Ministry of Higher Education and
Scientific Research
University of Diyala
College of Engineering
Electronic Engineering Department

Design & Simulation of low Hardware Computational Cost Multiplier

A PROJECT

Submitted to the Electronic Engineering Department College of Engineering Diyala University in partial fulfillment of the Requirements for the Degree of Bachelor of Science in Electronics Engineering

By

Ashti Yadallah

Nawras Hamza

Supervised By

Dr. Qahtan Khalaf Omran

May 2016 1437

وزارة التعليم العالي والبحث العلمي

جامعة ديالى

كلية الهندسة

قسم الهندسة الالكترونية

تصميم ومحاكاة كلفة الأجهزة المنخفظة الحسابية

مشروع مقدم إلى قسم الهندسة الالكترونية

في جامعة ديالى كلية الهندسة كجزء من متطلبات الحصول على درجة البكالوريوس

في الهندسة الالكترونية

من قبل

آشتي يدالله & نورس حمزة

بإشراف

الدكتور قحطان خلف عمران

1437 2016

CHAPTER I

INTRODUCTION

1.1 Binary Multiplier Background

Multipliers are key components of many high performance systems such as FIR filters, microprocessors, digital signal processors, etc [1]. A system 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 are improving speed result mostly in large areas. As a result, whole spectrums of multiplier with different area-speed constraints are designed with fully parallel processing or with fully digit serial processing where single digits consisting performance in both of several bits are operated on. These multipliers have low speed and moderate area. However, the growing market for fast floating-point co-processors, digital signal processing chips, and graphics processors has created a demand for high speed, area–efficient multipliers. Current architectures range from, small, low performance shift and add multipliers to large, high performance array and tree multipliers. Conventional, linear array multipliers achieve high performance in regular structures, but require large amounts of silicon. Tree structures achieve even higher performance than linear array, but the Tree interconnection is more complex and less regular, making them even larger than linear arrays. Ideally, one would want the speed benefits of a tree structure, the regularity of an array multiplier, and the small size of a shift and add multiplier.

The multiplier plays an important role in today's digital signal processing and various other applications. With advances in technology, many researchers have tried and are trying to design multipliers which offer either of the following design targets – high speed, low power consumption, hence the less area or even a combination of them in one multiplier thus making them suitable for various high speed, low power and compact VLSI implementation.

1.2 Application of Multiplier

Multiplication is one of the basic functions used in digital signal processing (DSP). It can be found in many DSP applications such as:

1. Vector product , convolution ,filtering and frequency analysis
2. Matrix multiplication
3. Weighted sums required in many DSP such as (neural network ,filtering ,etc)
4. Speed processing , loud speaker equalization ,echo cancelation , adaptive noise cancelation ,and various communication application including software defend radio (SDR) and so on
5. Microprocessors IIR& FIR digital filters

For example an 8 bits are needed in image compression .Typically, we see 16 bit multipliers used for digital signal processing and 64 bit multipliers used in microprocessors many current DSP applications are targeted at portable, battery-operated systems, so that power dissipation becomes one of the primary design constraints. Since multipliers are rather complex circuits and must typically operate at a high system clock rate,

reducing the delay of a multiplier is an essential part of satisfying the overall design.

1.3 Multiplication Techniques

Multiplication is a mathematical operation that at its simplest is an abbreviated process of adding an integer to itself a specified number of times. A number (multiplicand) is added to itself a number of times as specified by another number (multiplier) to form result (product) as shown in Figure 1.1. The circuit which is performing the multiplication process named a multiplier.

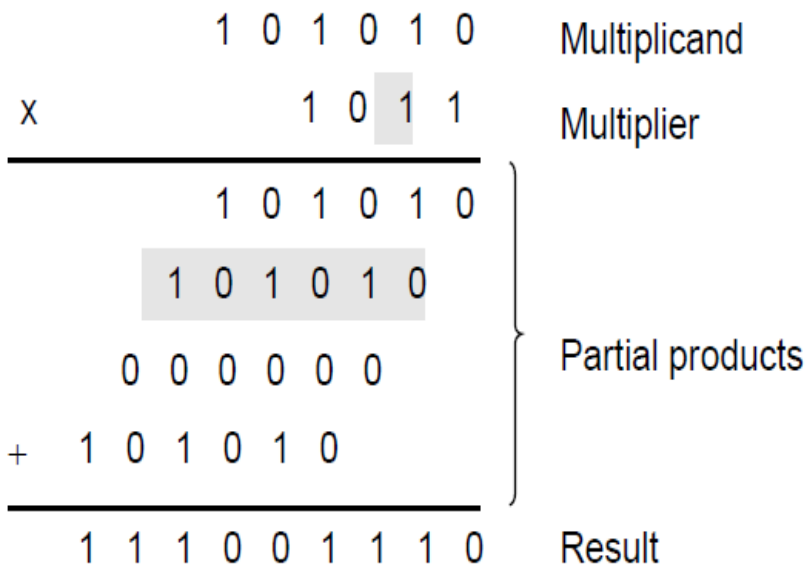


Figure 1.1 binary multiplication

The common multiplication method is “add and shift” algorithm. In parallel multipliers number of partial products to be added is the main parameter

that determines the performance of the multiplier. To reduce the silicon area (number of utilized logic gates), a serial processing has been used which is resulting in low speed performance. On the other hand “serial-parallel” multipliers compromise speed to achieve better performance for area, speed and power consumption. The selection of a parallel or serial multiplier actually depends on the nature of the application.

In this project we investigate fully parallel and fully serial multiplication algorithms by design and simulation fully parallel and fully serial multiplier and then observe their performance in terms of power consumption, logic utilization and speed.

CHAPTER II

MULTIPLICATION PROCESS

2.1 Multiplier Basic Operation

For the multiplication of an n-bit multiplicand with an m-bit multiplier, m partial products are generated and product formed is n + m bits long [2].
To perform N -bit by N-bit multiplication the N-bit multiplicand A is multiplied by N-bit multiplier B to produce product. The unsigned binary numbers A and B can be expressed as:

$$A = \sum_{i=0}^{n} A_i 2^i \qquad 2.1$$

$$B = \sum_{j=0}^{m} B_j 2^j \qquad 2.2$$

The product of A and B is P and it can be written in the following form

$$P = \sum_{i=0}^{n} \sum_{i=0}^{n} A_i \, B_j 2^{i+j} \qquad 2.3$$

An n*n multiplier requires n (n-1) ADDERS and n^2 AND gates.
Multiplication schemed used in digital systems are quite similar to pencil-and-paper multiplication. An array of partial products is found first, and these are then added to generate the product. Figure 2.1 shows a simple numerical example of the multiplication. As shown in the diagram, the first partial product is formed by multiplying 1110 by 0; the second partial product is formed by multiplying 1110 by 1, and so on. The multiplication of two bits produces a 1 if both bits are 1; otherwise it produces a 0. The summation of the partial products is accomplished by using full adders. In general, the multiplication of an m-bit multiplicand X ($x_m \ldots x_1 x_0$) by an n-bit multiplier Y ($y_{n1} \ldots y_1 y_0$) results in an (m n)-bit product. Each of the

m n 1-bit products x_i y_j may be generated by a 2-input AND gate; these products are then added by an array of full address. Figure 2-2(a) shows a 4-bit by 3-bit multiplier circuit.

Let us multiply $x_3x_2x_1x_0$ $0110(6_{10})$ by $y_2y_1y_0$ $101(5_{10})$ using the multiplier circuit of Figure2-2(a). The outputs of the AND gates and the full adders in the multiplier circuit corresponding to the applied input values are recorded in Figure2-2 (b). Note that the outputs of the AND gates form the partial products and the outputs of the full adders form the partial sum during the multiplication process. The final output pattern is p_6 p_5 p_4 p_3 p_2 p_1 p_0 0011110 (i.e., 30_{10}), which is the expected result. One of the problems in performing multiplication using this scheme is that when many partial products are to be added, it becomes difficult to handle the carries generated during the summation of partial products.

Multiplicand =	1 1 1 0
Multiplier =	1 0 1 0
	0 0 0 0
Partial product bits	1 1 1 0
	0 0 0 0
	1 1 1 0
Product p =	1 0 0 0 1 1 0 0

Figure 2.1 Binary multiplication examples.

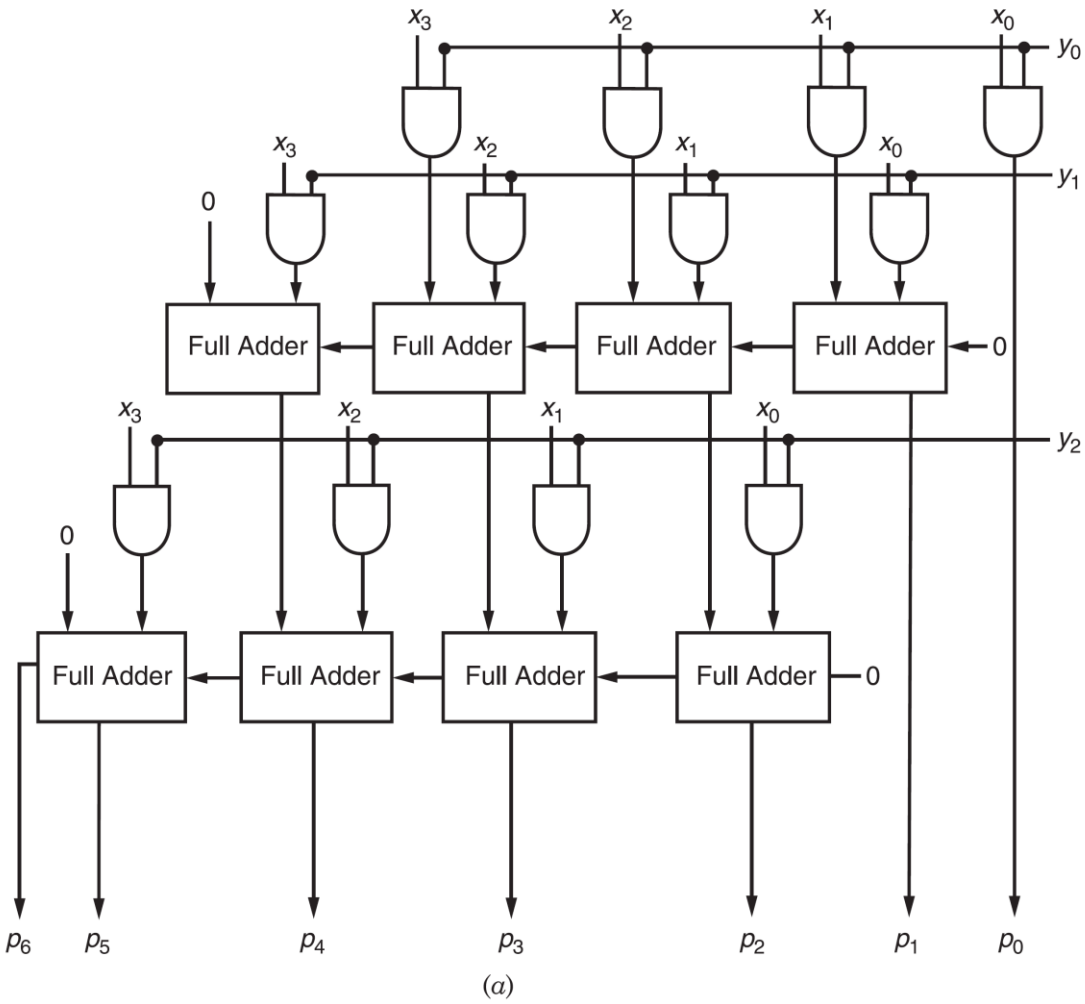
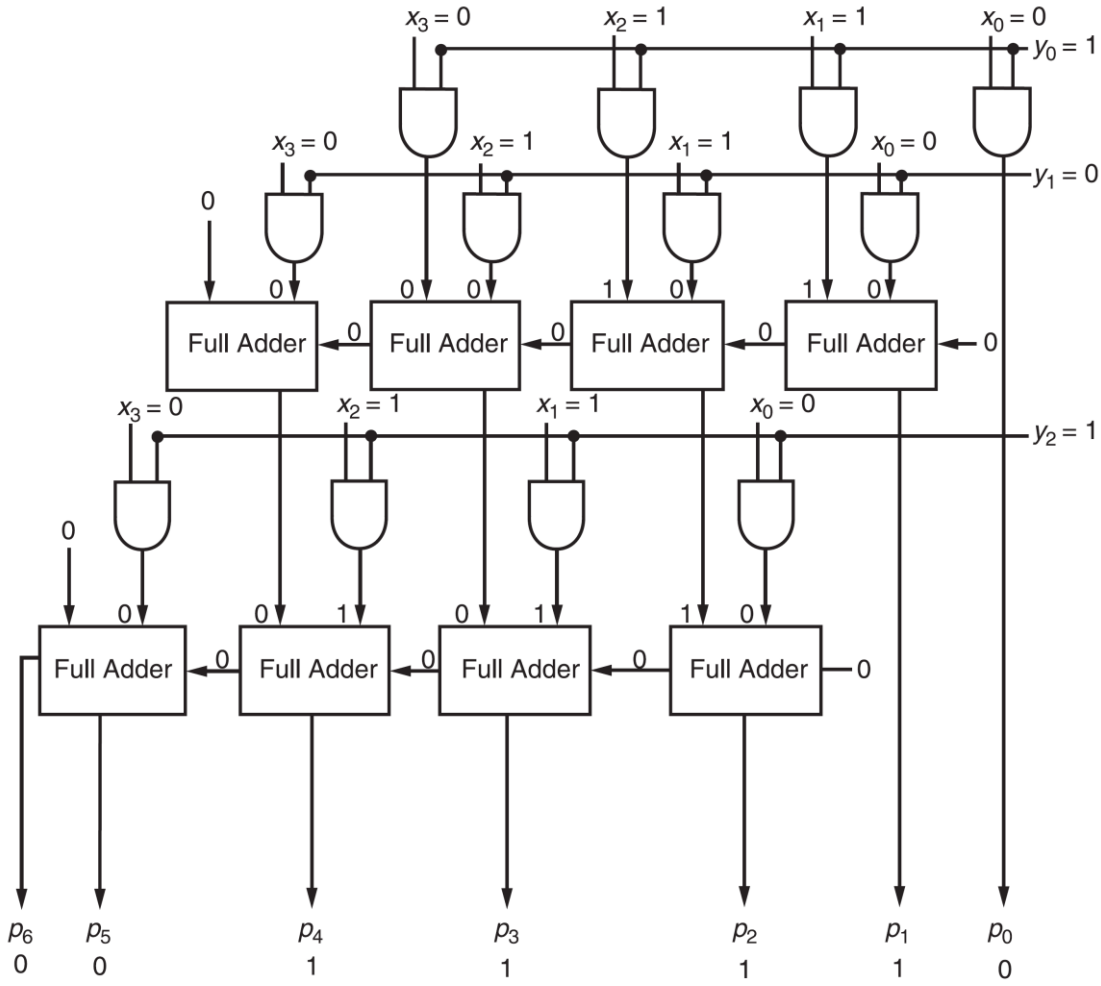


