VHDL Code
Simulation & Implementation
in ISE Xilinx

Signed 4BIT Binary Multiplication

Mohammad Niknam

یک ضرب کننده ٤بیتی علامت دار)و البته سنتزپذیر(را توصیف نمایید. توصیف باید استراکچرال و مبتنی بر جمع کننده ها باشد)که پیش از این توصیف کرده(. همچنین یک تست بنچ برای ارزیابی صحت عملکرد بنویسید که خروجی را برای زوج ورودی هایی با علامت های متفاوت نمایش دهد)مثلا یک بار دو عدد مثبت را در هم ضرب کند، بار دیگر یک عدد مثبت در منفی و بالاخره دو عدد منفی را.

1.6 SIGNED BINARY NUMBERS

Positive integers (including zero) can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with binary digits. It is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit 0 for positive and 1 for negative.

It is important to realize that both signed and unsigned binary numbers consist of a string of bits when represented in a computer. The user determines whether the number is signed or unsigned. If the binary number is signed, then the leftmost bit represents the sign and the rest of the bits represent the number. If the binary number is assumed to be unsigned, then the leftmost bit is the most significant bit of the number. For example, the string of bits 01001 can be considered as 9 (unsigned binary) or as +9 (signed binary) because the leftmost bit is 0. The string of bits 11001 represents the binary equivalent of 25 when considered as an unsigned number and the binary equivalent of −9 when considered as a signed number. This is because the 1 that is in the leftmost position designates a negative and the other four bits represent binary 9. Usually, there is no confusion in interpreting the bits if the type of representation for the number is known in advance.

The representation of the signed numbers in the last example is referred to as the *signed-magnitude* convention. In this notation, the number consists of a magnitude and a symbol (+ or −) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic. When arithmetic operations are implemented in a computer, it is more convenient to use a different system, referred to as the *signed-complement* system, for representing negative numbers. In this system, a negative number is indicated by its complement. Whereas the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement. Since positive numbers always start with 0 (plus) in the leftmost position, the complement will always start with a 1, indicating a negative number. The signed-complement system can use either the 1's or the 2's complement, but the 2's complement is the most common.

As an example, consider the number 9, represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent of 9, which gives 00001001. Note that all eight bits must have a value; therefore, 0's are inserted following the sign bit up to the first 1. Although there is only one way to represent +9, there are three different ways to represent −9 with eight bits:

signed-magnitude representation:	10001001
signed-1's-complement representation:	11110110
signed-2's-complement representation:	11110111

In signed-magnitude, −9 is obtained from +9 by changing only the sign bit in the leftmost position from 0 to 1. In signed-1's-complement, −9 is obtained by complementing all the bits of +9, including the sign bit. The signed-2's-complement representation of −9 is obtained by taking the 2's complement of the positive number, including the sign bit.

Table 1.3 lists all possible four-bit signed binary numbers in the three representations. The equivalent decimal number is also shown for reference. Note that the positive numbers in all three representations are identical and have 0 in the leftmost position. The signed-2's-complement system has only one representation for 0, which is always positive. The other two systems have either a positive 0 or a negative 0, something not encountered in ordinary arithmetic. Note that all negative numbers have a 1 in the leftmost bit position; that is the way we distinguish them from the positive numbers. With four bits, we can represent 16 binary numbers. In the signed-magnitude and the 1's-complement representations, there are eight positive numbers and eight negative numbers, including two zeros. In the 2's-complement representation, there are eight positive numbers, including one zero, and eight negative numbers.

The signed-magnitude system is used in ordinary arithmetic, but is awkward when employed in computer arithmetic because of the separate handling of the sign and the magnitude. Therefore, the signed-complement system is normally used. The 1's complement imposes some difficulties and is seldom used for arithmetic operations. It is useful as a logical operation, since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation, as will be shown in the next chapter. The discussion of signed binary arithmetic that follows deals exclusively with the signed-2's-complement

Table 1.3
Signed Binary Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

representation of negative numbers. The same procedures can be applied to the signed-1's-complement system by including the end-around carry as is done with unsigned numbers.

4.7 BINARY MULTIPLIER

Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

To see how a binary multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 4.15. The multiplicand

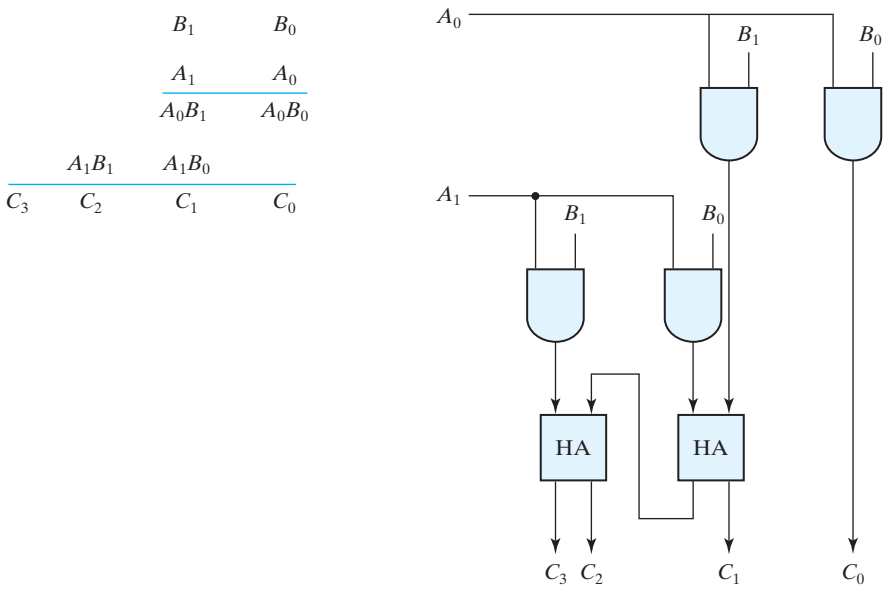


FIGURE 4.15
Two-bit by two-bit binary multiplier

bits are B_1 and B_0, the multiplier bits are A_1 and A_0, and the product is $C_3C_2C_1C_0$. The first partial product is formed by multiplying B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram. The second partial product is formed by multiplying B_1B_0 by A_1 and shifting one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products. Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added with the partial product of the previous level to form a new partial product. The last level produces the product. For J multiplier bits and K multiplicand bits, we need $(J \times K)$ AND gates and $(J - 1)$ K-bit adders to produce a product of $(J + K)$ bits.

