**MULTIPLICATION ACCELERATION THROUGH TWIN PRECISION**

**A BAUGH WOOLEY IMPLEMENTATION.**

***Abstract* — *Multiplication is a complex arithmetic operation, which is reflected in its relatively high signal propagation delay, high power dissipation, and large area requirement. The twin-precision technique can reduce the power dissipation by adapting a multiplier to the bit-width of the operands being computed. The technique also enables an increased computational throughput, by allowing several narrow-width operations to be computed in parallel. The project is implemented by applying the twin-precision technique to multiplier schemes, such as Baugh–Wooley multiplier for both signed and unsigned 8 bit numbers*.**

**Note:** The report is formatted into various sections. In section I, multiplication and basic principles of twin precision are discussed. Section II briefs about the Project details. Section III contains the information about the multiplication for unsigned and signed numbers using the BW algorithm, the modes of operation. The entire project implementation, the functioning of the multiplier in various modes is described in section IV. The synthesis and simulation results can be found in section VI.

1. INTRODUCTION

Multiplication is the most frequently performed operation in applications like ALU, MAC units, digital filters, communication units ...etc. It is hence very important that the speed and precision must be increased while the area being reduced. The multiplication algorithm for an N bit multiplicand by N bit multiplier shown in the Fig.1.1

Y= Yn-1 Yn-2 .............Y2 Y1 Y0 Multiplicand

X= Xn-1 Xn-2 ………..... X2 X1 X0 Multiplier

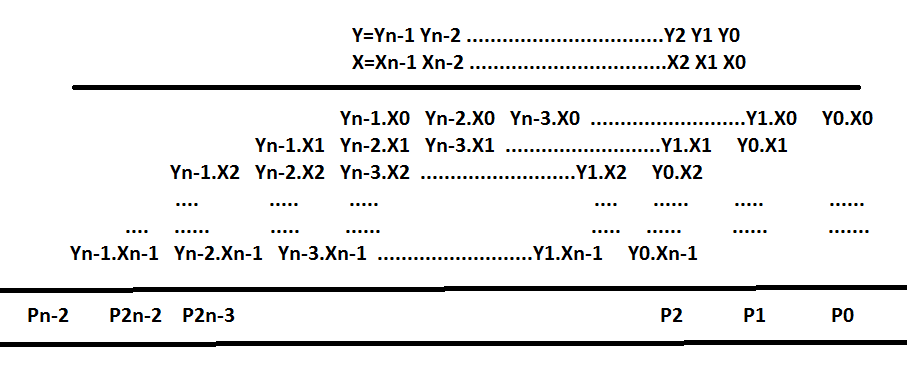


Fig.1.1: Multiplication of 2 binary numbers

1. *Multiplication algorithm and implementation logic*

Generally, AND gates are used to generate the Partial Products, PP, If the multiplicand is N-bits and the Multiplier is M-bits then there is N\* M partial products are generated. The way that the partial products are generated or summed up is the difference between the different architectures of various multipliers.

The equation for the addition is:

Multiplication of binary numbers can be decomposed into additions.

The two main blocks of the multiplier can be hence classified as Partial product generator and partial product adder.

Consider the multiplication of two 8-bit numbers A and B to generate the 16-bit product P.

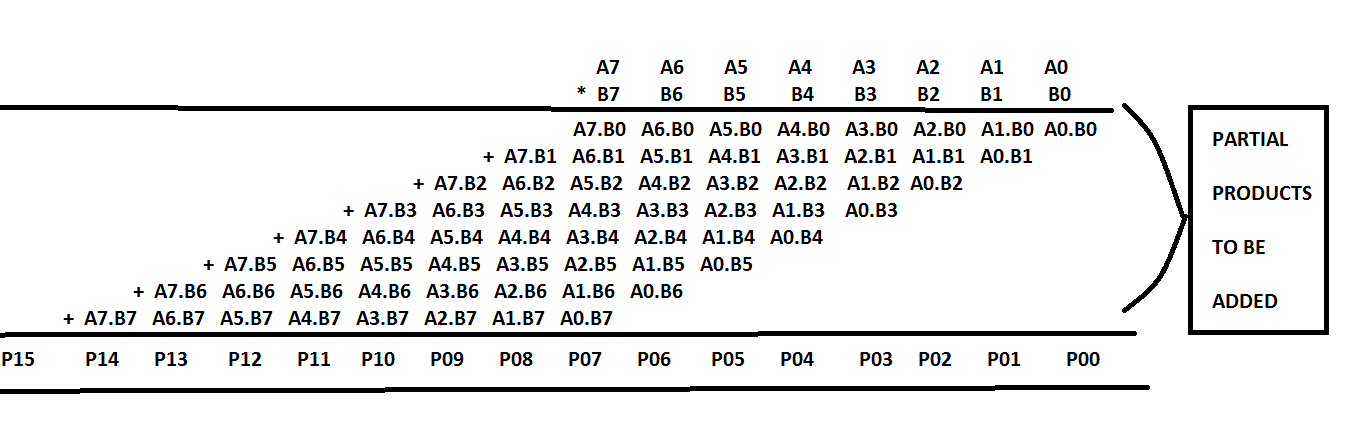


Fig1.2: Partial products and summation depiction of 8-bit multiplication

1. *Multiplication algorithm*
2. If the LSB of Multiplier is ‘1’, then add the multiplicand into an accumulator.
3. Shift the multiplier one bit to the right and multiplicand one bit to the left.
4. Stop when all bits of the multiplier are zero.

From above, multiplication has been changed to addition of numbers. If the Partial Products are added serially then a serial adder is used with least hardware. It is possible to add all the partial products with one combinational circuit using a parallel multiplier. However, it is possible also, to use compression technique then the number of partial products can be reduced before addition is performed.

1. *Digital Multipliers*

The classification of Digital multipliers is as shown in the next figure.

All the multipliers shown in the figure have some or the other limitation or a tradeoff between the throughput, speed and area - thus affecting the performance. The most commonly used multiplier for general multiplications is array multiplier, based on this the further improvements have been formed to develop the twin precision multiplication using Modified booth, Baugh-Wooley implementation.

Consider an array multiplier which is one of the traditionally used multipliers in which an array of identical cells generates new partial product and accumulation of it at the same time. Pipelines are used at each level and the result from the adder can be latched at each level and used as input to the next level circuit adder. For n-bit\*m-bit array multiplier, n\*m two-input AND gates and (m-1) units of n-bit adders are required. Which means that the number of logic gates required to design an array multiplier is large which makes it not a wise-selection when area and latency are considered.

A multiplier can be designed to be appropriate for the above requirements by following:

1. Power reduction by reducing operand bit width.
2. Parallel operation so that we can get high throughput.
3. *Multiplication and bit width operations*

As discussed in the introduction, multiplication has important function in digital signal processing, for example in digital filtering and Fast Fourier Transform(FFT) processing. Many multiplier schemes were purposed in early stages by Burton and Noaks(1968), De Mori(1969), and Guilt(1969) for positive numbers, and by Baugh-Wooley (1973) and Hwang (1979) for numbers in two's complement form.

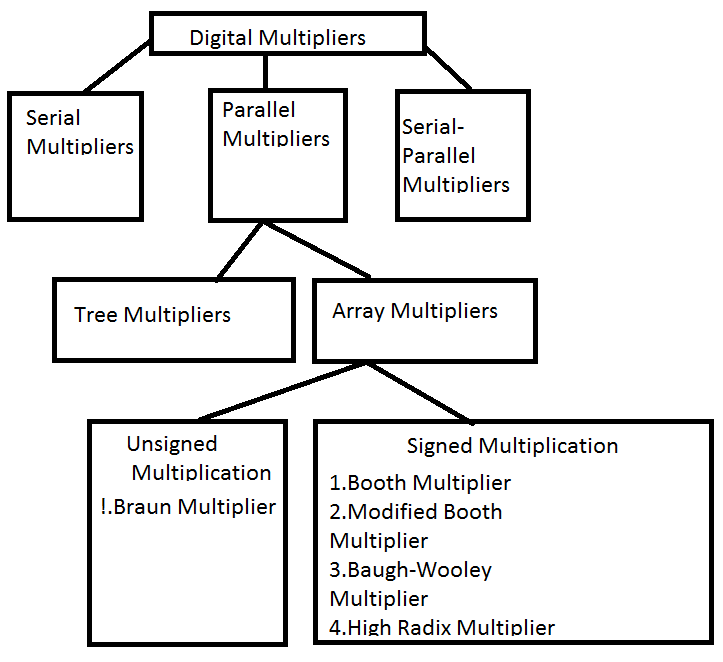


Fig 1.3: Classification of Digital Multipliers.

The performance requirements of the multipliers are increasing day by day for a wide range of bit widths. In most of the integer applications the bit width of the instructions used are the instructions with bit width less than 16 bits. Such operations are called as the *narrow width operations*, which improve the throughput. This can be achieved by computing several narrow width operations in parallel on a full width data path.

1. *Twin Precision*

In twin precision technique, narrow width operations are performed on a full width data path.

Precision is defined as the number of digits used to represent a number. Twin precision is nothing but using two different precisions with in a single multiplication i.e. dividing the bit width of the operands into two equal bit widths. Consider an operand with precision of octal, using the twin precision technique the precision of the operand is divided into two quad operands.

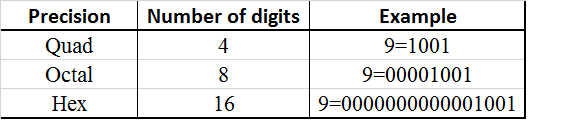


Fig 1.4: Diverse types of precision.

This technique provides the same power reduction as the operand guarding and the area overhead is also reduced, but it holds a penalty of small delay. The Twin precision technique can be applied for array multipliers, Baugh-Wooley multipliers and Modified-Booth Multipliers to obtain double throughput.

1. PROJECT DESCRIPTION
2. ***Purpose of the project***

The project is all about implementation of multiplier which has the characteristics of low energy requirement, high speed and high throughput and low area, acquiring twin precision. In the method to obtain Double or Twin precision, the operands of the multiplier will be made into narrow width operands, by which the power dissipation and throughput of the multipliers is improved.

For applications where the demand on precision is relaxed, the multiplier can perform N/2-bit multiplications in parallel while expending only a fraction of the energy of a conventional N-bit multiplier. For applications with high demands on throughput, the multiplier can perform two independent N/2-bit multiplications in parallel.

The multiplier designed in this projects functions in 3 modes which is described in later sections (section IIb, III.C). The mode in which the multiplier functions is controlled by the control signals.

1. ***Project Implementation***

Twin-precision technique can be applied to signed multiplier schemes, such as Baugh–Wooley and modified-Booth multipliers.

**How much is implemented in this project:** This project describes how to apply twin-precision technique to Baugh-Wooley multipliers for 8\*8 bit signed operation.

**How it works:** The Baugh-Wooley multiplier implemented in this project works in both full precision mode and twin precision modes for signed numbers and in general mode for unsigned numbers. Twin-precision multiplier in normal operation mode efficiently performs 8-bit multiplications, two 4-bit multiplications in parallel when high precision is demanded.

**Inputs**: The inputs for the multiplier are the signed or unsigned 8 bit numbers along with the control signals which determine the mode of operation.

**Software:** **Xilinx ISE 14.7** is used with VHDL coding.

**Xilinx Vivado** tool is also used for high level power and area analysis.

**What is leftover to be implemented:** The Twin precision technique can be applied to Modified booth multipliers also. The throughput for booth multipliers is high compared to the Baugh-Wooley multipliers. This implementation is not done in the project.

1. ***Difficulties and Challenges***

The key difficulties in outlining a twin-exactness multiplier are to constrain the effect of adaptability on power dissipation, latency, and range.

Large count of the AND gates and adders make the design complex. Since there are 3 modes of operation, the control signals design and the logical implementation for a multiplier to function in 3 modes was very complicated.

Not only the logical and physical implementation, but there were many difficulties during the initial stages to understand the functionality and code as per requirements, debug the logical errors.

1. MULTIPLIERS FOR SIGNED AND UNSIGNED NUMBERS
2. *Flexible multiplier*

Late advancement at the miniaturized scale design level demonstrates that there is an expanding interest for datapath parts that are equipped for performing calculations with variable operand estimate, e.g. adders fit for doing both N and N/2-b multiplication. By utilizing just, a piece of the datapath part for calculation, it has been shown that decreases in the aggregate power dissemination can be affected.

Datapath parts that can perform both one N, one single N/2, or two N/2-b operations give the designer an opportunity to outline a framework which can adapt to changing modes, for example, low-control, high-throughput, or high-precision operation. Such a datapath component could be used for dynamic power reduction by using the same kind of logic for detecting if the effective bit rate is within N/2-b precision, it is possible to control at what precision the datapath component should be operating. This versatile type of datapath component is also suitable for systems in which several applications, having quite different requirements on precision and/or throughput, are executed. Furthermore, such a datapath component could prove useful in processors that can support several instructions sets.