Figure 2.2 (a) A 4-bit by 3-bit multiplier circuit.



(*b*)

Figure 2.2 (b) Multiplication of 6 by 5.

CHAPTER II

ARRAY MULTIPLIER DESIGN

3.1 Introduction

In this chapter an 4×4 bit unsigned binary array (parallel) multiplier is designed and realized using the Altera FPGA device (Stratix II GX: EP2SGX30CF780C3). As mentioned before a fully parallel is used to achieve high speed at the expense of high area and power consumption. Qurtus II 11.0 software is used as a synthesis tool. The ModelSim Altera 6.6d is used as verification tools.

Tools used:

Simulation Software: QURTUS II 11.0 is used for design and implementation and ModelSim Altera 6.6d is used for modeling and simulation.

3.2 Design 4x4 bit parallel multiplier

Array multiplier is well known due to its regular structure. Multiplier circuit is based on add and shift algorithm. Each partial product is generated by the multiplication of the multiplicand with one multiplier bit using a simple AND gate. The partial product are shifted according to their bit orders and then added. The addition can be performed with normal carry

propagate adder. N-1 adders of M-bits size are required where N and M are the multiplier, and multiplicand length respectively.

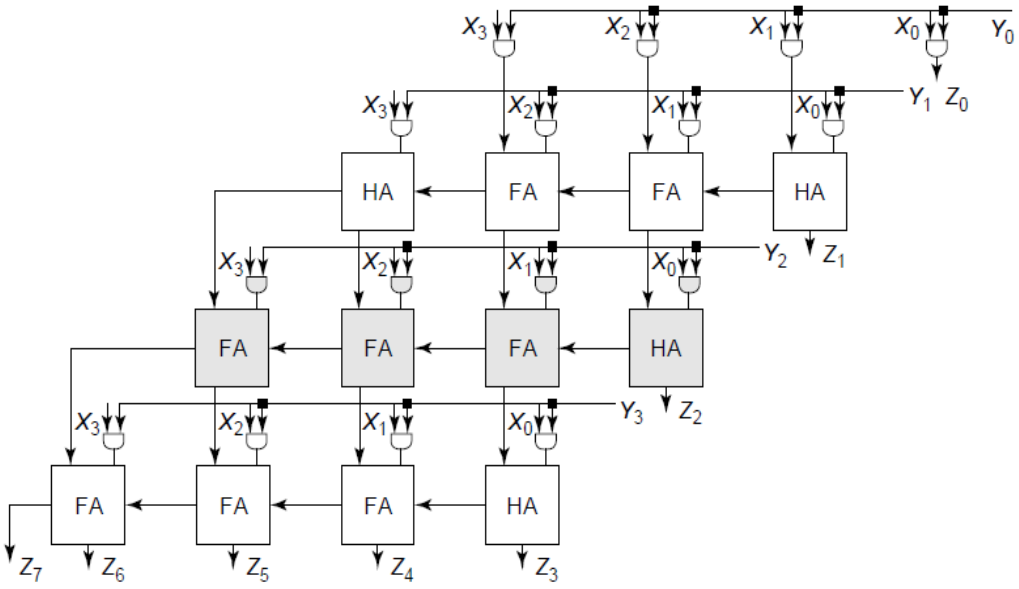


Figure 3.1 Array binary multiplier (parallel multiplier).

As shown in the Figure 3.1 the structure of multiplier include three basic component AND gate, Half adder (HA) and Full adder (FA). To generate the partial products AND gate is needed . The partial products then added using either HA or FA. For the 4x4 bit parallel multiplier design ,we need 4x4 AND gates and 3×4 FA . Note that some of the FA can be replaced by simple HA when the carry in C_{in} port is not used for further reduction in logic gates utilization.

3.2.1 Generation of Partial Products

As mentioned in the previous section , 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 product, hence it need N× M AND gates. For our design, 4×4 AND gates has been used. The way that the partial

products are generated or summed up is the difference between the different architectures of various multipliers.

3.2.2 Half Adder Design

A half adder is used to add two binary digits together, A and B. It produces S, the sum of A and B, and the corresponding carry out Co. Although by itself, a half adder is not extremely useful, it can be used as a building block for larger adding circuits (FA). One possible implementation is using two AND gates, two inverters, and an OR gate instead of a XOR gate as shown in figure 3.2.

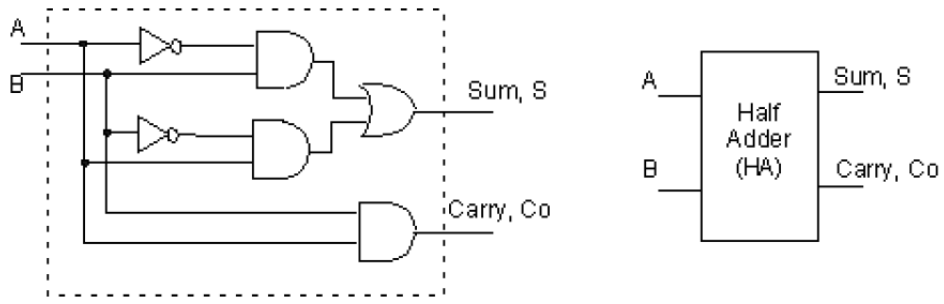


Figure 3.2 Half-Adder logic and block diagrams

Half-Adder truth table

A	B	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.2.2(A) VHDL Code of the Half Adder

A Half Adder can be synthesize using the Qurtus II 11.0 software .The VHDL code for a simple HA is

Use IEEE.STD_LOGIC_1164.ALL;

Entity HA_AN is

Port(X, Y: IN STD_LOGIC;

Cout, Sum: out STD_LOGIC);

End HA_AN;

Architecture Equation of HA_AN is

Begin

Sum <= Xxor Y;

Cout <= X and Y;

End Equation

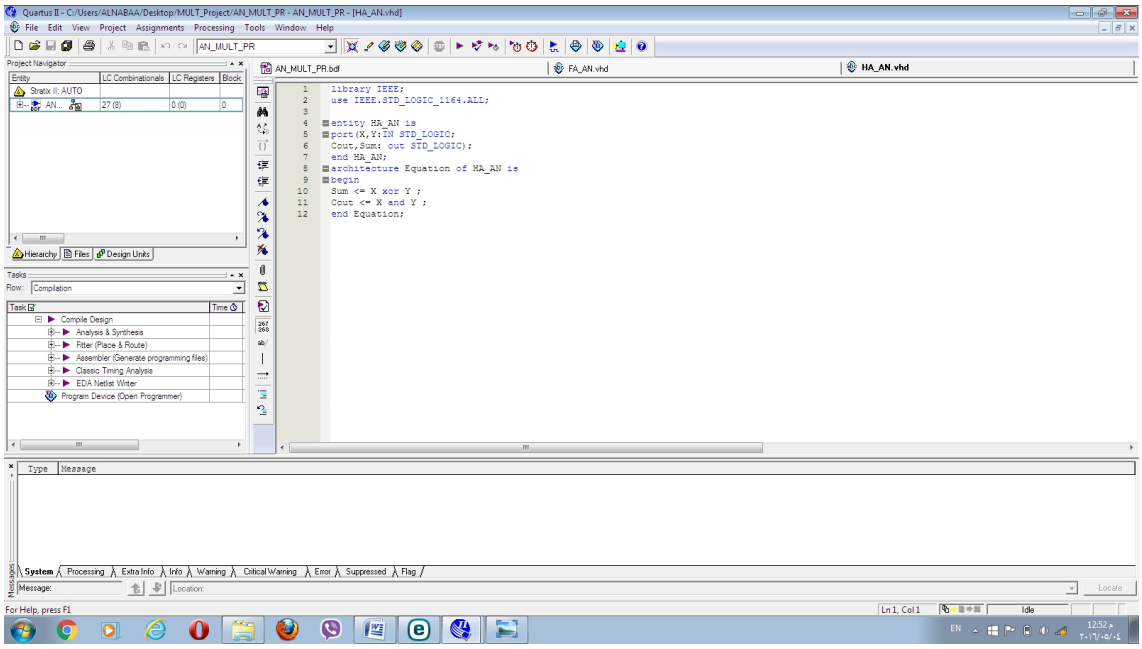


Figure 3.3 the half adder in VHDL code

3.2.3 Full Adder Design

Another component of the multiplier design is the FA of Figure 3.4. A full adder is a combinational circuit that performs the arithmetic sum of three bits: A, B and a carry in, C, from a previous addition, Also, as in the case of the half adder, the full adder produces the corresponding sum, S, and a carry out Co. As mentioned previously a full adder may be designed by two half adders in series as shown below in Figure (2.3)