As a second example, consider a multiplier circuit that multiplies a binary number represented by four bits by a number represented by three bits. Let the multiplicand be represented by $B_3B_2B_1B_0$ and the multiplier by $A_2A_1A_0$. Since $K = 4$ and $J = 3$, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig. 4.16.

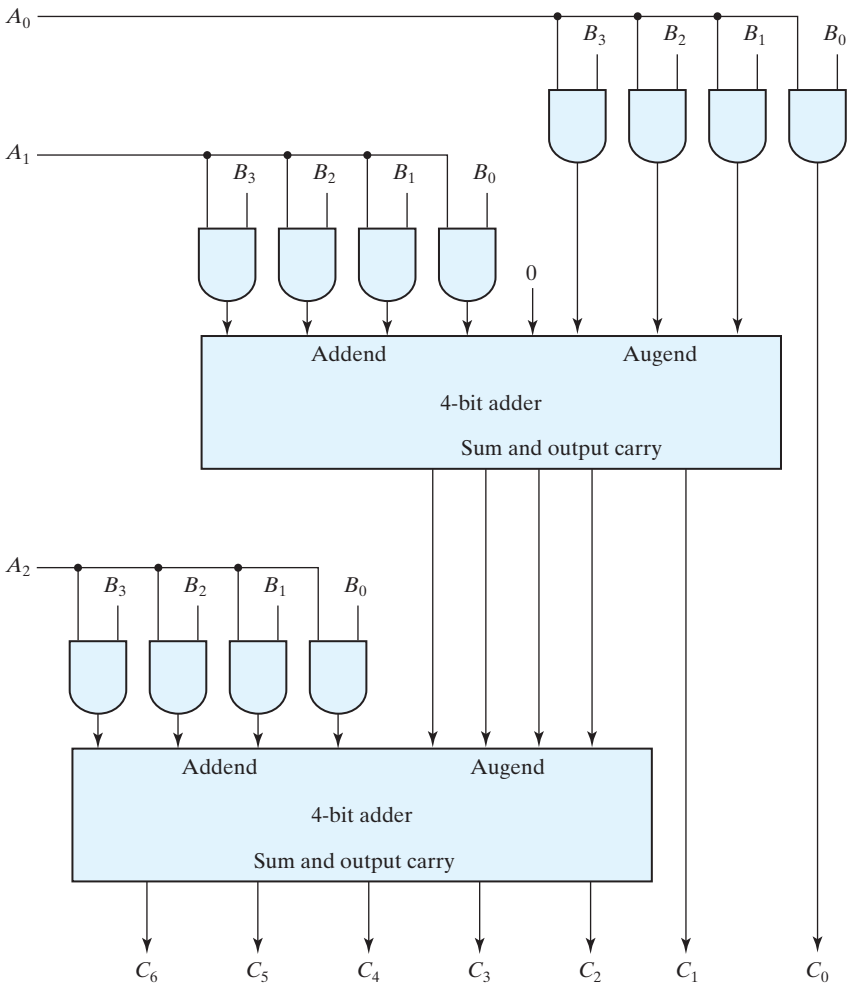
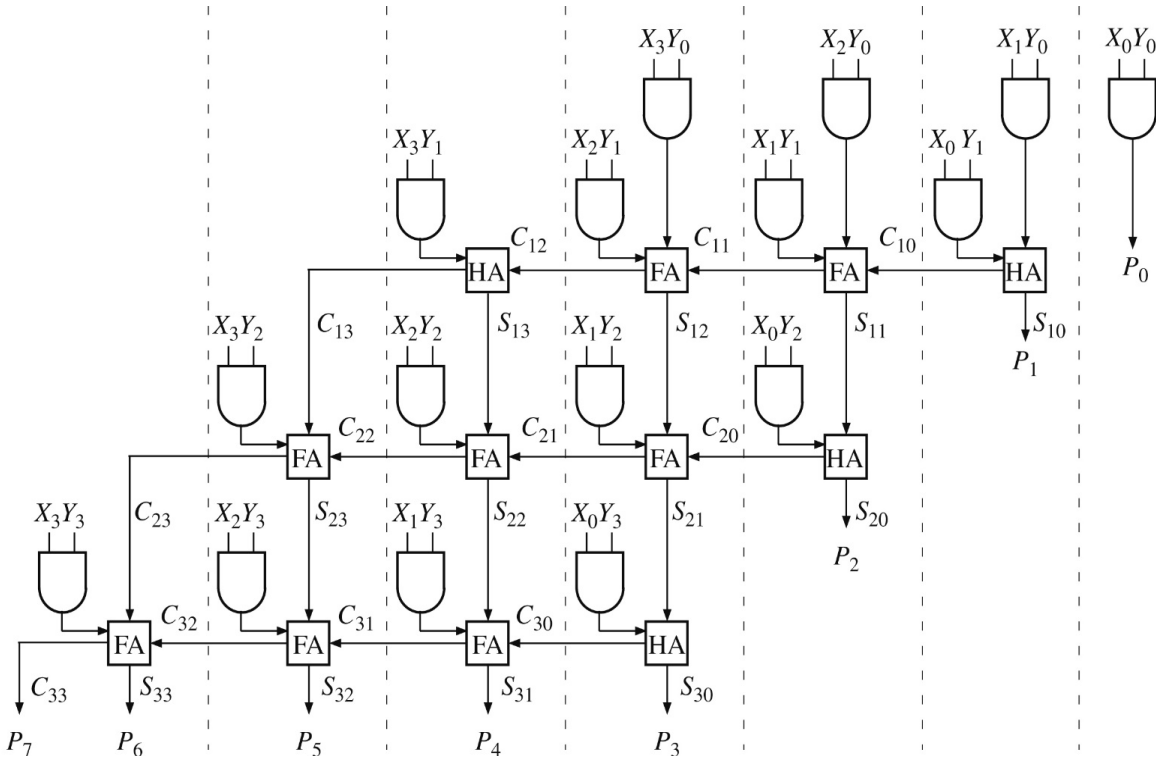