In the flexible multiplier, we can consolidate N and N/2-b duplications in a similar N-bit tree multiplier: we call this a twin-precision multiplier.

In comparison to tree multipliers, however, an array multiplier is slow and power hungry which makes it a poor design choice when a fast and efficient multiplier is needed.

1. *Modes of Operation*

The multiplier designed in this project is flexible and adapts according to the requirement. In this project, the multiplier operates in 3 modes.

The various Modes of operation:

*Mode 1*: Multiplier for Unsigned numbers. (This is like the array multiplier as described in section III.C).

*Mode 2*: Multiplier with BW- Full precision (section IV.A)

*Mode 3*: Multiplier with BW-Twin precision. (section IV.B)

1. *Multiplication of unsigned numbers*

The multiplication of two 8bit unsigned numbers is performed in this project in a similar fashion to the array multiplier.

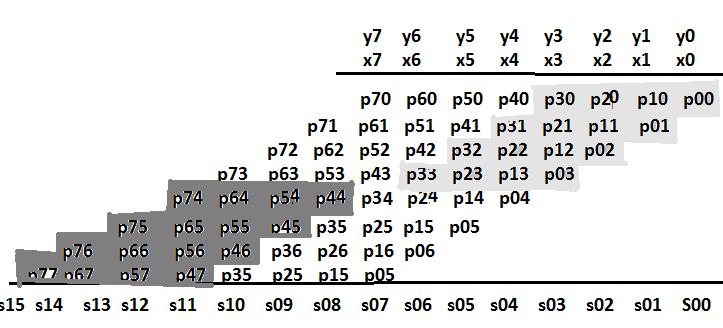
In view of a basic portrayal of an array multiplier, Figure 3.1, clearly if the incomplete item bits not being utilized as a part of a low-precision multiplication are set to zero, the array multiplier will deliver the right outcome without the need of any extra logic. The 2-input AND gates corresponding comparing to the partial product bits that are not being used in the low-precision multiplication can be replaced by 3-input AND gates to force those bits to zero (When performing just a single N/2-b multiplication it is possible to set the most significant bits of the operands to zero).

Fig 3.1: Partial product representation of a 4-bit multiplication in an 8-bit multiplier and Illustration of an unsigned multiplication, where a 4-bit multiplication shown

in light grey, is computed in parallel with a second 4-bit multiplication (shown in dark grey) (mode 1:BW-Unsigned numbers multiplication)

When doing an N/2-b multiplication within an N-b multiplier only one quarter of the logic is being used, as seen in grey. This makes it possible to use the multiplier for two parallel and independent N/2-b multiplications. We can partition the partial product bits of the N-b multiplier, such that an N/2-b multiplication can be performed in the Least Significant Part (LSP) of the multiplier in parallel with another N/2-b multiplication in the Most Significant Part (MSP), without using any additional logic in the partial product reduction tree, as seen in grey and black, respectively, in Fig 3.1. To be able to switch between N, N/2, or two N/2-b multiplications, the 2-input AND gates used to create the partial products need to be replaced with 3-input AND gates and two control signals for selecting the operating mode of the multiplier need to be introduced.

1. *Signed Multiplication According to Baugh-Wooley*

**Algorithm**: The BW algorithm is a relative straightforward way of doing signed multiplications. The algorithm for an N-bit case, where the creation of the reorganized partial-product array comprises three steps:

1. The most significant partial product of the first (N – 1) rows and the last row of partial products except the most significant must be negated,
2. A constant one is added to the Nth column.
3. the most significant bit (MSB) of the result is negated.

Baugh-Wooley algorithm Modified Booth does not impose any fundamental problems to the twin-precision concept. The Baugh-Wooley twin- precision multiplier proficiently performs possibly one N-b multiplication, one single N/2-b multiplication, or two N/2-b multiplications in parallel.

Here, signed multiplication is performed by:

1.Inverting all partial product bits that are results of the most significant bit (MSB) of exactly one of the operands, Figure 3.2.

2.For each executed multiplication, a logical one (framed) is added to column N (column 0 is to the far right in Figure 3.3).

3.The MSB of the product is inverted. This is directly mapped onto the tree multiplier as shown in Fig 3.3.

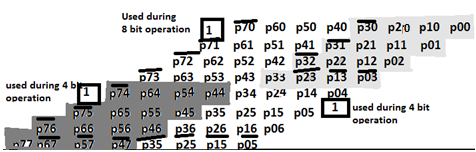


Fig 3.2: Example showing the inverted partial product bits of two signed 4-b multiplications within a signed 8-b multiplication.

To generate the inverted partial product bits, we replaced the AND gates relating to the reversed bits with NAND gate followed by XOR gate. The choice to either invert or not invert the signal from the NAND gate makes it conceivable to switch amongst signed and unsigned multiplication.

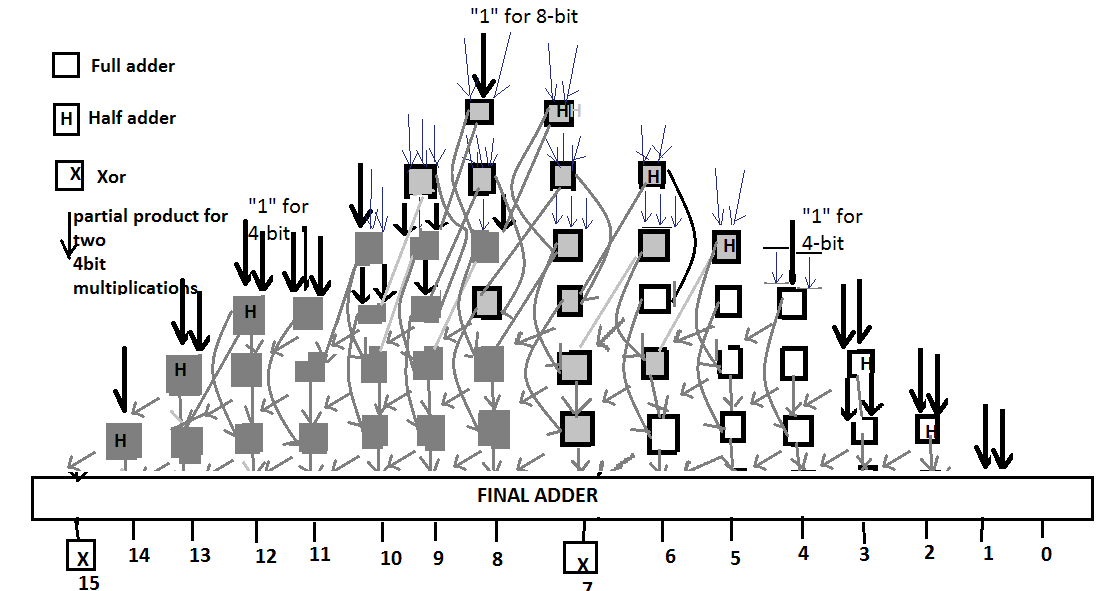


Fig 3.3: Signed 8-b multiplier capable of doing two signed 4-b multiplications using Baugh-Wooley.

The inversion of the MSB of the product is also done with an XOR gate. The insertion of the logical one to column N of the multiplication is straightforward for the N-b and the N/2-b multiplication in the LSP by changing the half adder of that column to a full adder and adding the logical one to the new adder. For the N/2-b multiplication in the MSP there is no half adder that can be replaced, but an extra level of half adders has to be added, seen at the far left in Figure 3.4. This added level of half adders does not increase the delay for the N-b multiplication, since none of the half adders are in the critical path.

1. FULL PRECISION AND TWIN PRECISION MODES OF BW MULTIPLIER-8 bits.
2. *Full precision implementation using the Baugh-Wooley(BW) Algorithm.*

In this mode, the multiplier functions in Full precision mode for the signed numbers using BW algorithm. Fig 4.1 illustrates this mode of operation. During this operation, a full 8-bit multiplication occurs with minor changes in the bits as per Baugh Wooley algorithm.

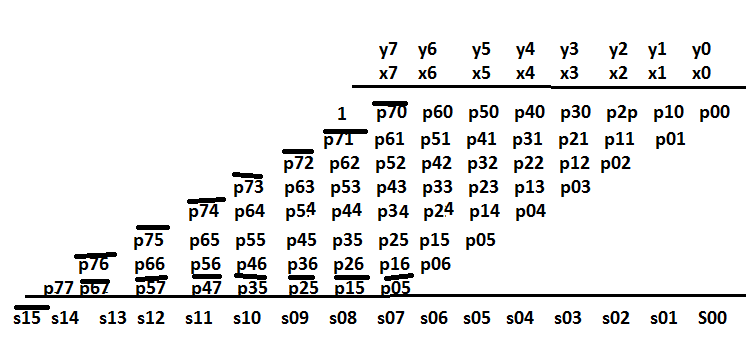
****

Fig 4.1: Illustration of an 8-bit Baugh-Wooley multiplication. (mode 2: Full precision mode for Signed numbers)

1. *Twin-Precision Implementation using the Baugh-Wooley(BW) algorithm.*

To combine twin-precision with BW is not as simple as for the unsigned multiplication, where only parts of the partial products needed to be set to zero. To be able to compute two signed N=2 multiplications, it is necessary to make a more sophisticated modification of the partial-product array. Fig. 4.2 shows an illustration of an 8-bit BW multiplication, where two 4-bit multiplications have been depicted in white and black, i.e. twin precision mode – mode3.

When comparing the illustration of Fig. 4.1 with that of Fig. 4.2 one can see that the only modification needed to compute the 4-bit multiplication in the MSP of the array is an extra sign bit '1' in column S12. For the 4-bit multiplication in the LSP of the array, there is a need for some more modifications.

Looking at the active partial-product array of the 4-bit LSP multiplication (shown in white), we see that the most significant partial product of all rows, except the last, needs to be negated. For the last row it is the opposite, here all partial products, except the most significant, are negated. Also for this multiplication a sign bit '1' is needed, but this time in column S4. At last, the MSB of the result needs to be negated to get the correct result of the two 4-bit multiplications.

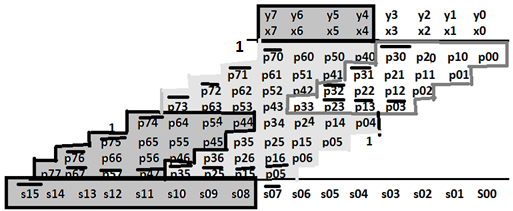


Fig 4.2: Illustration of a signed 8-bit multiplication, using the Baugh-Wooley algorithm, where a 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in dark grey. (mode 3: Twin precision mode for signed numbers)

To allow for the full-precision multiplication of size N to coexist with two multiplications of size N=2 in the same multiplier, it is necessary to modify the partial-product generation and the reduction tree. For the N=2-bit multiplication in the MSP of the array all that is needed is to add a control signal that can be set to high, when the N=2-bit multiplication is to be computed and to low, when the full precision N multiplication is to be computed. To compute the N=2-bit multiplication in the LSP of the array, certain partial products need to be negated.

This can easily be accomplished by changing the 2-input AND gate that generates the partial product to a 2-input NAND gate followed by an XOR gate. The second input of the XOR gate can then be used to invert the output of the NAND gate. When computing the N=2-bit LSP multiplication, the control input to the XOR gate is set to low making it work as a buffer. When computing a full-precision N multiplication the same signal is set to high making the XOR work as an inverter.

At last, the MSB of the result needs to be negated and this can again be achieved by using an XOR gate together with an inverted version of the control signal for the XOR gates used in the partial-product generation. Setting unwanted partial products to zero can be done by 3-input AND gates as for the unsigned case.

Fig. 4.3 shows an implementation of a twin-precision 8-bit BW multiplier. The modifications of the reduction tree compared to the unsigned 8-bit multiplier in Fig. 3.3 consist of three things;

1. the half adders in column 4 and 8 have been changed to full adders in order to fit the extra sign bits that are needed,
2. for the sign bit of the 4-bit MSP multiplication there is no half adder that can be changed in column 12, so here an extra half adder has been added which makes it necessary to also add half adders for the following columns of higher precision, and
3. XOR gates have been added at the output of column 7 and 15 so that they can be inverted.