The sum of A and B are fed to a second half adder, which then adds it to the carry in C (from a previous addition operation) to generate the final sum S. The carry out, Co, is the result of an OR operation taken from the carry outs of both half adders. There are a variety of adders in the literature both at the gate level and transistor level each giving different performances

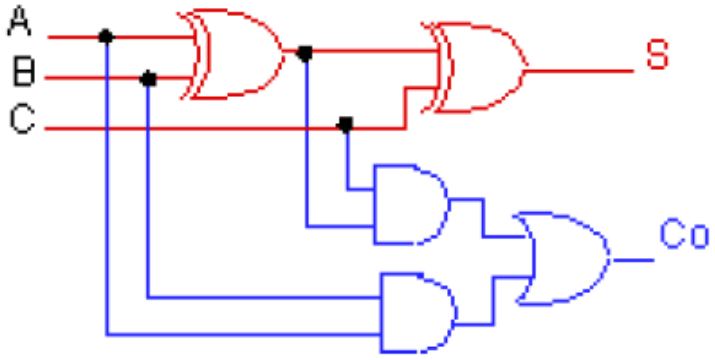


Figure 3.4 Full Adder block diagram

Full Adder truth table

A	*B*	*C*	*S*	*Co*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3.2.3(A) Full adder in VHDL

A Full Adder can be synthesize using the Qurtus II 11.0 software .The VHDL code for a simple FA is

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
Entity FA_AN is
Port(X, Y, and Cin: IN STD_LOGIC;
Cout, Sum: OUT STD_LOGIC);
End FA_AN;
Architecture equations of FA_AN is
Begin
Sum <= X xor Y xor Cin ;
Cout <= (X and Y) or (X and Cin) or (Y and Cin);
End equations;
```

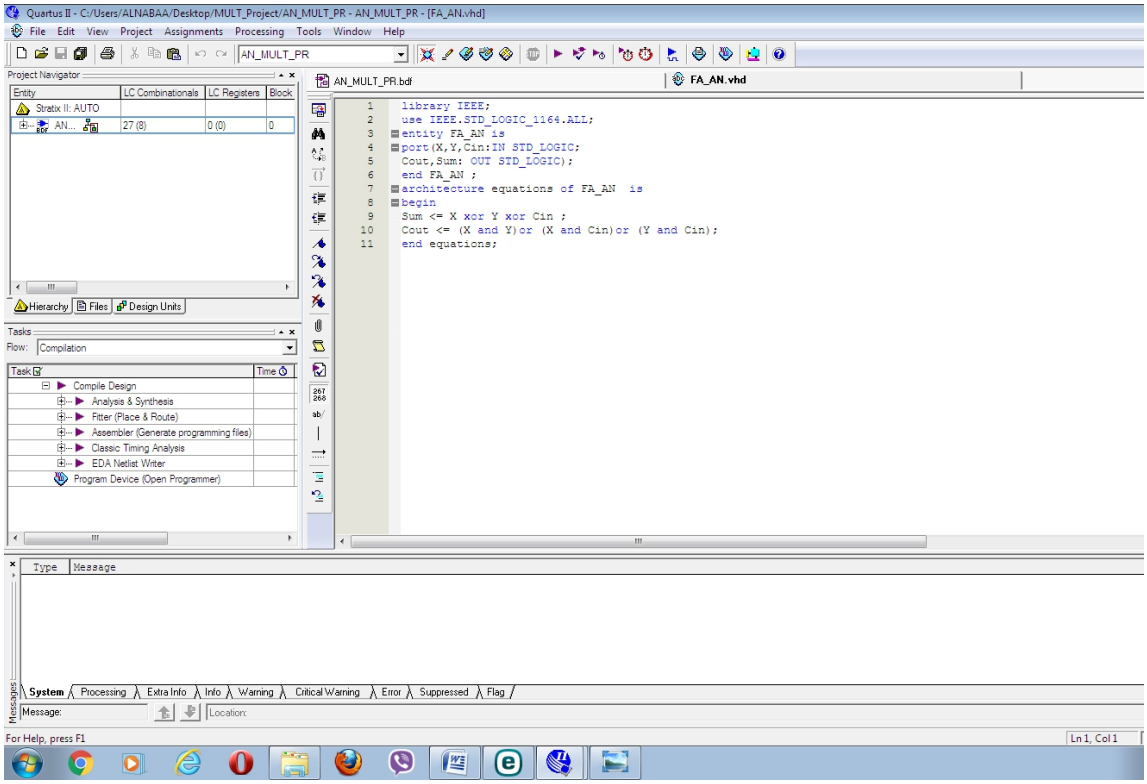


Figure 3.5 the full adder VHDL code

3.3 VHDL code of parallel multiplier

Using the schematic additer in Qurtus II software, it can be realize the final multiplier structure as shown in Figure 3.6 by combining the AND gates, HA, and FA blocks.

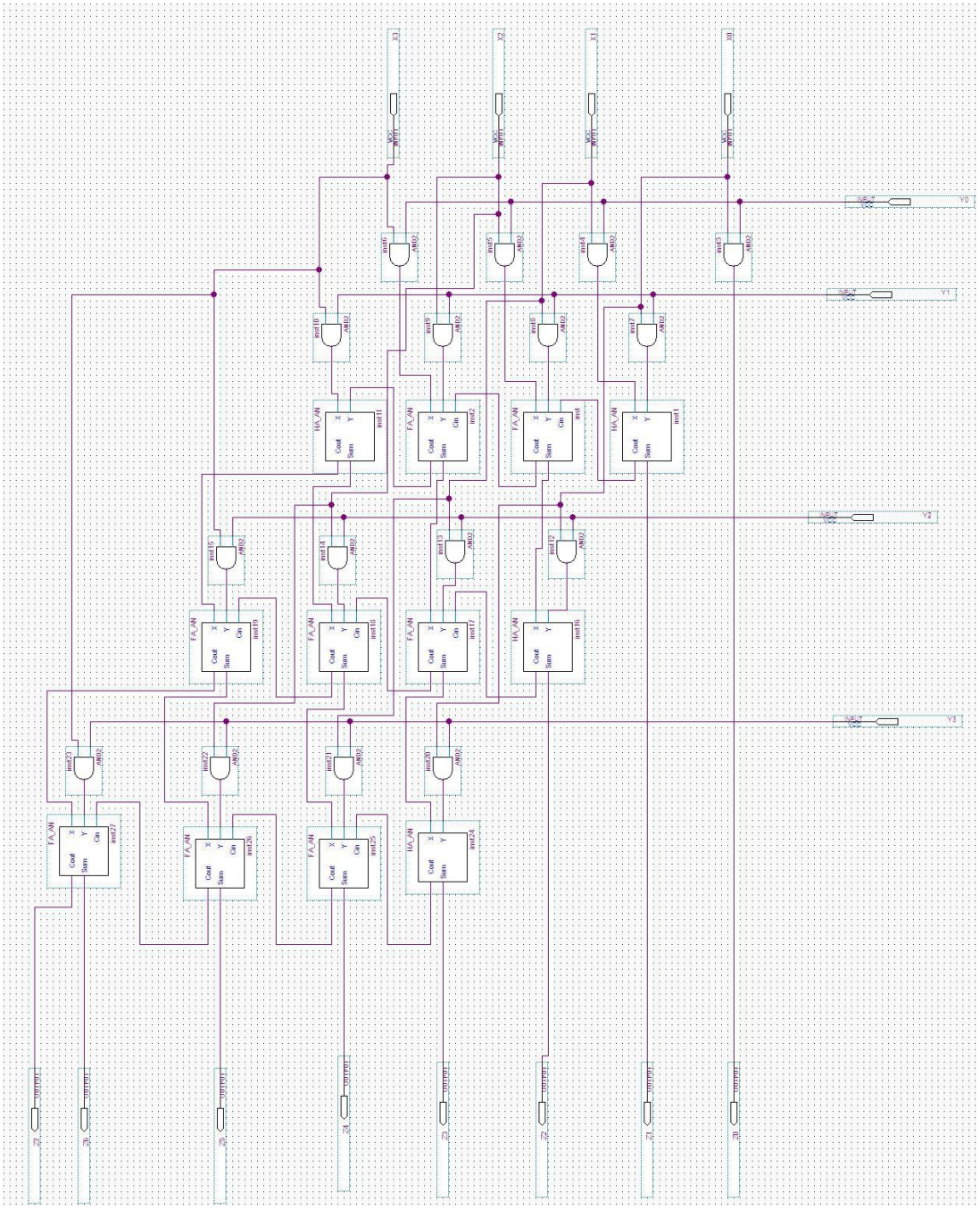


Figure 3.6 the schematic diagram of 4×4 array multiplier

The final VHDL code o the array multiplier is

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
LIBRARY work;
ENTITY AN_MULT_PR IS
        PORT
        (
                Y0:  IN STD_LOGIC;
                X0:  IN STD_LOGIC;
                X1:  IN STD_LOGIC;
                X2:  IN STD_LOGIC;
                X3:  IN STD_LOGIC;
                Y1:  IN STD_LOGIC;
                Y2:  IN STD_LOGIC;
                Y3:  IN STD_LOGIC;
                Z0:  OUT STD_LOGIC;
                Z1:  OUT STD_LOGIC;
                Z2:  OUT STD_LOGIC;
                Z3:  OUT STD_LOGIC;
                Z4:  OUT STD_LOGIC;
                Z5:  OUT STD_LOGIC;
                Z6:  OUT STD_LOGIC;
                Z7:  OUT STD_LOGIC
        );
END AN_MULT_PR;
ARCHITECTURE bdf _type OF AN_MULT_PR IS
```

```
COMPONENT fa _an
	PORT(X: IN STD_LOGIC;
		Y: IN STD_LOGIC;
		Cin: IN STD_LOGIC;
		Cout: OUT STD_LOGIC;
		Sum: OUT STD_LOGIC
	);
END COMPONENT;
COMPONENT ha_an
	PORT(X: IN STD_LOGIC;
		Y: IN STD_LOGIC;
		Cout: OUT STD_LOGIC;
		Sum: OUT STD_LOGIC
	);
END COMPONENT;
SIGNAL	SYNTHESIZED_WIRE_0:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_1:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_2:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_3:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_4:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_5:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_6:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_7:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_8:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_9:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_10:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_11:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_12:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_13:  STD_LOGIC;
```

```
SIGNAL	SYNTHESIZED_WIRE_14:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_15:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_16:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_17:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_18:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_19:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_20:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_21:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_22:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_23:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_24:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_25:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_26:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_27:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_28:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_29:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_30:  STD_LOGIC;
SIGNAL	SYNTHESIZED_WIRE_31:  STD_LOGIC;
BEGIN
b2v_inst:  fa _an
PORT MAP(X => SYNTHESIZED_WIRE_0,
		Y => SYNTHESIZED_WIRE_1,
		Cin => SYNTHESIZED_WIRE_2,
		Cout => SYNTHESIZED_WIRE_20,
		Sum => SYNTHESIZED_WIRE_7);
b2v_inst1: ha _an
PORT MAP(X => SYNTHESIZED_WIRE_3,
		Y => SYNTHESIZED_WIRE_4,
		Cout => SYNTHESIZED_WIRE_2,
```

```
                Sum => Z1);
SYNTHESIZED_WIRE_5 <= Y1 AND X3;
b2v_inst11: ha _an
PORT MAP(X => SYNTHESIZED_WIRE_5,
                Y => SYNTHESIZED_WIRE_6,
                Cout => SYNTHESIZED_WIRE_15,
                Sum => SYNTHESIZED_WIRE_12);
SYNTHESIZED_WIRE_8 <= Y2 AND X0;
SYNTHESIZED_WIRE_10 <= Y2 AND X1;
SYNTHESIZED_WIRE_13 <= Y2 AND X2;
SYNTHESIZED_WIRE_16 <= Y2 AND X3;
b2v_inst16: ha _an
PORT MAP(X => SYNTHESIZED_WIRE_7,
                Y => SYNTHESIZED_WIRE_8,
                Cout => SYNTHESIZED_WIRE_11,
                Sum => Z2);
b2v_inst17: fa _ an
PORT MAP(X => SYNTHESIZED_WIRE_9,
                Y => SYNTHESIZED_WIRE_10,
                Cin => SYNTHESIZED_WIRE_11,
                Cout => SYNTHESIZED_WIRE_14,
                Sum => SYNTHESIZED_WIRE_21);
b2v_inst18: fa_an
PORT MAP(X => SYNTHESIZED_WIRE_12,
                Y => SYNTHESIZED_WIRE_13,
                Cin => SYNTHESIZED_WIRE_14,
                Cout => SYNTHESIZED_WIRE_17,
                Sum => SYNTHESIZED_WIRE_23);
b2v_inst19: fa _an
```

```
PORT MAP(X => SYNTHESIZED_WIRE_15,
         Y => SYNTHESIZED_WIRE_16,
         Cin => SYNTHESIZED_WIRE_17,
         Cout => SYNTHESIZED_WIRE_29,
         Sum => SYNTHESIZED_WIRE_26);
b2v_inst2: fa _an
PORT MAP(X => SYNTHESIZED_WIRE_18,
         Y => SYNTHESIZED_WIRE_19,
         Cin => SYNTHESIZED_WIRE_20,
         Cout => SYNTHESIZED_WIRE_6,
         Sum => SYNTHESIZED_WIRE_9);
SYNTHESIZED_WIRE_22 <= Y3 AND X0;
SYNTHESIZED_WIRE_24 <= Y3 AND X1;
SYNTHESIZED_WIRE_27 <= Y3 AND X2;
SYNTHESIZED_WIRE_30 <= Y3 AND X3;
b2v_inst24: ha_an
PORT MAP(X => SYNTHESIZED_WIRE_21,
         Y => SYNTHESIZED_WIRE_22,
         Cout => SYNTHESIZED_WIRE_25,
         Sum => Z3);


b2v_inst25: fa_an
PORT MAP(X => SYNTHESIZED_WIRE_23,
         Y => SYNTHESIZED_WIRE_24,
         Cin => SYNTHESIZED_WIRE_25,
         Cout => SYNTHESIZED_WIRE_28,
         Sum => Z4);
b2v_inst26: fa _an
PORT MAP(X => SYNTHESIZED_WIRE_26,
```

```
                Y => SYNTHESIZED_WIRE_27,
                Cin => SYNTHESIZED_WIRE_28,
                Cout => SYNTHESIZED_WIRE_31,
                Sum => Z5);
b2v_inst27: fa_ an
PORT MAP(X => SYNTHESIZED_WIRE_29,
                Y => SYNTHESIZED_WIRE_30,
                Cin => SYNTHESIZED_WIRE_31,
                Cout => Z7,
                Sum => Z6);
Z0 <= Y0 AND X0;
SYNTHESIZED_WIRE_3 <= Y0 AND X1;
SYNTHESIZED_WIRE_0 <= Y0 AND X2;
SYNTHESIZED_WIRE_18 <= Y0 AND X3;
SYNTHESIZED_WIRE_4 <= Y1 AND X0;
SYNTHESIZED_WIRE_1 <= Y1 AND X1;
SYNTHESIZED_WIRE_19 <= Y1 AND X2;
END bdf_type;
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
Entity FA_AN is
Port(X, Y, Cin: IN STD_LOGIC;
Cout, Sum: OUT STD_LOGIC);
End FA_AN;
Architecture equations of FA_AN is
Begin
Sum <= X xor Y xor Cin;
Cout <= (X and Y) or (X and Cin) or (Y and Cin);
End equations;
```