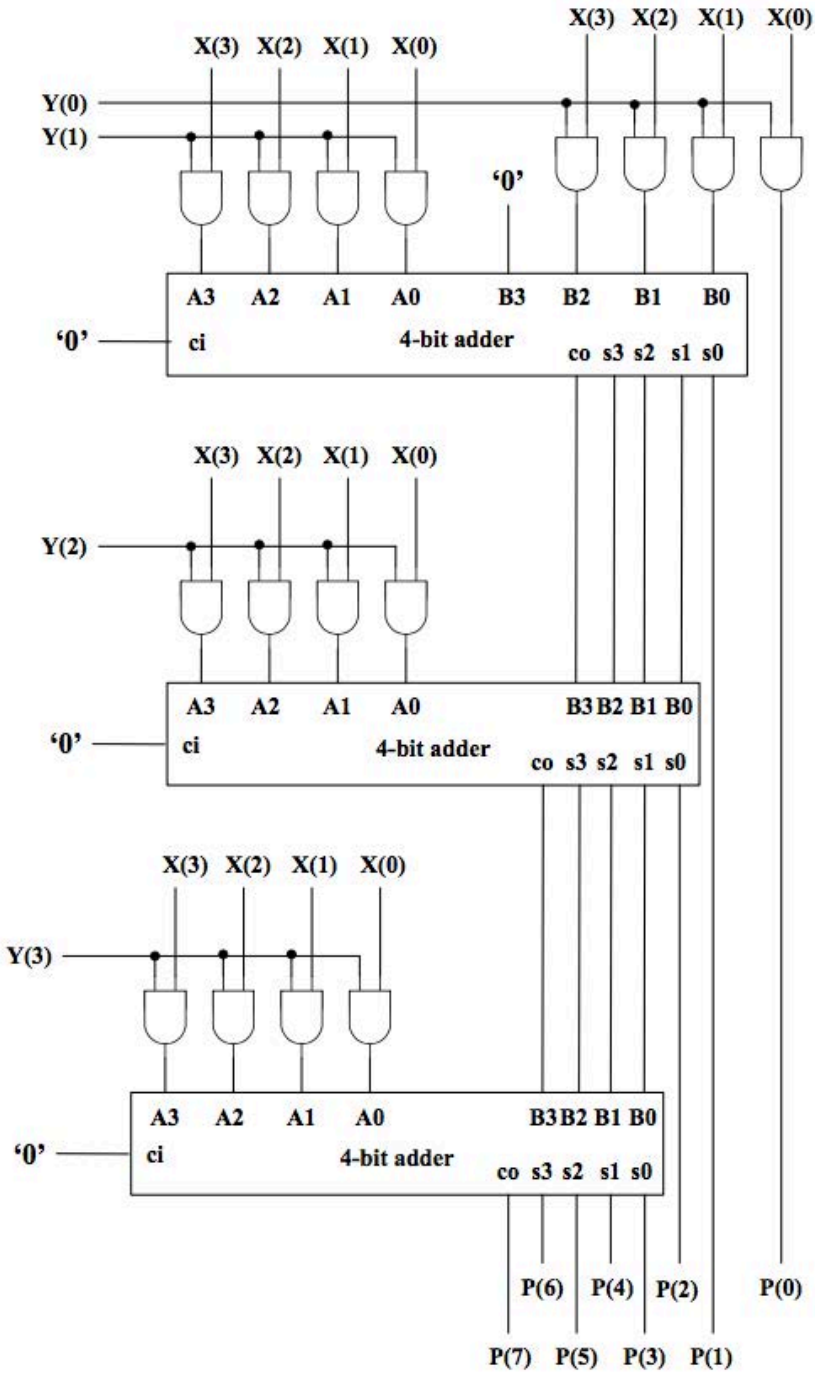
FIGURE 4.16
Four-bit by three-bit binary multiplier

Unsigned Multiplication

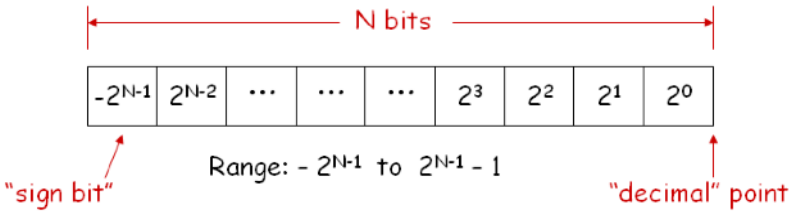
			X_3	X_2	X_1	X_0	
			Y_3	Y_2	Y_1	Y_0	
			X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	
		X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		
		C_{12}	C_{11}	C_{10}			
	C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		
	X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			
	C_{22}	C_{21}	C_{20}				
C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			
X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				
C_{32}	C_{31}	C_{30}					
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0