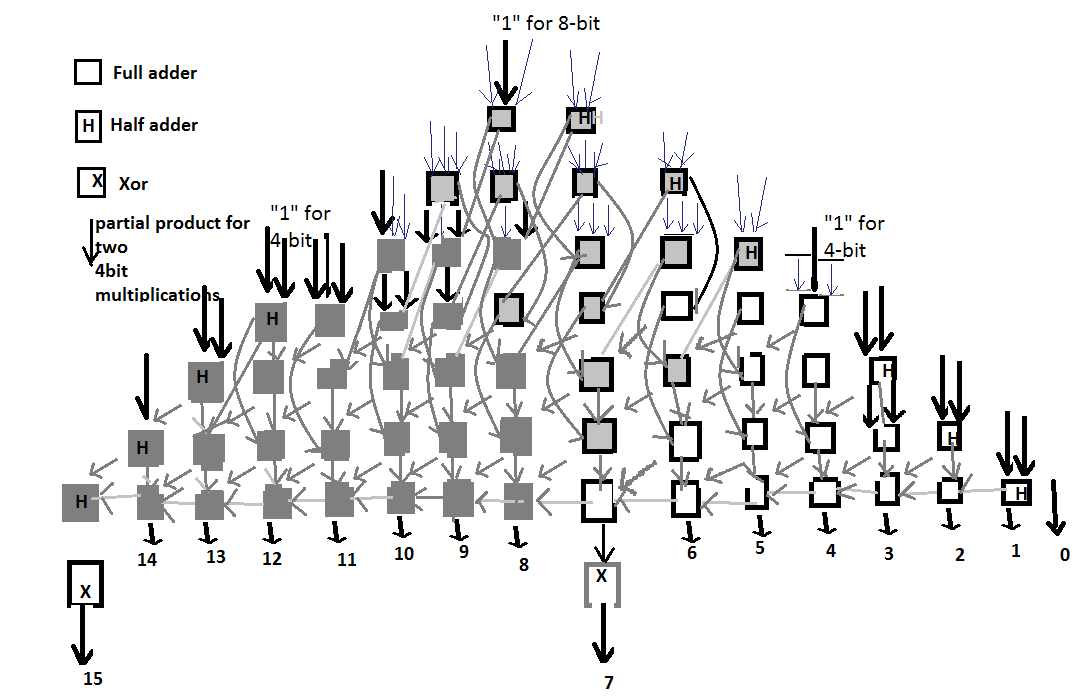


Fig 4.3: Block diagram of a signed 8-bit multiplication, using the Baugh-Wooley algorithm, where a 4-bit multiplication, shown in white, is computed in parallel with a second 4-bit multiplication, shown in black.

The simplicity of the BW implementation makes it easy to also compute unsigned multiplications. All that is needed is to set the control signals accordingly, such that none of the partial products are negated, the XOR gates are set to not negate the result and all the sign bits are set to zero.

The logic gates that are to be used as control signals follow NAND-XOR, NAND-AND, AND\_AND logic.

For NX logic: C1=1, Y= AB for Full Precision,

C1=0, Y= (AB)’ for Twin Precision- Certain partial products to be negated.

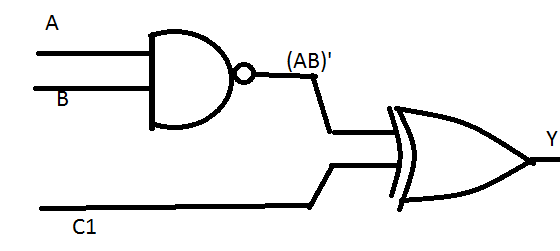
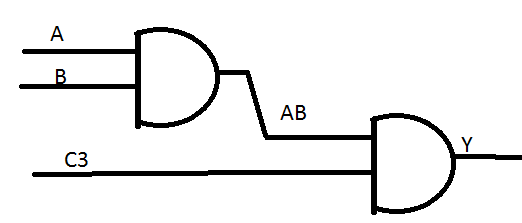


Fig 4.4: NAND-XOR(NX) logic.

For AA logic, C3 = 1, Y = AB For Full Precision,

C3 = 0, Y = 0 For Twin Precision- unwanted partial products to be set to zero.

Fig 4.5: AA (AND -AND logic)

****

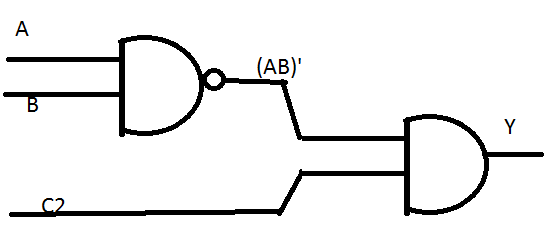


Fig 4.6: NAND-AND logic(NA)

For NA logic: C2 = 1, Y= (AB)’ For Full Precision,

C2 = 0 Y= 0 For Twin Precision

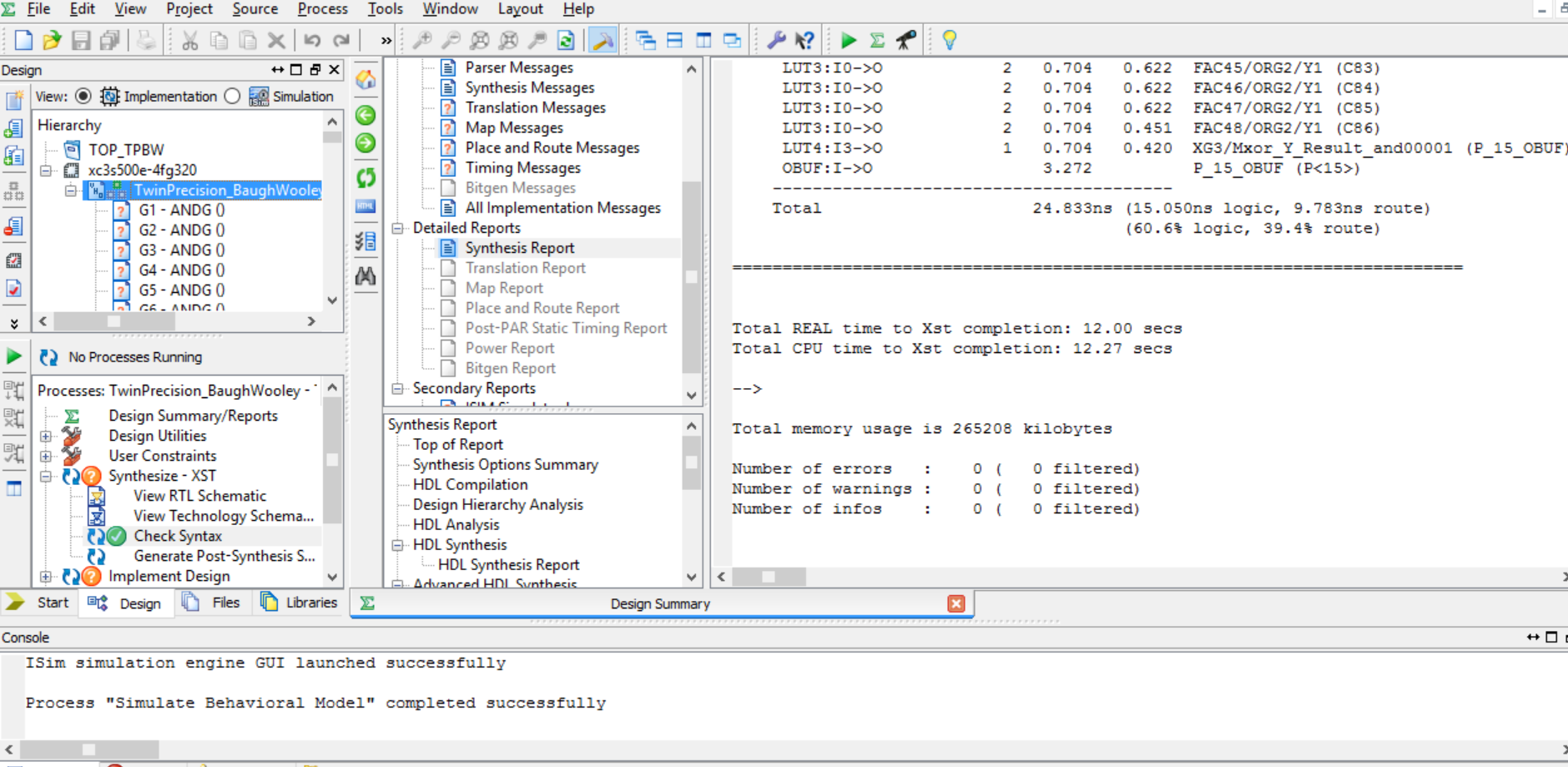
*C.Final adder*

The choice of final adder is very important to get short delay for both N and N/2-b multiplications. The delay profile of the multiplier varies with the multiplication precision. If adder scheme is used to reduce the delay for the N-b multiplication, it could bring lengthy delays for the two independent N/2-b multiplications. In order to not increase the delay too much for the N/2-b multiplications, it is important to have a final adder that is fast for both N and N/2-b multiplications. In this project, I have used ripple carry adder.

1. SYNTHESIS AND SIMULATION RESULTS

***IMPLEMENTATION:***

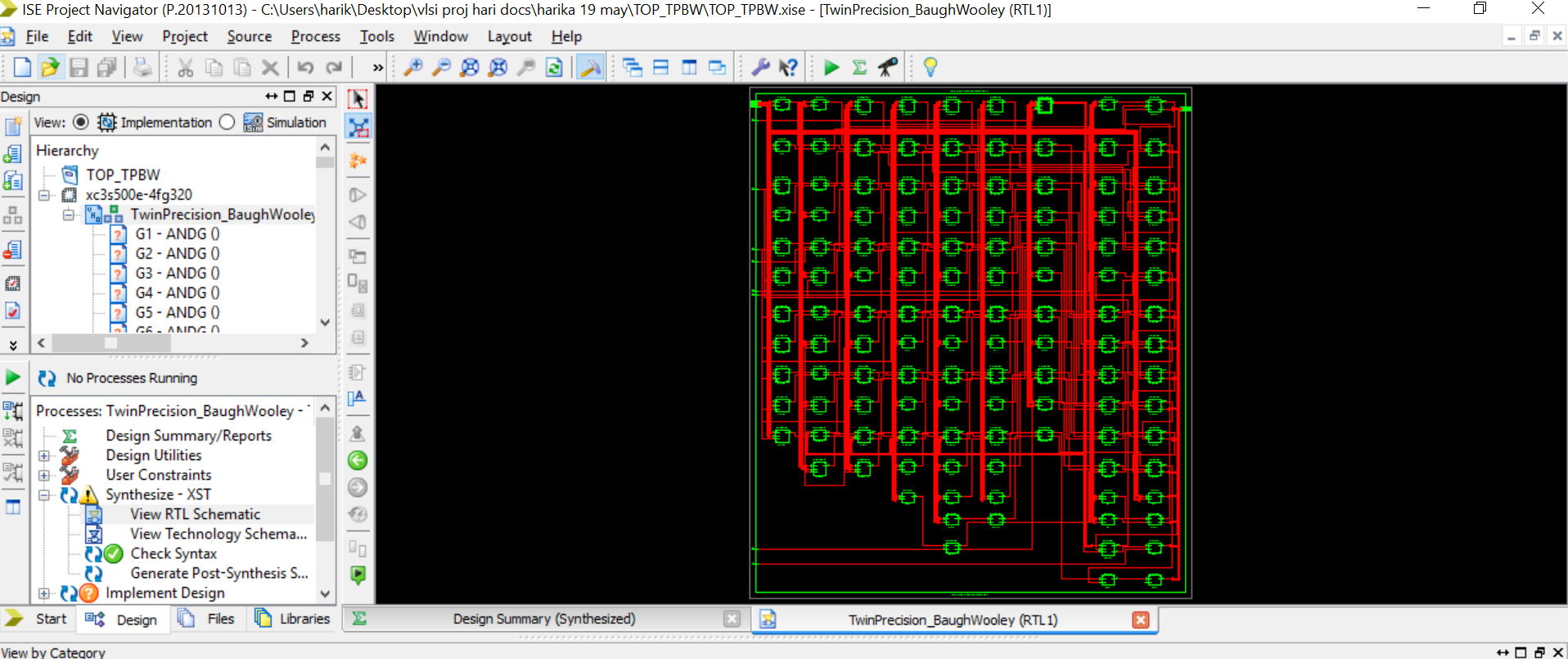
1.Syntax Check:



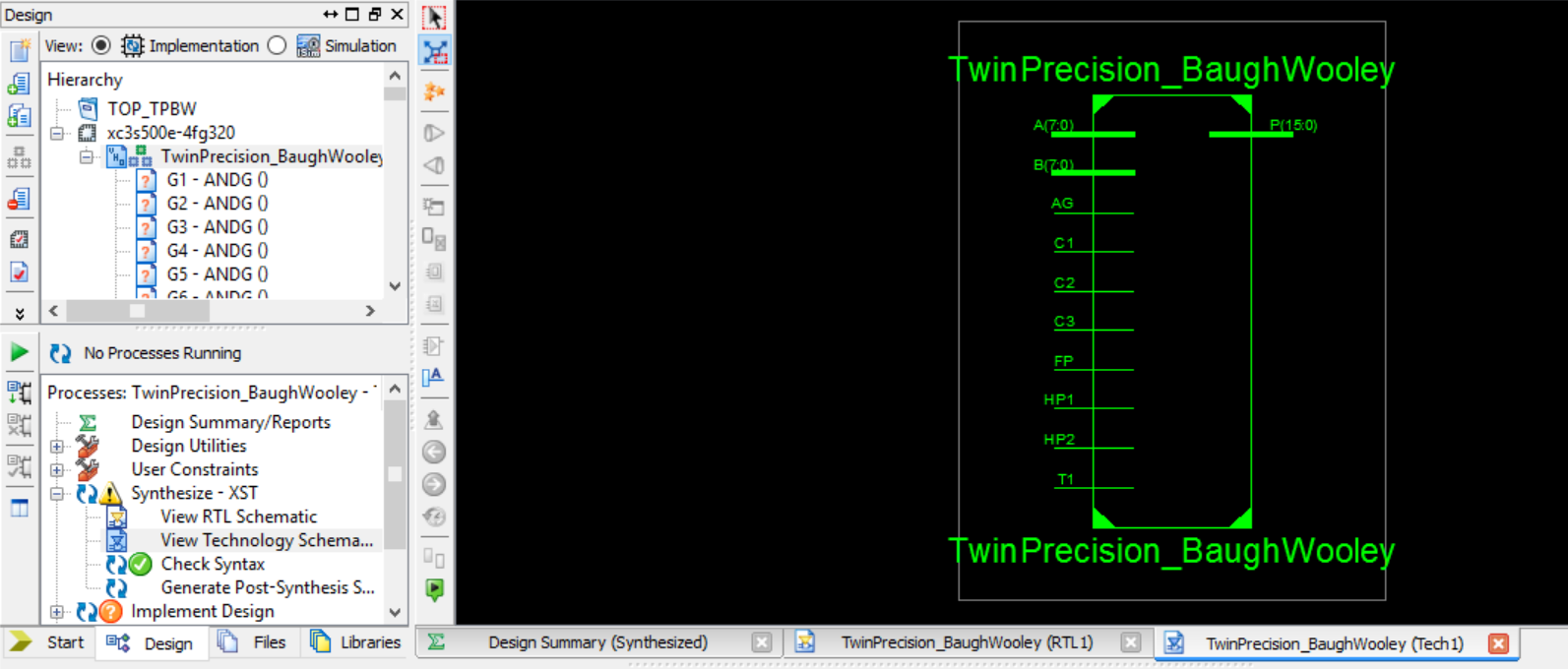
2.RTL schematic:



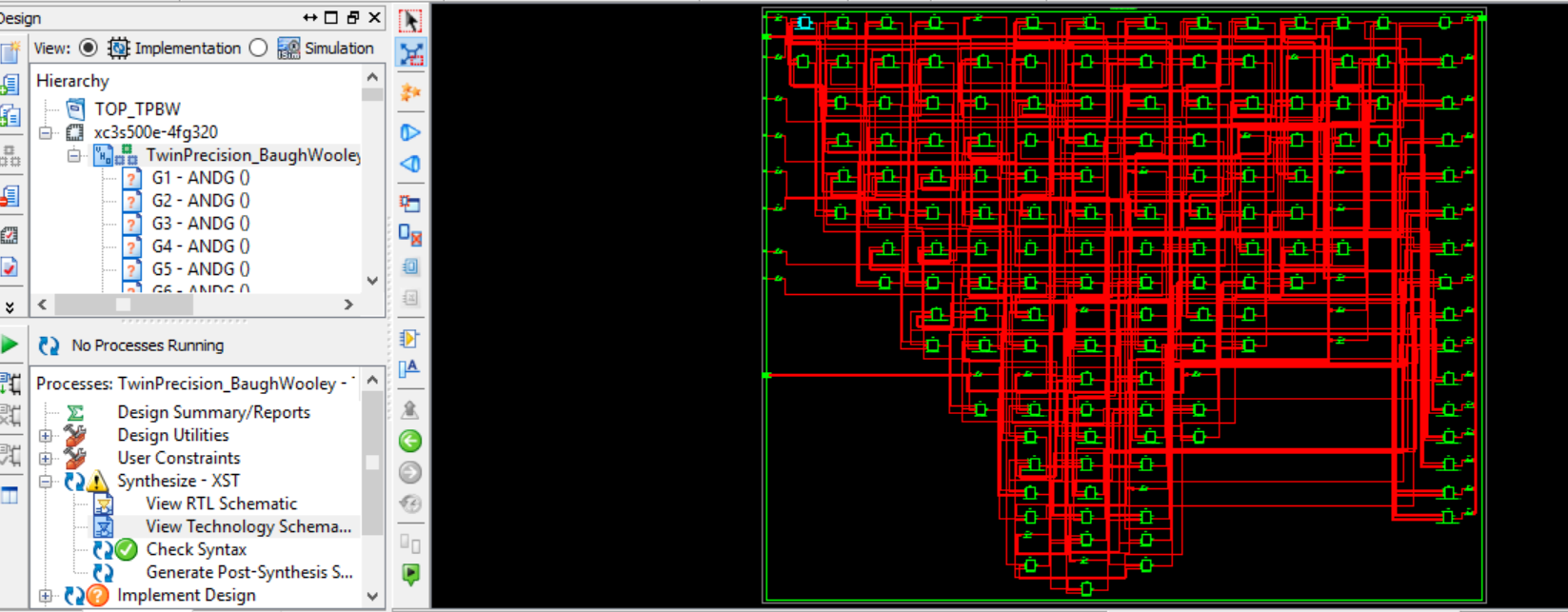
3.SUB RTL:



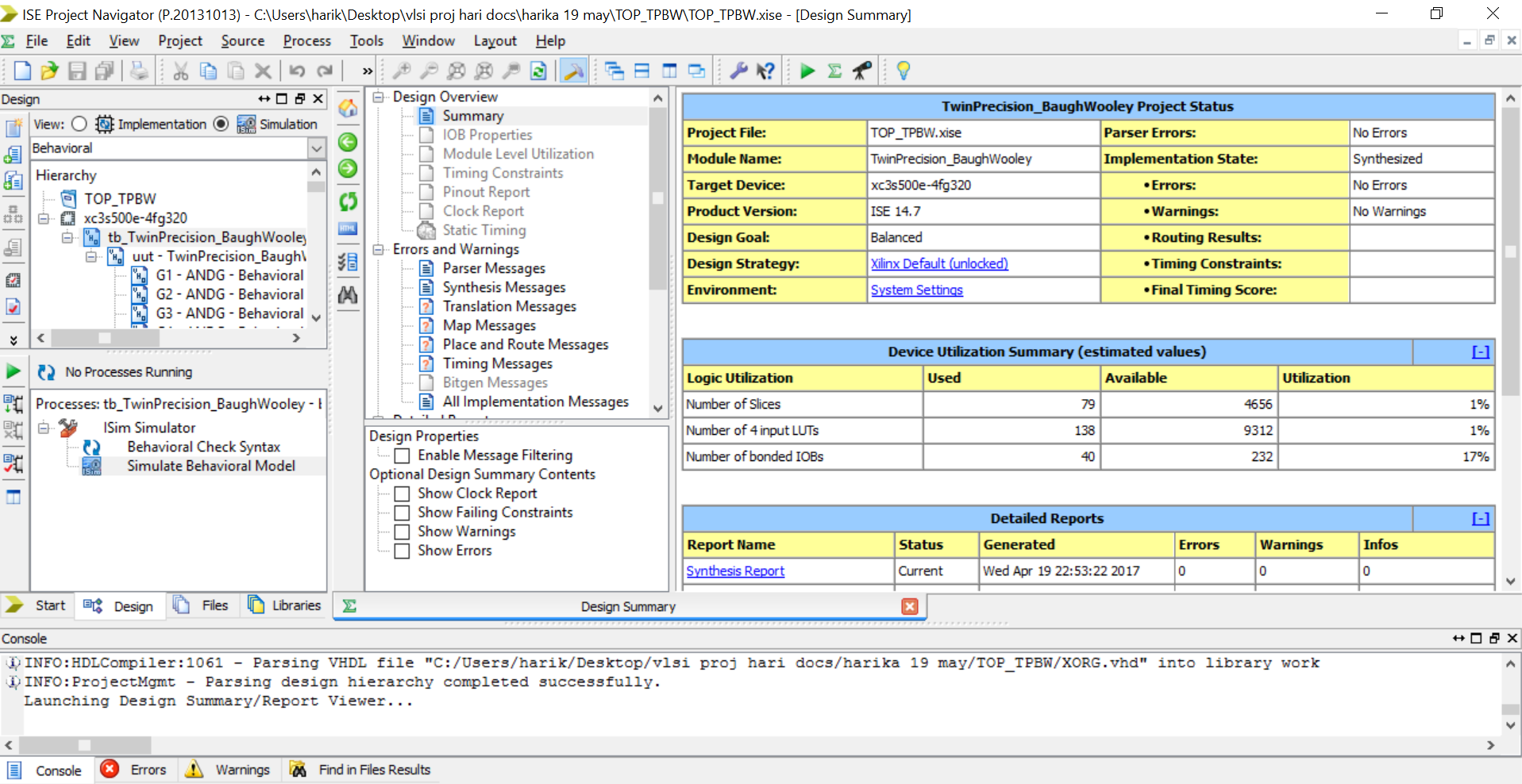
4.TECH:



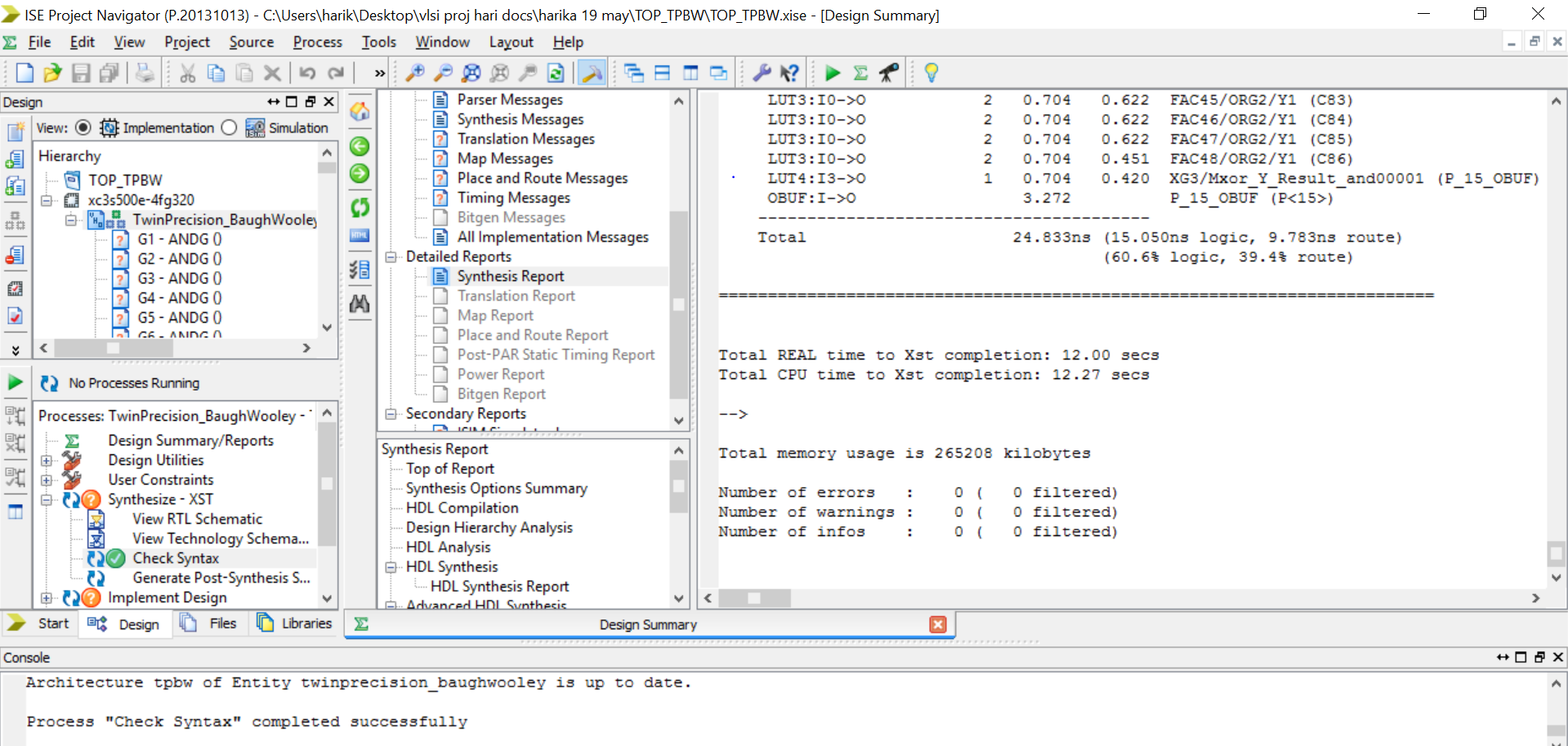
5.SUB TECH:



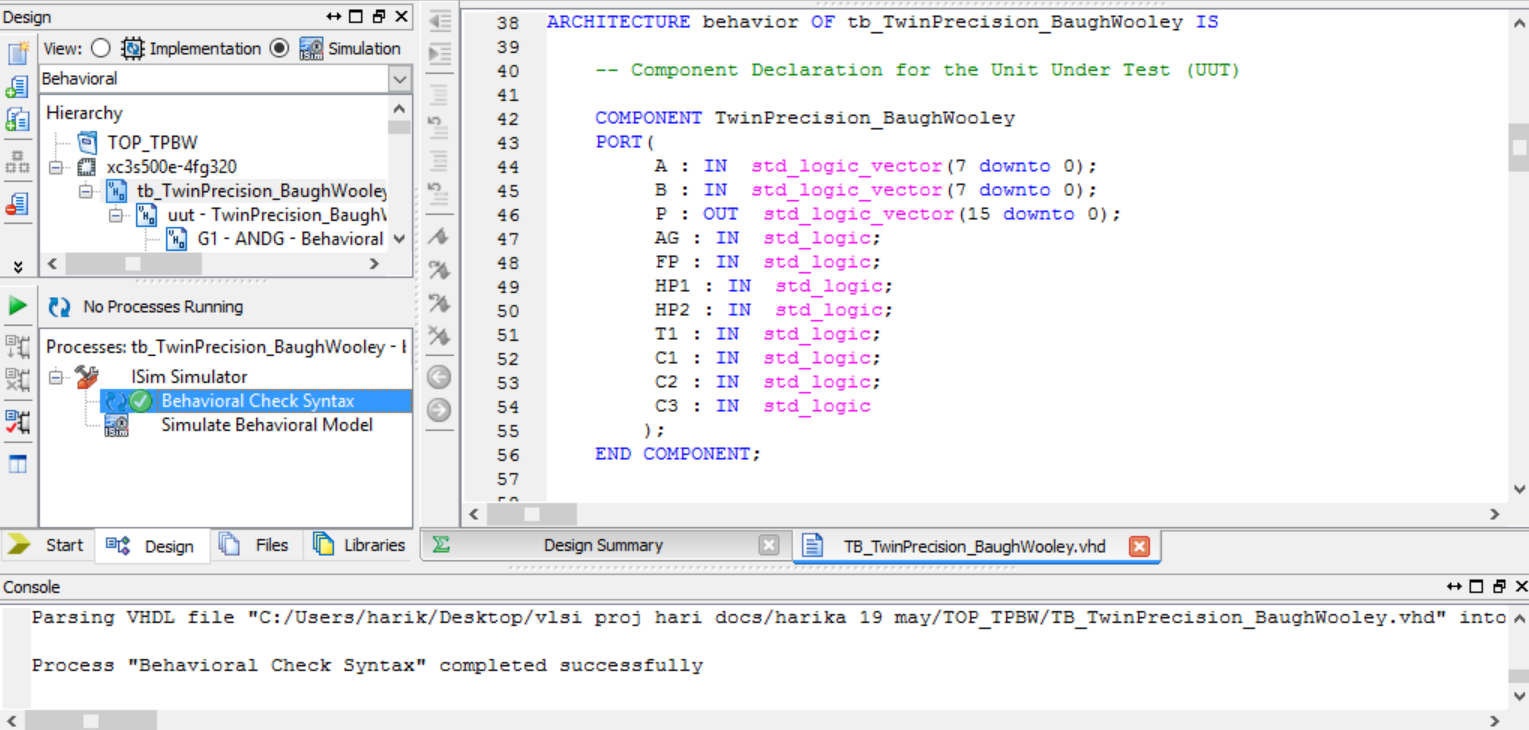
6.DESIGN SUMMARY:



7.SYNTHESIS REPORT:



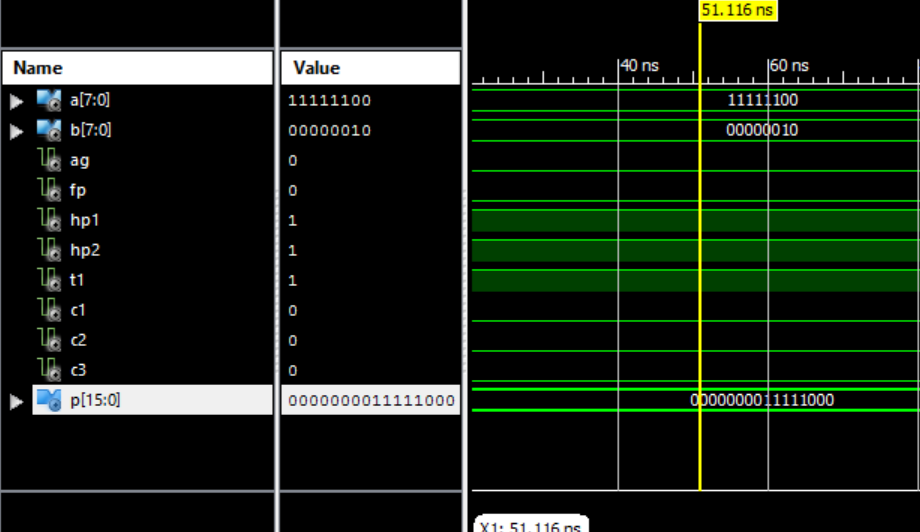
***SIMULATION:*** Simulation using the test bench:



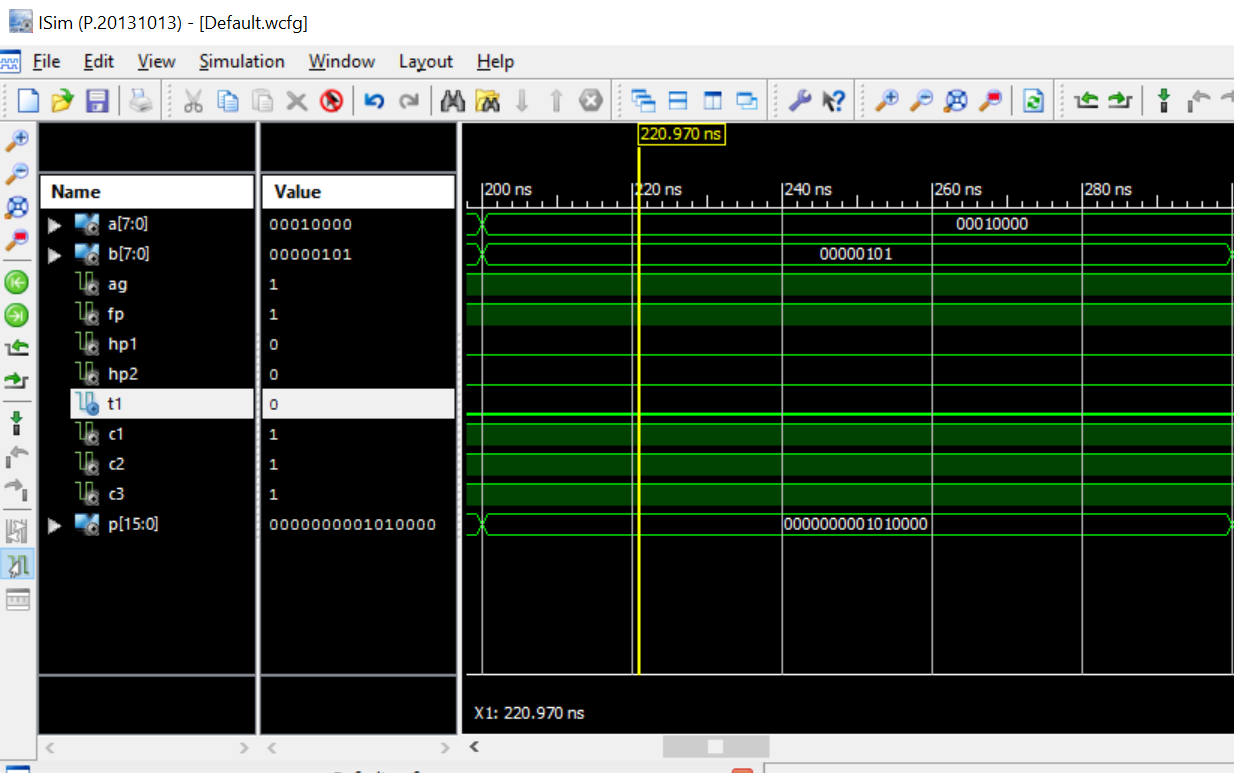
SIMULATED OUTPUT:

The outputs can be checked by changing the radix, since the test bench includes both signed and unsigned numbers, radix must be changed accordingly. The last 8digits of the outputs represents the final product.

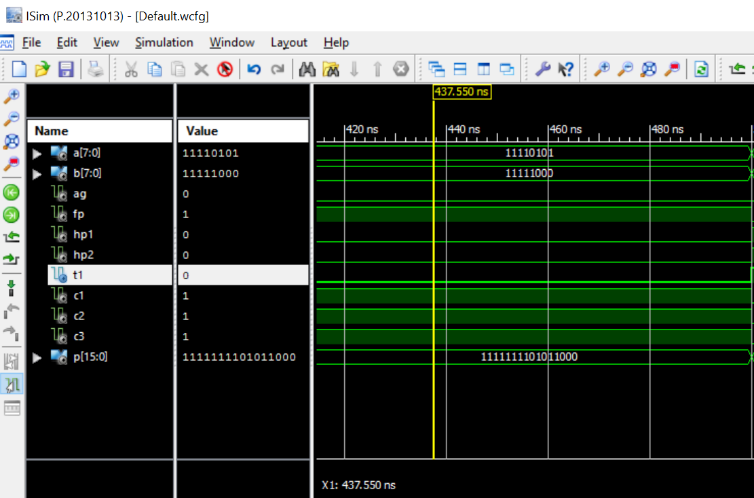
* Output 1: BW-twin precision case. [-4\*2=-8(11111000)]



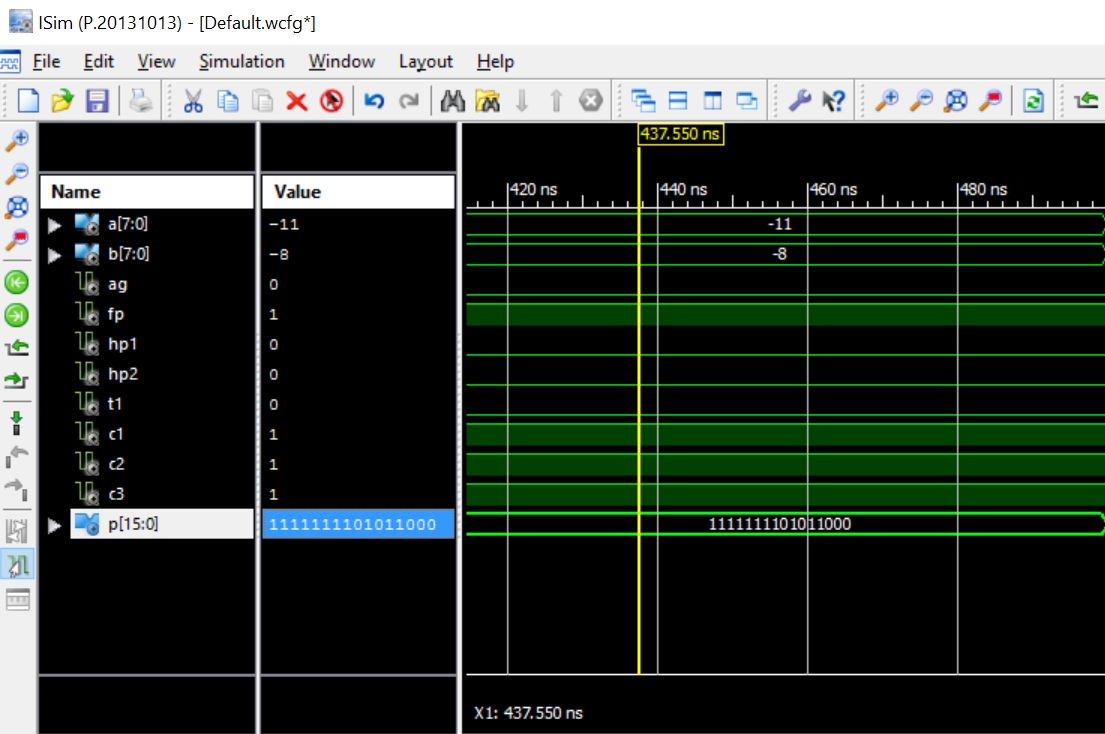
* Output2: unsigned number multiplication (last 8 bits of p denotes the final product) [ 16\*5= +80(01010000)]