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
Entity HA_AN is
Port(X, Y: IN STD_LOGIC;
Cout, Sum: out STD_LOGIC);
End HA_AN;
Architecture Equation of HA_AN is
Begin
Sum <= X xor Y;
Cout <= X and Y;
End Equation;
```

After compilation the synthesis process, we can verify the functionality o the final design project by observing the RTL view as shown in Figure 3.7.

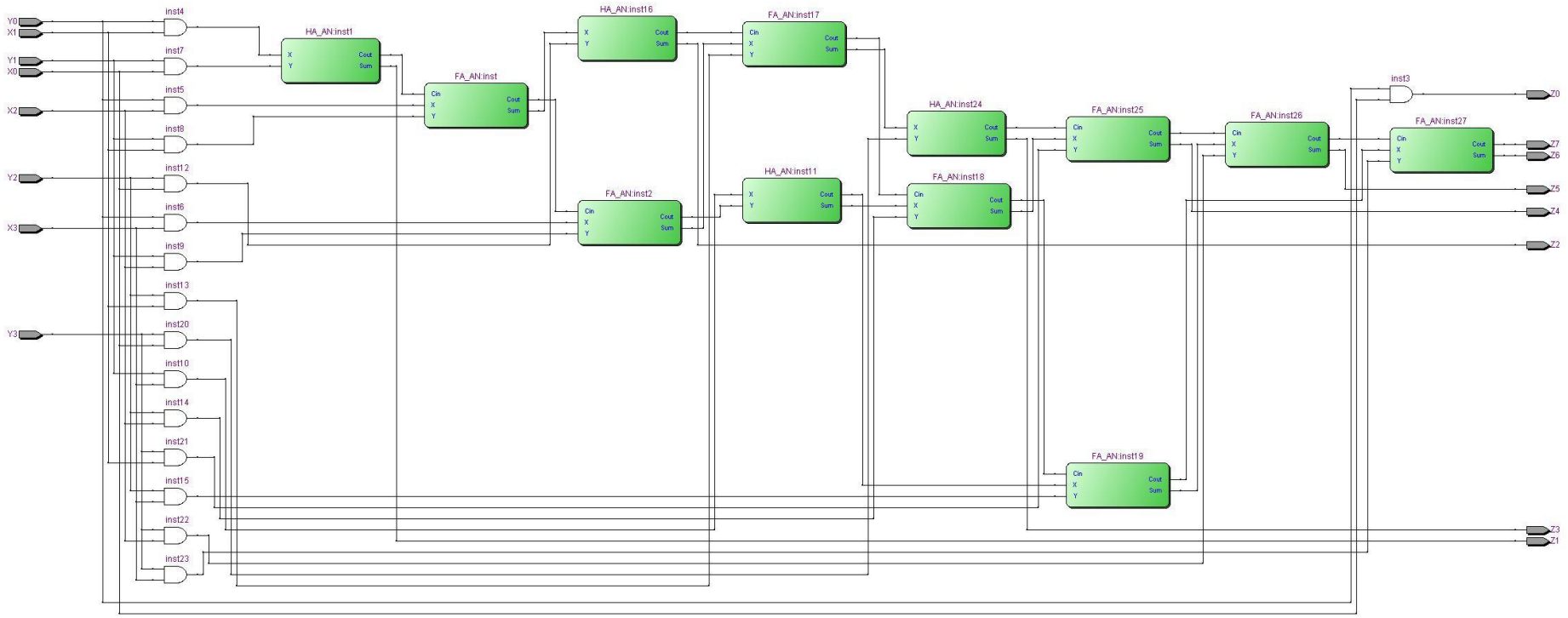


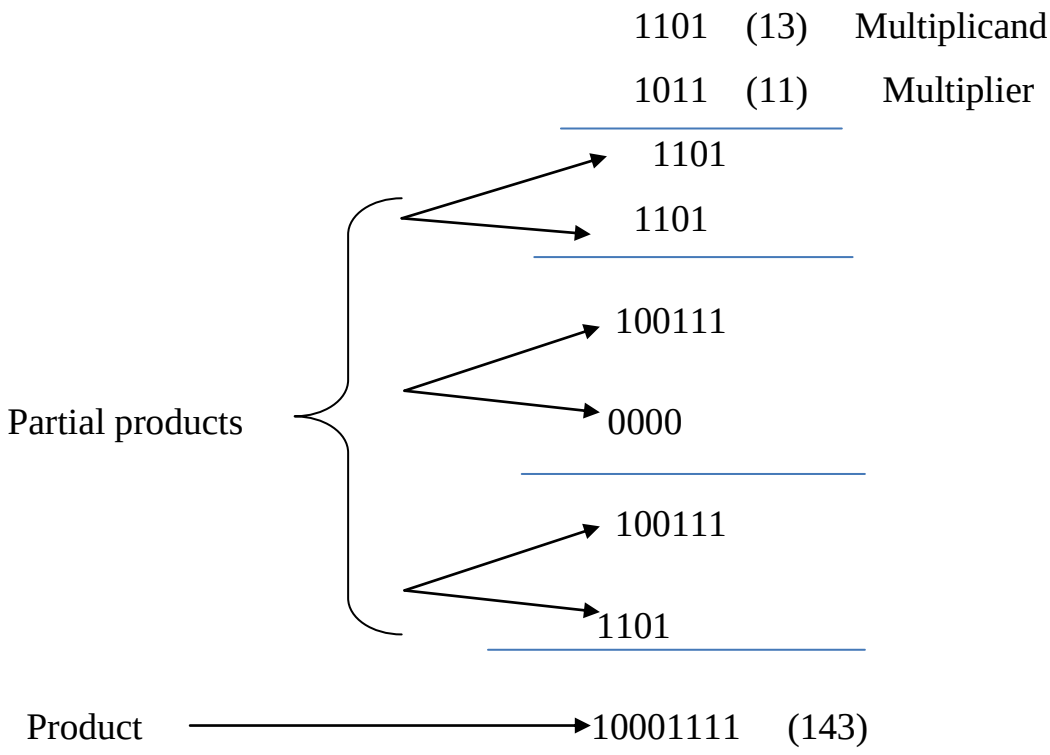
Figure 3.7 RTL view of synthesized 4×4 array multiplier

CHAPTER IV

SERIAL MULTIPLIER DESIGN

4.1 Design of Serial binary multiplier

In this chapter a 4×4 bit serial multiplier for unsigned binary numbers is designed. Encoded in VHDL code using the Qurtus II 11.0 software and then simulated by the ModelSim Altera 6.6d. The performance of the project can be verified and compared with the design architecture presented in Chapter 4. To illustrate the design procedure a binary numbers 1101 multiply by 1011 shown below is used as example .It is clear that the binary multiplication requires only shifting and adding. The following example shows how each partial product is added in as soon as it is formed. This eliminates the need for adding more than two binary numbers at a time



The multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4- bit multiplier register, and an 8 bit register for the product. The product register serves as an accumulate the sum of the partial products. Instead of shifting the multiplicand left each time before it is added, as was done in the previous example, it is more convenient to shift the product register to the right each time.

Figure 4-1 shows a block diagram for such a multiplier. As indicated by the arrows on the diagram, 4 bit from the accumulator and 4 bit from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. Followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs.

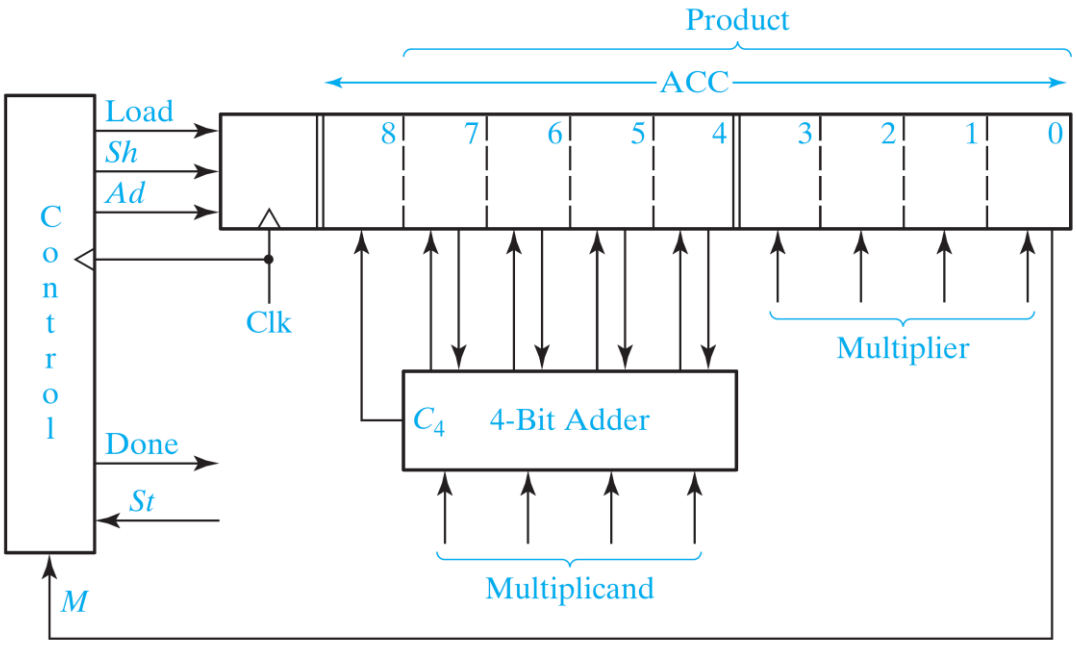
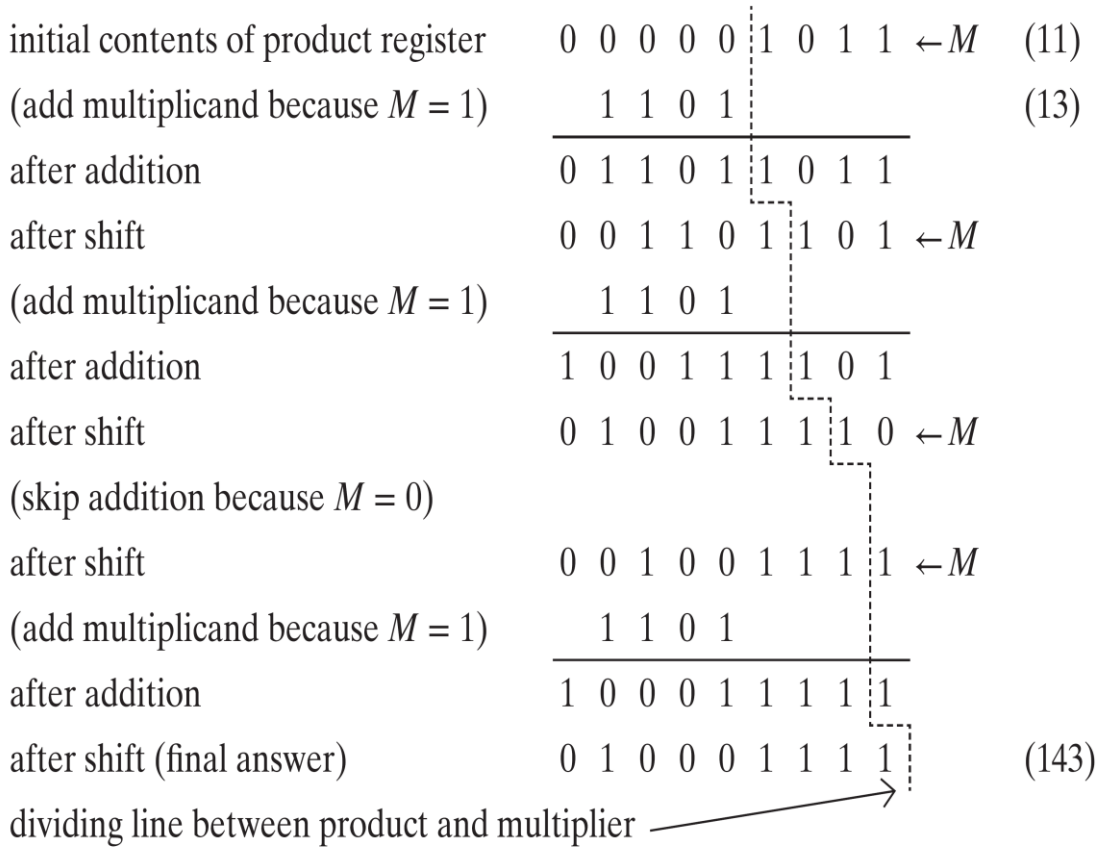


Figure 4.1 block diagram for binary multiplier

The adder calculates the sum of its inputs, and when an add signal (Ad) occurs, the adder outputs are stored in the accumulator by the next rising clock edge, thus causing the multiplicand to the added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry (C_4) which is generated when the multiplicand is added to the accumulator. Note that when ACC is shifted right, the input to port 8 of ACC must be 0. Then, a following shift will shift the correct value in to bit 7 of ACC. The shift input to ACC is not explicitly shown in Figure 4.1. Because the lower four bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. The load signal loads the multiplier into the lower four bits of ACC and at the same time clears the upper 4 bits the shift signal (sh) causes the contents of the product register (including the multiplier) to be shifted one place to the right when the next rising clock edge occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal (st=1) has been received. If the current multiplier pit (M) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit 0, the addition is skipped and only the right shit occurs.



4.1.1 Design of Control Circuit