Unsigned Multiplication



Signed Multiplication

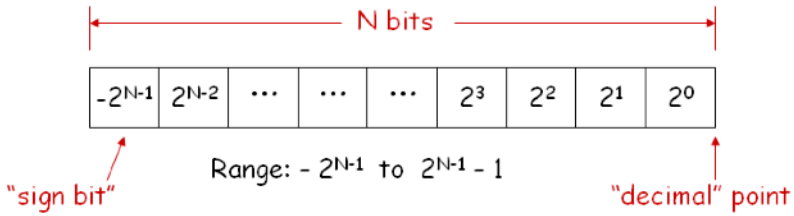


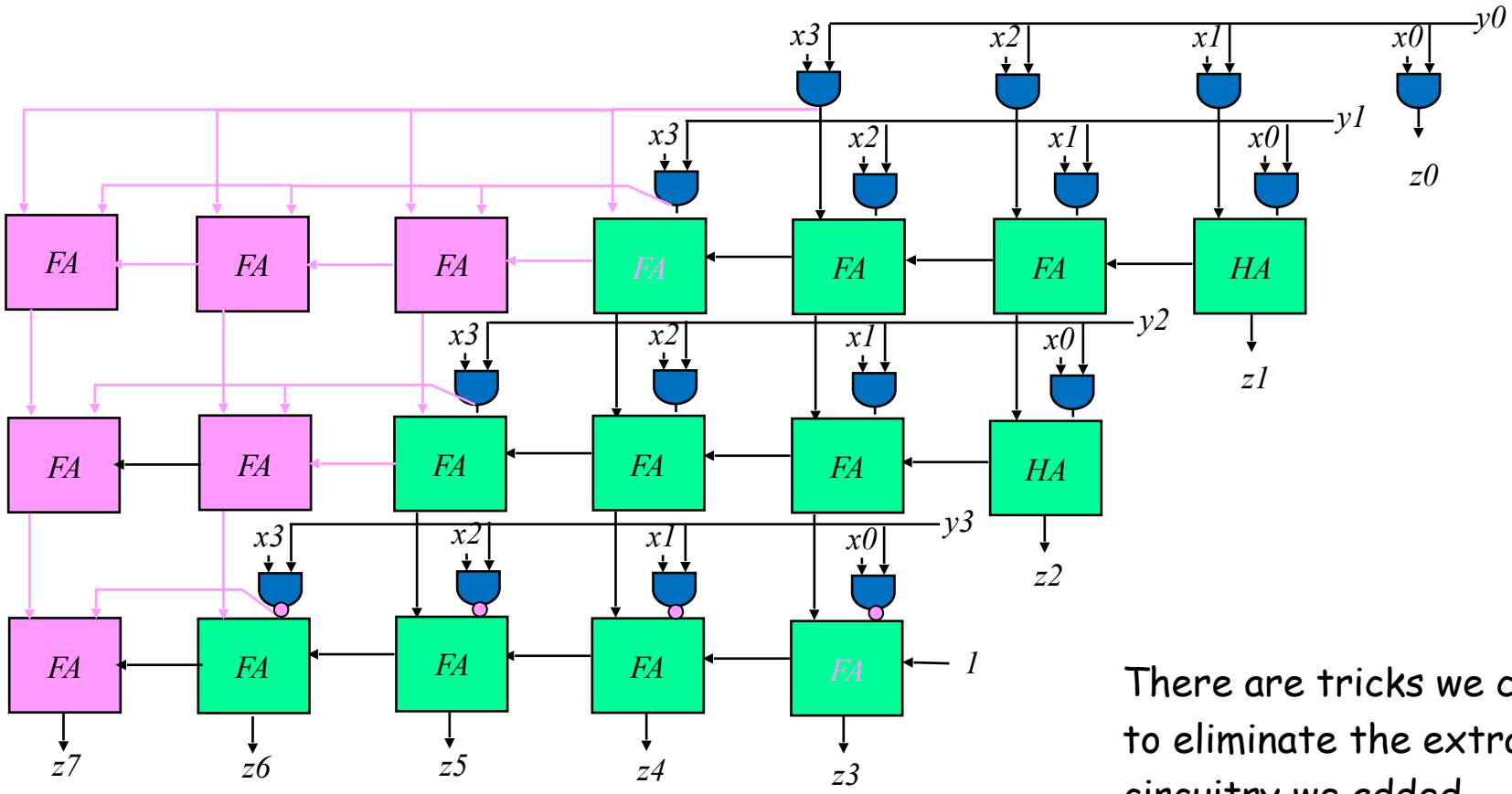
(-3) * (-2)

```
(-3)                    1   0   1        (X)
(-2)              *     1   1   0        (Y)
                    ---------------------
                    0 0 0 0 0 0          Y0*X =  0
                  + 1 1 1 0 1            2Y1*X = -6
                  - 1 1 0 1              4Y2*X = -12
                  ----------------------
(+6)                0 0 0 1 1 0
```

Signed Multiplication

```
                          X3    X2    X1    X0
                      *   Y3    Y2    Y1    Y0
                      ------------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
---------------------------------------
   Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```





There are tricks we can use to eliminate the extra circuitry we added…

(Baugh-Wooley Method)

Step 1: two's complement operands so high order bit is -2^{N-1}. Must sign extend partial products and subtract the last one

```
                          X3    X2    X1    X0
                      *   Y3    Y2    Y1    Y0
                      ------------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
-----------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```

Step 2: don't want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

```
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+                         1
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+                   1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
+             1
+ X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
+                         1
+       1
-       1     1     1     1
```

$-B = \sim B + 1$

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

```
                          X3Y0  X2Y0  X1Y0  X0Y0
+               X3Y1  X2Y1  X1Y1  X0Y1
+         X2Y2  X1Y2  X0Y2
+   X3Y3  X2Y3  X1Y3  X0Y3
+
+                         1
-       1     1     1     1
```

