A LOW POWER MULTIPLIER WITH SPURIOUS POWER SUPPRESSION TECHNIQUE

A PROJECT REPORT

Submitted by

BALASUBRAMANIAN R (71204105014)

KARTHIKEYAN C (71204105027)

SARAVANAN A (71204105047)

In partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

ELECTRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE-641006

ANNA UNIVERSITY: CHENNAI 600025

APRIL 2008

BONAFIDE CERTIFICATE

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report entitled **"A LOW POWER MULTIPLIER WITH SPURIOUS POWER SUPPRESSION TECHNIQUE"** is the bonafide work of

BALASUBRAMANIAN R - Register No. **71204105014**
KARTHIKEYAN C - Register No. **71204105027**
SARAVANAN A - Register No. **71204105047**

Who carried out the project work under my supervision.

Prof. K.Regupathy Subramanian
HEAD OF THE DEPARTMENT

Mrs. R.Mahalakshmi
SUPERVISOR
Senior Lecturer

Internal Examiner

External Examiner

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING
KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE 641 006

ABSTRACT

ABSTRACT

Multiplier represents a fundamental building block in all DSP task. A low power, high speed multiplier is essential for all DSP application in which multiplication are frequently used for key computations, such as Fast Fourier transform (FFT) , Discrete cosine transform(DCT), Quantization and Filtering. Our project deals with developing a highly efficient multiplier algorithm and implementing it in FPGA with VHDL as programming language.

A technique called 'Spurious Power Suppression Technique' is employed in the multiplication algorithm which helps in eliminating redundant computation thus saving unnecessary power consumption thereby increasing the speed of computation. This technique equipped with booth algorithm results in highly efficient multiplier algorithm.

The multiplier algorithm is programmed using VHDL. It is simulated using MODELSIM 6.0a and synthesized by XILINX ISE9.1i Web pack. It is implemented in a Xilinx FPGA device XC3S400. It is inferred from the power report that 50% reduction in power consumption is obtained with SPST equipped with booth multiplier compared with the booth multiplier without SPST.

ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

The completion of our project can be attributed to the combined efforts made by us and the contribution made in one form or the other by the individuals we hereby acknowledge.

We are highly privileged to thank **Dr. Joseph V Thanikal**, Principal, Kumaraguru College of Technology for allowing us to do this project.

We express our heart felt gratitude and thanks to the Dean / HOD of Electrical & Electronics Engineering, **Prof. K.Regupathy Subramanian,** for encouraging us and for being with us right from beginning of the project and guiding us at every step.

We wish to place on record our deep sense of gratitude and profound thanks to our guide **Mrs.R.Mahalakshmi**, Senior Lecturer, Electrical and Electronics Engineering Department, for her valuable guidance, constant encouragement, continuous support and co-operation rendered throughout the project.

We are also thankful to our teaching and non-teaching staffs of Electrical and Electronics Engineering department, for their kind help and encouragement.

Last but not least, we extend our sincere thanks to all our parents and friends who have contributed their ideas and encouraged us for completing the project.

CONTENTS

CONTENTS

TABLES&FIGURES

LIST OF TABLES

LIST OF FIGURES

SYMBOLS& ABBREVATIONS

LIST OF SYMBOLS AND ABBREVIATIONS

NO.	SYMBOLS	ABBREVIATIONS
01	ASIC	Application Specific Integrated Circuit
02	CLB	Configurable Logic Block
03	DCI	Digitally Controlled Impedance
04	DCM	Digital Clock Manager
05	DDR	Double Data Rate
06	DSP	Digital Signal Processing
07	FPGA	Field Programmable Gate Array
08	FFT	Fast Fourier Transform.
09	HDL	Hardware Descriptive Language
10	IOB	Input Output Block
11	I/O	Input / Output
12	LSB	Least Significant Bit
13	PCB	Printed Circuit Board
14	PLD	Programmable Logic Device
15	VHDL	VHSIC Hardware Descriptive Language
16	VLSI	Very Large Scale Integrated Circuit
17	SPST	Spurious Power Suppression Technique

CHAPTER- 1

1.INTRODUCTION

1.1 NEED FOR THE PROJECT

Multipliers play an important role in today's digital signal processing and various other applications. Multiplier represents a fundamental building block in all DSP applications. Multiplication is one of the most common operation performed in signal processing – convolution, IIR filtering, Fourier transform etc. Since multiplication dominates the execution time of most DSP algorithm, using a high speed multiplier is desirable.

FPGA are on the verge of revolutionizing digital signal processing in the manner that programmable digital signal processors did nearly two decades ago. Many front-end digital signal processing algorithms such as FFT, FIR (or) IIR , to name just a few, previously built with ASIC (or) PDSP are now most often replaced by FPGA.

FPGA have many features in common with ASIC, such as reduction in size, weight and power dissipation, higher throughput, better security against unauthorized copies, reduced device and inventory cost, and reduced board test cost, and claim advantages over ASIC, such as a reduction in development time(rapid prototyping), in-circuit reprogram ability, lower NRE cost, resulting in more economical design for solution requiring less than 1000 units. FPGA allow a variety of computer arithmetic implementation for the desired digital signal processor algorithm, because of the physical bit level programming architecture.

1.2 OBJECTIVE

The objective of this project is to develop a highly efficient multiplier algorithm and to implement a 16x16 bit multiplier in FPGA with VHDL as programming language.

1.3 ORGANIZATION OF THE REPORT

Chapter 1: Introduction to the project stating the need for this approach, objective and Methodology

Chapter 2: Multiplier – An overview about types of multiplier and its algorithm.

Chapter 3: SPST equipped booth multiplier (16x16 bit) – Gives a brief idea about the implementation of 16x16 bit multiplier with SPST equipped booth algorithm.

Chapter 4: Software implementation – this chapter gives an introduction about VHDL, VHDL programming of 16x16 bit multiplier, and simulating it in software named 'Modelsim 6.0a'.

Chapter 5: FPGA – Describes about the FPGA, its architecture, design flow and applications. Details the architecture and features of the FPGA device XC3S400 used.

Chapter 6: Gives the details about the hardware implementation. The synthesis report of the project is discussed.

Chapter 7: Conclusion – The project done and its results are discussed.

Appendices: Gives the features and the specification the FPGA device used.

1.4 METHODOLOGY

The sequence of work carried out is been listed below,

VHDL coding:	Describes the circuit structure
Simulation:	To check the functionality of the code
Synthesis:	RTL is turned into design implementation in terms of logic gates creates a net list
Mapping:	Fits the design into available resources in the tool
Place & Route:	Determining the position & interconnecting the sub blocks
Bit stream:	A binary file in terms of 0s and 1s to program FPGA
Hardware verification:	The VHDL coding for 16x16 bit multiplier is downloaded in the FPGA device and output is verified.

CHAPTER-2

2. MULTIPLIER – AN OVERVIEW

2.1 INTRODUCTION

Multiplication can be considered as a series of repeated additions. The number to be added is called multiplicand, the number of times it is added is called the multiplier and the result obtained is called the product. The basic operation involved in multiplication includes generating and accumulating (or) adding the partial products. Consequently to speed up the entire multiplication process these two major steps must be optimized. There are two types of multipliers

1. Array Multipliers and

2. Booth Multipliers.

2.2 ARRAY 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. 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 are required where N is the multiplier length.

2.2.1 4-Bit multiplication method

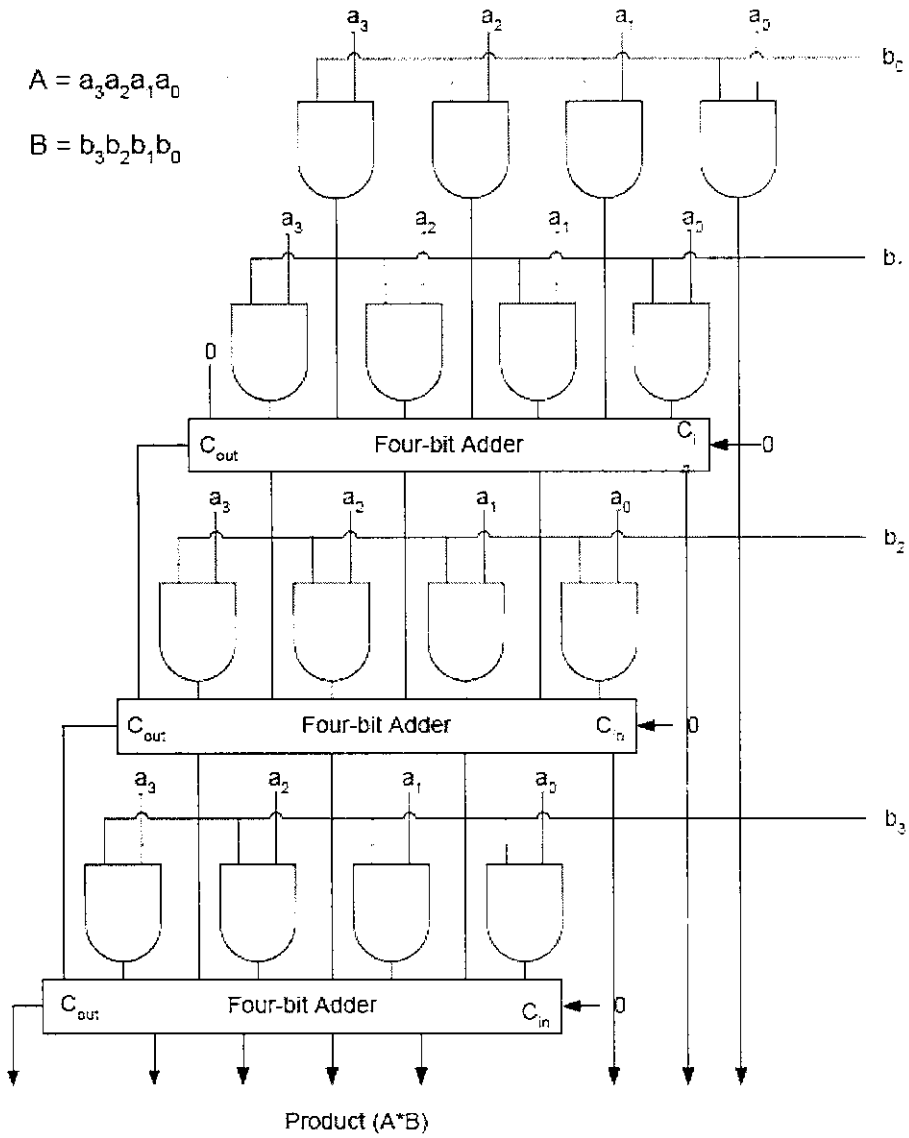


Fig 2.1 Array multiplier

Although the method is simple as it can be seen from this example, the addition is done serially as well as in parallel. To improve on the delay the CRAs are replaced with Carry Save Adders, in which every carry and sum signal is passed to the adders of the next stage. Final product is obtained in a final adder by any fast adder. In array multiplication we need to add, as many partial products as there are multiplier bits.

This arrangements is shown in the figure below

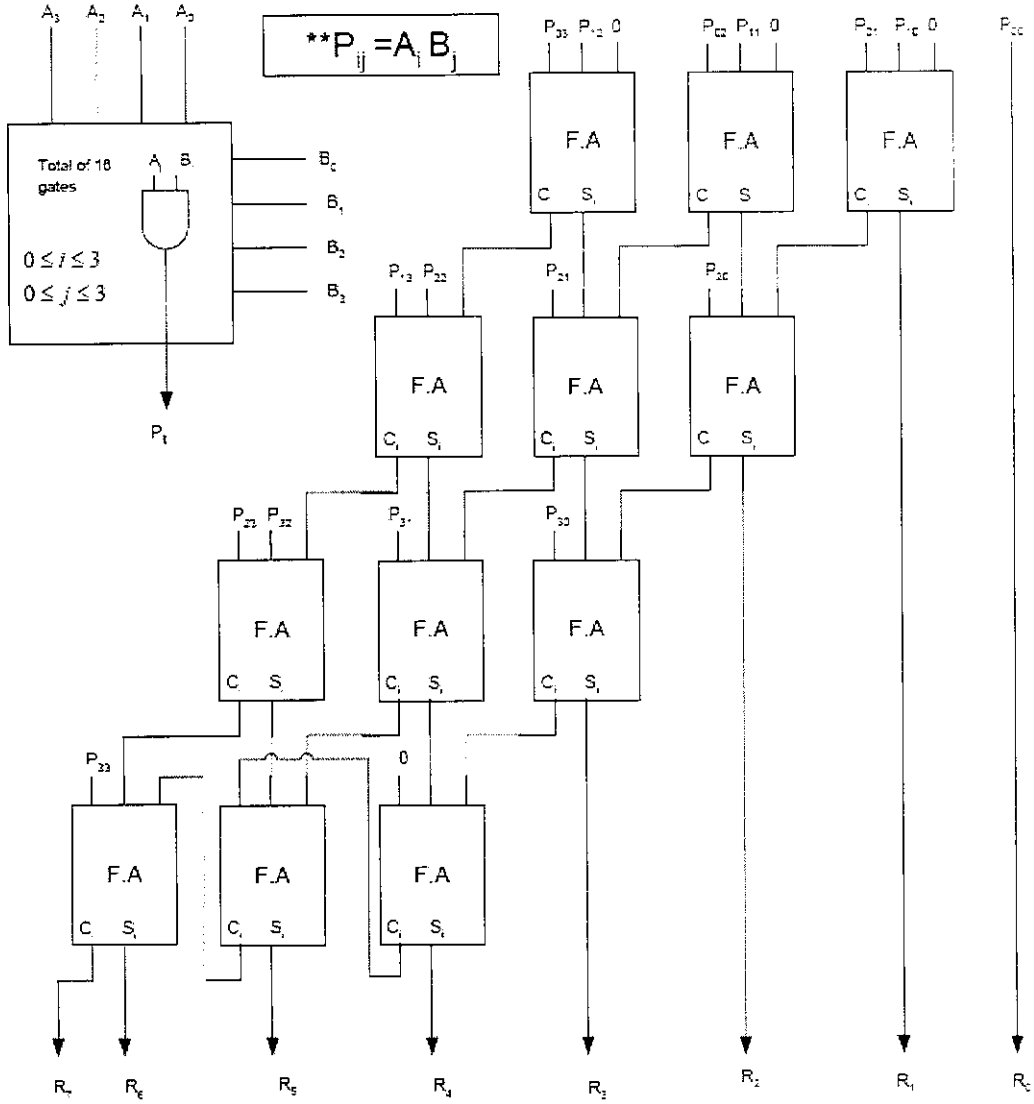


Fig 2.2 Array multiplier with carry and save adder.