The control must be designed to output the proper sequence of add and shift signals. Figure 4-2 shows a state graph for control circuit. M/Ad means if M=1, then the output Ad is 1 (and the other output are 0). M'/Sh means if M' = 1 (M=0), then output Sh is 1 (and other output are 0). In figure 4-2, S_0 is the reset state, and the circuit says in S_0 until a start signal (St=1) is received. This generated a Load signal , which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of ACC to cleared on to the lower 4 bits of the accumulator (ACC) and the upper 5 bits of ACC to be cleared on the next rising clock edge . In state S_1, the low order bit of the multiplier (M) is tested. If M=1, an add signal is generated and then, a shift signal is generated in S_1, a shift signal I generated because adding 0 can be omitted. Similarly, in states S_3,

$S_{5,}$ and $S_{7,}$ M is tested to determine whether to generate an add signal followed by shift or just a shift signal. A shift signal is always generated at the next clock time following an add signal (states $S_{2,}$ $S_{4,}$ $S_{6,}$ and S_8). After four shifts have been generated, all four multiplier bits have been processed, and the control circuit goes to a done state and terminates the multiplication process.

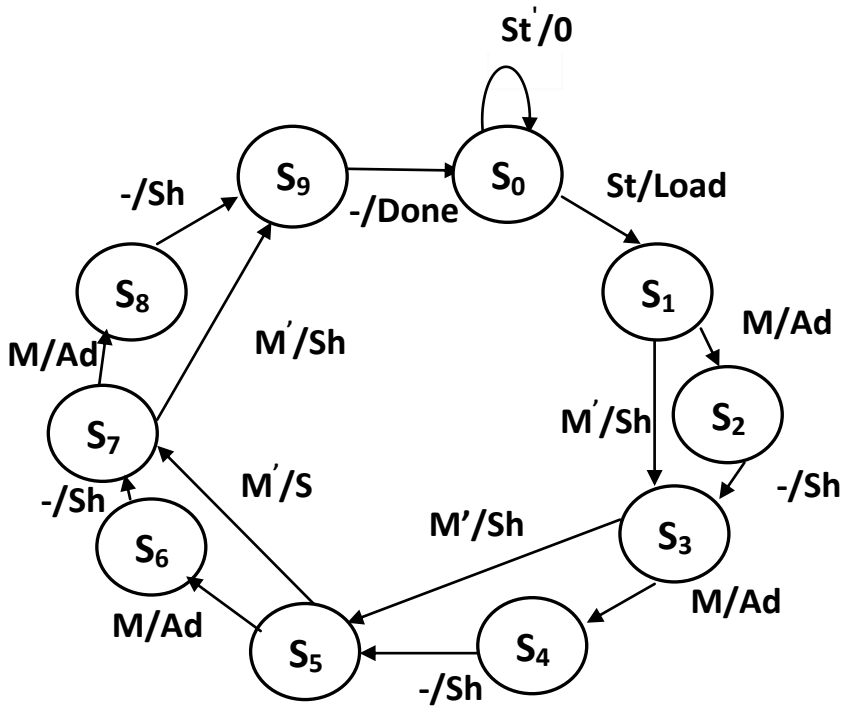


Figure 4.2 State Graphs for Multiplier Control

Note that the done signal cannot be turned on in state S_7 or S_8 because the last shift not occurs until the next active clock edge. The last shift occurs at the same time the transition to S9 occurs .The control signal Sh enable the shifting to occur. In general, if a control signal is turned on in state S_n, the resulting action does not occur until the next active clock edge takes the circuit to the next state. State S9 could be eliminated if it was acceptable to turn on the done signal in state S_0 when St =0.

4.1.2 Operation of Control Circuit

In section 4.1, we designed a multiplier for unsigned binary number. In the section we will shows several ways of writing VHDL code to describe the multiplier operation. As in Fig 4.1 bits from accumulator (ACC) and 4 bit from the multiplicand register are connected to the adder inputs; the 4 sum bit and the carry output from the adder are connected back to the accumulator. When an add signal (Ad) occurs, the adder outputs are loaded into the accumulator by the next clock pulse, thus, causing the multiplicand to be add to the accumulator. An extra bit at the left end of the product register temporarily stores any carry which is generated when the multiplicand is added to the accumulator. When shift signal (Sh) occurs, all 9bits of ACC are shifted right by the next clock pulse.

4.2 VHDL Code For A binary Multiplier

In this section, we will write a behavioral VHDL model for multiplier based on the block diagram of Fig 4.1 and state graph of Fig 4.2.This model will allows as to check out the basic design of the multiplier and multiplication algorithm before proceeding with a more detailed design. Because the control circuit has ten states, we have declared an integer in the range 0 to 9 for the state signal. The signal ACC represents the 9-bit accumulator output.