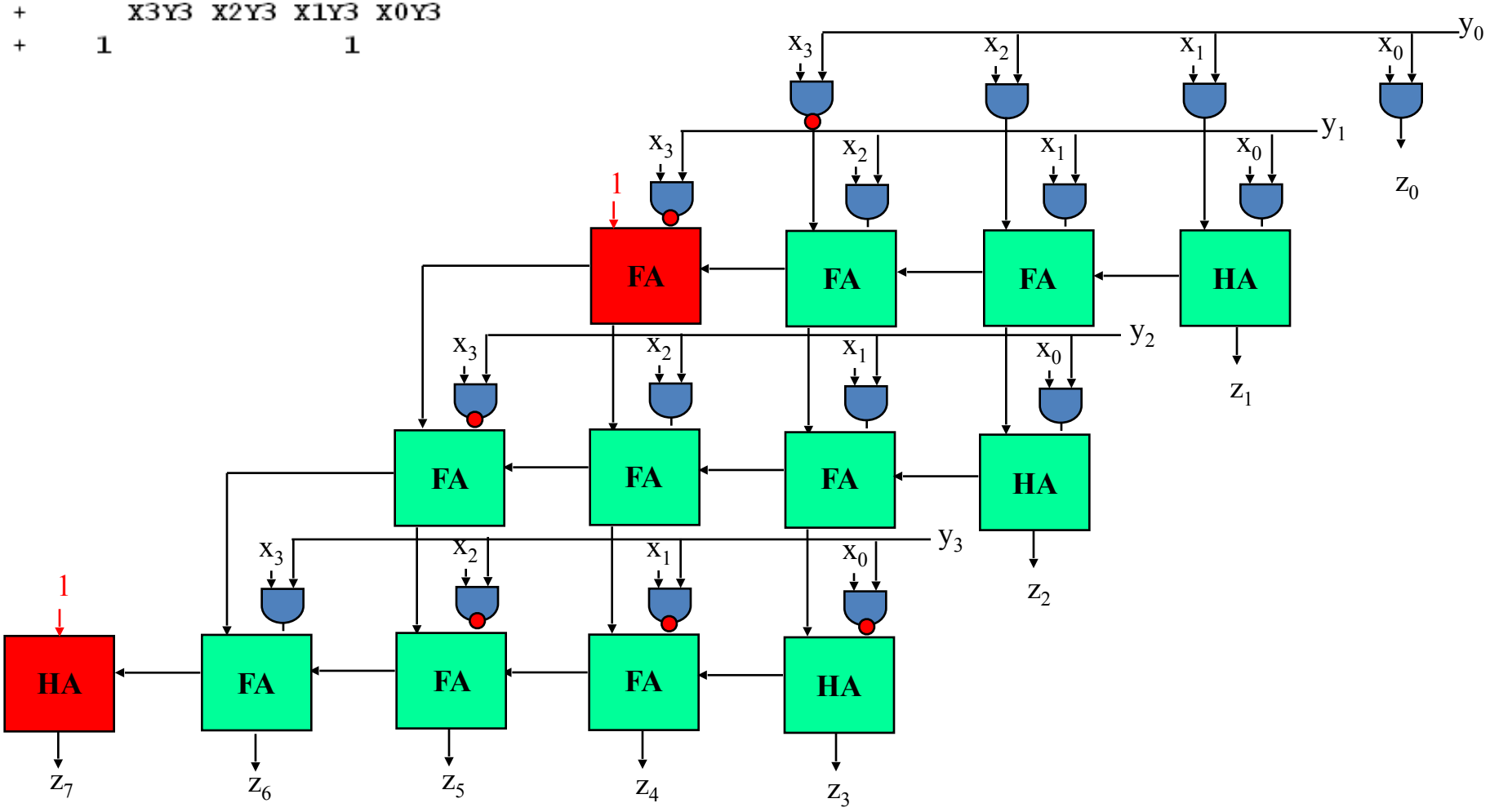
Step 4: finish computing the constants…

```
                          X3Y0  X2Y0  X1Y0  X0Y0
+               X3Y1  X2Y1  X1Y1  X0Y1
+         X2Y2  X1Y2  X0Y2
+   X3Y3  X2Y3  X1Y3  X0Y3
+                         1
+ 1                       1
```

Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

Signed Multiplication

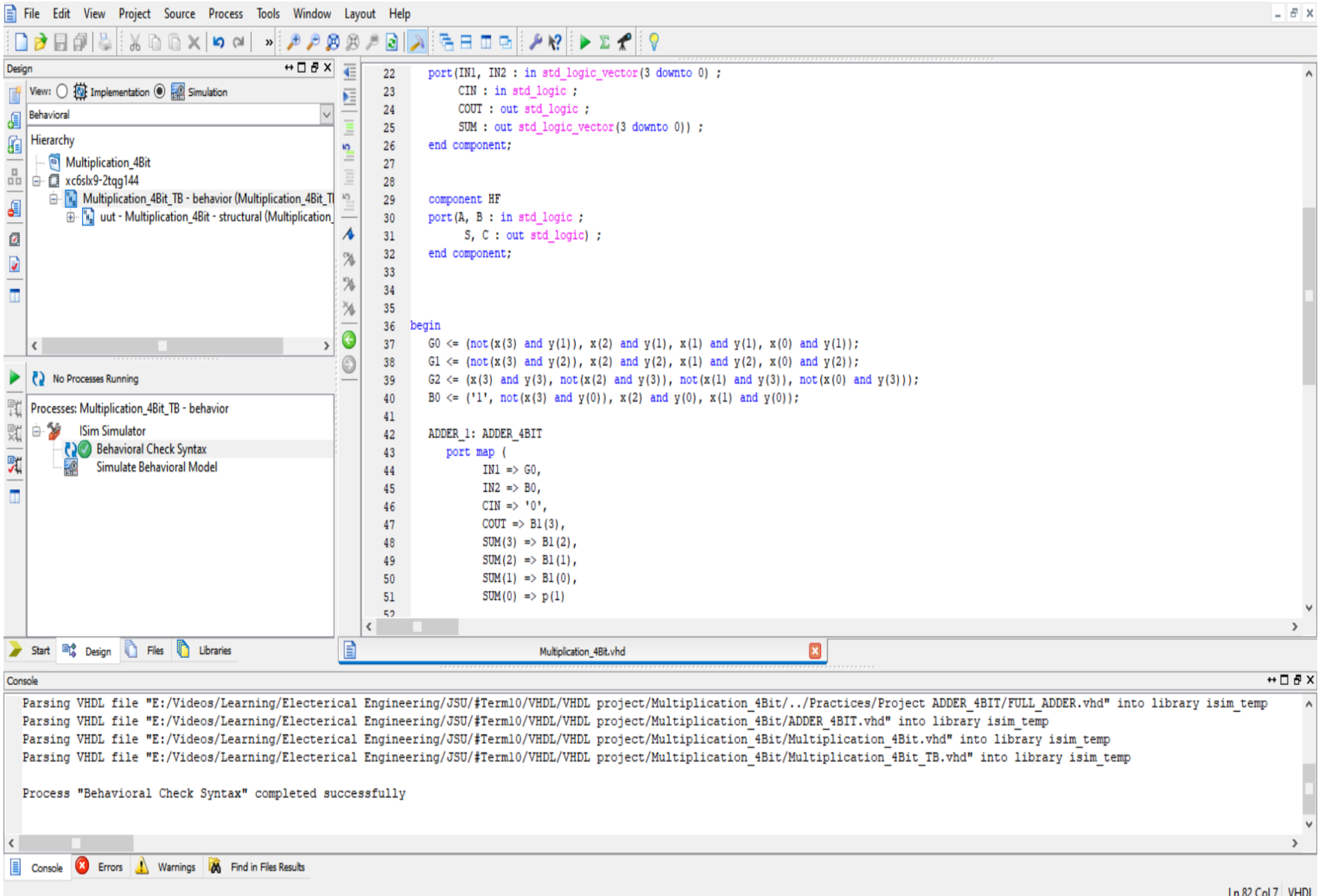
$$
\begin{array}{llllll}
 & & & \overline{X3Y0} & X2Y0 & X1Y0 & X0Y0 \\
+ & & \overline{X3Y1} & X2Y1 & X1Y1 & X0Y1 & \\
+ & & \overline{X2Y2} & X1Y2 & X0Y2 & & \\
+ & X3Y3 & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & \\
+ & 1 & & 1 & & &
\end{array}
$$



