ 1001)_2 \times (0000\ 0110)_2 = (1111\ 1111\ 1101\ 0110)_2$



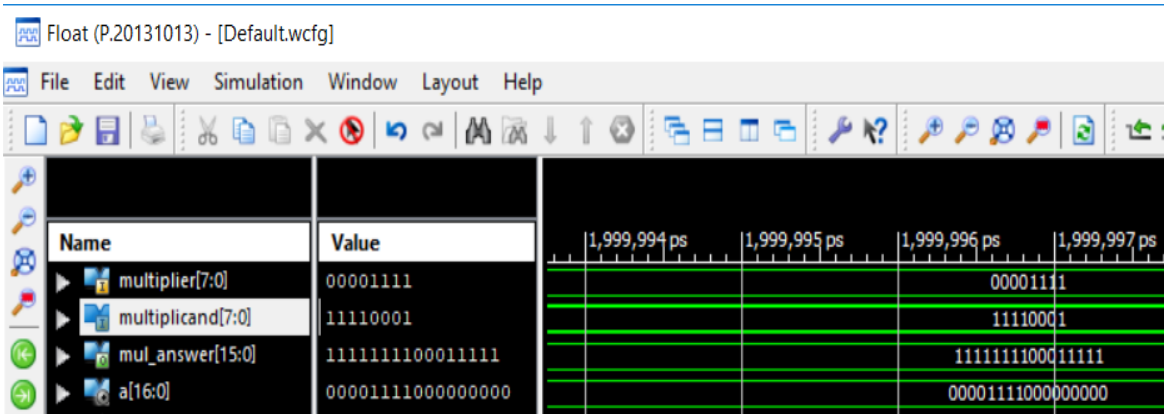
3) $(-09)_{10} \times (-09)_{10} = (81)_{10}$

$(1111\ 0111)_2 \times (1111\ 0111)_2 = (0000\ 0000\ 0101\ 0001)_2$



4) $(15)_{10} \times (-15)_{10} = (-225)_{10}$

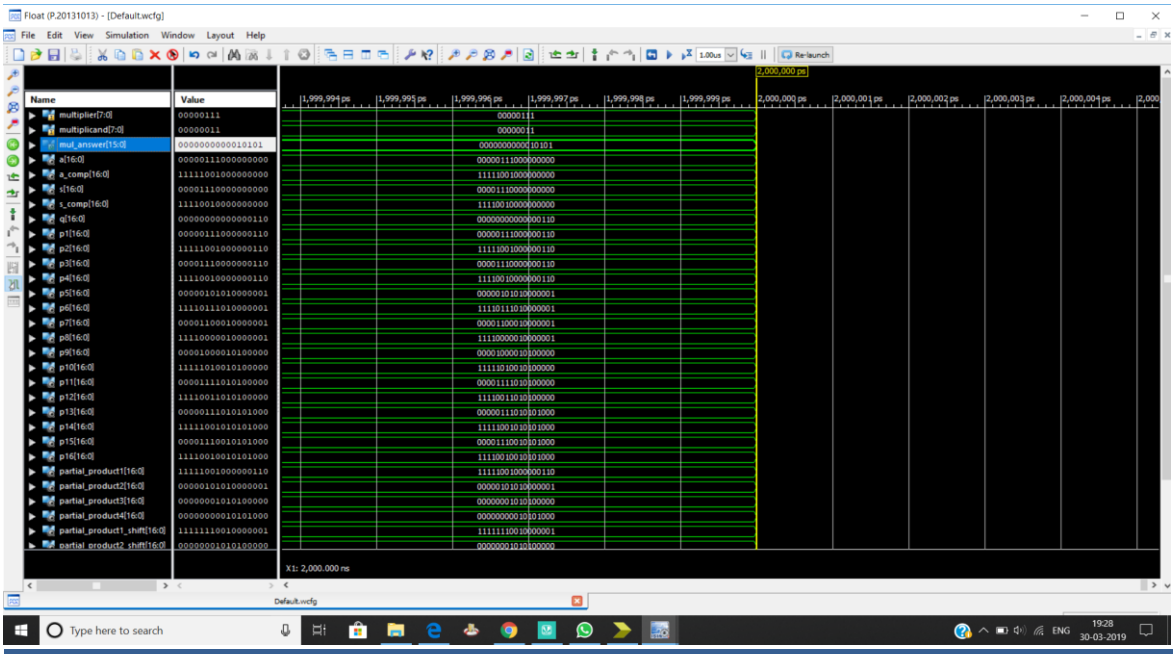
$(0000\ 1111)_2 \times (1111\ 0001)_2 = (1111\ 1111\ 0001\ 1111)_2$