```
Library IEEE;
Use IEEE.STD_LOGIC_1164. ALL;
Use IEEE.STD_LOGIC_ARITH.ALL;
Use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
Entity mult4B is
Port (Clk, St: in std_logic;
Mplier, Mcand: in std_logic_vector (3 downto 0);
Done: out std_logic;
Product: out std_logic_vector (7 downto 0));
End mult4B;
Architecture behave1 of mult4B is
Signal state: integer range 0 to 9;
Signal ACC: std_logic_vector (8 downto 0);
Alias M: std_logic is ACC (0);
Begin
Product<=ACC (7 downto 0);
Process (ClK)
Begin
If Clk'event and Clk='1'then
Case state is
When 0=>
If St='1' then
ACC (8 downto 4) <= "00000";
ACC (3 downto 0) <=Mplier;
State<=1;
End if;
When 1|3|5|7=>
If M='1'then
ACC (8 downto 4) <= ('0'& ACC (7 downto 4)) + Mcand;
State <=State+1;
Else ACC<='0'&ACC (8 downto 1);
State<=state+2;
End if;
```

```
When 2|4|6|8=>
ACC<='0'&ACC (8 downto 1);
State<=state+1;
When 9=>
State<=0;
End case;
End if;
End process;
Done <= '1'when state =9 else'0';
End behave1;
```

Figure 4.3 shows the RTL view of the design after the synthesis process is accomplished using ModelSim Altera simulator.

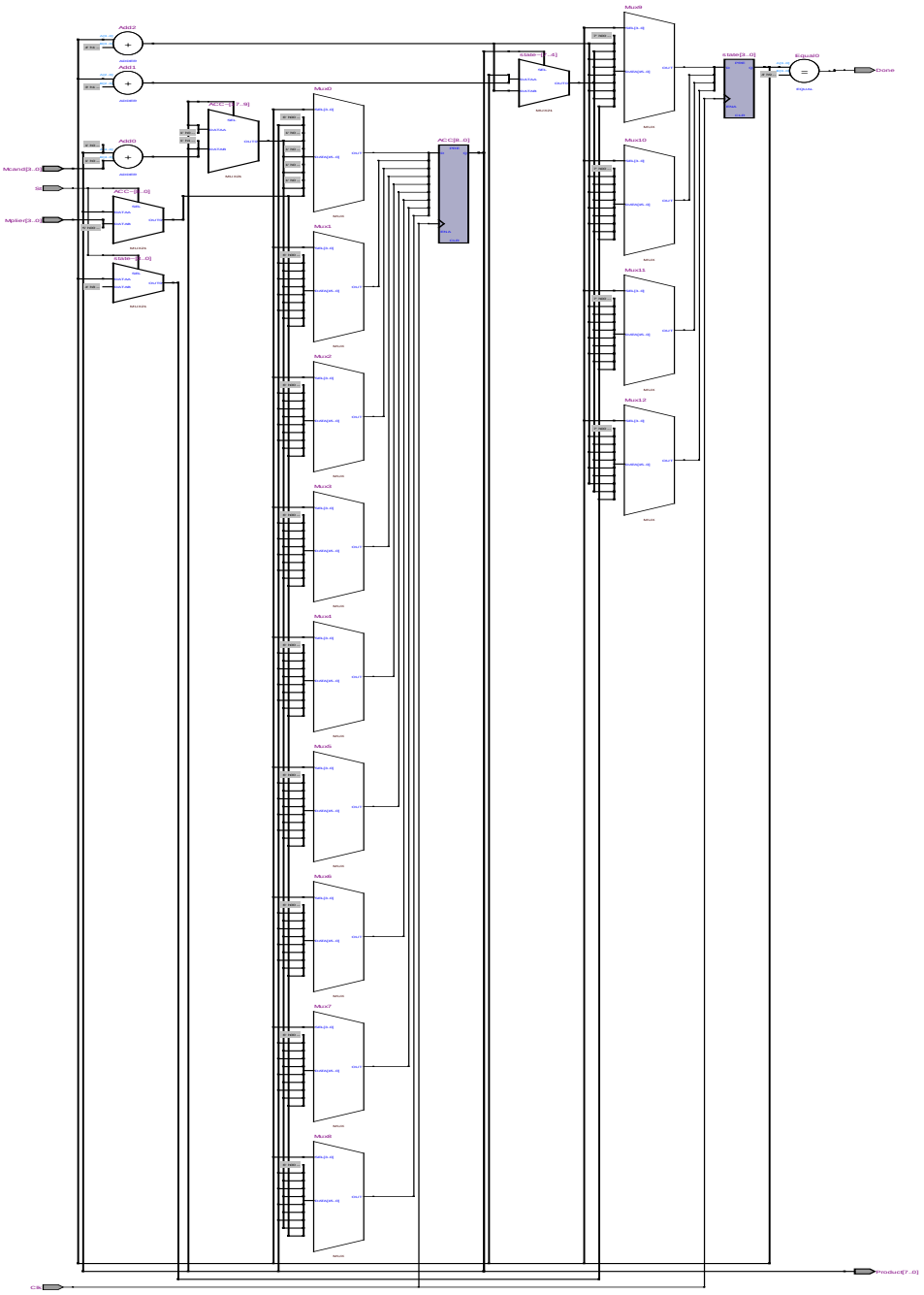


Figure 4.3 RTL view of synthesized serial multiplier

CHAPTER V

MULTIPLIERS RESULT

5.1 Parallel Multiplier Result

1. Power dissipation 0.74 mw
2. Clock 412 MHz
3. Combinational ALUTs 28
4. Logic register 16

Fig 5.1 shows decimal multiplication in parallel multiplier and binary multiplication in parallel multiplier shows in Fig 5.3

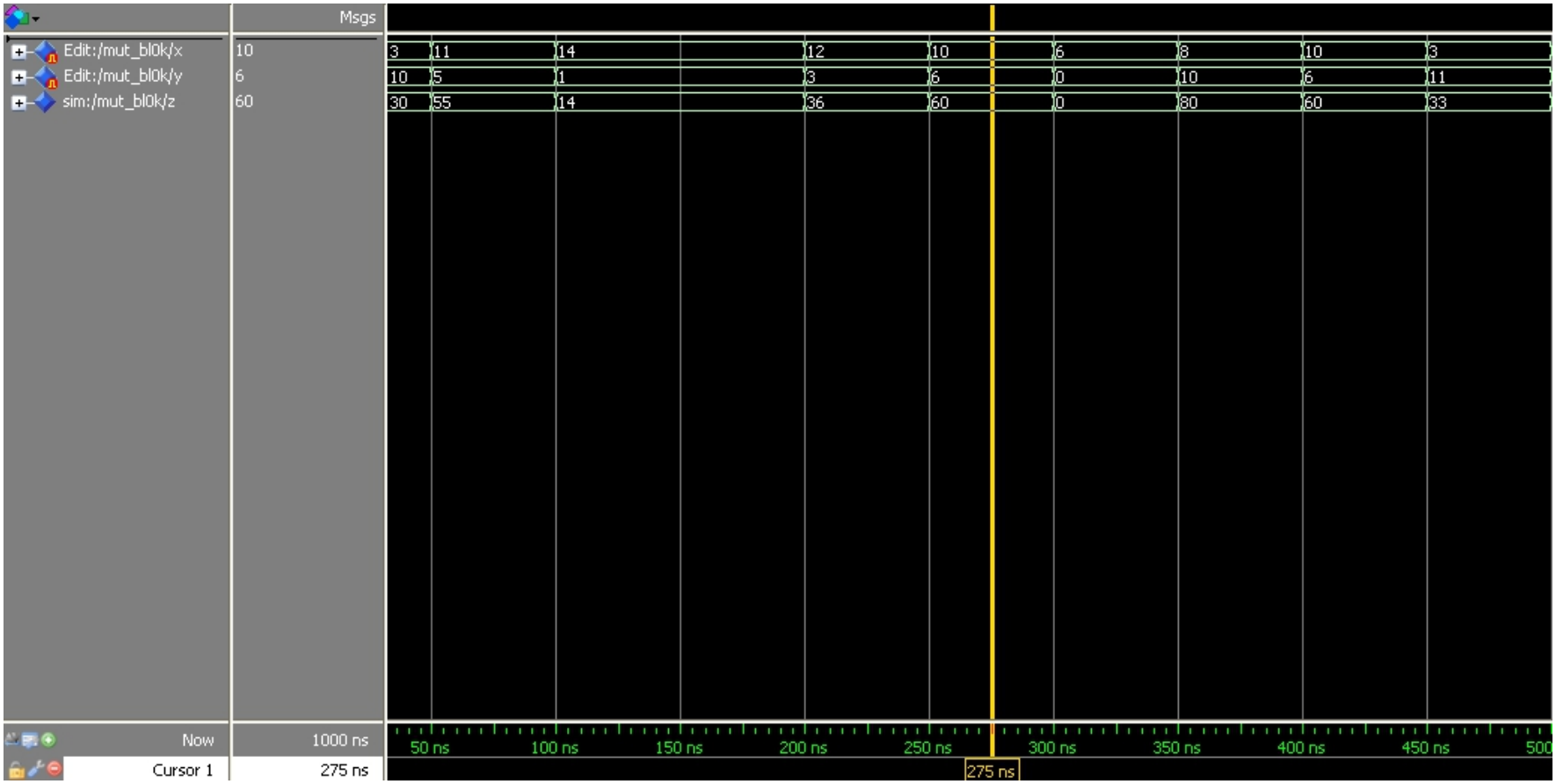


Figure 5.1 decimal parallel multiplication (3×10)

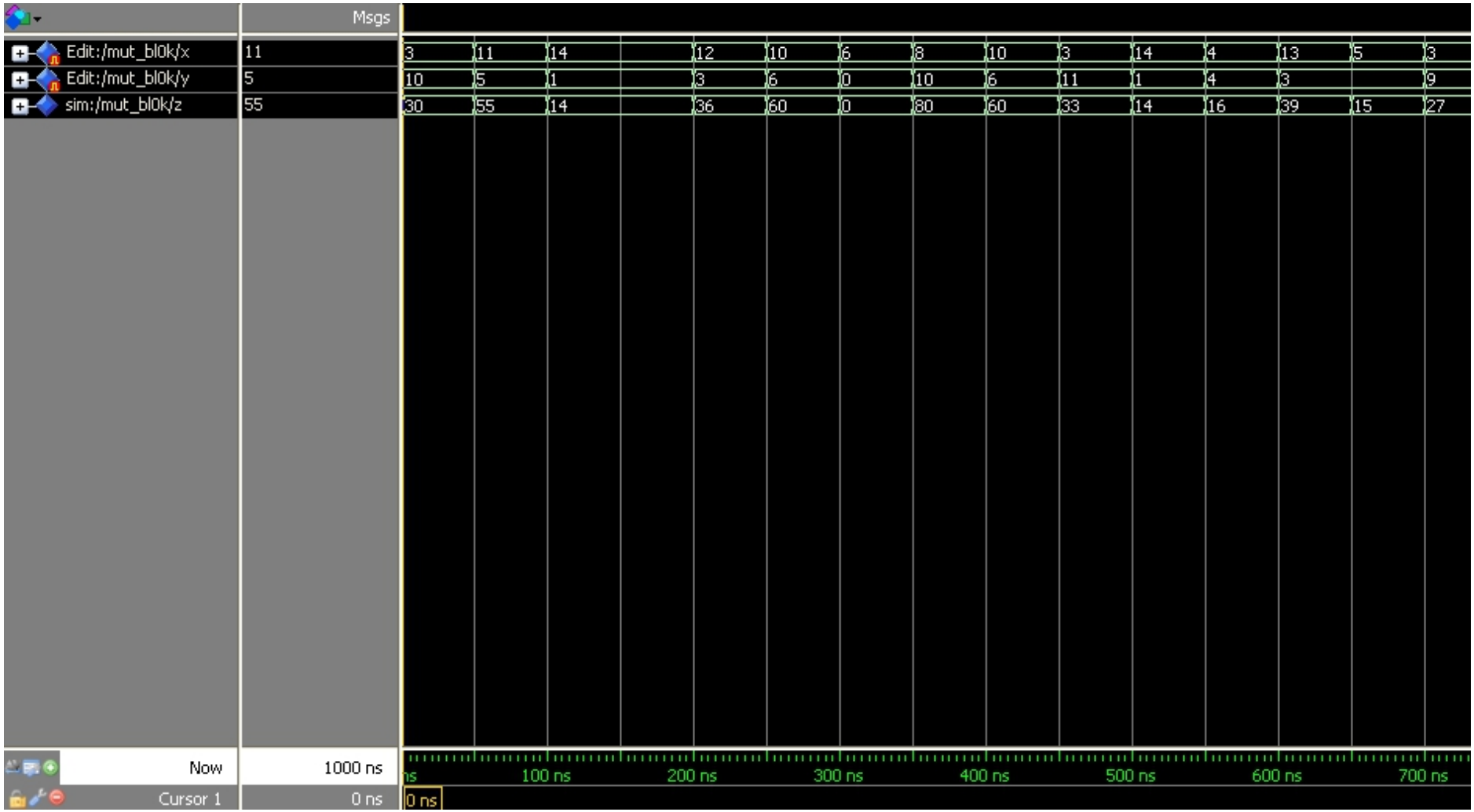


Figure 5.2 decimal parallel multiplication (11×5)

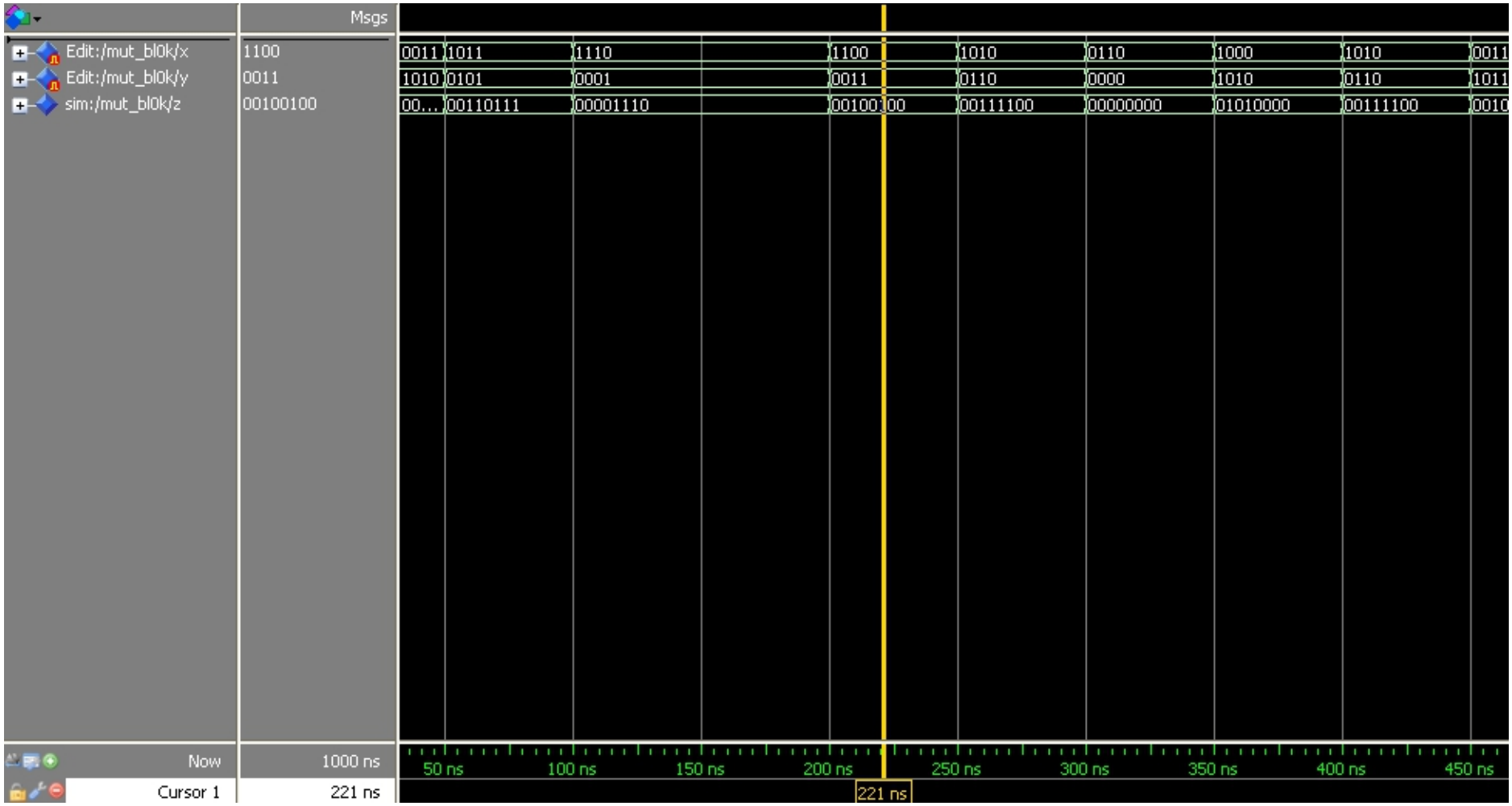


Figure 5.3 binary parallel multiplication (12×3)

5.2 Serial Multiplier Result

1. Power dissipation 0.62 mw
2. Clock 524 MHz (clock pulses period≈2 ns)
3. Combinational ALUTs 22
4. Logic registers 13

To calculate the number of clock that depend on multiplier that mean dependent on ADD or ADD SHIFT