Multiplication_4Bit

```vhdl
--Mohammad Niknam 951843189
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Multiplication_4Bit is
    port (
        x: in  std_logic_vector (3 downto 0);
        y: in  std_logic_vector (3 downto 0);
        p: out std_logic_vector (7 downto 0));
end Multiplication_4Bit;

architecture structural of Multiplication_4Bit is
    signal G0, G1, G2:  std_logic_vector (3 downto 0);
    signal B0, B1, B2:  std_logic_vector (3 downto 0);
    signal TMP: std_logic;

    component ADDER_4BIT
    port(IN1, IN2 : in std_logic_vector(3 downto 0) ;
          CIN : in std_logic ;
          COUT : out std_logic ;
          SUM : out std_logic_vector(3 downto 0)) ;
    end component;

    component HF
    port(A, B : in std_logic ;
            S, C : out std_logic) ;
    end component;

begin
    G0 <= (not(x(3) and y(1)), x(2) and y(1), x(1) and y(1), x(0) and y(1));
    G1 <= (not(x(3) and y(2)), x(2) and y(2), x(1) and y(2), x(0) and y(2));
    G2 <= (x(3) and y(3), not(x(2) and y(3)), not(x(1) and y(3)), not(x(0) and y(3)));
    B0 <= ('1', not(x(3) and y(0)), x(2) and y(0), x(1) and y(0));

    ADDER_1: ADDER_4BIT
        port map (
            IN1 => G0,
            IN2 => B0,
            CIN => '0',
            COUT => B1(3),
            SUM(3) => B1(2),
            SUM(2) => B1(1),
            SUM(1) => B1(0),
            SUM(0) => p(1) );

    ADDER_2: ADDER_4BIT
        port map (
            IN1 => G1,
            IN2 => B1,
            CIN => '0',
            COUT => B2(3),
            SUM(3) => B2(2),
            SUM(2) => B2(1),
            SUM(1) => B2(0),
            SUM(0) => p(2));

    ADDER_3: ADDER_4BIT
        port map (
            IN1 => G2,
            IN2 => B2,
            CIN => '0',
            COUT => TMP,
            SUM => p(6 downto 3));

    Half_Adder: HF
        port map (
        A => TMP,
        B => '1',
        S => P(7)
        );

    p(0) <= x(0) and y(0);
end structural;
```