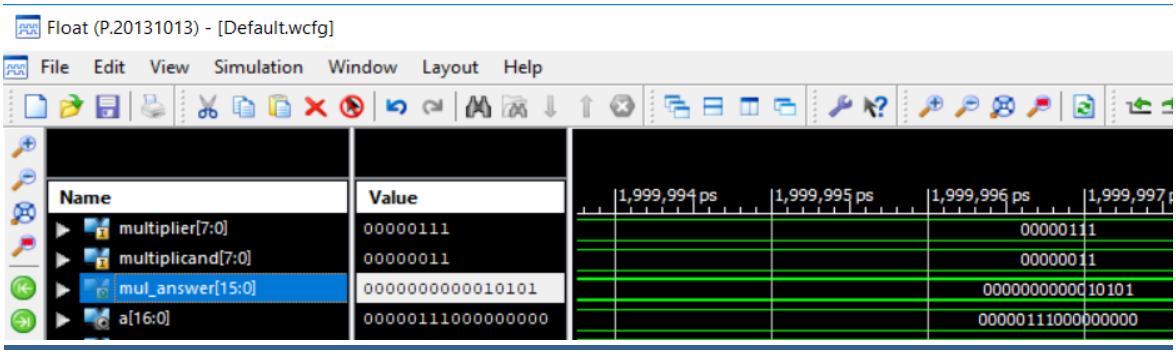


Multiplication Using Booth Algorithm:

1) $(07)_{10} \times (03)_{10} = (21)_{10}$

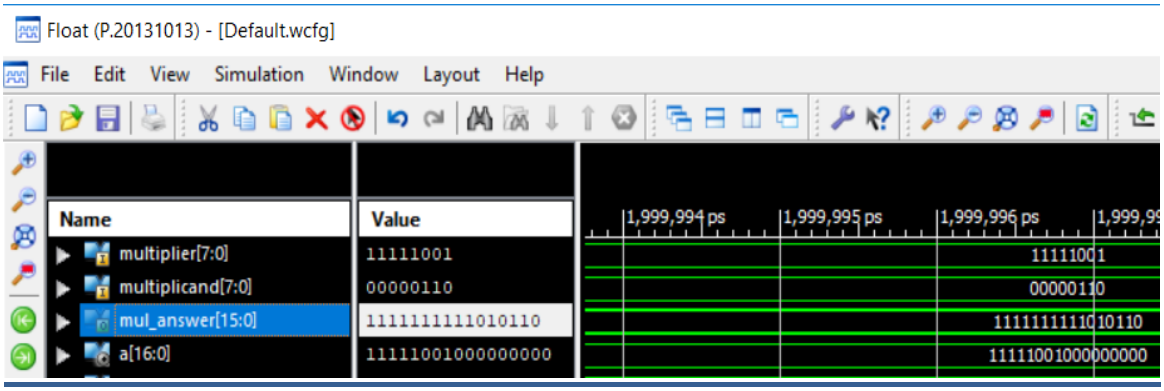
$(0000\ 0111)_2 \times (0000\ 0011)_2 = (0000\ 0000\ 0001\ 0101)_2$





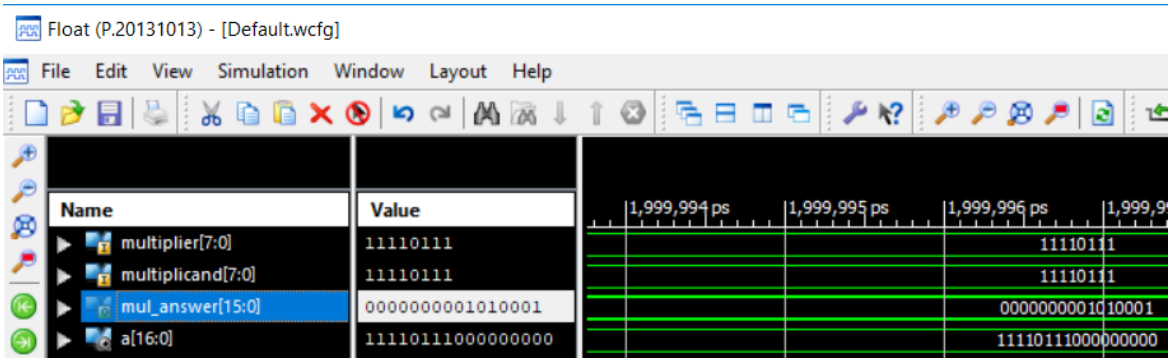
2) $(-07)_{10} \times (06)_{10} = (42)_{10}$