2.2.2 Algorithm- array multiplier

STEP 1: START

STEP 2: STORE MULTIPLICAND IN ONE VARIABLE, MULTIPLIER IN ANOTHER VARIABLE AND DECLCARE ONE OUTPUT VARIABLE.

STEP 3: GET THE LSB BIT OF MULTIPLIER

IF LSB=1, STORE THE MULTIPLICAND IN ONE TEMPORARY VARIABLE ELSE TEMPORARY VARIABLE EQUALS TO ZERO.

STEP 4: GET THE NEXT BIT OF MULTIPLIER AND SAME OPERATION IS PERFORMED AS THE PREVIOUS STEP AND THE RESULT IS STORED IN ANOTHER VARIABLE.

STEP 5: THE 2^{ND} PARTIAL PRODUCT IS SHIFTED ONE BIT TO THE LEFT . AFTER GETTING TWO PARTIAL PRODUCTS FULL ADDER OPERATION IS PERFORMED ON IT AND THE RESULT IS STORED IN OUTPUT VARIABLE.

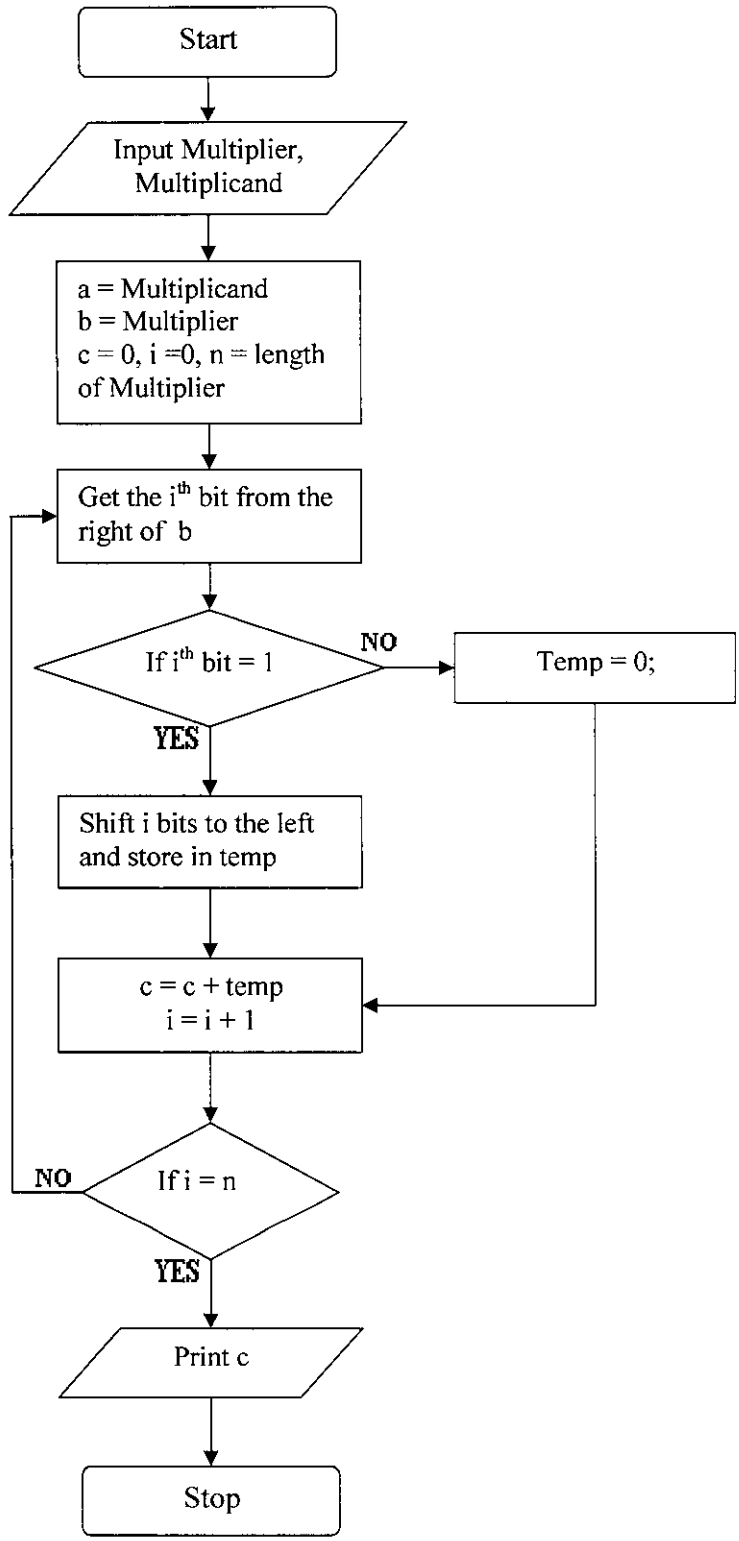
STEP 6: NOW GET NEXT BIT OF MULTIPLIER PERFORM THE MULTIPLICATION OPERATION AND STORE THE RESULT IN A TEMPORARY VARIABLE.

STEP 7: WE HAVE ONE PARTIAL PRODUCT IN OUTPUT VARIABLE AND ANOTHER IN A TEMPORARY VARIABLE NOW PERFORM THE FULL ADDER OPERATION AND STORE THE RESULT IN OUTPUT VARIABLE.

STEP 8: THIS OPERATION IS TILL N-BIT OF THE MULTIPLIER.

STEP 9:STOP.

2.2.3 Flowchart



2.2.4 Examples

```
                                          A7    A6    A5    A4    A3    A2    A1    A0
                                        X B7    B6    B5    B4    B3    B2    B1    B0
                                        ------------------------------------------------
                                          A7.B0 A6.B0 A5.B0 A4.B0 A3.B0 A2.B0 A1.B0 A0.B0
                                    +   A7.B1 A6.B1 A5.B1 A4.B1 A3.B0 A2.B0 A1.B0 A0.B0
                              +   A7.B2 A6.B2 A5.B2 A4.B2 A3.B0 A2.B0 A1.B0 A0.B0
                        +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
                  +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
            +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
      +   A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
  +  A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
+A7.B3 A6.B3 A5.B3 A4.B3 A3.B0 A2.B0 A1.B0 A0.B0
------------------------------------------------------------------------------------
P15   P14   P13   P12   P11   P10   P9    P8    P7    P6    P5    P4    P3    P2    P1    P0
```

The equation is: $P(m+n) = A(m)B(n) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_i b_j 2^{i-j}$.

$Y = Y_{n-1}\ Y_{n-2}\ \ldots\ldots\ldots\ldots\ldots Y_2\ Y_1\ Y_0$ Multiplicand

$X = X_{n-1}\ X_{n-2}\ \ldots\ldots\ldots\ldots\ldots X_2\ X_1\ X_0$ Multiplier

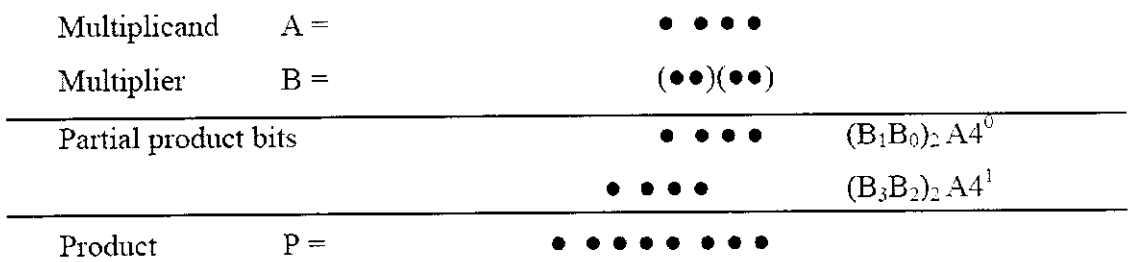
Example

```
    1101   4-bits
    1101   4-bits
    ----------------
    1101
   0000
  1101
 1101
----------------
10010101
```

2.3 BOOTH MULTIPLIER

It is a powerful algorithm for multiplication, which treats both positive and negative numbers uniformly. For the standard add-shift operation, each multiplier bit generates one multiple of the multiplicand to be added to the partial product. If the multiplier is very large, then a large number of multiplicands have to be added. In this case the delay of multiplier is determined mainly by the number of additions to be performed. If there is a way to reduce the number of the additions, the performance will get better. Booth algorithm is a method that will reduce the number of multiplicand multiples. For a given range of numbers to be represented, a higher representation radix leads to fewer digits. Since a k-bit binary number can be interpreted as K/2-digit radix-4 number, a K/3-digit radix-8 number, and so on, it can deal with more than one bit of the multiplier in each cycle by using ix multiplication.

This is shown for Radix-4 in the example below.



Radix-4 multiplication in dot notation.

Fig 2.3 Radix 4 multiplication

As shown in the figure above, if multiplication is done in radix-4, in each step, the partial product term needs to be formed and added to the cumulative partial product. Whereas in radix-2 multiplication, each row of dots in the partial products matrix represents 0 (or) a shifted version must be included and added.

Table-1 is used to convert a binary number to radix-4 number.

Initially, a 0 is placed to the right most bit of the multiplier. Then three bit of the multiplicand is recoded according to table below or according to the following equation:

$Z_i = -2X_{i+1} + X_i + X_{i-1}$

Example:

Multiplier is equal to 0 1 0 1 1 10

then a 0 is placed to the right most bit which gives 0 1 0 1 1 10 0

0 added

the 3 digits are selected at a time with overlapping left most bit as follows:

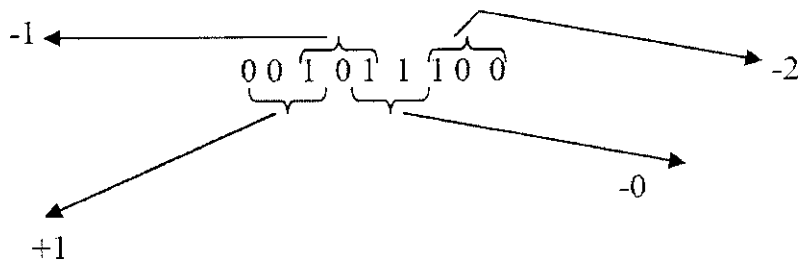


Table 2.1 Radix 4 booth recoding

X_{i+1}	X	X_{i-1}	Z_{i+2}
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 2.2 Multiplier recoding

0	0	0	+0*multiplicand
0	0	1	+1*multiplicand
0	1	0	+1*multiplicand
0	1	1	+2*multiplicand
1	0	0	-2*multiplicand
1	0	1	-1*multiplicand
1	1	0	-1*multiplicand
1	1	1	-0*multiplicand

Here -2 multiplicand is actually the 2's complement of the multiplicand with an equivalent left shift of one bit position. Also +2 multiplicand is the multiplicand shifted left one bit position which is equivalent to multiplying by 2.

To enter +2(or)-2 multiplicand into the adder, an (n+1)-bit adder is required. In this case the multiplicand is offset one bit to the left to enter into the adder while for the low order multiplicand position a 0 is added. Each time the partial product is shifted two bit position to the right and the sign is extended to the left.

During each add shift cycle , different version of the multiplicand are added to the new partial products depends on the equation derived from the bit pair recoding table above

2.3.1 Algorithm

STEP 1: START

STEP 2: STORE THE MULTIPLICAND IN 'A', STORE THE MULTIPLIER IN 'B' AND DECLARE ONE OUTPUT VARIABLE 'C'.

STEP 3: INITIALLY THE MULTIPLIER IS APPENDED WITH ZERO TO IT'S RIGHT.

STEP 4: THE MULTIPLIER IS PARTIONED INTO OVERLAPPING GROUP OF 3-BITS AND THE OPERATION FOR CORRESPONDING THREE BITS CAN BE KNOWN FROM THE BOOTH RECODING TABLE.

STEP 5: NOW THE FIRST RECODED VALUE IS TAKEN AND ITS CORRESPONDING OPERATION IS PERFORMED ON THE MULTIPLICAND AND THE RESULT IS STORED IN A VARIABLE 'PP1'.

STEP 6: NEXT RECODED VALUE IS TAKEN AND ITS RESULT IS STORED IN VARIABLE 'PP2'.

STEP 7: NOW FULL ADDER OPERATION IS PERFORMED ON THESE TWO PARTIAL PRODUCTS AND THE RESULT IS STORED IN THE OUTPUT VARIABLE 'C'.