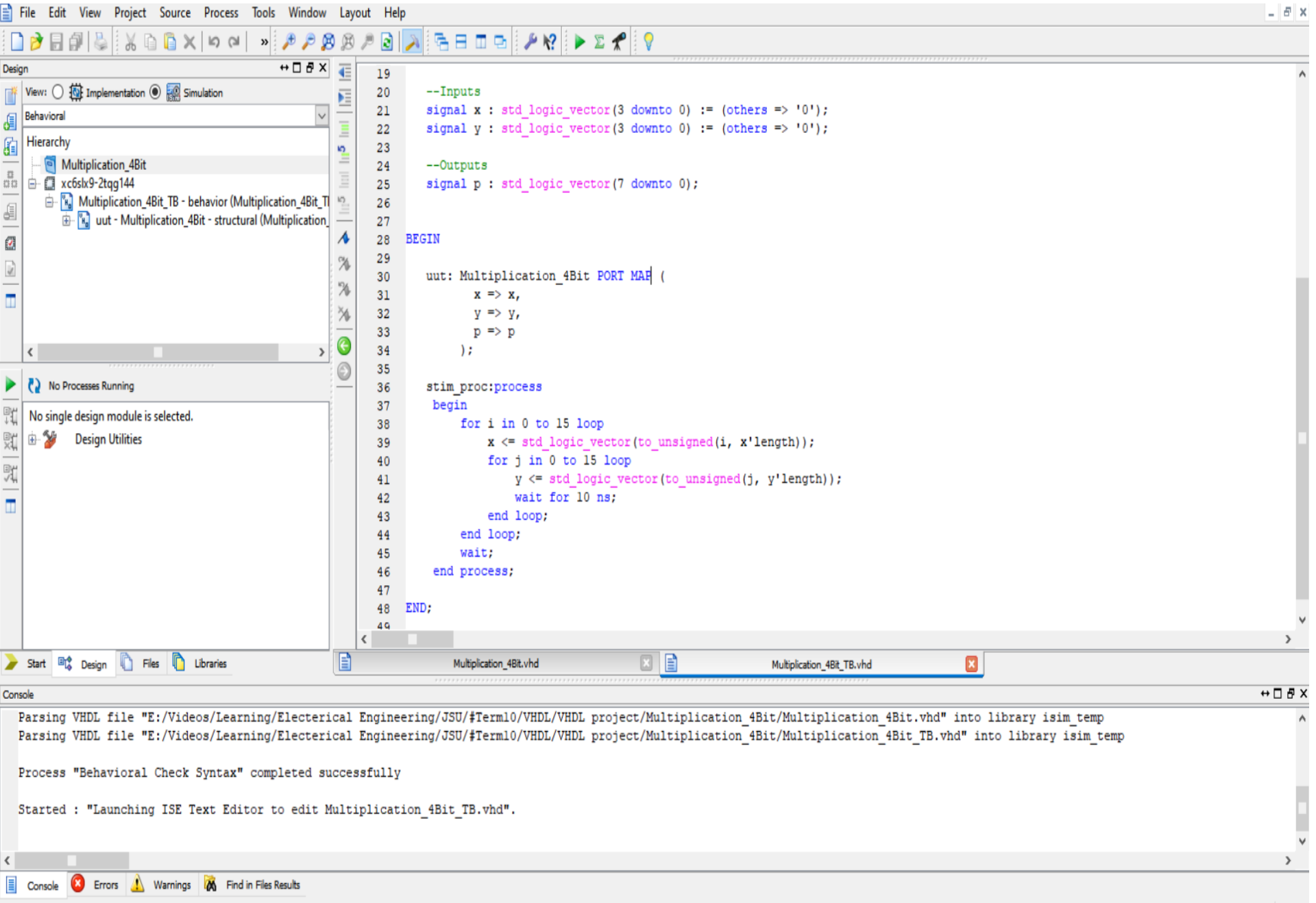
Multiplication_4Bit



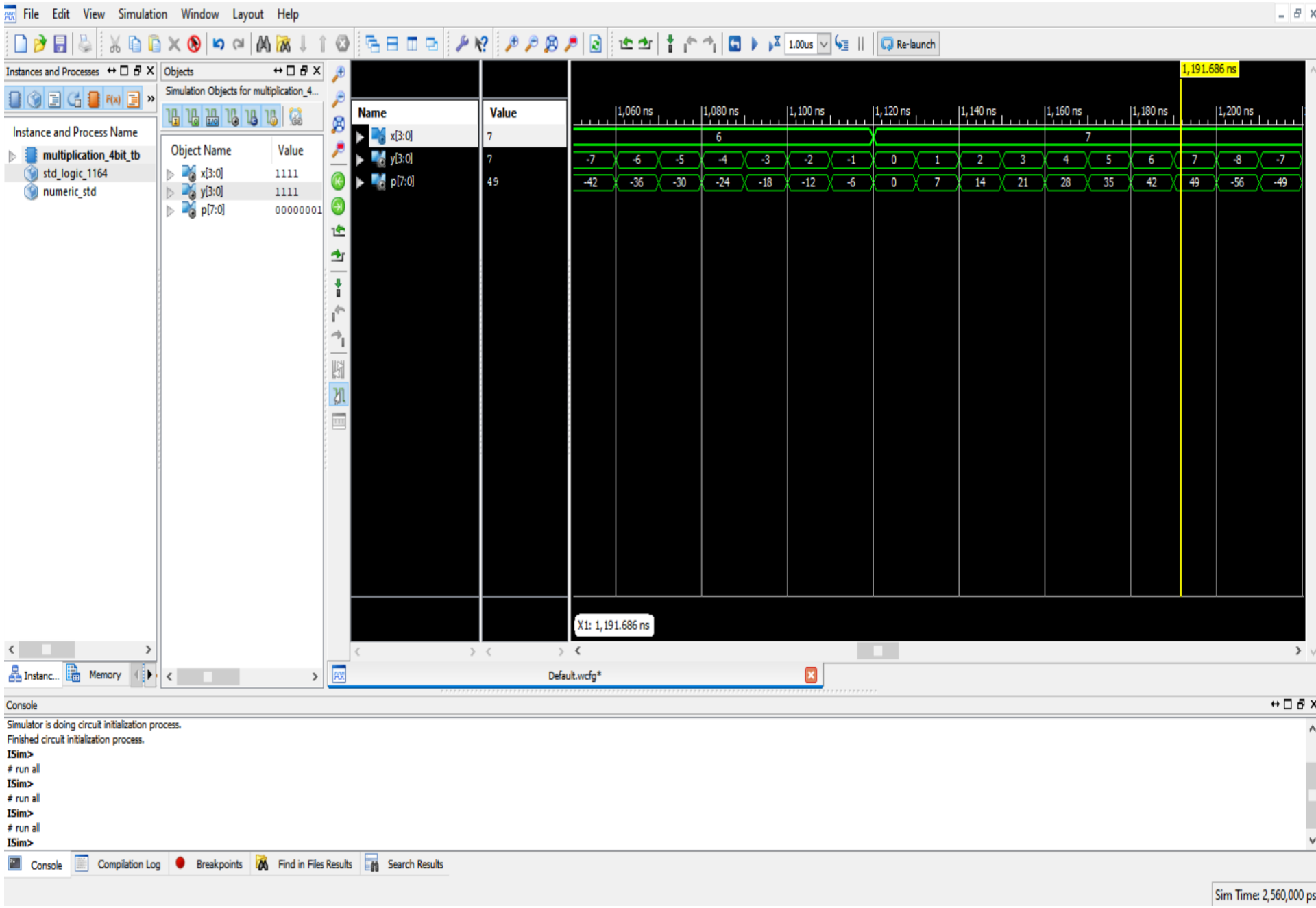
Multiplication_4Bit_TB

```
1   --Mohammad Niknam 951843189
2
3   librarLIBRARY ieee;
4   USE ieee.std_logic_1164.ALL;
5   use ieee.numeric_std.all;
6
7   ENTITY Multiplication_4Bit_TB IS
8   END Multiplication_4Bit_TB;
9
10  ARCHITECTURE behavior OF Multiplication_4Bit_TB IS
11
12
13      COMPONENT Multiplication_4Bit
14      PORT(
15           x : IN  std_logic_vector(3 downto 0);
16           y : IN  std_logic_vector(3 downto 0);
17           p : OUT  std_logic_vector(7 downto 0)
18          );
19      END COMPONENT;
20
21
22     --Inputs
23     signal x : std_logic_vector(3 downto 0) := (others => '0');
24     signal y : std_logic_vector(3 downto 0) := (others => '0');
25
26      --Outputs
27     signal p : std_logic_vector(7 downto 0);
28
29
30  BEGIN
31
32     uut: Multiplication_4Bit PORT MAP (
33            x => x,
34            y => y,
35            p => p
36          );
37
38      stim_proc:process
39      begin
40          for i in 0 to 15 loop
41              x <= std_logic_vector(to_unsigned(i, x'length));
42              for j in 0 to 15 loop
43                  y <= std_logic_vector(to_unsigned(j, y'length));
44                  wait for 10 ns;
45              end loop;
46          end loop;
47          wait;
48      end process;
49
50  END;
```