$(1111\ 1001)_2 \times (0000\ 0110)_2 = (1111\ 1111\ 1101\ 0110)_2$



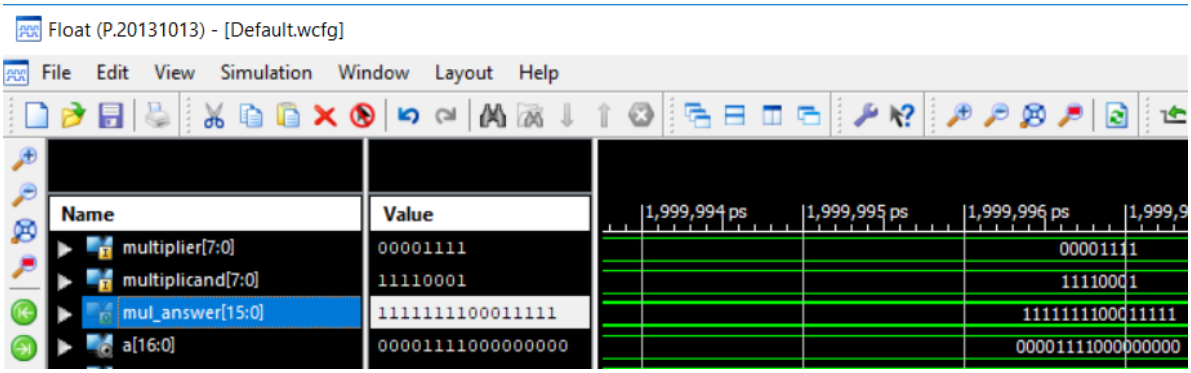
3) $(-09)_{10} \times (-09)_{10} = (81)_{10}$

$(1111\ 0111)_2 \times (1111\ 0111)_2 = (0000\ 0000\ 0101\ 0001)_2$



4) $(15)_{10} \times (-15)_{10} = (-225)_{10}$

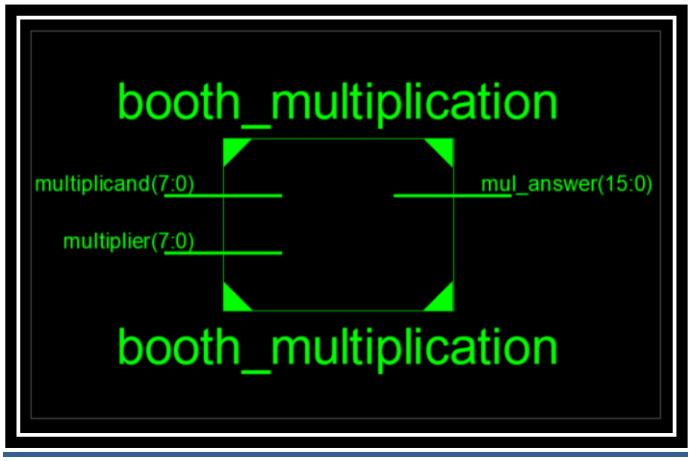
$(0000\ 1111)_2 \times (1111\ 0001)_2 = (1111\ 1111\ 0001\ 1111)_2$