In the example 4-1 the multiplier (1011)

1. The first digit is one (M=1) we need two clock cycle one for add and the second for shift
2. The second digit is zero (M=0) so we just need one clock cycle for shift
3. The third digit is 1 (M=1) so we need two clock cycle one for add and the second for shift
4. The fourth clock is 1 (M=1) so we need two clock cycle one for add and the second for shift

And one clock for loading the result two +one +two +two +one clock =eight clock is the number of clock for example 4-1.

And to calculate the time of one clock we divide one on number of clock the result 2ns and to calculate the total number of clock we multiply the total number of clock in the time for one clock. we conclude that the maximum number of clock is 9 for multiplier 1111 .and minimum number of clock is 5 for multiplier 0000 we use the Modelsim to obtain the result this program used to simulate the circuit do you work or not, and display

the result In fig 5.6 the multiplier is 0010 the result in binary and the number of clock is 6 clock and for example if the multiplier is 1o11 the number of clock required is 8 clock as shown in fig 5.7 and the result in decimal number.

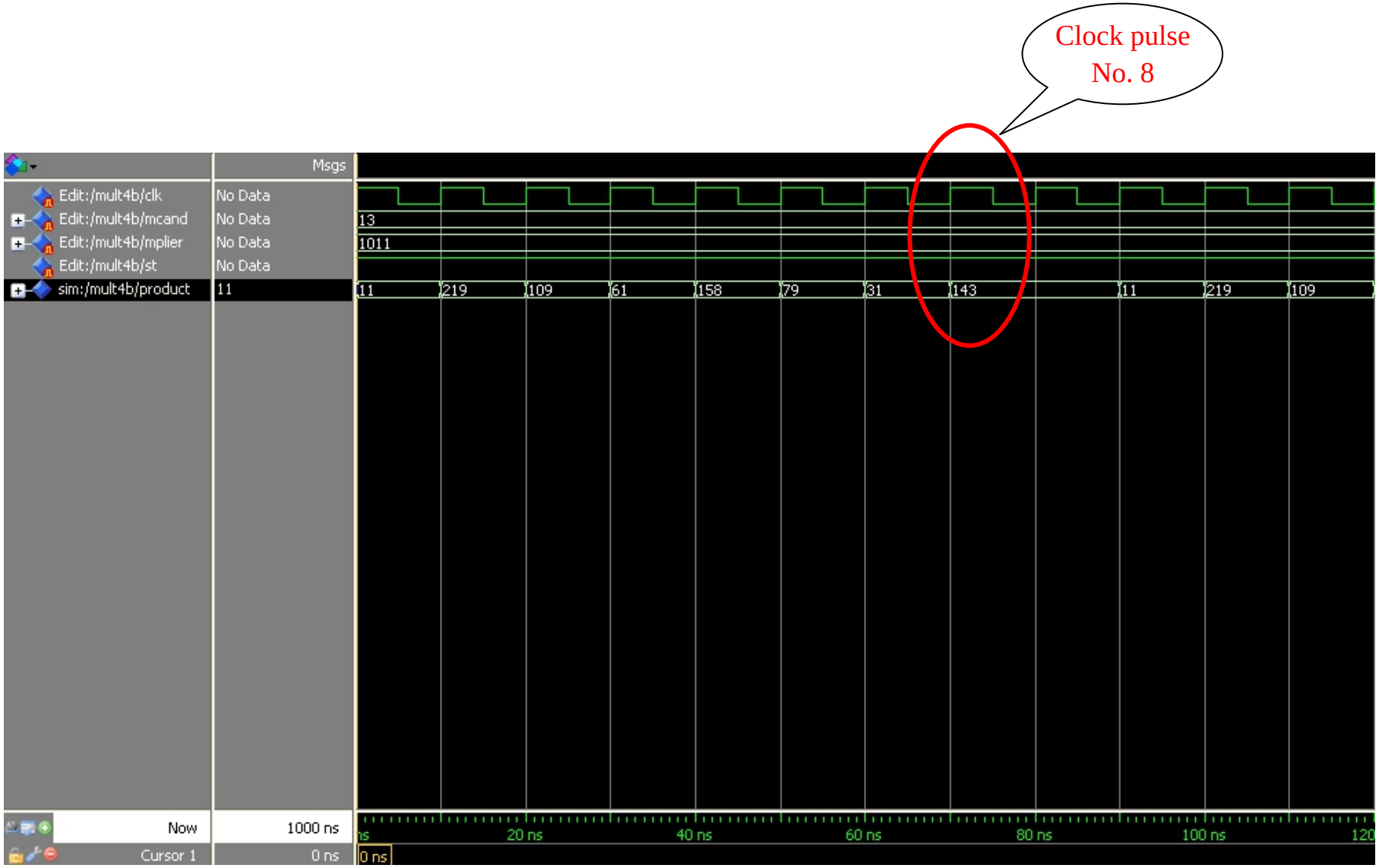


Figure 5.4 decimal serial multiplication (13× 11)

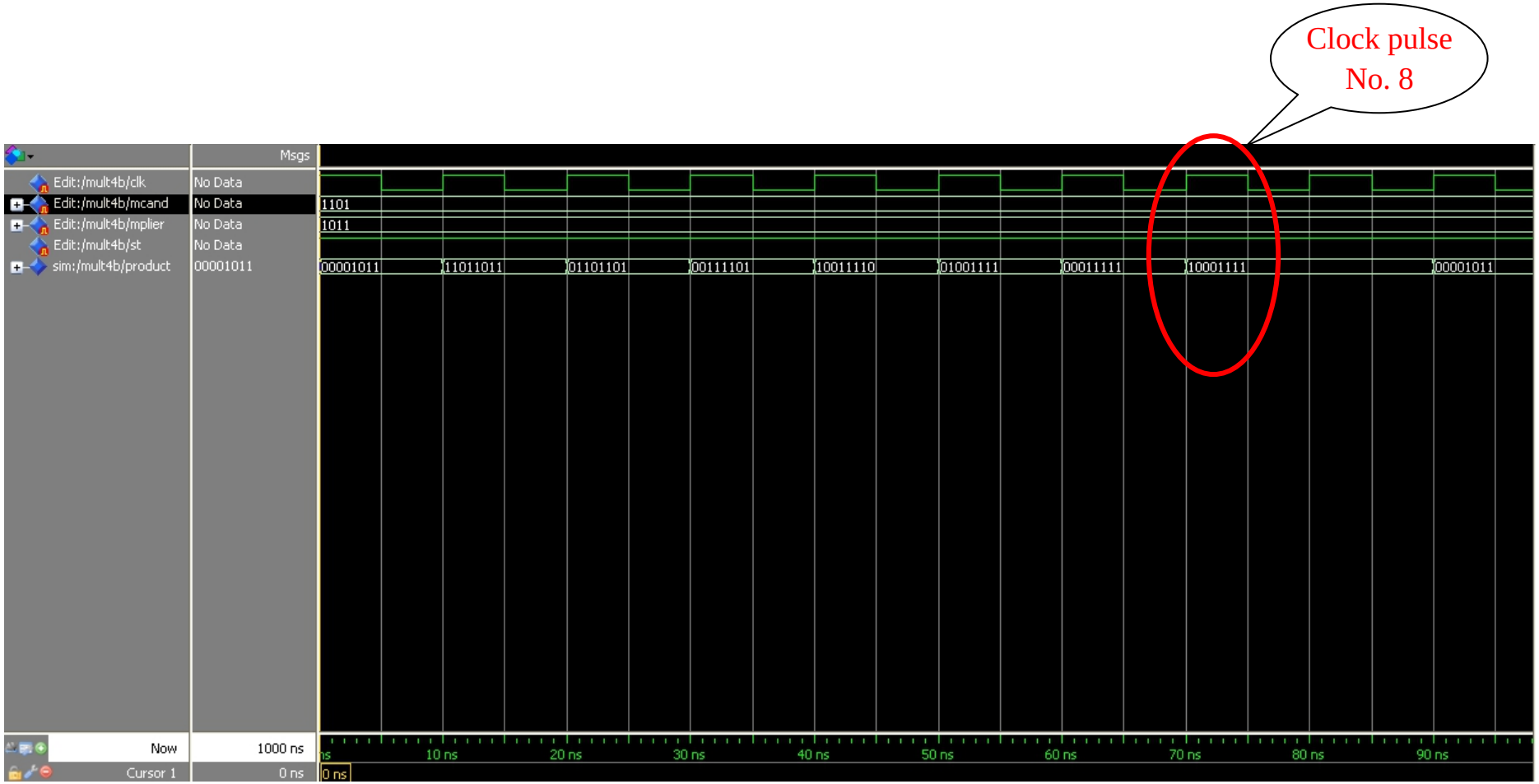


Figure 5.5 binary serial multiplication (13× 11)

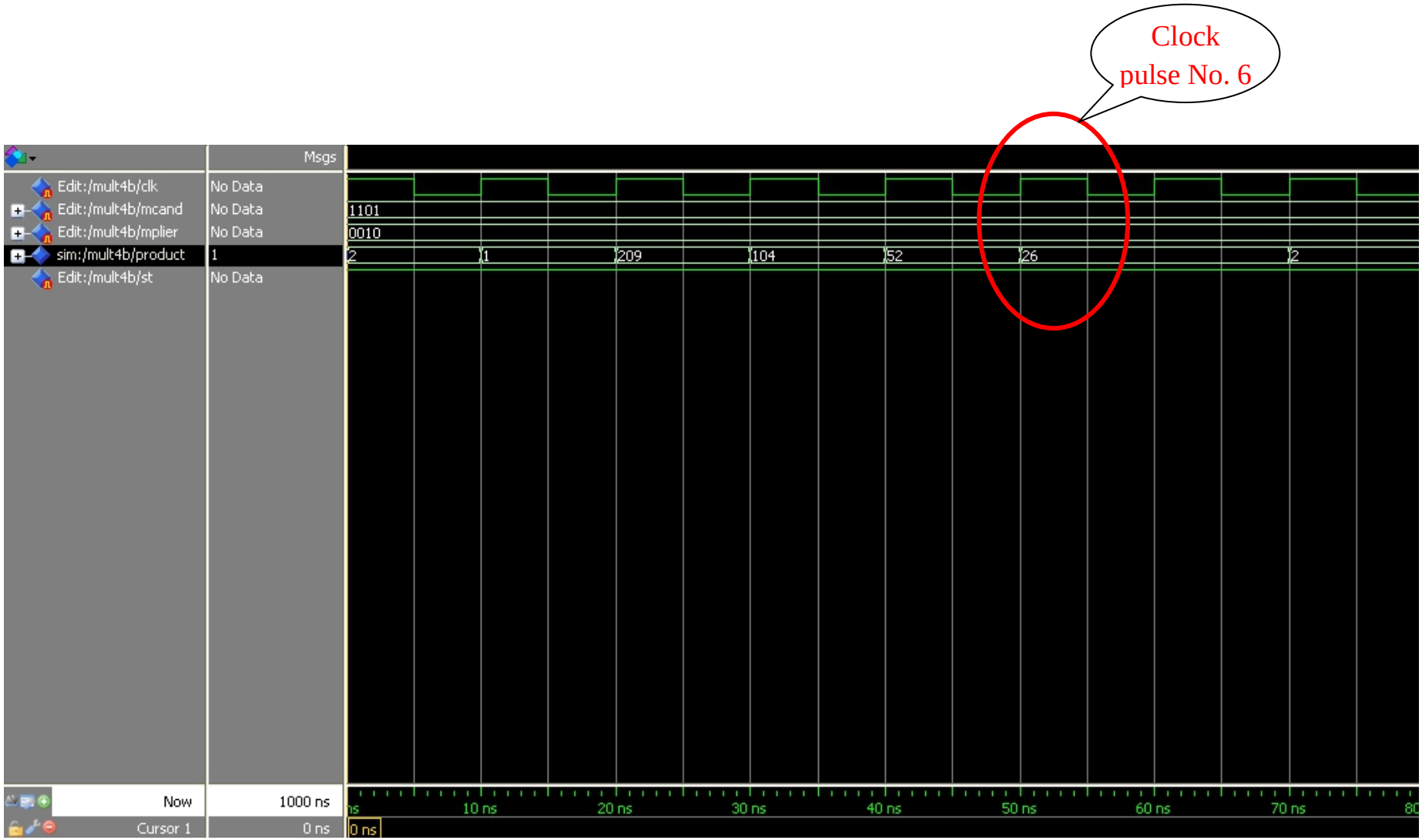


Figure 5.6 decimal serial multiplication (13× 2)

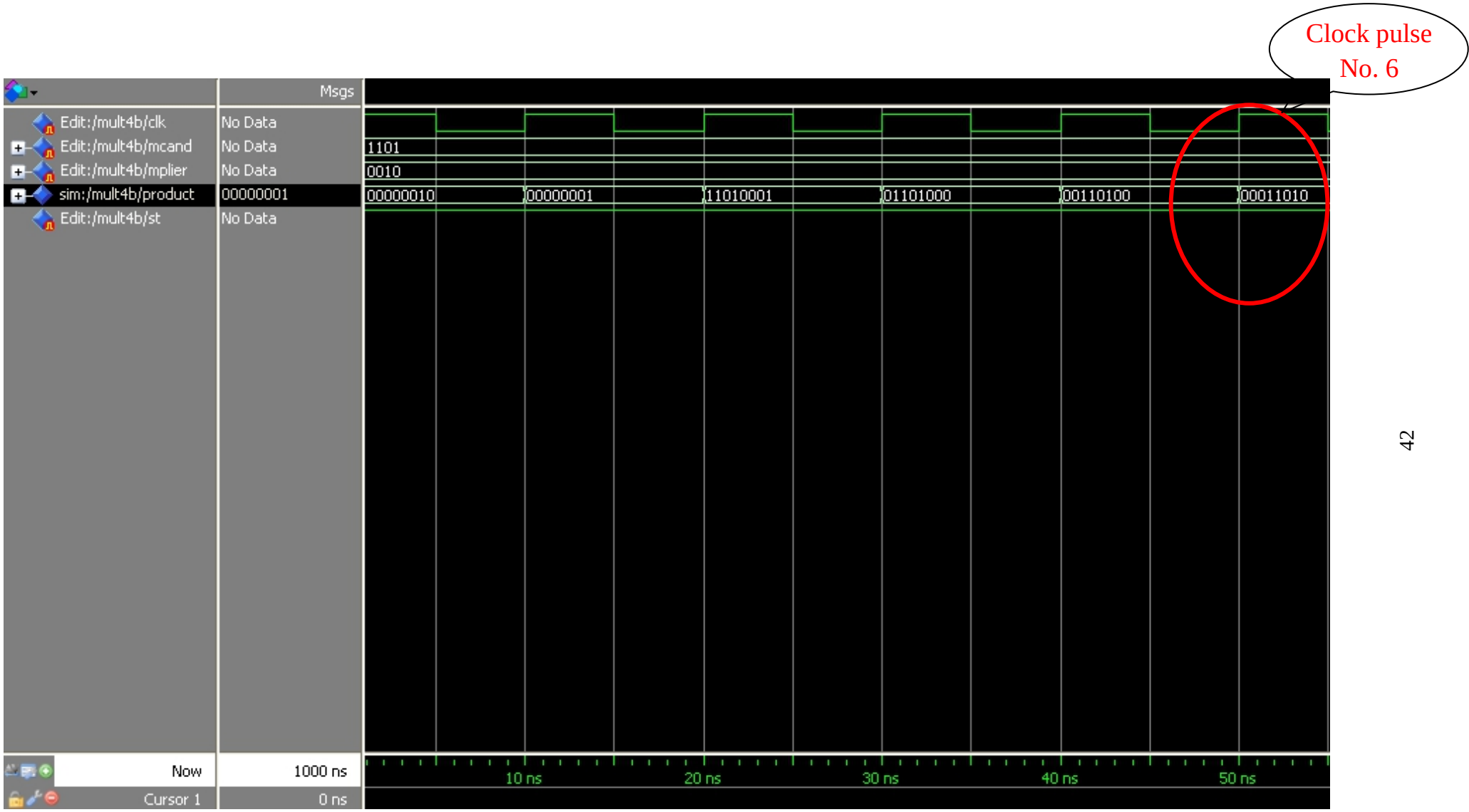


Figure 5.7 binary serial multiplication (13× 11)

CHAPTER VI

CONCLUTIONS

This project was presented two techniques to be used unsigned binary multiplier. The aim is to realize simple, low hardware architecture. The proposed multipliers are analyzed at the RTL level using Modelsim6.6d simulator. The performance of the two designs are evaluates terms of speed, logic utilization and power consumption.

According to the report generated by the Quartus II 11.0 soft ware and the RTL simulation result, we can conclude the following.