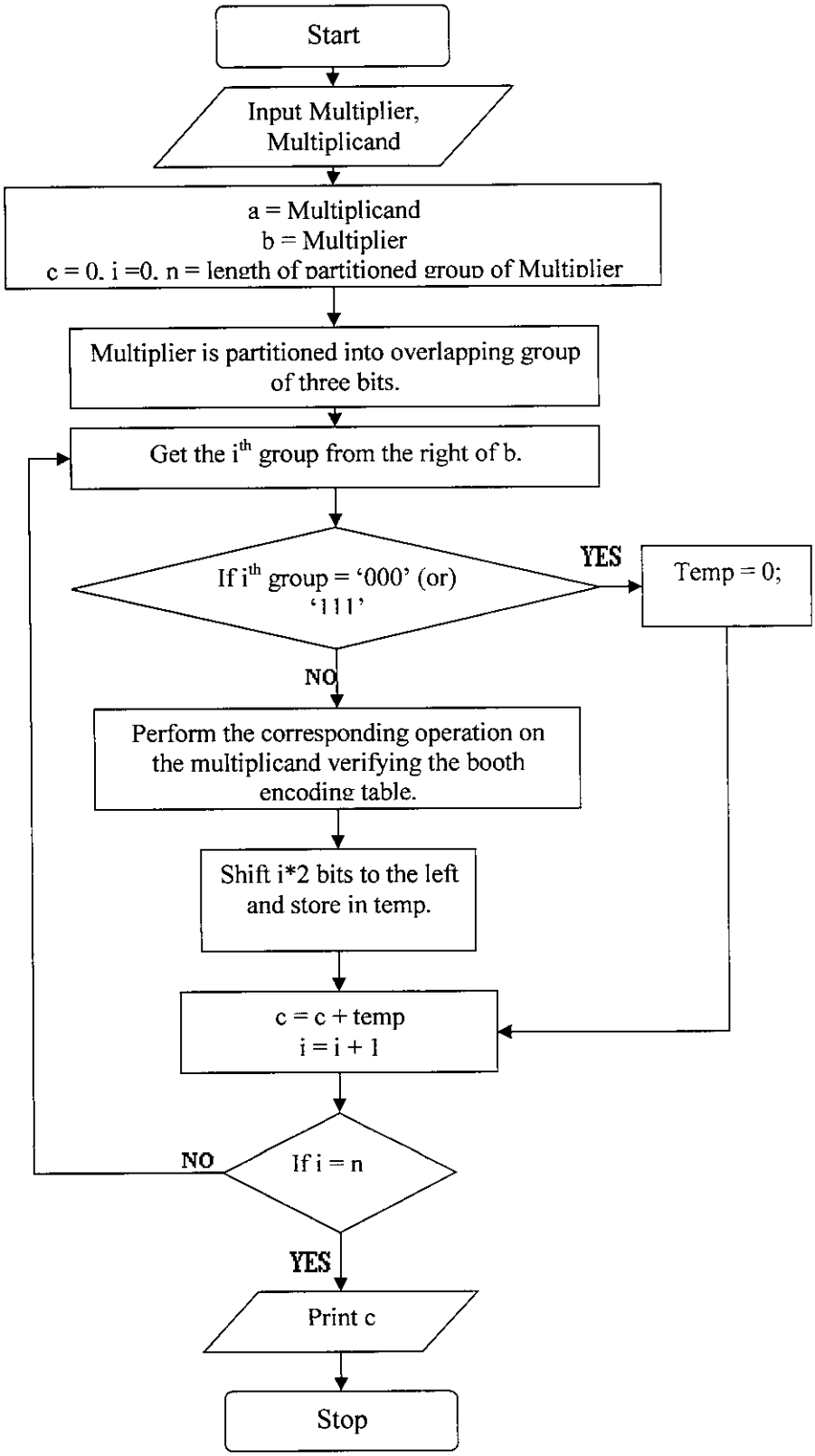
STEP 8: 'PP3' IS GENERATED AND FULL ADDER OPERATION IS PERFORMED WITH 'PP3' AND 'C' AND 'C'IS UPDATED WITH THE RECENT RESULT.

STEP 9: SIMILARLY OPERATIONS ARE PERFORMED WITH 'PP4', 'PP5', ETC UNTIL ALL RECODED VALUE.

STEP 10: FINAL RESULT IS OBTAINED IN 'C'.

STEP 11: STOP.

2.3.2 Flowchart



2.3.3 Examples

Example 1:

```
          000011      (+3)
      ×   011101 0    (+29)
          +2 -1 +1
      ------------
      000000000011
      1111111101
      00000110
      ------------
1 ←   000001010111    (+87)
```

Example 2:

```
          111101      (-3)
      ×   011101 0    (+29)
          +2 -1 +1
      ------------
      111111111101
      0000000011   ← 2s complement of multiplicand
      11111010
      ------------
1 ←   111110101001    (-87)
```

Example 3:

```
          111101      (-3)
      ×   100011 0    (-29)
          -2 +1 -1
      ------------
      000000000011
      1111111101   ← Shifted 2s complement
      00000110
      ------------
1 ←   000001010111    (+87)
```

CHAPTER- 3

3. SPST EQUIPPED BOOTH MULTIPLIER(16X16 bit)

3.1 BOOTH MULTIPLIER

3.1.1. Introduction

The multiplication consists of two factors one is named the multiplicand, the other one multiplier. In general multiplication takes place by adding the multiplicands after shifting depending on the position of the positive correlated bit of the multiplier. This leads to a defined number of partial products which equals the number of bits of the multiplier. As a result logic has to be designed for as many partial products as bits of the multiplier. Using the modified Booth recoding technique leads to the advantage that the number of partial products to be added is reduced to one half of the original word length. As a consequence the delay and area occupation shrinks substantially. A reduced number of partial products minimize the number of additions to consolidate the result.

The realization of the multiplier can be divided into three sections. The first section calculates the partial products to be added. These partial products are reduced in a second stage to two final bit vectors and in a third step the final addition of those two bit vectors is realized. For the calculation of the partial products the multiplier is partitioned into three-bit-groups that overlap by one bit and are recoded in the range of {-2, -1, 0, 1, 2}. The first group consists of the two LSBs of the multiplier and '0'. The following groups are constructed by the next two consecutive bits of the multiplier plus the MSB of the previous group. Whereas the order of significance of the bits in the groups remains the same as in the multiplier.

Booth multiplier is an efficient multiplier for multiplication of larger bit-length. Booth algorithm is a method that will reduce the number of multiplicand multiples. Number of partial products determines the performance of a multiplier and this multiplier reduces the number of partial product to a great extend. Compared with array multiplier booth multiplier is found to be more efficient. SPST equipped with booth multiplier makes it highly efficient.

Table 3.1 Booth Encoding Table

Modified Booth recoding	
Partial Product Selection Table	
Multiplier Bits	Recoded Bits
000	0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	- 2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	0

As a result we receive the fixed number of partial products as mentioned. In the case of a 32 x 32 bit multiplication 16 partial products are produced. Depending on the result of the recoding the multiplicand affects addition process positive, negative, once, twice or will not be added. The recoding result will change the multiplicand as follows:

- 0: all bits of the multiplicand change to ´0´
- + multiplicand: the multiplicand remains the same
- +2 x multiplicand: the multiplicand is shifted to the left
- -2 x multiplicand: the multiplicand is shifted to the left. the bits are inverted and one bit is added to the LSB
- -1 x multiplicand: the bits are inverted and one bit is added to the LSB

This multiplicand modification adds one bit to the original word length of the multiplicand. The new multiplicand has now 33 instead of 32 bits. In the case of not shifting the new multiplicand's 32 bit corresponds to the multiplicand's 31 bit.

In most of the DSP applications involving fast Fourier transform (FFT), discrete cosine transform (DCT), quantization, filtering deals with 16x16 bit multipliers. This chapter deals with implementation of 16x16 bit multiplier with booth algorithm equipped with SPST.

3.1.2. Advantages of Booth multiplier

1. High speed.
2. Cuts the number of additions in half.
3. Both for signed and unsigned operands.

3.1.3 Partial Product Generator

Partial product generator of a 16x16 bit multiplier first converts the binary number to radix-4 number (i.e. partitioning the multiplier to overlapping group of three bits). This partitioned value performs its operation on the multiplicand based on booth's algorithm. Before starting partitioning a zero will be appended to the right of the multiplier.

The operation performed on the multiplicand for partial product formation when the partitioned value is 011 is shifting the multiplicand one bit to the right. When it is 100 2's complement operation on the multiplicand and shifting the multiplicand one bit to the right is performed. For 110 (or) 101 2's complement operation is performed on multiplicand. When the partitioned value is 001 (or) 010 no operation is performed on the multiplicand.

E.g.

0010101011001001 X 100 = 11010101001101110

(16 bit input) (Encoded value) (Output)

3.1.4 Full adder

After the generation of two partial products the next operation is carried out in the full adder block. Before performing the operation the second partial product is shifted two bit to the left. During the gate level implementation of the full adder the sum

operation is performed using an XOR gate and the carry operation is performed using AND gate. The carry generated will be propagated to the next bit.

As the number of partial products generated is less the number of full adders needed also get reduced thus saving the increase in area.

e.g.

(+) 1101010100110111 0(first partial product)

1110101010011011 1(second partial product)

10001111111101001010(output)

3.2 SPURIOUS POWER SUPPRESSION TECHNIQUE

SPST stands for "spurious power suppression technique". By this technique unneccessary power consumption by the multiplier is saved. This SPST operation is implemented before the full adder operation is performed. Before the two partial products entering into the full adder the SPST block checks the two partial products one after the other whether it carry a value (or) it an zero. If the first partial product is found to be zero the second partial product is entered as output. In the either case first partial is entered as output. Thus the unneccessary transitions are avoided. Therefore power consumption is reduced.

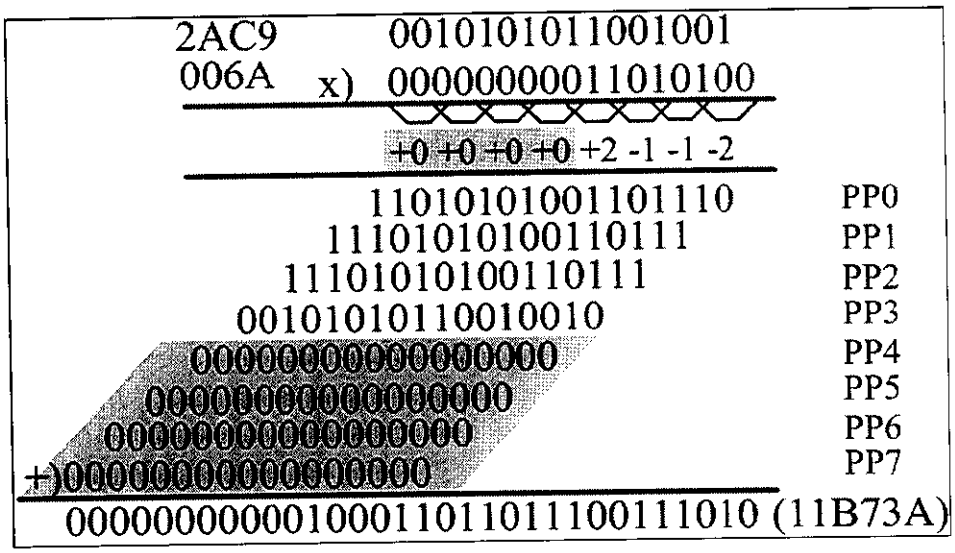


Fig 3.1 SPST Equipped Booth Multiplication

3.2.1 Algorithm (16x16 bit booth multiplier with SPST)

STEP 1: START

STEP 2: STORE THE MULTIPLICAND IN THE VARIABLE 'A', MULTIPLIER IN THE VARIABLE 'B' AND DECLARE AN OUTPUT VARIABLE 'C'.

STEP 3: APPEND A ZERO TO THE RIGHT OF THE MULTIPLIER.

STEP 4: PARTITION THE MULTIPLIER TO OVERLAPPING GROUP OF 3-BITS.

STEP 5: TAKE THE FIRST PATITIONED VALUE AND PERFORM THE CORRESPONDING OPERATION ON THE MULTIPLICAND AND STORE THE RESULT IN THE VARIABLE 'PP1'.

STEP 6: GENERATE THE OUTPUT FOR THE NEXT PARTITIONED VALUE AND STORE IT IN VARIABLE 'PP2'.

STEP 7: NOW IF 'PP1' IS FOUND TO BE ZERO STORE 'PP2' IN 'C'
ELSE IF 'PP2' IS FOUND TO BE ZERO STORE 'PP1' IN 'C'
IN NEITHER OF TWO CASE GOTO NEXT STEP.

STEP 8: GET TWO PARTIAL PRODUCTS 'PP1' AND 'PP2' AND PERFORM THE FULL ADDER OPERATION AND THE RESULT IS STORED IN 'C'.

STEP 9: BY TAKING THE NEXT PARTITIONED VALUE NEXT PARTIAL PRODUCT 'PP3' IS OBTAINED.

STEP 10: FULL ADDER OPERATION IS PERFORMED WITH 'PP3' AND 'C' AND THE RESULT IS STORED IN 'C'.

STEP 11: THIS OPERATION IS PERFORMED TILL ALL PARTITIONED VALUES TAKEN AND 'C' IS UPDATED WITH THE NEW RESULT.

STEP 12: STOP.

3.2.2 Block Representation Of Spst Equipped Booth Multiplier(16X16 bit)

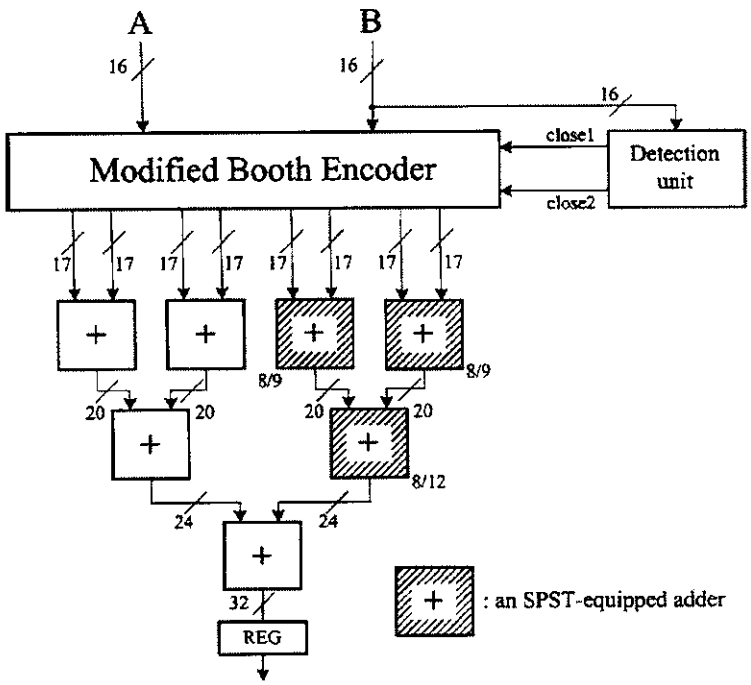
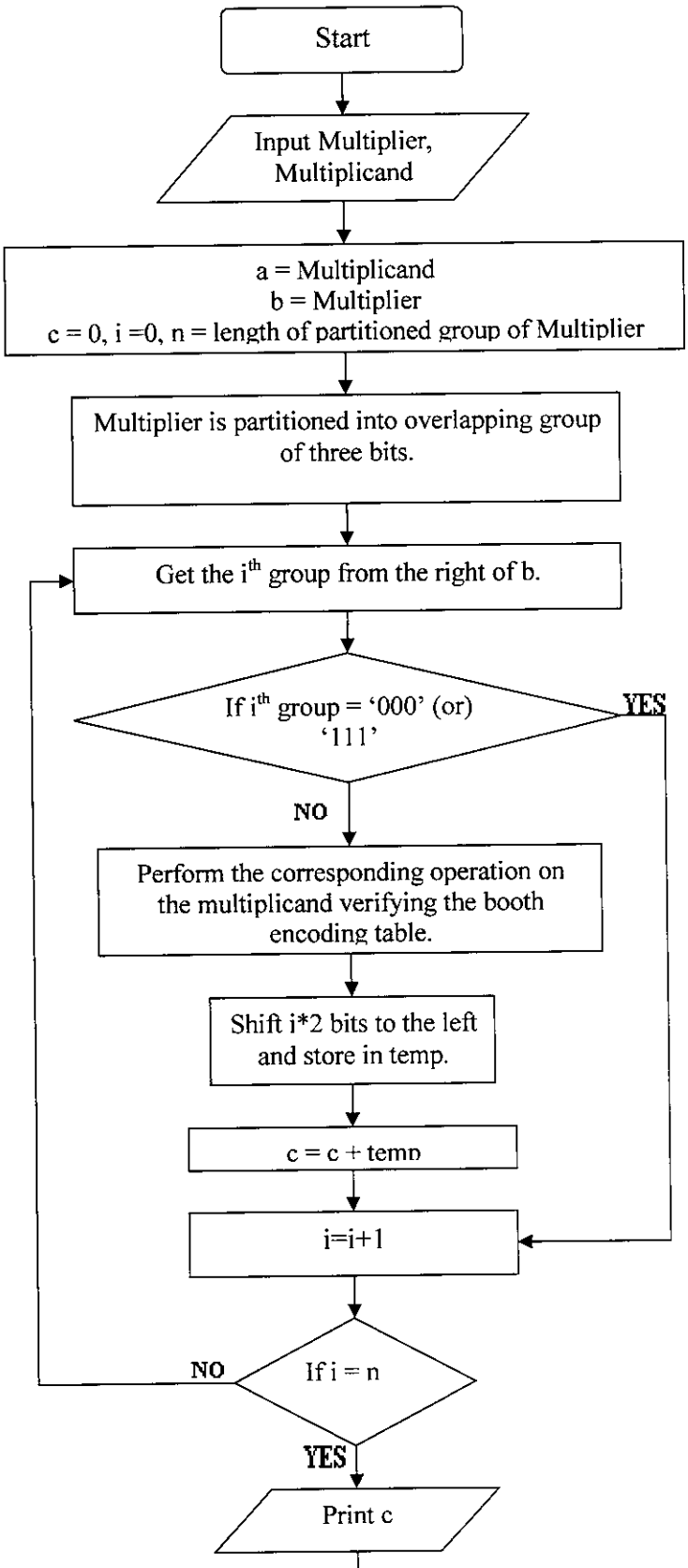