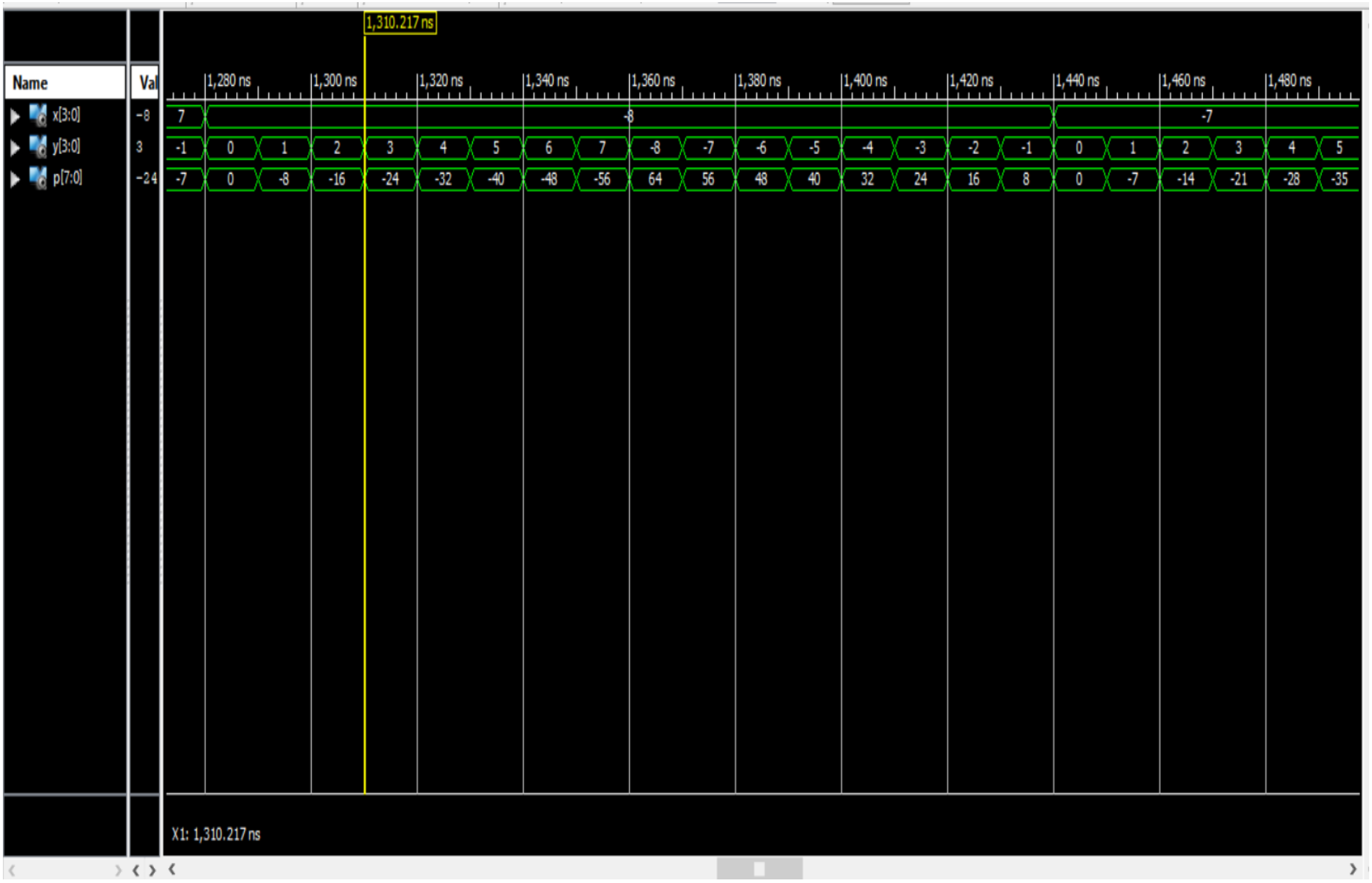
Multiplication_4Bit_TB



Simulation:


File Edit View Simulation Window Layout Help
Re-launch
Instances and Processes
Instance and Process Name
multiplication_4bit_tb
std_logic_1164
numeric_std
Objects
Simulation Objects for multiplication_4...
Object Name
Value
x[3:0]
1111
y[3:0]
1111
p[7:0]
00000001
Name
Value
x[3:0]
7
y[3:0]
7
p[7:0]
49
1,191.686 ns
1,060 ns
1,080 ns
1,100 ns
1,120 ns
1,140 ns
1,160 ns
1,180 ns
1,200 ns
X1: 1,191.686 ns
Default.wcfg*
Console
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
run all
ISim>
run all
ISim>
run all
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 2,560,000 ps

Simulation:


1,310.217 ns
Name
Val
1,280 ns
1,300 ns
1,320 ns
1,340 ns
1,360 ns
1,380 ns
1,400 ns
1,420 ns
1,440 ns
1,460 ns
1,480 ns
x[3:0]
-8
7
-8
-7
y[3:0]
3
-1 0 1 2 3 4 5 6 7 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5
p[7:0]
-24
-7 0 -8 -16 -24 -32 -40 -48 -56 64 56 48 40 32 24 16 8 0 -7 -14 -21 -28 -35
X1: 1,310.217 ns

Synthesize & RTL Schematic