1. Array multiplier exhibits high speed performance while it consumes much power in comparison to the serial bit multiplier. Also the array multiplier as expected utilizes large number of logic gate in comparison the serial.
2. Several bit multiplier is running low bit it occupy a low chip area and consume of little bit power than the parallel multiplier
3. The array multiplier is suitable for the application which is need a fast processing at the expense of area. While the bit serial is suitable for low chip area. And the speed is not a critical factor.

6.1 Suggestion for future work

We suggest for future work the following:

1. Design and implementation of signed binary multiplier
2. Design and implementation of fixed width multiplier
3. FPGA based implementation of the proposed design

REFRENCE

[1] B. Nag arjun and poonam Sharma, “design and FPGA Implementation of a simple time efficient kom multiplier “IOSR journal of Electronic and communications Engineering 2014 –pp 35.39

[2] P. Meier, R. Rutenbar, and L.R. Carley, “Exploring Multiplier Architecture and Layout
For Low Power,” *IEEE Custom Integrated Circuits Conference* 1996, pp. 513-516.

[3] R. Fried, “Minimizing Energy Dissipation in High-Speed Multipliers,” *International*
Symposium on Low Power Electronics and Design, 1997

[4] B. Ackland, C.J. Nicol, “High Performance DSPs - What’s Hot and what’s Not?” *International*
Symposium on Low Power Electronics and Design, 1998

[5] C.J. Nicol, P. Larsson, “Low Power Multiplication for FIR Filtering,” *International Symposium*
On Low Power Electronics and Design, 1997

[6] E.d.Angel, "Low Power Digital Multiplication," in *Application Specific Processors*, E.E.
Swartzlander, ed., Kluwer Academic Publishers, Norwell, Mass, 1997.

[7] E. Musoll and J. Cortadella, “Low-Power Array Multipliers with Transition-Retaining

[8] P. Meier, R. Rutenbar, and L.R. Carley, “Exploring Multiplier Architecture and Layout For Low Power,” *IEEE Custom Integrated Circuits Conference* 1996.