Fig 3.2 Block Representation of SPST Equipped Booth Multiplier(16x16)

3.2.3 Flowchart



3.4MULTIPLIER APPLICATIONS:

1. Mobile telephony.

    -speech and channel coding.

    -voice and data processing.

2.Personal computer.

    -data storage and retrival.

    -multimedia.

    -error correction.

3.Medical electronics.

    -digital X-ray.

    -EEG analysers.

4. Automotive.

    -digital audio.

    -digital radio.

CHAPTER- 4

4. SOFTWARE IMLEMENTATION

4.1 INTRODUCTION

In the first half of the 1990s, the electronics industry experienced an explosion in the demand for personal computers, cellular telephones and high-speed data communication devices. Vying for market share, vendors built products with increasingly greater functionality, higher performance, lower cost, lower power consumption and smaller dimensions. To do this vendor created highly integrated complex system with fewer ICs and less printed circuit board (PCB) area.

4.2 WHAT IS VHDL?

VERY HIGH SPEED INTEGRATED CIRCUIT HARDWARE DESCRIPTION LANGUAGE

VHDL is a programming language that has been designed and optimized for describing the behavior of hardware digital circuits and systems. It combines the features of a simulation modeling language, a design entry language, a test language, and a net list language. As a simulation modeling language, includes many feature appropriate for describing the behavior of electronic components ranging from simple logic gates to complete microprocessor and custom chips.

Design entry using VHDL is very much like software design using a software programming language like Pascal, C and C++. VHDL includes features useful for the structured design techniques and offer a rich set of control and data representation features. Unlike the other programming languages VHDL provides features allowing concurrent events to be described. This is important because the hardware being described using VHDL is inherently concurrent in its operation. VHDL follows IEEE standard 1076.

4.3 FEATURES OF VHDL

VHDL is an advantage rendered to every design engineer in the electronic industry to keep pace with the productivity of competitors. Equivalent designs described with the schematics or Boolean equations at transfer level can require several months of work by one person.

4.4 ADVANTAGES

- VHDL supports the development environment for digital development and various methods Top-down, Bottom-up or any mix.
- It is possible to VHDL to change technology using automatic tools.
- VHDL supports modifiability and hence it is easy to be read and structured.
- The language supports block diagram, reusable components, error management and verification.
- The language also supports concurrent and sequential language constructions.
- VHDL models of commercial IC standard components can now be bought, which is a great advantage when it comes to verifying entire circuit boards.
- The code for a VHDL component can be verified functionally in a simulator. It executes the codes with input signals and produces a signal diagram and error message on the basis of the components. Verification of the design is also possible.
- One of the great advantages of designing in VHDL is that the designer can concentrate on function, (i.e.). Implement the requirement specification and does not need to devote time and energy to technology specific factors, which do not affect function.

4.5 VHDL COMPONENTS

Components are a general concept in VHDL and can be a complete design or a small part of a system. It consist of two main parts:

4.5.1 Entity

It describes the input and output. Each input/output signal in an entity declaration is referred to as a port, which is analogous to a pin in a schematic symbol. A port is a data object and it can assign values and used in expressions. The set of ports defined for an entity is referred to as port declaration. Each port declared must have a name, a direction and a data type.

4.5.2 Architecture

Architecture defines a body for a component entity. An architecture body specifies a behavior between inputs and outputs.

4.5.3 Behavioral Description

It is referred as high level description because of the resemblance to high level programming languages. Instead of specifying the structure or net list of a circuit, a set of statement are specified, which when executed in sequence, model the function or behavior of the entity. The advantage to the high level description language is that a clear focus on the gate level implementation of a design is not needed. Instead an effort on accurately modeling its function is necessary.

4.5.4 Structural Description

It consists of VHDL net lists. These net list are very much like schematic net list-components are instantiated and connected together with signal. To instantiate the components are to be placed in a hierarchical design. An instantiation is therefore either an act of placing a component or an instance of a component.

4.6 VHDL CODING

```
--------------------------------------------------------------------------
-----------------------ARRAY 4 MULTIPLIER-------------------------------
--------------------------------------------------------------------------
```

Coding:

```
library ieee;
use ieee.std_logic_1164.all;

entity array4 is

port(
      clk : in std_logic;
      y   : in std_logic_vector(0 to 3);
      x   : in std_logic_vector(0 to 3);
      p   : out std_logic_vector(7 downto 0)

     );
end array4;

architecture df  of  array4 is component m2

port(clk,cin,x,y : in std_logic;
         s,cout      : out std_logic);

end component;

component m0

port(clk,x,y : in std_logic;
     xy      : out std_logic);

end component;

component m4

port(clk,a,b : in std_logic;
      s,c    : out std_logic);

end component;


signal pp1,pp2,pp3,pp4,pp6,pp7,pp8,pp9 : std_logic;
signalpc0,pc1,pc2,pc3,pc4,pc5,pc6,pc7,pc8,pc9,pc10,pc11,pc12,pc13,pc14:
std_logic;
signal w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16 : std_logic;
signal z : std_logic:='0';
begin


u0:m0 port map(clk,x(3),y(3),p(0));
u1:m0 port map(clk,x(2),y(3),w2);
u2:m0 port map(clk,x(1),y(3),w3);
```

```
u3:m0 port map(clk,x(0),y(3),w4);
u4:m0 port map(clk,x(3),y(2),w5);
u5:m0 port map(clk,x(2),y(2),w6);
u6:m0 port map(clk,x(1),y(2),w7);
u7:m0 port map(clk,x(0),y(2),w8);
u8:m4 port map(clk,w2,w5,p(1),pc0);
u9:m2 port map(clk,w3,w6,pc0,pp1,pc1);
u10:m2 port map(clk,w4,w7,pc1,pp2,pc2);
u11:m2 port map(clk,w8,z,pc2,pp3,pc3);
u12:m4 port map(clk,z,pc3,pp4,pc4);
u13:m0 port map(clk,x(3),y(1),w9);
u14:m0 port map(clk,x(2),y(1),w10);
u15:m0 port map(clk,x(1),y(1),w11);
u16:m0 port map(clk,x(0),y(1),w12);
u17:m4 port map(clk,pp1,w9,p(2),pc5);
u18:m2 port map(clk,w10,pp2,pc5,pp6,pc6);
u19:m2 port map(clk,w11,pp3,pc6,pp7,pc7);
u20:m2 port map(clk,w12,pp4,pc7,pp8,pc8);
u21:m4 port map(clk,z,pc8,pp9,pc9);
u22:m0 port map(clk,x(3),y(0),w13);
u23:m0 port map(clk,x(2),y(0),w14);
u24:m0 port map(clk,x(1),y(0),w15);
u25:m0 port map(clk,x(0),y(0),w16);
u26:m4 port map(clk,pp6,w13,p(3),pc10);
u27:m2 port map(clk,pp7,w14,pc10,p(4),pc11);
u28:m2 port map(clk,pp8,w15,pc11,p(5),pc12);
u29:m2 port map(clk,pp9,w16,pc12,p(6),pc13);
u30:m4 port map(clk,z,pc13,p(7),pc14);


end df;
```

Component 1:

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

entity m0 is
port(clk,x,y:in std_logic;
          xy:out std_logic);

end m0;
architecture df of m0 is

begin
process(clk,x,y)
begin
if(clk='1')and(clk'EVENT) then
xy<=x and y;
end if;
end process;
end df;
```

Component 2:

```
library ieee;
use ieee.std_logic_1164.all;

entity m2 is
port(clk,cin,x,y :in std_logic;
s,cout:out std_logic);

end m2;

architecture df of m2 is

begin
process(clk,x,y,cin)
begin
if(clk='1')and(clk'EVENT) then
s<=x xor y xor cin;
cout<=(x and y) or (cin and x) or (cin and y);
end if;
end process;
end df;
```

Component 3:

```
library ieee;
use ieee.std_logic_1164.all;

entity m4 is
port(clk,a,b:in std_logic;
     s,c:out std_logic);

end m4;

architecture df of m4 is
begin

process(clk,a,b)

begin
if(clk='1')and(clk'EVENT) then
s<= a xor b;
c<=a and b;
end if;
end process;
end df;
```

4.7 SIMULATION RESULTS

SIMULATION RESULT USING "MODELSIM 6.0a"

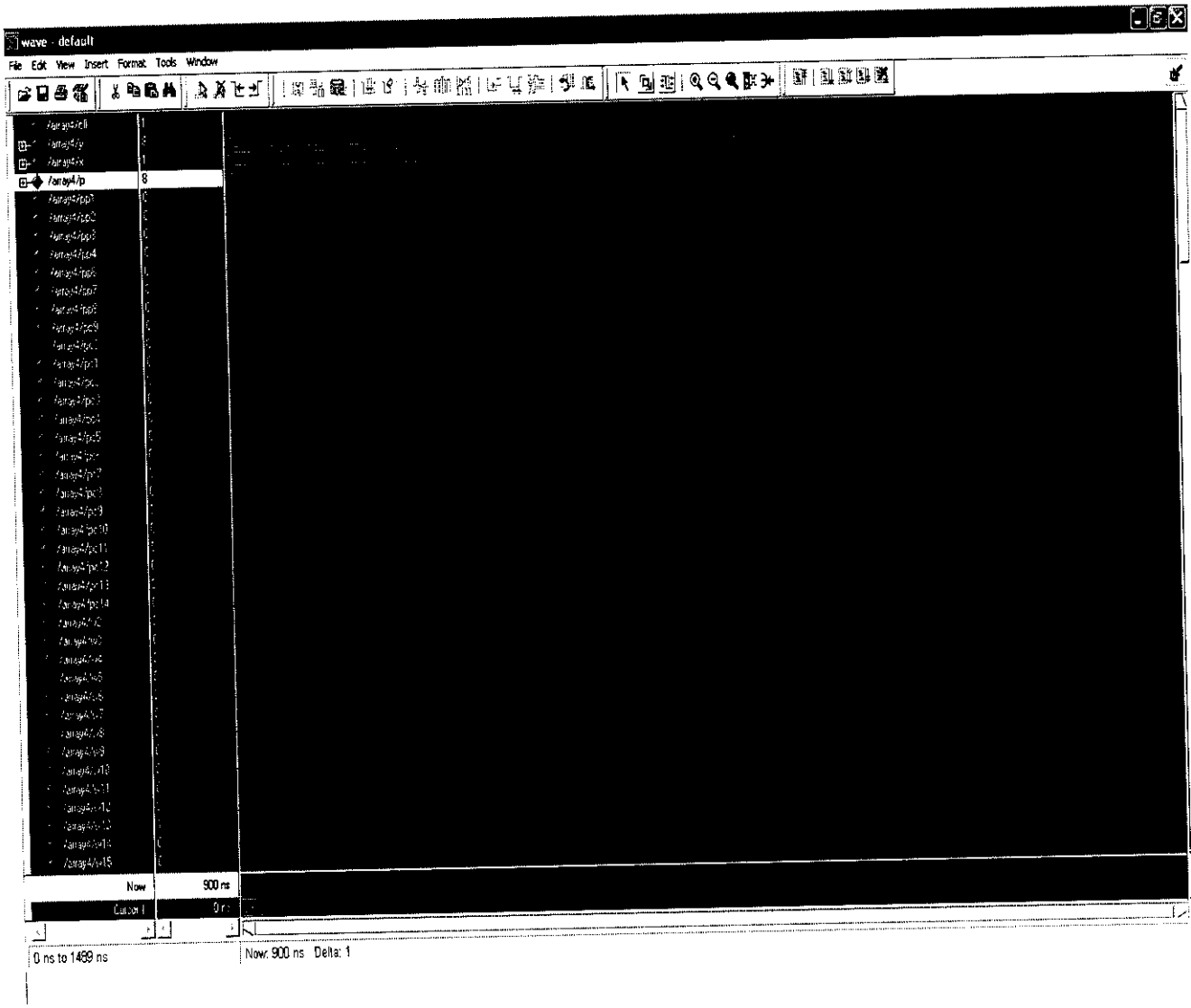


Fig4.1 SIMULATION RESULT OF 4X4 ARRAY MULTIPLIER

4x4 booth multiplier

Coding:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.math1.all;


entity modifiedbooth1 is
    Port ( input1 : in std_logic_vector(3 downto 0);
           input2 : in std_logic_vector(3 downto 0);
           output : out std_logic_vector(7 downto 0);
           clk : in std_logic);
end modifiedbooth1;

architecture Behavioral of modifiedbooth1 is

begin
process(clk,input1,input2)
variable temp_p1,temp_p2,temp_p5 : std_logic_vector(4 downto 0);
--variable temp_p3,temp_p4 : std_logic_vector(17 downto 0);
variable temp_output : std_logic_vector(9 downto 0);
begin

if(clk='1')and(clk'EVENT) then

temp_output:="0000000000";
temp_p2(3 downto 0):=input2;        --input2
temp_p2(4):=temp_p2(3);              --sig ext
temp_p1(3 downto 0):=input1;          --input1
temp_p1(4):=temp_p1(3);              --sig ext
temp_output(4 downto 0):=ppgen(temp_p2(3 downto 0),temp_p1(1
downto 0)&'0');  --ist ppgen
temp_p5:=ppgen(temp_p2(3 downto 0),temp_p1(3 downto 1));        --
2nd ppgen
temp_output(7 downto 2):=ispctadd(temp_output(6 downto
2),temp_p5(4 downto 0),'0');
temp_output(7):=temp_output(6);
output<=temp_output(7 downto 0);
end if;
end process;
end Behavioral;
```

Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package math1 is

function ppgen(pp : in std_logic_vector(3 downto 0); code : in
std_logic_vector(2 downto 0)) return std_logic_vector;
function  add(op1,op2  :  in  std_logic_vector(4  downto  0);cin:  in
std_logic) return std_logic_vector;
function ispctadd(op1,op2 : in std_logic_vector(4 downto 0);cin: in
std_logic)
return std_logic_vector;
  function     fulladder(op1,op2,cin:     in     std_logic)     return
std_logic_vector;
end math1;


package body math1 is


  function ppgen   (pp : in std_logic_vector(3 downto 0);code : in
std_logic_vector(2 downto 0) ) return std_logic_vector is
    variable temp1 : std_logic_vector(4 downto 0);
    variable output : std_logic_vector(4 downto 0);
  begin
    temp1:="00000";
    if((code = "001")or(code = "010")) then      -- +1
         temp1(3 downto 0):=pp;
    elsif(code = "011") then        --+2
          temp1(4 downto 1):=pp;
    elsif(code = "100") then       ---2
          temp1(3 downto 0):=pp;
          temp1(4 downto 0):= not temp1(4 downto 0);
          temp1(4 downto 0):=temp1(4 downto 0) + "1";
          temp1(4 downto 1):=temp1(3 downto 0);
          temp1(0):='0';
    elsif((code = "101") or (code = "110")) then ---1

          temp1(3 downto 0):=pp;
          temp1(4 downto 0):= not temp1(4 downto 0);
          temp1(4 downto 0):=temp1(4 downto 0) + "1";

    end if;
    output:=temp1;
    return output;
  end ppgen;

function fulladder(op1,op2,cin: in std_logic) return std_logic_vector
is
      variable output : std_logic_vector(1 downto 0);
      begin
      output:="00";
```

```
     output(0):=op1 xor op2 xor cin;
      output(1):=(op1 and op2)or(op1 and cin)or(op2 and cin);
      return output;
      end fulladder;

function  add(op1,op2  :  in  std_logic_vector(4  downto  0);cin:  in
std_logic) return std_logic_vector is
      variable output : std_logic_vector(5 downto 0);
      begin
      output(1 downto 0):=fulladder(op1(0),op2(0),cin);
      output(2 downto 1):=fulladder(op1(1),op2(1),output(1));
      output(3 downto 2):=fulladder(op1(2),op2(2),output(2));
      output(4 downto 3):=fulladder(op1(3),op2(3),output(3));
      output(5 downto 4):=fulladder(op1(4),op2(4),output(4));

      return output;
      end add;

function  ispctadd(op1,op2  :  in  std_logic_vector(4  downto  0);cin:  in
std_logic) return std_logic_vector is
      variable output : std_logic_vector(5 downto 0);
      variable temp : std_logic_vector(4 downto 0);
      begin
      temp:=op1;
      temp(4):=temp(2);        --sig ext
      temp(3):=temp(2);
      if((op1="00000") or (op2="00000"))then
          if(op1="00000") then
          output(4 downto 0):=op2;
          else
          output(4 downto 0):=temp;
          end if;
      else
            output:=add(temp,op2,cin);
      end if;

      return output;
      end ispctadd;


end math1;
```

SIMULATION RESULT USING "MODELSIM 6.0a"

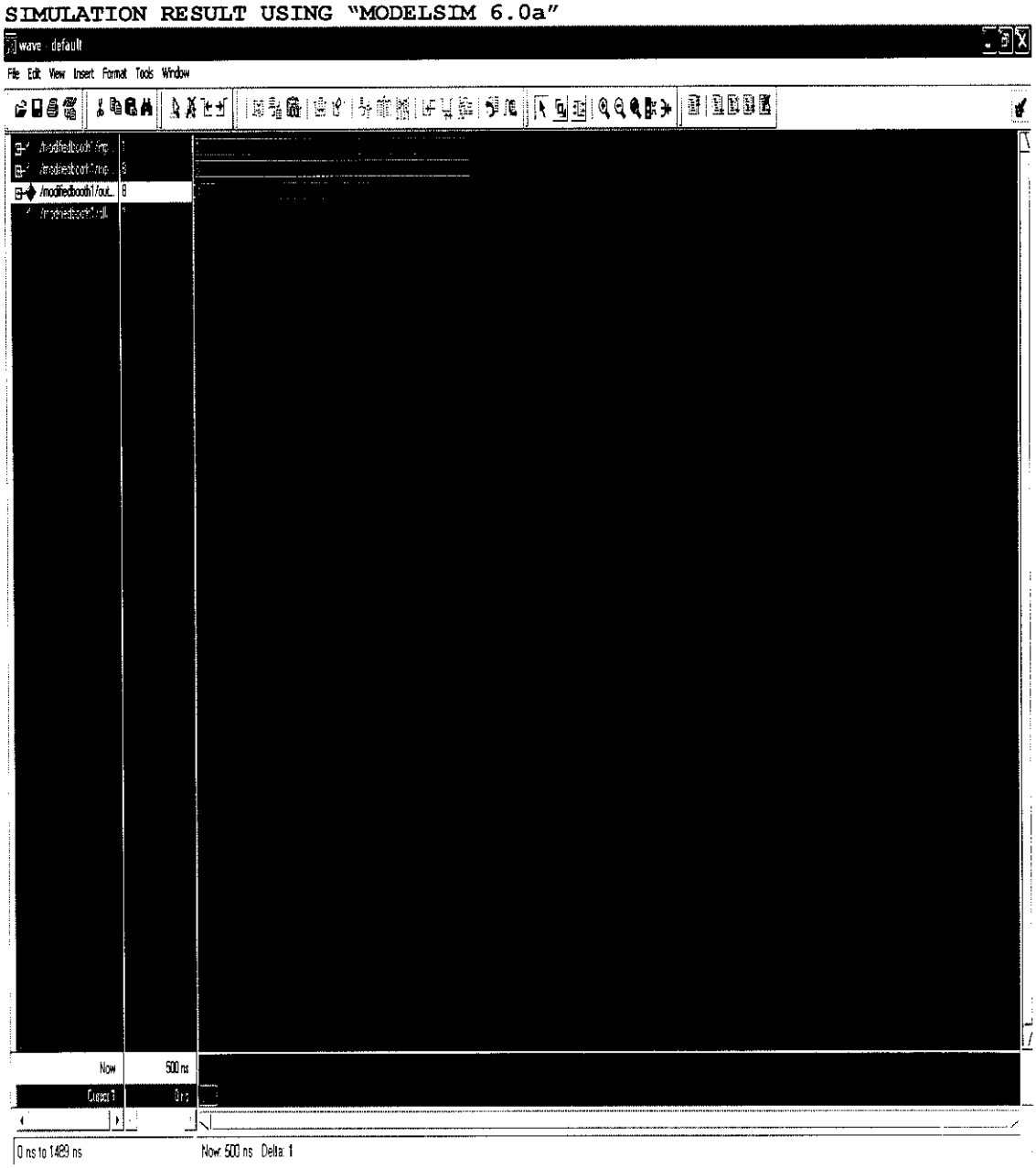


Fig 4.2 SIMULATION RESULT OF 4X4 BOOTH MULTIPLIER

---------- **16x16 bit SPST equipped booth multiplier**----------

---

coding

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.math.all;


entity modifiedbooth is
   Port ( input1 : in std_logic_vector(15 downto 0);
          input2 : in std_logic_vector(15 downto 0);
          output : out std_logic_vector(31 downto 0);
          clk : in std_logic);
end modifiedbooth;

architecture Behavioral of modifiedbooth is

begin
process(clk,input1,input2)
variable temp_p1,temp_p2,temp_p5 : std_logic_vector(16 downto 0);
--variable temp_p3,temp_p4 : std_logic_vector(17 downto 0);
variable temp_output : std_logic_vector(33 downto 0);
begin

if(clk='1')and(clk'EVENT) then
temp_output:="00000000000000000000000000000000";

temp_p2(15 downto 0):=input2;      --input2

temp_p2(16):=temp_p2(15);          --sig ext

temp_p1(15 downto 0):=input1;       --input1

temp_p1(16):=temp_p1(15);          --sig ext

temp_output(16 downto 0):=ppgen(temp_p2(15 downto 0),temp_p1(1 downto 0)&'0');     --ist
ppgen

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(3 downto 1));    --2nd ppgen

temp_output(19 downto 2):=spstadd(temp_output(18 downto 2),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(5 downto 3));    --3rd ppgen

temp_output(21 downto 4):=spstadd(temp_output(20 downto 4),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(7 downto 5));    --4th ppgen
```

```
temp_output(23 downto 6):=spstadd(temp_output(22 downto 6),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(9 downto 7));    --5th ppgen

temp_output(25 downto 8):=spstadd(temp_output(24 downto 8),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(11 downto 9));          --6th ppgen

temp_output(27 downto 10):=spstadd(temp_output(26 downto 10),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(13 downto 11));         --7th ppgen

temp_output(29 downto 12):=spstadd(temp_output(28 downto 12),temp_p5(16 downto 0),'0');

temp_p5:=ppgen(temp_p2(15 downto 0),temp_p1(15 downto 13));         --8th ppgen

temp_output(31 downto 14):=spstadd(temp_output(30 downto 14),temp_p5(16 downto 0),'0');


 temp_output(31):=temp_output(30);

 output<=temp_output(31 downto 0);

  end if;

   end process;
end Behavioral;
```

package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package math is

  function ppgen(pp : in std_logic_vector(15 downto 0); code : in std_logic_vector(2 downto 0))
return std_logic_vector;
  function add(op1,op2 : in std_logic_vector(16 downto 0);cin: in std_logic) return
std_logic_vector;
function spstadd(op1,op2 : in std_logic_vector(16 downto 0);cin: in std_logic) return
std_logic_vector;
function fulladder(op1,op2,cin: in std_logic) return std_logic_vector;
end math;


package body math is
```

```
function ppgen  (pp : in std_logic_vector(15 downto 0);code : in      std_logic_vector(2 downto
0) ) return std_logic_vector is
variable temp1 : std_logic_vector(16 downto 0);
variable output : std_logic_vector(16 downto 0);
begin
temp1:="00000000000000000";
if((code = "001")or(code = "010")) then  -- +1
temp1(15 downto 0):=pp;
elsif(code = "011") then                    --+2
temp1(16 downto 1):=pp;
elsif(code = "100") then              ---2
temp1(15 downto 0):=pp;
temp1(16 downto 0):= not temp1(16 downto 0);
temp1(16 downto 0):=temp1(16 downto 0) + "1";
temp1(16 downto 1):=temp1(15 downto 0);
temp1(0):='0';
elsif((code = "101") or (code = "110")) then ---1
temp1(15 downto 0):=pp;
temp1(16 downto 0):= not temp1(16 downto 0);
temp1(16 downto 0):=temp1(16 downto 0) + "1";

end if;
output:=temp1;
return output;
end ppgen;

function fulladder(op1,op2,cin: in std_logic) return std_logic_vector is
variable output : std_logic_vector(1 downto 0);
begin
output:="00";
output(0):=op1 xor op2 xor cin;
output(1):=(op1 and op2)or(op1 and cin)or(op2 and cin);
return output;
end fulladder;

function add(op1,op2 : in std_logic_vector(16 downto 0);cin: in std_logic) return
std_logic_vector is
variable output : std_logic_vector(17 downto 0);
begin
output(1 downto 0):=fulladder(op1(0),op2(0),cin);
output(2 downto 1):=fulladder(op1(1),op2(1),output(1));
output(3 downto 2):=fulladder(op1(2),op2(2),output(2));
output(4 downto 3):=fulladder(op1(3),op2(3),output(3));
output(5 downto 4):=fulladder(op1(4),op2(4),output(4));
output(6 downto 5):=fulladder(op1(5),op2(5),output(5));
output(7 downto 6):=fulladder(op1(6),op2(6),output(6));
output(8 downto 7):=fulladder(op1(7),op2(7),output(7));
output(9 downto 8):=fulladder(op1(8),op2(8),output(8));
output(10 downto 9):=fulladder(op1(9),op2(9),output(9));
output(11 downto 10):=fulladder(op1(10),op2(10),output(10));
```