* Output 3: Full precision mode of multiplier for signed numbers -BW

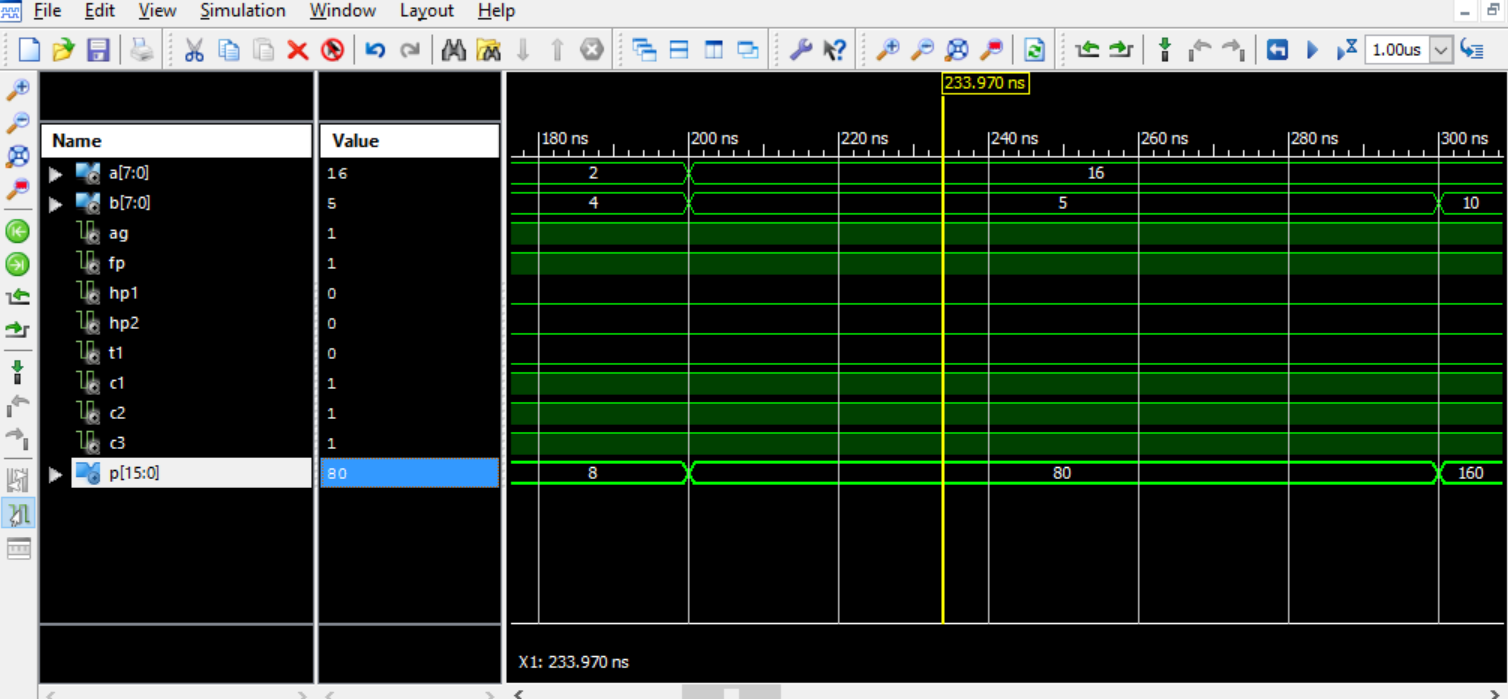


Expressed the inputs in decimal form for clear understanding.

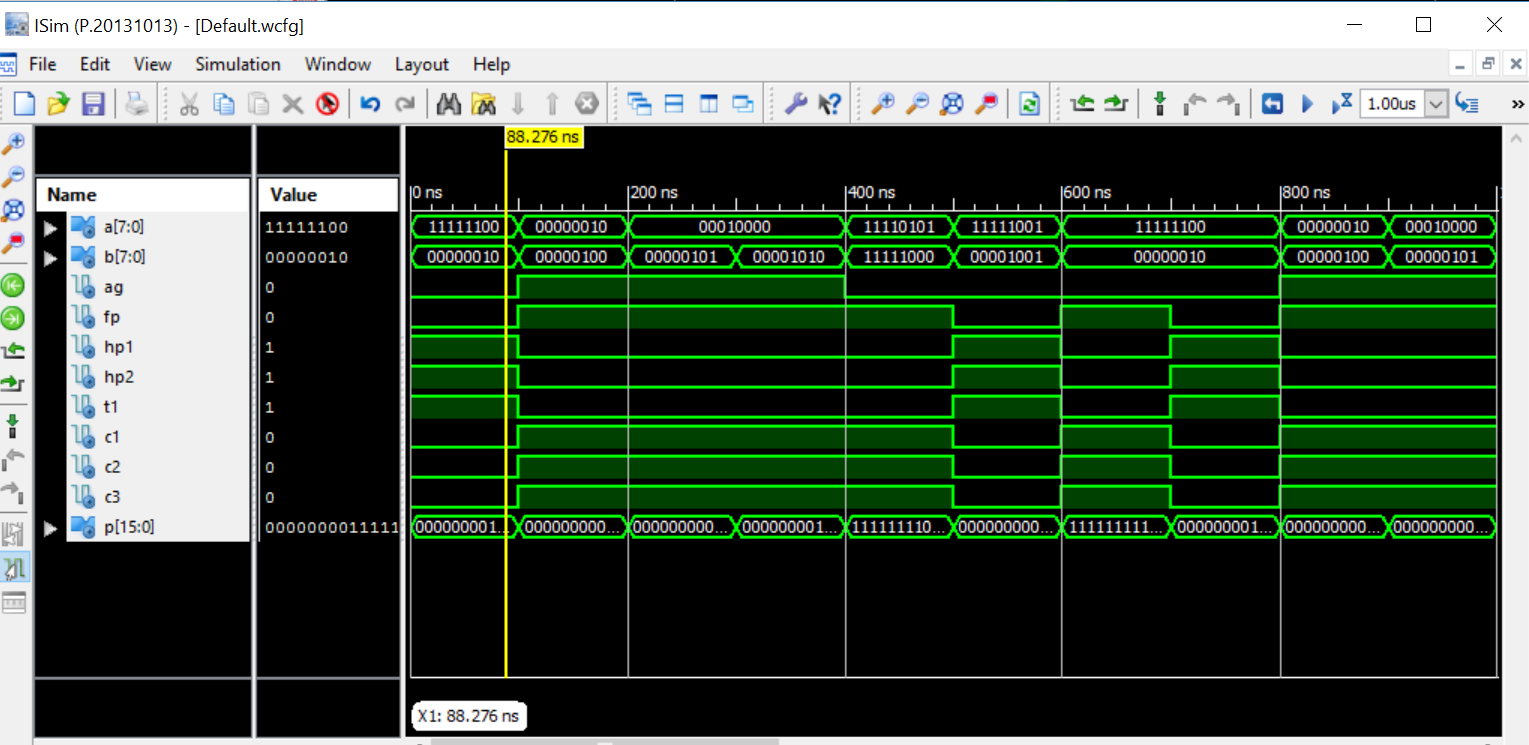


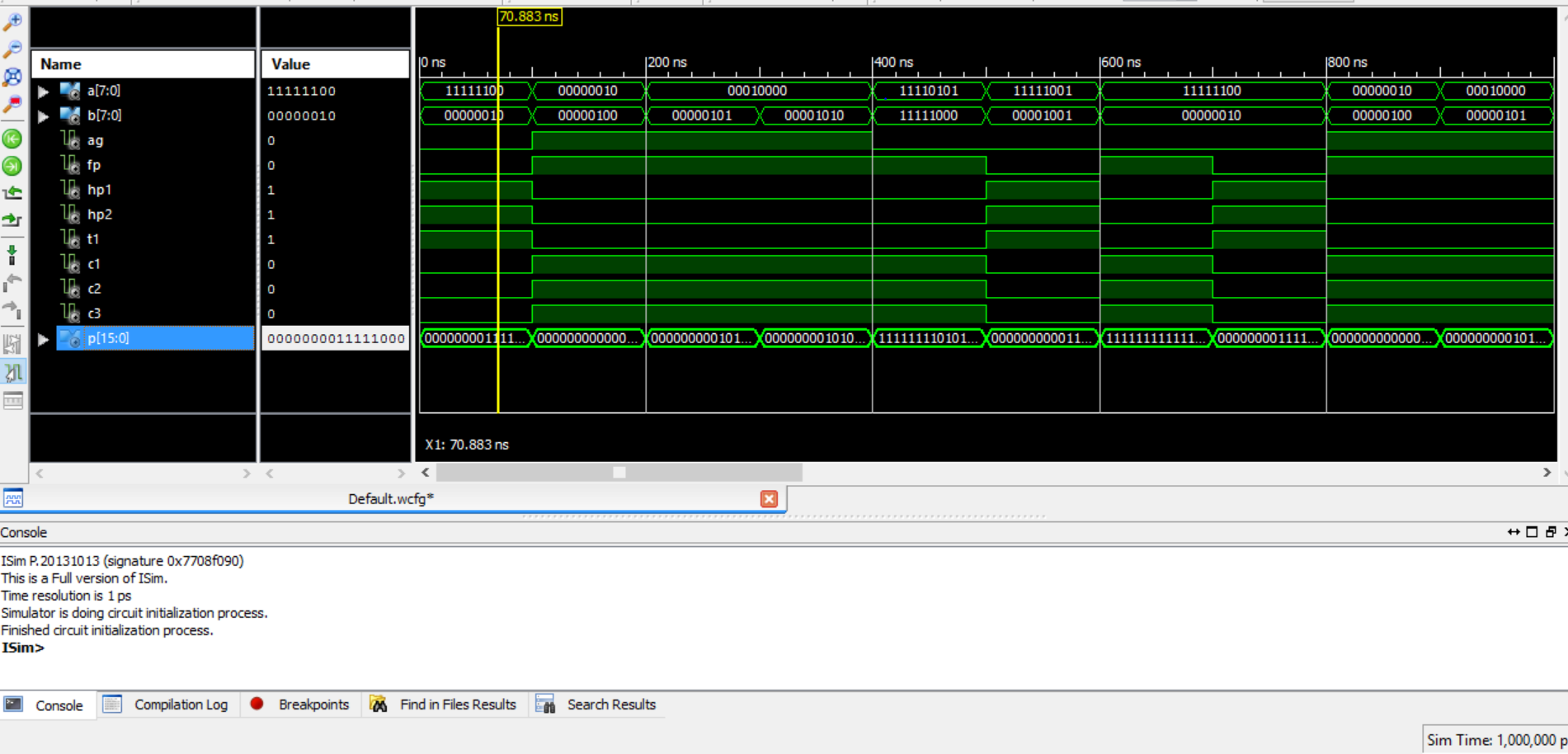
The last 8 bits-> -11\*-8 = +88 -> 01011000 represent 88 in decimal.

Changing the radix for clear understanding of the outputs:

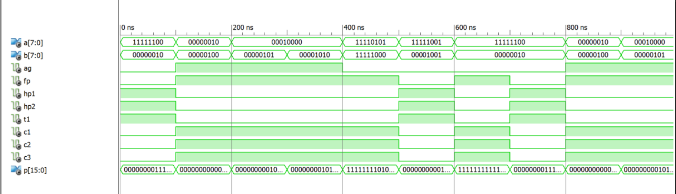


Full View of the test bench results:





Print view from Xilinx simulation.



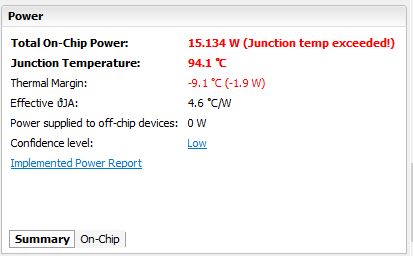
1. ADVANTAGES OF TWIN PRECISION MULTIPLIER

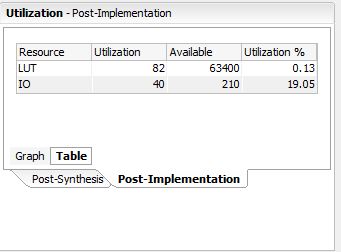
The twin-precision multiplier offers a good tradeoff between precision flexibility, area, delay and power dissipation by using the same multiplier for doing N, N/2 or two N/2-b multiplications. In comparison to a conventional 16-bit multiplier, a 16-bit twin-precision multiplier has 8% higher transistor count and 9% longer delay. The relative transistor count overhead decreases for larger multipliers, since the number of AND gates needed to set the partial products to zero does not grow as fast as the number of adders in the tree.