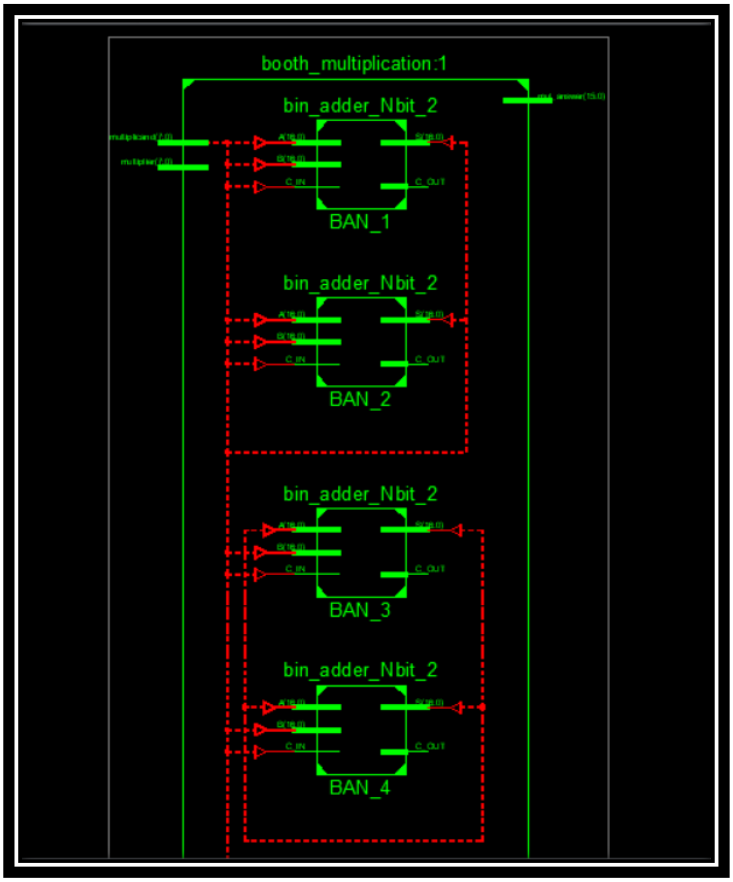


RESULT

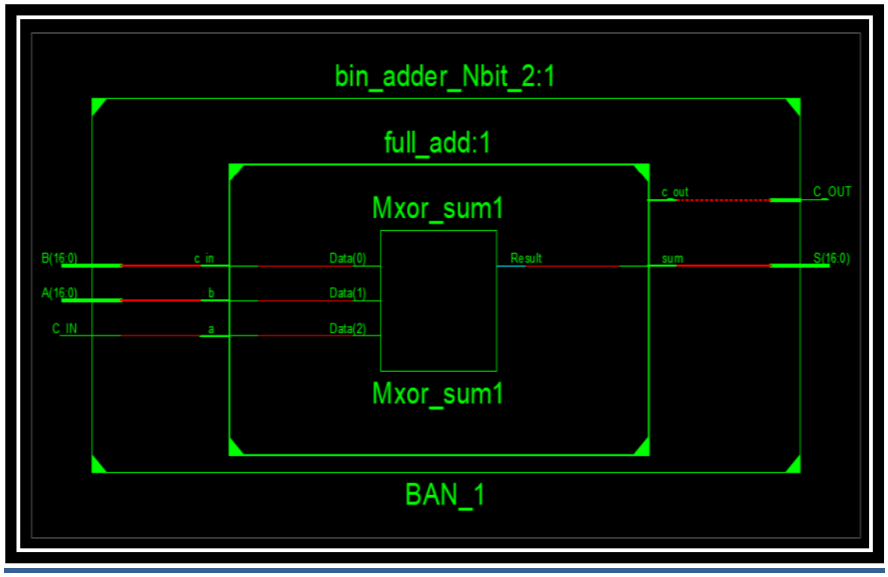
RTL SCHEMATICS :

Booth Multiplier Entity :