```
output(12 downto 11):=fulladder(op1(11),op2(11),output(11));
output(13 downto 12):=fulladder(op1(12),op2(12),output(12));
output(14 downto 13):=fulladder(op1(13),op2(13),output(13));
output(15 downto 14):=fulladder(op1(14),op2(14),output(14));
output(16 downto 15):=fulladder(op1(15),op2(15),output(15));
output(17 downto 16):=fulladder(op1(16),op2(16),output(16));
return output;
end add;

function spstadd(op1,op2 : in std_logic_vector(16 downto 0);cin: in std_logic) return
std_logic_vector is
variable output : std_logic_vector(17 downto 0);
variable temp : std_logic_vector(16 downto 0);
begin
temp:=op1;
temp(16):=temp(14);                --sig ext
temp(15):=temp(14);
if((op1="00000000000000000") or (op2="00000000000000000"))then
if(op1="00000000000000000") then
output(16 downto 0):=op2;
else
output(16 downto 0):=temp;
end if;
else
output:=add(temp,op2,cin);
end if;

return output;
end spstadd;
end math;
```

SIMULATION RESULT USING 'MODELSIM 6.0a'

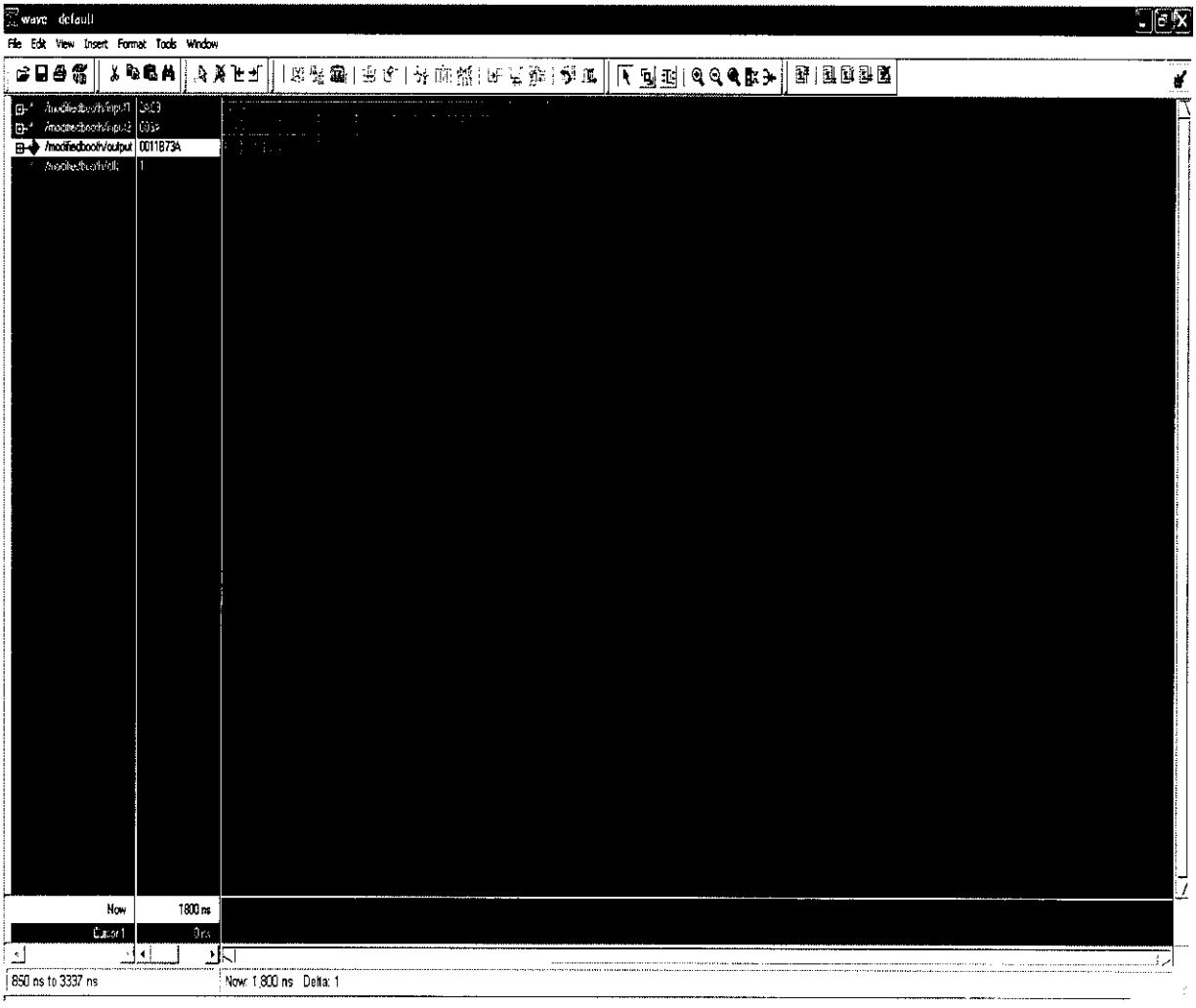


Fig 4.3 SIMULATION RESULT IS 16X16 SPST EQUIPPED BOOTH MULTIPLIER

CHAPTER- 5

5. FIELD PROGRAMMABLE GATE ARRAY

5.1 INTRODUCTION

A Field Programmable Gate Array (FPGA) is a semiconductor device containing electrically *programmable* logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories. An FPGA is similar to a PLD, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates.

A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform any logical function needed.

5.2 ARCHITECTURE

The typical basic architecture consists of an array of configurable logic blocks (CLBs) and routing channels. Multiple I/O pads may fit into the height of one row or the width of one column in the array. Generally, all the routing channels have the same width (number of wires). The basic components of FPGA are

- Configurable Logic Block (CLBs)
- Interconnect
- Input Output Block(IOBs)
- Memory
- Complete Clock Management

An application circuit must be mapped into an FPGA with adequate resources. The typical FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop, as shown

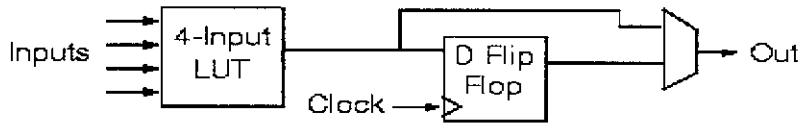


Fig 5.1. Logic block

There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed. For this example architecture, the locations of the FPGA logic block pins are shown

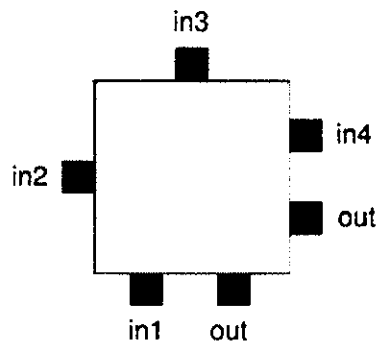


Fig 5.2. Logic Block Pin Locations

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block. Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

5.3 FPGA MANUFACTURERS

Field programmable devices are manufactured by several companies and a few of them are listed

- Altera Corporation
- Xilinx Inc
- Atmel Corporation
- QuickLogic Corporation
- Actel Corporation
- Aeroflex Inc
- Lattice semiconductor corporation
- Leopard Logic Inc

Xilinx first formed in 1984, are the major manufacturers of CPLDs and SRAM based FPGAs. Xilinx leads FPGA industry by its products

- Virtex series
- Spartan series.

The PWM control IC designed is programmed into Spartan II XC2S200 PQ208

5.4 FPGA DESIGN FLOW

The standard design flow for Spartan generation FPGAs include following three major steps

- Design Entry and Synthesis
- Design Implementation
- Design Verification

Design Description:

Designer describes design functionality either by using schema editors or by using one of the various Hardware Description Languages (HDLs) like Verilog or VHDL. The control IC is designed in VHDL code. A standard simulator which supports VHDL is used to verify the correctness of the design. Data can be analyzed in a number of ways. Waveform display and tabular display are generally used to trace down the errors in VHDL code. Functional simulation is done by creating testbench. Testbench is an environment, where a design is checked by applying stimuli and monitoring responses.

Synthesis:

Next step is to synthesize the design. The goal of VHDL synthesis is to create a design that implements the required functionality and matches the designer's constraints an area, speed or power. The synthesis tool reads the VHDL design and reports syntax errors and synthesis errors. If there are no syntax errors, the designer can synthesize the design and map to target technology. If any changes are to be made in the VHDL description, then the description needs to be simulated again and the output is validated for correctness. The synthesizer produces an output netlist in the target technology and a number of report files. From the netlist, it can be determined that the design is reasonable or not. The reports such as timing report and device utilization summary helps to determine the quality of the design.

Design Implementation

Implementation includes Partition, Place and route. Place and Route translates the logic design into physical design, maps the components used in the design into specific elements, places them and routes the interconnection between them. Place and Route also helps to do timing analysis. After all cells are placed and routed, the output of

the place and route tools consists of data files that can be used to implement the chip. The output of design implementation phase is bit-stream file. These files describe all the connections needed to make the FPGA macro cells implement the functionality required.

Design Verification

Bit stream file is fed to a simulator which simulates the design functionality and reports errors in desired behavior of the design. Timing tools are used to determine maximum clock frequency of the design. Now the design is loading onto the target FPGA device and testing is done in real environment.

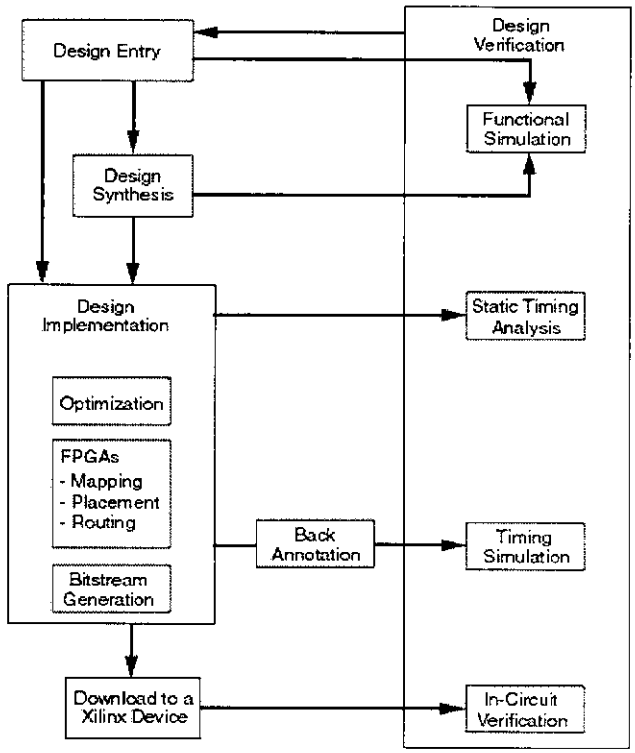


Fig 5.3 Design Flow

5.5 ARCHITECTURE OF XC3S400

Block Diagram

The 1.2V Spartan™-3 families of Field-Programmable Gate Arrays is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The eight-member family offers densities ranging from 50,000 to five million system gates. The Spartan-3 family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.

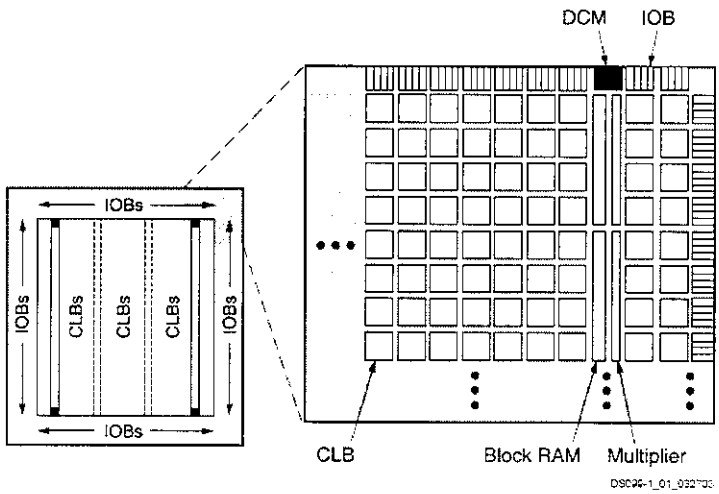


Fig 5.4 Basic Architecture of XC3S400

5.5.1 Architecture overview

The Spartan-3 family architecture consists of five fundamental programmable functional elements:

Configurable Logic Blocks (CLBs):It contains RAM-based Look-Up Tables (LUTs) to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logical functions as well as to store data.

Input/Output Blocks (IOBs):It controls the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. Twenty-three different signal standards, including six high-performance differential standards, are available. Double Data-Rate (DDR) registers are included. The Digitally Controlled Impedance (DCI) feature provides automatic on-chip terminations, simplifying board designs.