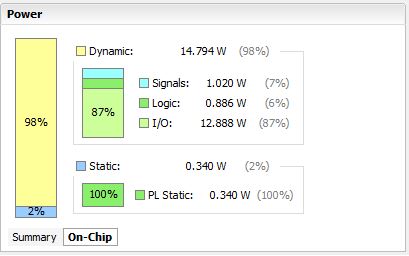
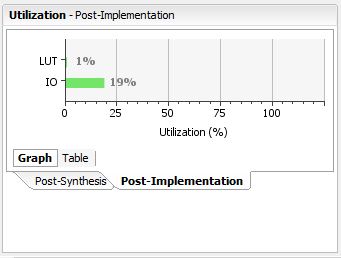
The main advantages that can be observed through this project are:

1. Power reduction- by using precision technique.
2. Double throughput -by performing two 4-bit multiplications in single operation.
3. Low Area –by implementing both multipliers in single design.

These observations are made based on detailed analysis using the Xilinx Vivado tool. The following are the Power and area requirements for this multiplier.







VII.CONCLUSION

I could successfully implement the B-W multiplier applying the twin precision technique to it for 8 bit multiplications.

The number of LUT’s represents the area required for implementation. We can observe from the synthesis report that the area overhead is very low for the B-W twin precision multiplier as it requires few LUT’s compared to general multipliers.

Also, the twin-precision technique, makes a multiplier able to adapt to different requirements. By adapting to actual multiplication bit-width using the twin-precision technique, it is possible to save power, increase speed and double computational throughput.

REFERENCES

[1] This project is reproduced based on an IEEE paper “Multiplication Acceleration Through Twin Precision” published in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 17, No. 9, September 2009.

[2] Nanoelectronic Mixed-Signal System Design by Saraju P. Mohanty, McGraw-Hill, 2015, 1 st Edition, ISBN-10: 0071825711, ISBN-13: 978-0071825719.

[3] M. Själander, H. Eriksson, and P. Larsson-Edefors, “An efficient twinprecision multiplier,” in Proc. 22nd IEEE Int. Conf. Comput. Des., Oct. 2004, pp. 30–33.

[4]. Andrew D. Booth, “A Signed Binary Multiplication Technique,” Quarterly Journal of Mechanics and Applied Mathematics, vol. 4, pp. 236-240, 1951.

[5]. Charles R. Baugh and Bruce. A. Wooley, “A Two’s Complement Parallel Array Multiplication Algorithm,” IEEE Transactions on Computers, vol. C-22, pp. 1045-1047, 1973.

[6]. PramodiniMohanty, RashmiRanjan, “An Efficient Baugh-Wooley Architecture for both Signed & Unsigned Multiplication” in International Journal of Computer Science & Engineering Technology (IJCSET), ISSN: 2229- 3345, Vol. 3 No. 4, April 2012.

CODE

---- Top level --main file-------

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity TwinPrecision\_BaughWooley is

port (A,B:in STD\_LOGIC\_VECTOR(7 downto 0);

P:out STD\_LOGIC\_VECTOR(15 downto 0);

AG,FP,HP1,HP2,T1,C1,C2,C3:in STD\_LOGIC);

end TwinPrecesion\_BaughWooley;

architecture TPBW of TwinPrecesion\_BaughWooley is

component ANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component XORG is

port (A,B :in STD\_LOGIC;

Y:out STD\_LOGIC );

end component;

component NANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component Full\_Add is

port(A,B,CIN:in STD\_LOGIC;

S,COUT:out STD\_LOGIC);

end component;

component Half\_Add is

port(A,B:in STD\_LOGIC;

S,C:out STD\_LOGIC);

end component;

component Nand\_Xor is

port(A,B,C:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component And\_And is

port(A,B,C:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component Nand\_And is

port(A,B,C:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

signal A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,

A21,A22,A23,A24,A25,A26,A27,A28,

A31,A32,A33,A34,A35,A36,A37,A38,

A41,A42,A43,A44,A45,A46,A47,A48,

A51,A52,A53,A54,A55,A56,A57,A58,

A61,A62,A63,A64,A65,A66,A67,A68,

A71,A72,A73,A74,A75,A76,A77,A78:STD\_LOGIC;

signal S11,S12,S13,S14,S15,S16,S1,S2,

S21,S22,S23,S24,S25,S26,

S31,S32,S33,S34,S35,S36,

S41,S42,S43,S44,S45,S46,

S51,S52,S53,S54,S55,S56,S57,

S61,S62,S63,S64,S65,S66,S67,

S71,S72,S73,S74,S75,S76,S77,S78:STD\_LOGIC;

signal C11,C12,C13,C14,C15,C16,C17,

C21,C22,C23,C24,C25,C26,C27,

C31,C32,C33,C34,C35,C36,C37,

C41,C42,C43,C44,C45,C46,C47,

C51,C52,C53,C54,C55,C56,C57,C58,

C61,C62,C63,C64,C65,C66,C67,C68,

C71,C72,C73,C74,C75,C76,C77,C78,

C81,C82,C83,C84,C85,C86,C87:STD\_LOGIC;

begin

------------ANDG -----------

G1:ANDG port map (A=>A(0),B=>B(0),Y=>P(0));

G2:ANDG port map (A=>A(1),B=>B(0),Y=>A1);

G3:ANDG port map (A=>A(2),B=>B(0),Y=>A3);

G4:ANDG port map (A=>A(0),B=>B(1),Y=>A2);

G5:ANDG port map (A=>A(1),B=>B(1),Y=>A4);

G6:ANDG port map (A=>A(2),B=>B(1),Y=>A6);

G7:ANDG port map (A=>A(0),B=>B(2),Y=>A21);

G8:ANDG port map (A=>A(1),B=>B(2),Y=>A22);

G9:ANDG port map (A=>A(2),B=>B(2),Y=>A23);

G10:ANDG port map (A=>A(3),B=>B(3),Y=>A34);

G11:ANDG port map(A=>A(4),B=>B(4),Y=>A45);

G12:ANDG port map(A=>A(5),B=>B(4),Y=>A46);

G13:ANDG port map(A=>A(6),B=>B(4),Y=>A47);

G14:ANDG port map(A=>A(4),B=>B(5),Y=>A55);

G15:ANDG port map(A=>A(5),B=>B(5),Y=>A56);

G16:ANDG port map(A=>A(6),B=>B(5),Y=>A57);

G17:ANDG port map(A=>A(4),B=>B(6),Y=>A65);

G18:ANDG port map(A=>A(5),B=>B(6),Y=>A66);

G19:ANDG port map(A=>A(6),B=>B(6),Y=>A67);

G20:ANDG port map(A=>A(7),B=>B(7),Y=>A78);

G21:ANDG port map(A=>AG,B=>C71,Y=>S2);---AG

-----------NANDG -----------

NG1:NANDG port map (A=>A(7),B=>B(4),Y=>A48);

NG2:NANDG port map (A=>A(7),B=>B(5),Y=>A58);

NG3:NANDG port map (A=>A(7),B=>B(6),Y=>A68);

NG4:NANDG port map (A=>A(4),B=>B(7),Y=>A75);

NG5:NANDG port map (A=>A(5),B=>B(7),Y=>A76);

NG6:NANDG port map (A=>A(6),B=>B(7),Y=>A77);

----------XORG ------------

XG1:XORG port map (A=>S1,B=>T1,Y=>P(7)); -----t1

XG2:XORG port map (A=>C78,B=>C87,Y=>S78);

XG3:XORG port map (A=>S78,B=>'1',Y=>P(15));

---------Nand\_Xor --------------

NXG1:Nand\_Xor port map (A=>A(3),B=>B(0),C=>C1,Y=>A5); ---c1

NXG2:Nand\_Xor port map (A=>A(3),B=>B(1),C=>C1,Y=>A8);

NXG3:Nand\_Xor port map (A=>A(3),B=>B(2),C=>C1,Y=>A24);

NXG4:Nand\_Xor port map (A=>A(0),B=>B(3),C=>C1,Y=>A31);

NXG5:Nand\_Xor port map (A=>A(1),B=>B(3),C=>C1,Y=>A32);

NXG6:Nand\_Xor port map (A=>A(2),B=>B(3),C=>C1,Y=>A33);

--------And\_Aand -------------------

AAG1:And\_And port map (A=>A(4),B=>B(0),C=>C3,Y=>A7); ----c3

AAG2:And\_And port map (A=>A(5),B=>B(0),C=>C3,Y=>A9);

AAG3:And\_And port map (A=>A(6),B=>B(0),C=>C3,Y=>A11);

AAG4:And\_And port map (A=>A(4),B=>B(1),C=>C3,Y=>A10);

AAG5:And\_And port map (A=>A(5),B=>B(1),C=>C3,Y=>A12);

AAG6:And\_And port map (A=>A(6),B=>B(1),C=>C3,Y=>A14);

AAG7:And\_And port map (A=>A(4),B=>B(2),C=>C3,Y=>A25);

AAG8:And\_And port map (A=>A(5),B=>B(2),C=>C3,Y=>A26);

AAG9:And\_And port map (A=>A(6),B=>B(2),C=>C3,Y=>A27);

AAG10:And\_And port map (A=>A(4),B=>B(3),C=>C3,Y=>A35);

AAG11:And\_And port map (A=>A(5),B=>B(3),C=>C3,Y=>A36);

AAG12:And\_And port map (A=>A(6),B=>B(3),C=>C3,Y=>A37);

AAG13:And\_And port map (A=>A(0),B=>B(4),C=>C3,Y=>A41);

AAG14:And\_And port map (A=>A(1),B=>B(4),C=>C3,Y=>A42);

AAG15:And\_And port map (A=>A(2),B=>B(4),C=>C3,Y=>A43);

AAG16:And\_And port map (A=>A(3),B=>B(4),C=>C3,Y=>A44);

AAG17:And\_And port map (A=>A(0),B=>B(5),C=>C3,Y=>A51);

AAG18:And\_And port map (A=>A(1),B=>B(5),C=>C3,Y=>A52);

AAG19:And\_And port map (A=>A(2),B=>B(5),C=>C3,Y=>A53);

AAG20:And\_And port map (A=>A(3),B=>B(5),C=>C3,Y=>A54);

AAG21:And\_And port map (A=>A(0),B=>B(6),C=>C3,Y=>A61);

AAG22:And\_And port map (A=>A(1),B=>B(6),C=>C3,Y=>A62);

AAG23:And\_And port map (A=>A(2),B=>B(6),C=>C3,Y=>A63);

AAG24:And\_And port map (A=>A(3),B=>B(6),C=>C3,Y=>A64);

----------------Nand\_And -------------------

NAG1:Nand\_And port map (A=>A(7),B=>B(0),C=>C2,Y=>A13);---c2

NAG2:Nand\_And port map (A=>A(7),B=>B(1),C=>C2,Y=>A15);

NAG3:Nand\_And port map (A=>A(7),B=>B(2),C=>C2,Y=>A28);

NAG4:Nand\_And port map (A=>A(7),B=>B(3),C=>C2,Y=>A38);

NAG5:Nand\_And port map (A=>A(0),B=>B(7),C=>C2,Y=>A71);

NAG6:Nand\_And port map (A=>A(1),B=>B(7),C=>C2,Y=>A72);

NAG7:Nand\_And port map (A=>A(2),B=>B(7),C=>C2,Y=>A73);

NAG8:Nand\_And port map (A=>A(3),B=>B(7),C=>C2,Y=>A74);

---------------Half\_Add ---------------

HAC1:Half\_Add port Map (A=>A1,B=>A2,S=>P(1),C=>C11);

HAC2:Half\_Add port Map (A=>A3,B=>A4,S=>S11,C=>C12);

HAC3:Half\_Add port Map (A=>A5,B=>A6,S=>S12,C=>C13);

HAC4:Half\_Add port Map (A=>A9,B=>A10,S=>S14,C=>C15);

HAC5:Half\_Add port Map (A=>A11,B=>A12,S=>S15,C=>C16);

HAC6:Half\_Add port Map (A=>A13,B=>A14,S=>S16,C=>C17);

HAC7:Half\_Add port Map (A=>A58,B=>HP2,S=>S57,C=>C58);----HP2

HAC8:Half\_Add port Map (A=>A68,B=>C58,S=>S67,C=>C68);

HAC9:Half\_Add port Map (A=>A78,B=>C68,S=>S77,C=>C78);

--------------Full\_Add ----------------

FAC1:Full\_Add port map (A=>A21,B=>C11,CIN=>S11,S=>P(2),COUT=>C21);

FAC2:Full\_Add port map (A=>A22,B=>C12,CIN=>S12,S=>S21,COUT=>C22);

FAC3:Full\_Add port map (A=>A23,B=>C13,CIN=>S13,S=>S22,COUT=>C23);

FAC4:Full\_Add port map (A=>A24,B=>C14,CIN=>S14,S=>S23,COUT=>C24);

FAC5:Full\_Add port map (A=>A25,B=>C15,CIN=>S15,S=>S24,COUT=>C25);

FAC6:Full\_Add port map (A=>A26,B=>C16,CIN=>S16,S=>S25,COUT=>C26);

FAC7:Full\_Add port map (A=>A27,B=>C17,CIN=>A15,S=>S26,COUT=>C27);