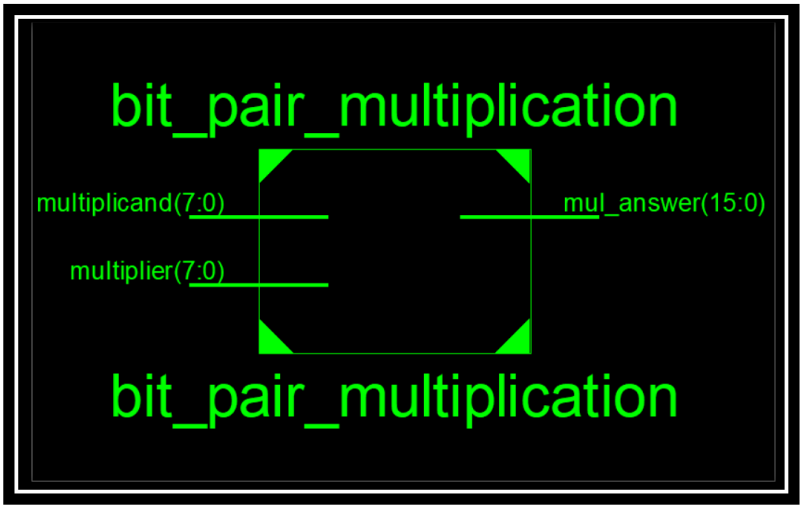


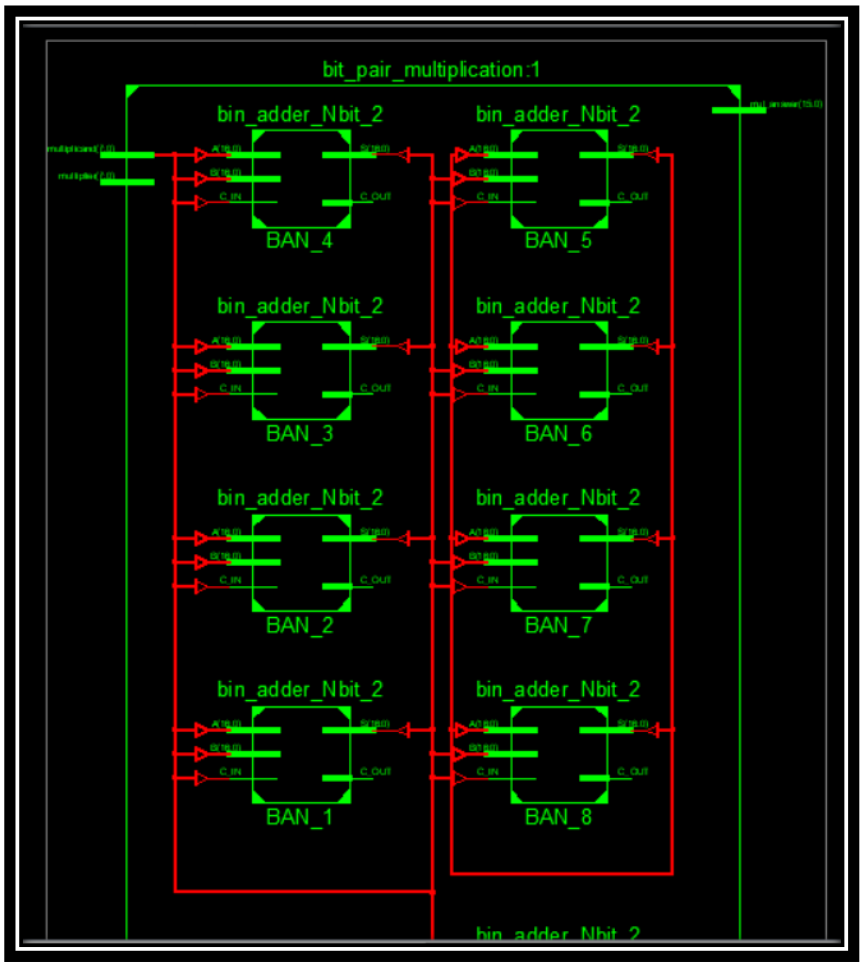


N Bit Binary Adder Entity :



Bit Pair Multiplier Entity :





CONCLUSION

This project gives a clear concept of different multiplier and their implementation . We found that the parallel multipliers are much better option than the serial multiplier. We concluded this from the result of power consumption and the total area. In case of parallel multipliers, the total area is much less than that of serial multipliers. Hence the power consumption is also less. This is clearly depicted in our results. This speeds up the calculation and makes the system faster.

While comparing the Booth Multiplier (Radix 2) , Bit Pair Redocding Multiplier (Radix 4) we found that radix 4 consumes lesser power than that of radix 2. This is because it uses almost half number of iteration and adders when compared to radix 2.

Multipliers are one the most important component of many systems. So we always need to find a better solution in case of multipliers. Our multipliers should always consume less power and cover less power. So through our project we try to determine which of the two algorithms works the best. In the end we determine that radix 4 modified booth Bit Pair Recoding algorithm works the best.

BIBLIOGRAPHY

Websites :

1) https://www.geeksforgeeks.org/
2) https://stackoverflow.com/

Books :

1) A VHDL Primer By Jayaram Bhaskar .
2) Computer Organization And Embedded Systems By Carl Hamacher .