Block RAM: provides data storage in the form of 18-Kbit dual-port blocks.

Multiplier blocks: accept two 18-bit binary numbers as inputs and calculate the product.

Digital Clock Manager (DCM) block: provide self calibrating, fully digital solution for distributing, delaying, multiplying, dividing, and phase shifting clock signals.

5.6 APPLICATIONS OF FPGA

Applications of FPGAs include DSP, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation and a growing range of other areas. FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SOC). Due to their programmable nature, FPGAs are an ideal fit for many different markets such as

- Aerospace & Defense
- Automotive
- Broadcast
- Consumer.
- Industrial/Scientific/Medical
- Storage & Server
- Wireless Communications
- Wired Communications

CHAPTER- 6

6. FPGA BASED SPST EQUIPPED BOOTH MULTIPLIER

6.1 RTL SCHEMATIC

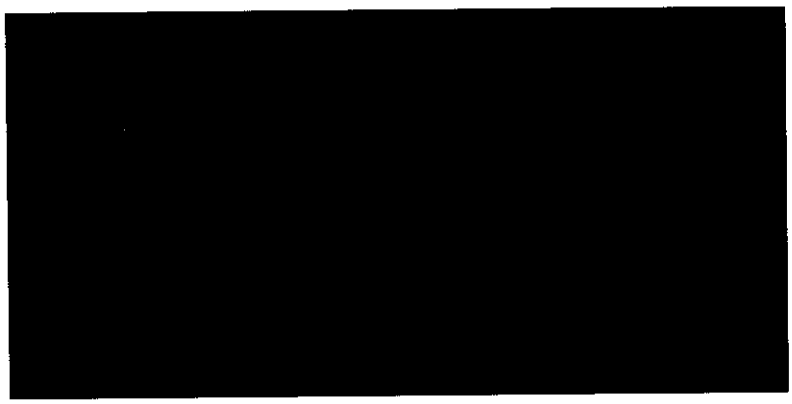


Fig 6.1 Top Level Schematic

6.2 HARDWARE IMPLEMENTATION

Xilinx ISE 8.2i Webpack was used for Synthesis and Implementation.

6.2.1 SYNTHESIS REPORT

Synthesis report details about the cell usage and device utilization summary.

HDL SYNTHESIS REPORT

ADDERS/SUBTRACTORS 17-BIT ADDER	: 1
MULTIPLEXERS	: 136
XORS	: 119
1-BIT 4-TO-1 MULTIPLEXER XORS	: 136
1-BIT XOR2	: 7
1-BIT XOR3	: 112

CELL USAGE:

#	BELS	: 971
#	GND	: 1
#	INV	: 15
#	LUT1	: 1
#	LUT2	: 1
#	LUT3	: 198
#	LUT4	: 559
#	MUXCY	: 16
#	MUXF5	: 163
#	VCC	: 1
#	XORCY	: 16
#	IO BUFFERS	: 64
#	IBUF	: 32
#	OBUF	: 32

DEVICE UTILIZATION SUMMARY:

SELECTED DEVICE: 3S400PQ208-4

NUMBER OF SLICES:	428 OUT OF	3584	11%
NUMBER OF 4 INPUT LUTS:	774 OUT OF	7168	10%
NUMBER OF IOS:	64		
NUMBER OF BONDED IOBS:	64 OUT OF	141	45%

6.2.2 ASSIGNING PACAKAGE PINS

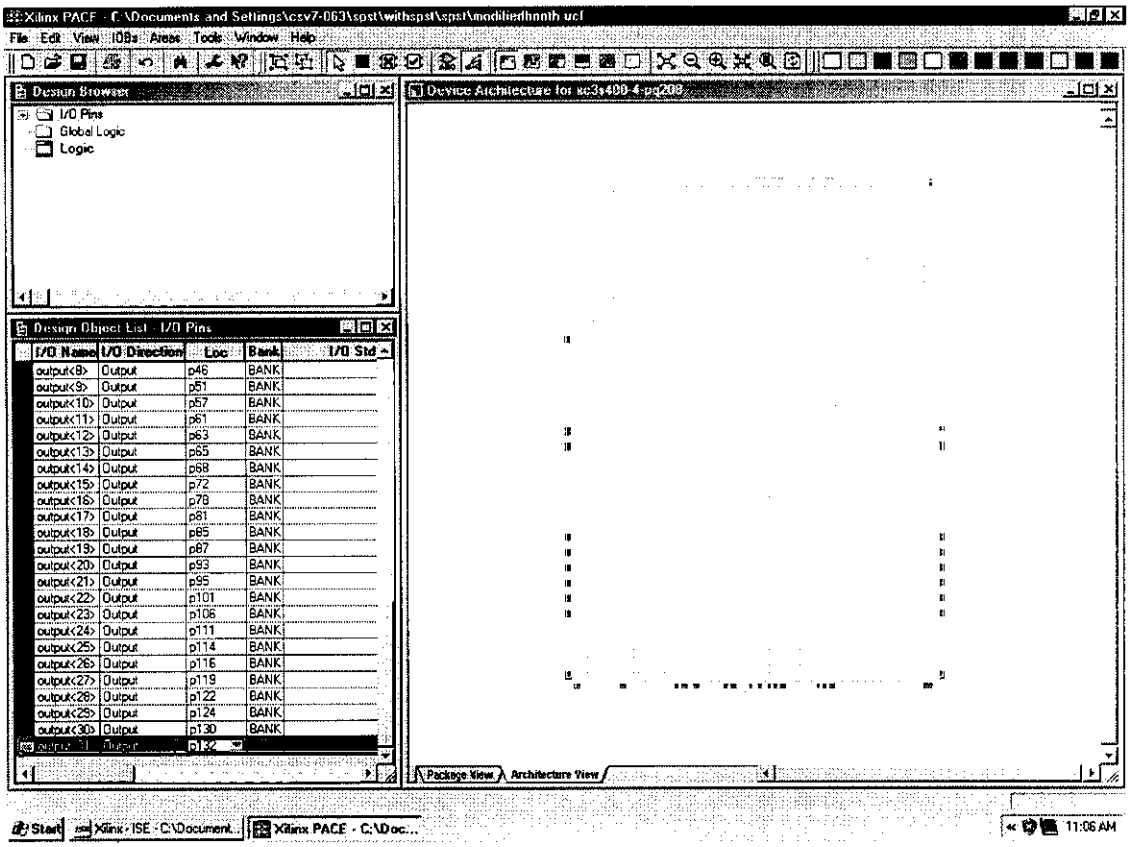


fig 6.2 Assigning Package Pins

6.2.3 PROGRAMMING OF FPGA DEVICE

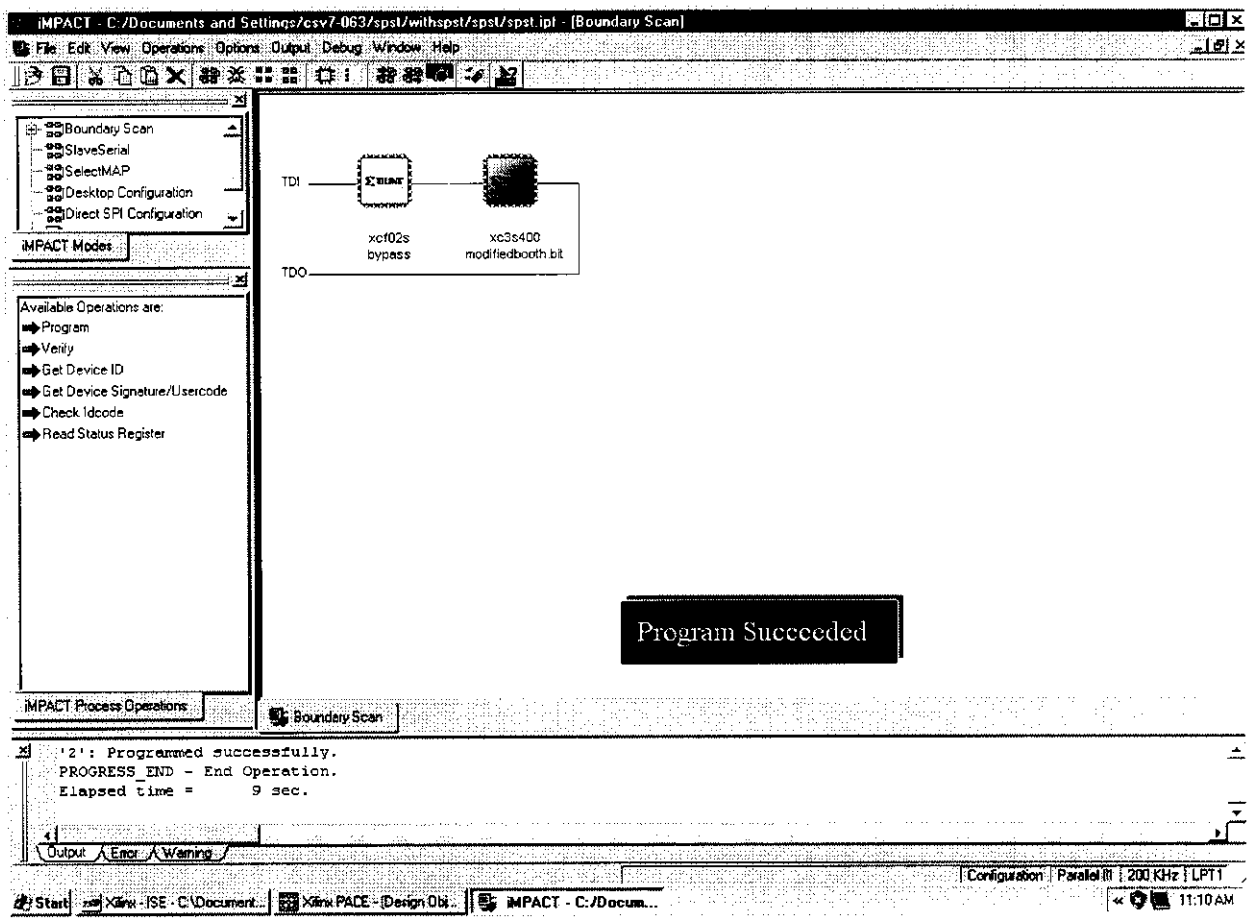


fig 6.3 Programming of FPGA Device

6.2.4 POWER REPORT OF BOOTH MULTIPLIER (WITHOUT SPST)

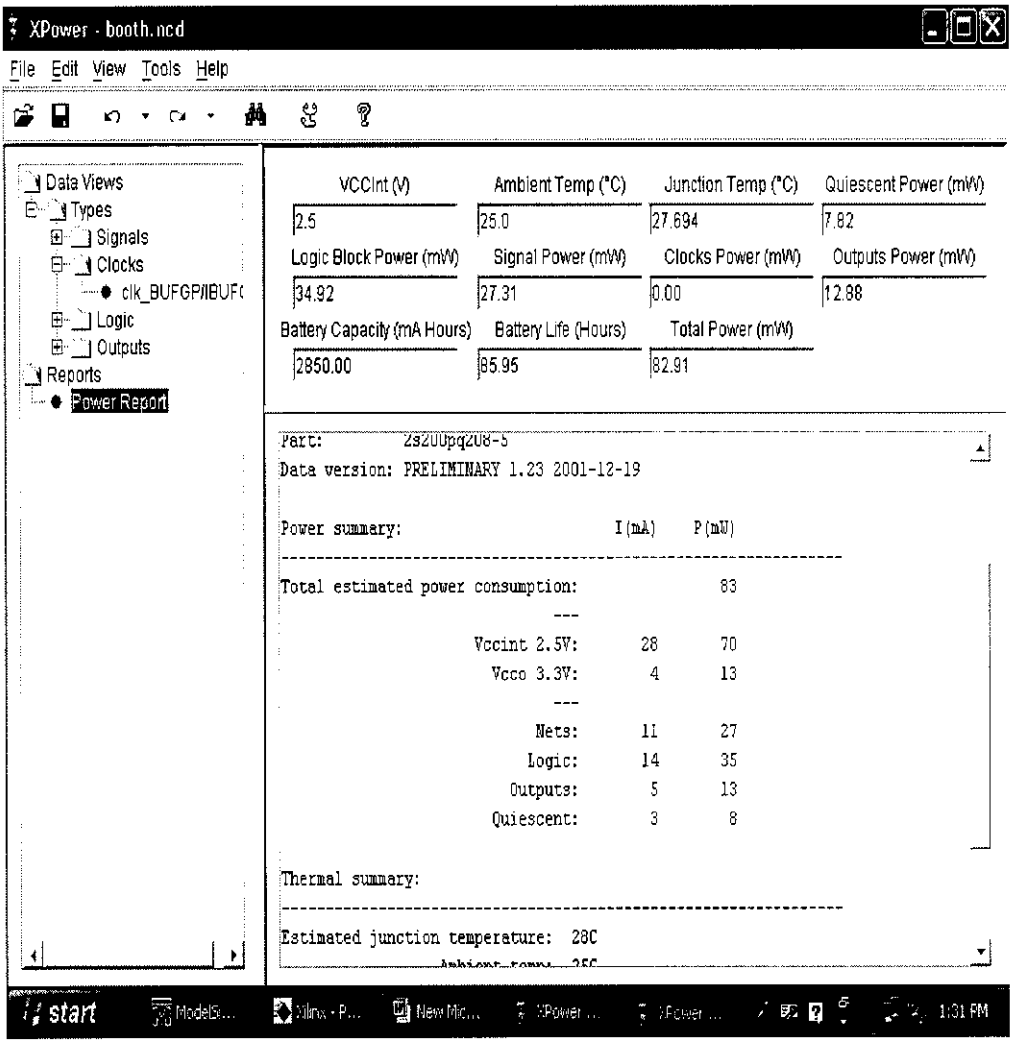


fig 6.4 Power Report of Booth Multiplier(Without SPST)

6.2.5 POWER REPORT OF SPST EQUIPPED BOOTH MULTIPLIER

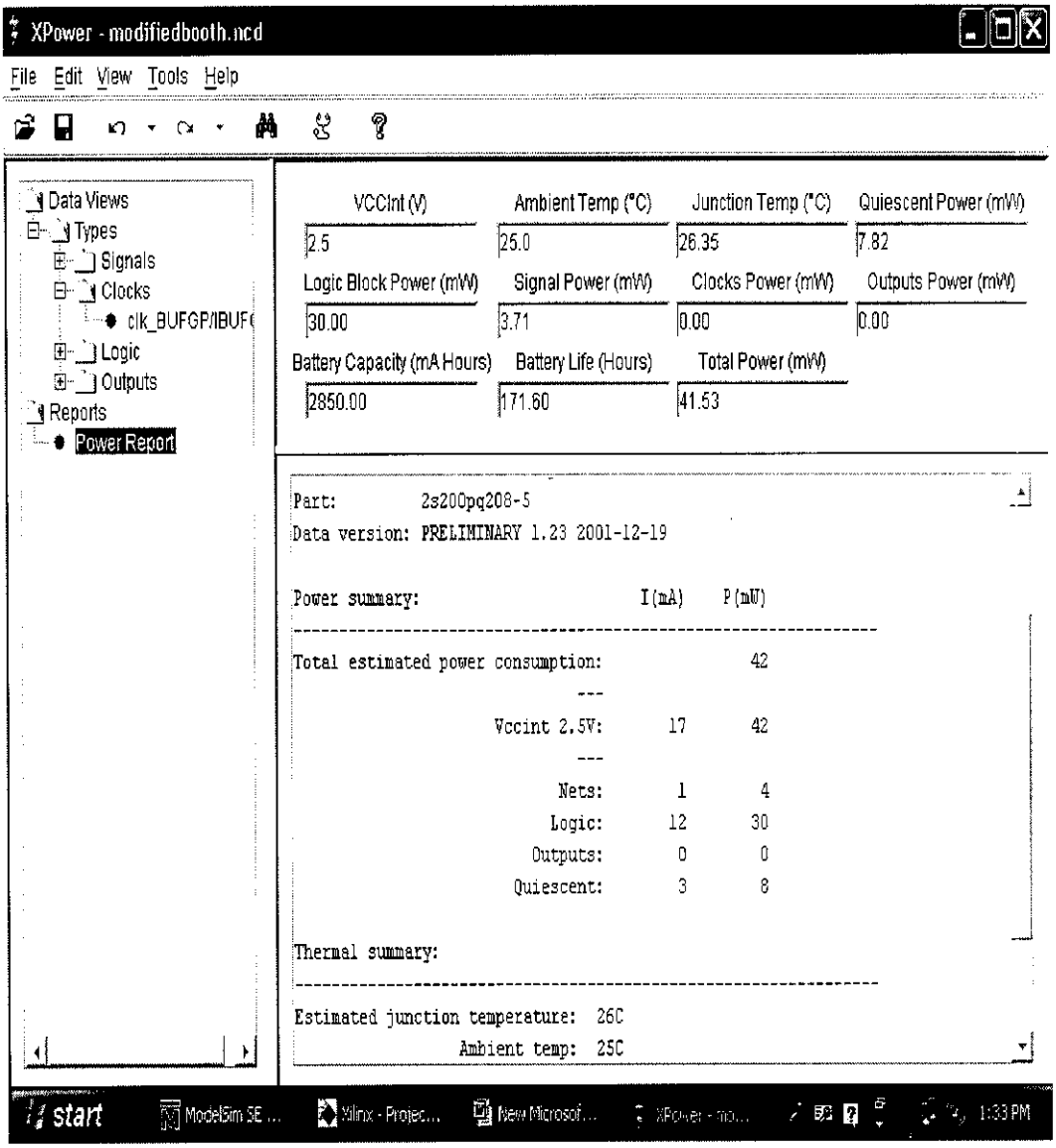


fig 6.5 Power Report of SPST Equipped Booth Multiplier

CHAPTER- 7

CONCLUSION

In this project, the multiplier algorithm is programmed using VHDL. It is simulated using MODELSIM 6.0a and the implementation of FPGA based SPST equipped booth multiplier is done that consumes less power compared with booth multiplier without SPST.

The SPST equipped booth multiplier is implemented and verified on XILINX SPARTAN-III FPGA device. From the power report obtained we can infer that about 50% reduction in power consumption is achieved with SPST compared with the booth multiplier without SPST.

REFERENCES:

[1] Kuan-Hung Chen and Yuan-Sun Chu , "A Low Power Multiplier With Spurious Power Suppression Technique", IEEE Transactions Very Large Scale Integration Systems, Volume-15, No.7,Pageno 846-850, July (2007).

[2] O.Chen,R.Sheen, and S.Wang, "A Low-Power Adder Operating On Effective Dynamic Data Ranges", IEEE Transactions Very Large Scale Integration Systems, Volume-10, No.4,Pageno 435-453,August(2002).

[3] A.Dempster M.Macleod, "Use of minimum adder multiplier block in FIR digital signal processing", IEEE Transactions on circuits and Systems, volume-II, No.42, Pageno 569-577, November (1999).

[4] U.Meyer Baese, "Digital Signal Processing with Field Programmable Gate Array", Springer International Edition, Second Edition (2006).

[5] W.Jenlans, G.Jullian, F.Taylor "Modern Application in Digital Signal Processing", Prentice Hall India (1995).

[6] E.Swartzlander, "Computer Arithmetic", Pearson Education India (2000).

APPENDIX-A

Spartan-3 1.2V FPGA Family: Introduction and Ordering Information

DS099-1 (v1.1) April 24, 2003

Advance Product Specification

Introduction

The 1.2V Spartan™-3 family of Field-Programmable Gate Arrays is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The eight-member family offers densities ranging from 50,000 to five million system gates, as shown in Table 1.

The Spartan-3 family builds on the success of the earlier Spartan-IIE family by increasing the amount of logic resources, the capacity of internal RAM, the total number of I/Os, and the overall level of performance as well as by improving clock management functions. Numerous enhancements derive from state-of-the-art Virtex™-II technology. These Spartan-3 enhancements, combined with advanced process technology, deliver more functionality and bandwidth per dollar than was previously possible, setting new standards in the programmable logic industry.

Because of their exceptionally low cost, Spartan-3 FPGAs are ideally suited to a wide range of consumer electronics applications, including broadband access, home networking, display/projection and digital television equipment.

The Spartan-3 family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.

Features

- Revolutionary 90-nanometer process technology
- Very low cost, high-performance logic solution for high-volume, consumer-oriented applications
  - Densities as high as 74,880 logic cells
  - 326 MHz system clock rate
  - Three separate power supplies for the core (1.2V), I/Os (1.2V to 3.3V), and special functions (2.5V)
- SelectIO™ signaling
  - Up to 784 I/O pins
  - 622 Mb/s data transfer rate per I/O
  - Seventeen single-ended signal standards
  - Six differential signal standards including LVDS
  - Termination by Digitally Controlled Impedance
  - Signal swing ranging from 1.14V to 2.46V
  - Double Data Rate (DDR) support
- Logic resources
  - Abundant, flexible logic cells with registers
  - Wide multiplexers
  - Fast look-ahead carry logic
  - Dedicated 18 x 18 multipliers
  - JTAG logic compatible with IEEE 1149.1/1532 standards
- SelectRAM™ hierarchical memory
  - Up to 1,872 Kbits of total block RAM
  - Up to 520 Kbits of total distributed RAM
- Digital Clock Manager (up to four DCMs)
  - Clock skew elimination
  - Frequency synthesis
  - High resolution phase shifting
- Eight global clock lines and abundant routing
- Fully supported by Xilinx ISE development system
  - Synthesis, mapping, placement and routing

Table 1: **Summary of Spartan-3 FPGA Attributes**

Device	System Gates	Logic Cells	CLB Array (One CLB = Four Slices)			Distributed RAM (bits[1])	Block RAM (bits[1])	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000	4M	62,208	96	72	6,912	432K	1,728K	96	4	712	312
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	784	344

Notes:
1. By convention, one Kb is equivalent to 1,024 bits.

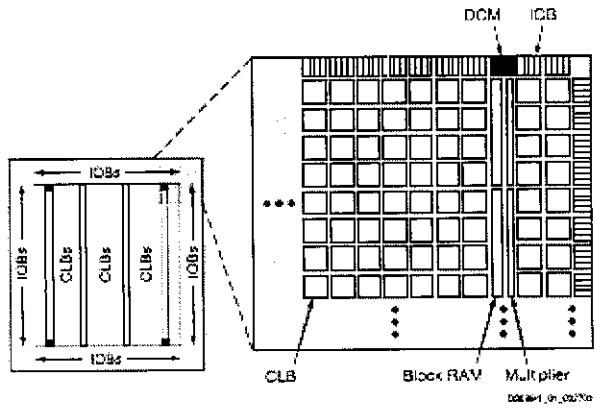
Architectural Overview

The Spartan-3 family architecture consists of five fundamental programmable functional elements:

- Configurable Logic Blocks (CLBs) contain RAM-based Look-Up Tables (LUTs) to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logical functions as well as to store data.
- Input/Output Blocks (IOBs) control the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. Twenty-three different signal standards, including six high-performance differential standards, are available as shown in Table 2. Double Data-Rate (DDR) registers are included. The Digitally Controlled Impedance (DCI) feature provides automatic on-chip terminations, simplifying board designs.
- Block RAM provides data storage in the form of 18-Kbit dual-port blocks.
- Multiplier blocks accept two 18-bit binary numbers as inputs and calculate the product.
- Digital Clock Manager (DCM) blocks provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase shifting clock signals.

These elements are organized as shown in Figure 1. A ring of IOBs surrounds a regular array of CLBs. The XC3S50 has a single column of block RAM embedded in the array. Those devices ranging from the XC3S200 to the XC3S2000 have two columns of block RAM. The XC3S4000 and XC3S5000 devices have four RAM columns. Each column is made up of several 18K-bit RAM blocks; each block is associated with a dedicated multiplier. The DCMs are positioned at the ends of each block RAM column.

The Spartan-3 family features a rich network of traces and switches that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing.



Notes:

1. The two additional block RAM columns of the XC3S4000 and XC3S5000 devices are shown with dashed lines. The XC3S50 has only the block RAM column on the far left.

Figure 1: **Spartan-3 Family Architecture**

Configuration

Spartan-3 FPGAs are programmed by loading configuration data into robust static memory cells that collectively control all functional elements and routing resources. Before powering on the FPGA, configuration data is stored externally in a PROM or some other nonvolatile medium either on or off the board. After applying power, the configuration data is written to the FPGA using any of five different modes: Master Parallel, Slave Parallel, Master Serial, Slave Serial and Boundary Scan (JTAG). The Master and Slave Parallel modes use an 8-bit wide SelectMAP™ Port.

The recommended memory for storing the configuration data is the low-cost Xilinx Platform Flash PROM family, which includes XCF00S PROMs for serial configuration and XCF00P PROMs for parallel configuration.

I/O Capabilities

The SelectIO feature of Spartan-3 devices supports 17 single-ended standards and six differential standards as listed in Table 2. Table 3 shows the number of user I/Os as well as the number of differential I/O pairs available for each device/package combination.

Table 2: **Signal Standards Supported by the Spartan-3 Family**

Standard Category	Description	V_{CCO} (V)	Class	Symbol
Single-Ended				
GTL	Gunning Transceiver Logic	N/A	Terminated	GTL
			Plus	GTLP
HSTL	High-Speed Transceiver Logic	1.5	I	HSTL_
			III	HSTL_III
		1.8	I	HSTL__18
			II	HSTL_I_18
			III	HSTL_III_18
LVCMOS	Low-Voltage CMOS	1.2	N/A	LVCMOS12
		1.5	N/A	LVCMOS15
		1.8	N/A	LVCMOS18
		2.5	N/A	LVCMOS25
		3.3	N/A	LVCMOS33
LVTTL	Low-Voltage Transistor-Transistor Logic	3.3	N/A	LVTTL
PCI	Peripheral Component Interconnect	3.0	33 MHz	PCI33_3
SSTL	Stub Series Terminated Logic	1.8	N/A	SSTL18_I
		2.5	I	SSTL2_I
			II	SSTL2_II
Differential				
LDT	Lightning Data Transport (HyperTransport™)	2.5	N/A	LDT_25
LVDS	Low Voltage Differential Signaling		Standard	LVDS_25
			Bus	BLVDS_25
			Extended Mode	LVDSEXT_25
			Ultra	ULVDS_25
RSDS	Reduced-Swing Differential Signaling	2.5	N/A	RSDS_25

Table 3: Spartan-3 User I/O Chart

	Available User I/Os and Differential (Diff) I/O Pairs															
	VQ100		TQ144		PQ208		FT256		FG456		FG676		FG900		FG1156	
Device	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff	User	Diff
XC3S50	63	29	97	45	124	56	-	-	-	-	-	-	-	-	-	-
XC3S200	63	29	97	45	141	62	173	76	-	-	-	-	-	-	-	-
XC3S400	-	-	97	45	141	62	173	76	264	116	-	-	-	-	-	-
XC3S1000	-	-	-	-	-	-	173	76	333	149	391	175	-	-	-	-
XC3S1500	-	-	-	-	-	-	-	-	333	149	487	221	-	-	-	-
XC3S2000	-	-	-	-	-	-	-	-	-	-	489	221	565	270	-	-
XC3S4000	-	-	-	-	-	-	-	-	-	-	-	-	633	300	712	312
XC3S5000	-	-	-	-	-	-	-	-	-	-	-	-	633	300	784	344

Notes:
1. All device options listed in a given package column are pin-compatible.

Product Ordering and Availability

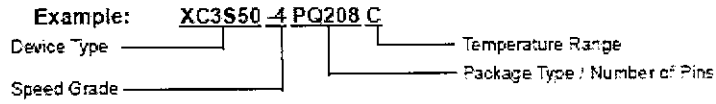
Table 4 shows all valid device ordering combinations of device density, speed grade, package, and temperature range parameters for the Spartan-3 family as well as the availability status of those combinations.

Table 4: Spartan-3 Device Availability

Package Type:	VQFP	TQFP	PQFP	FTBGA	FBGA			
No. of Pins:	100	144	208	256	456	676	900	1156
Code:	VQ100	TQ144	PQ208	FT256	FG456	FG676	FG900	FG1156
Device[1]								
XC3S50	(C, I)	(C, I)	(C, I)	-	-	-	-	-
XC3S200	(C, I)	(C, I)	(C, I)	(C, I)	-	-	-	-
XC3S400	-	(C, I)	(C, I)	(C, I)	(C, I)	-	-	-
XC3S1000	-	-	-	(C, I)	(C, I)	(C, I)	-	-
XC3S1500	-	-	-	-	(C, I)	(C, I)	-	-
XC3S2000	-	-	-	-	-	(C, I)	(C, I)	-
XC3S4000	-	-	-	-	-	-	(C, I)	(C, I)
XC3S5000	-	-	-	-	-	-	(C, I)	(C, I)

Notes:
1. Commercial devices are offered in the -4 and -5 speed grades; industrial devices are only in the -4 speed grade.
2. C = Commercial, T_J = 0° to +85° C; I = Industrial, T_J = –40° C to +100° C.
3. Parentheses indicate that a given product is not yet released to production. Contact sales for availability information.

Ordering Information



Device
XC3S50
XC3S200
XC3S400
XC3S1000
XC3S1500
XC3S2000
XC3S4000
XC3S5000

Speed Grade	
-4	Standard Performance
-5	High Performance

Package Type / Number of Pins	
VQ100	100-pin Very Thin Quad Flat Pack (VQFP)
TQ144	144-pin Thin Quad Flat Pack (TQFP)
PQ208	208-pin Plastic Quad Flat Pack (PQFP)
FT256	256-ball Fine-Pitch Thin Ball Grid Array (FTBGA)
FG456	456-ball Fine-Pitch Ball Grid Array (FBGA)
FG676	676-ball Fine-Pitch Ball Grid Array (FBGA)
FG900	900-ball Fine-Pitch Ball Grid Array (FBGA)
FG1156	1156-ball Fine-Pitch Ball Grid Array (FBGA)

Temperature Range (T_J)	
C	Commercial (0°C to 85°C)
I	Industrial (–40°C to 100°C)

Revision History

Date	Version No.	Description
04/11/03	1.0	Initial Xilinx release.
04/24/03	1.1	Updated block RAM, DCM, and multiplier counts for the XC3S50.

The Spartan-3 Family Data Sheet

DS099-1, *Spartan-3 1.2V FPGA Family: Introduction and Ordering Information* (Module 1)

DS099-2, *Spartan-3 1.2V FPGA Family: Functional Description* (Module 2)

DS099-3, *Spartan-3 1.2V FPGA Family: DC and Switching Characteristics* (Module 3)

DS099-4, *Spartan-3 1.2V FPGA Family: Pinout Tables* (Module 4)

APPENDIX-B

PHOTO:

XILINX SPARTAN-III FPGA (XC3S 400)