FAC8:Full\_Add port map (A=>A31,B=>C21,CIN=>S21,S=>P(3),COUT=>C31);

FAC9:Full\_Add port map (A=>A32,B=>C22,CIN=>S22,S=>S31,COUT=>C32);

FAC10:Full\_Add port map (A=>A33,B=>C23,CIN=>S23,S=>S32,COUT=>C33);

FAC11:Full\_Add port map (A=>A34,B=>C24,CIN=>S24,S=>S33,COUT=>C34);

FAC12:Full\_Add port map (A=>A35,B=>C25,CIN=>S25,S=>S34,COUT=>C35);

FAC13:Full\_Add port map (A=>A36,B=>C26,CIN=>S26,S=>S35,COUT=>C36);

FAC14:Full\_Add port map (A=>A37,B=>C27,CIN=>A28,S=>S36,COUT=>C37);

FAC15:Full\_Add port map (A=>A41,B=>C31,CIN=>S31,S=>P(4),COUT=>C41);

FAC16:Full\_Add port map (A=>A42,B=>C32,CIN=>S32,S=>S41,COUT=>C42);

FAC17:Full\_Add port map (A=>A43,B=>C33,CIN=>S33,S=>S42,COUT=>C43);

FAC18:Full\_Add port map (A=>A44,B=>C34,CIN=>S34,S=>S43,COUT=>C44);

FAC19:Full\_Add port map (A=>A45,B=>C35,CIN=>S35,S=>S44,COUT=>C45);

FAC20:Full\_Add port map (A=>A46,B=>C36,CIN=>S36,S=>S45,COUT=>C46);

FAC21:Full\_Add port map (A=>A47,B=>C37,CIN=>A38,S=>S46,COUT=>C47);

FAC22:Full\_Add port map (A=>A51,B=>C41,CIN=>S41,S=>P(5),COUT=>C51);

FAC23:Full\_Add port map (A=>A52,B=>C42,CIN=>S42,S=>S51,COUT=>C52);

FAC24:Full\_Add port map (A=>A53,B=>C43,CIN=>S43,S=>S52,COUT=>C53);

FAC25:Full\_Add port map (A=>A54,B=>C44,CIN=>S44,S=>S53,COUT=>C54);

FAC26:Full\_Add port map (A=>A55,B=>C45,CIN=>S45,S=>S54,COUT=>C55);

FAC27:Full\_Add port map (A=>A56,B=>C46,CIN=>S46,S=>S55,COUT=>C56);

FAC28:Full\_Add port map (A=>A57,B=>C47,CIN=>A48,S=>S56,COUT=>C57);

FAC29:Full\_Add port map (A=>A61,B=>C51,CIN=>S51,S=>P(6),COUT=>C61);

FAC30:Full\_Add port map (A=>A62,B=>C52,CIN=>S52,S=>S61,COUT=>C62);

FAC31:Full\_Add port map (A=>A63,B=>C53,CIN=>S53,S=>S62,COUT=>C63);

FAC32:Full\_Add port map (A=>A64,B=>C54,CIN=>S54,S=>S63,COUT=>C64);

FAC33:Full\_Add port map (A=>A65,B=>C55,CIN=>S55,S=>S64,COUT=>C65);

FAC34:Full\_Add port map (A=>A66,B=>C56,CIN=>S56,S=>S65,COUT=>C66);

FAC35:Full\_Add port map (A=>A67,B=>C57,CIN=>S57,S=>S66,COUT=>C67);

FAC36:Full\_Add port map (A=>A71,B=>C61,CIN=>S61,S=>S1,COUT=>C71);

FAC37:Full\_Add port map (A=>A72,B=>C62,CIN=>S62,S=>S71,COUT=>C72);

FAC38:Full\_Add port map (A=>A73,B=>C63,CIN=>S63,S=>S72,COUT=>C73);

FAC39:Full\_Add port map (A=>A74,B=>C64,CIN=>S64,S=>S73,COUT=>C74);

FAC40:Full\_Add port map (A=>A75,B=>C65,CIN=>S65,S=>S74,COUT=>C75);

FAC41:Full\_Add port map (A=>A76,B=>C66,CIN=>S66,S=>S75,COUT=>C76);

FAC42:Full\_Add port map (A=>A77,B=>C67,CIN=>S67,S=>S76,COUT=>C77);

FAC43:Full\_Add port map (A=>FP,B=>S2,CIN=>S71,S=>P(8),COUT=>C81);-----FP

FAC44:Full\_Add port map (A=>S72,B=>C72,CIN=>C81,S=>P(9),COUT=>C82);

FAC45:Full\_Add port map (A=>S73,B=>C73,CIN=>C82,S=>P(10),COUT=>C83);

FAC46:Full\_Add port map (A=>S74,B=>C74,CIN=>C83,S=>P(11),COUT=>C84);

FAC47:Full\_Add port map (A=>S75,B=>C75,CIN=>C84,S=>P(12),COUT=>C85);

FAC48:Full\_Add port map (A=>S76,B=>C76,CIN=>C85,S=>P(13),COUT=>C86);

FAC49:Full\_Add port map (A=>S77,B=>C77,CIN=>C86,S=>P(14),COUT=>C87);

FAC50:Full\_Add port map (A=>A7,B=>HP1,CIN=>A8,S=>S13,COUT=>C14);---HP1

END TPBW;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity And\_And is

Port ( A,B,C : in STD\_LOGIC;

Y : out STD\_LOGIC);

end And\_And;

architecture Behavioral of And\_And is

component ANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

signal S:STD\_LOGIC;

begin

G1:ANDG port map(A=>A,B=>B,Y=>S);

G2:ANDG port map(A=>S,B=>C,Y=>Y);

end Behavioral;

--ANDG

entity ANDG is

Port ( A,B : in STD\_LOGIC;

Y : out STD\_LOGIC);

end ANDG;

architecture Behavioral of ANDG is

begin

Y<= A AND B;

end Behavioral;

entity Full\_Add is

Port ( A,B,CIN : in STD\_LOGIC;

S,COUT : out STD\_LOGIC);

end Full\_Add;

architecture Behavioral of Full\_Add is

component Half\_Add is

port(A,B:in STD\_LOGIC;

S,C:out STD\_LOGIC);

end component;

component ORG is

port (A,B :in STD\_LOGIC;

Y:out STD\_LOGIC );

end component;

signal S0,S1,S2:STD\_LOGIC;

begin

HA1 :Half\_Add port map(A=>A,B=>B,S=>S0,C=>S1);

HA2 :Half\_Add port map(A=>S0,B=>CIN,S=>S,C=>S2);

ORG2:ORG port map(A=>S2,B=>S1,Y=>COUT);

end Behavioral;

entity Half\_Add is

Port ( A,B : in STD\_LOGIC;

S,C : out STD\_LOGIC);

end Half\_Add;

architecture Behavioral of Half\_Add is

component ANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component XORG is

port (A,B :in STD\_LOGIC;

Y:out STD\_LOGIC );

end component;

begin

G1:ANDG port map(A=>A,B=>B,Y=>C);

G2:XORG port map(A=>A,B=>B,Y=>S);

end Behavioral;

entity Nand\_And is

Port ( A,B,C : in STD\_LOGIC;

Y : out STD\_LOGIC);

end Nand\_And;

architecture Behavioral of Nand\_And is

component NANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component ANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

signal S:STD\_LOGIC;

begin

G1: NANDG port map(A=>A,B=>B,Y=>S);

G2: ANDG port map(A=>S,B=>C,Y=>Y);

end Behavioral;

entity Nand\_Xor is

Port ( A,B,C : in STD\_LOGIC;

Y : out STD\_LOGIC);

end Nand\_Xor;

architecture Behavioral of Nand\_Xor is

component NANDG is

port(A,B:in STD\_LOGIC;

Y:out STD\_LOGIC);

end component;

component XORG is

port (A,B :in STD\_LOGIC;

Y:out STD\_LOGIC );

end component;

signal S:STD\_LOGIC;

begin

G1:NANDG port map(A=>A,B=>B,Y=>S);

G2:XORG port map(A=>S,B=>C,Y=>Y);

end Behavioral;

entity NANDG is

Port ( A,B : in STD\_LOGIC;

Y : out STD\_LOGIC);

end NANDG;

architecture Behavioral of NANDG is

begin

Y<= A NAND B;

end Behavioral;

entity ORG is

Port ( A,B : in STD\_LOGIC;

Y : out STD\_LOGIC);

end ORG;

architecture Behavioral of ORG is

begin

Y<=A OR B;

end Behavioral;

entity XORG is

Port ( A,B : in STD\_LOGIC;

Y : out STD\_LOGIC);

end XORG;

architecture Behavioral of XORG is

begin

Y<=A XOR B;

end Behavioral;

***Test bench***

--------------------------------------------------------------------------------

-- simulation model.

--------------------------------------------------------------------------------

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

ENTITY tb\_TwinPrecision\_BaughWooley IS

END tb\_TwinPrecision\_BaughWooley;

ARCHITECTURE behavior OF tb\_TwinPrecision\_BaughWooley IS

COMPONENT TwinPrecision\_BaughWooley

PORT(

A : IN std\_logic\_vector(7 downto 0);

B : IN std\_logic\_vector(7 downto 0);

P : OUT std\_logic\_vector(15 downto 0);

AG : IN std\_logic;

FP : IN std\_logic;

HP1 : IN std\_logic;

HP2 : IN std\_logic;

T1 : IN std\_logic;

C1 : IN std\_logic;

C2 : IN std\_logic;

C3 : IN std\_logic

);

END COMPONENT;

--Inputs

signal A : std\_logic\_vector(7 downto 0) := (others => '0');

signal B : std\_logic\_vector(7 downto 0) := (others => '0');

signal AG : std\_logic := '0';

signal FP : std\_logic := '0';

signal HP1 : std\_logic := '0';

signal HP2 : std\_logic := '0';

signal T1 : std\_logic := '0';

signal C1 : std\_logic := '0';

signal C2 : std\_logic := '0';

signal C3 : std\_logic := '0';

--Outputs

signal P : std\_logic\_vector(15 downto 0);

BEGIN

-- Instantiate the Unit Under Test (UUT)

uut: TwinPrecision\_BaughWooley PORT MAP (

A => A,

B => B,

P => P,

AG => AG,

FP => FP,

HP1 => HP1,

HP2 => HP2,

T1 => T1,

C1 => C1,

C2 => C2,

C3 => C3

);

-- Stimulus process

stim\_proc: process

begin

--input 1

AG <= '0';

FP <= '0';

HP1 <= '1';

HP2 <= '1';

T1 <= '1';

C1 <= '0';

C2 <= '0';

C3 <= '0';

A <= "11111100";--- -4

B<= "00000010";--- +2

wait for 100 ns;---- -4\*2=-8(11111000) twin precision signed numbers BW

--input 2

AG <= '1';

FP <= '1';

HP1 <= '0';

HP2 <= '0';

T1 <= '0';

C1 <= '1';

C2 <= '1';

C3 <= '1';

A <= "00000010";--- +2

B <= "00000100";--- +4

wait for 100 ns;---- 4\*2=8(00001000) full precision unsigned

--input 3

AG <= '1';

FP <= '1';

HP1 <= '0';

HP2 <= '0';

T1 <= '0';

C1 <= '1';

C2 <= '1';

C3 <= '1';

A <= "00010000";--- +16

B <= "00000101";--- +5

wait for 100 ns;---- +80(01010000) unsigned

--input 4

AG <= '1';

FP <= '1';

HP1 <= '0';

HP2 <= '0';

T1 <= '0';

C1 <= '1';

C2 <= '1';

C3 <= '1';

A <= "00010000";--- 16

B <= "00001010";--- 10

wait for 100 ns;-- 160(010100000 requires 9 bits) unsigned

--input 5

AG <= '0';

FP <= '1';

HP1 <= '0';

HP2 <= '0';

T1 <= '0';

C1 <= '1';

C2 <= '1';

C3 <= '1';

A <= "11110101";--- -11

B <= "11111000";--- -8

wait for 100 ns;-- 88(01011000) Full precision Signed BW

--input 6

AG <= '0';

FP <= '0';

HP1 <= '1';

HP2 <= '1';

T1 <= '1';

C1 <= '0';

C2 <= '0';

C3 <= '0';

A <= "11111001";--- -7 new case

B<= "00001001";--- +9

wait for 100 ns;---- -7\*9=-63(11000001) --Twin precision BW signed numbers

--input 7

AG <= '0';

FP <= '1';

HP1 <= '0';

HP2 <= '0';

T1 <= '0';

C1 <= '1';

C2 <= '1';

C3 <= '1';

A <= "11111100";--- -4

B<= "00000010";--- +2

wait for 100 ns;---- -4\*2=-8(11111000) full precision signed numbers BW

end process;

END;

**<END OF REPORT>**