



# **DIGIT-LEVEL SERIAL - IN PARALLEL - OUT MULTIPLIER USING REDUNDANT REPRESENTATION FOR A CLASS OF FINITE FIELDS**

**AMAR HASSABALKAREEM ALMAN SOB, A.DINESH REDDY, G.MADHAV,  
P.NAGAVENI**

## **ABSTRACT**

Two digit-level finite field multipliers in  $F_{2^m}$  using redundant representation are presented. Embedding  $F_{2^m}$  in cyclotomic field  $F(n)$  causes a certain amount of redundancy and consequently performing field multiplication using redundant representation would require more hardware resources. Based on a specific feature of redundant representation in a class of finite fields, two new multiplication algorithms along with their pertaining architectures are proposed to alleviate this problem. Considering area-delay product as a measure of evaluation, it has been shown that both the proposed architectures considerably outperform existing digit-level multipliers using the same basis. It is also shown that for a subset of the fields, the proposed multipliers are of higher performance in terms of area-delay complexities among several recently proposed optimal normal basis multipliers. The main characteristics of the postplace&route application specific integrated circuit implementation of the proposed multipliers for three practical digit sizes are also reported.

## **CHAPTER-1 INTRODUCTION**

### **1.1 SYNOPSIS**

High performance Arithmetic units are essential since the speed of the digital processor depends heavily on the speed of the Arithmetic units used in the system. Digital arithmetic operations are very important in the design of digital processors and application-specific systems. Arithmetic circuits form an important class of circuits in digital systems.

Adders are most commonly used in various electronic applications e.g. Digital signal processing in which adders are used to perform various algorithms like FIR, IIR etc. In past, the major challenge for VLSI designer is to reduce

area of chip by using efficient optimization techniques. Then the next phase is to increase the speed of operation to achieve fast calculations like, in today's microprocessors millions of instructions are performed per second.

Speed of operation is one of the major constraints in designing DSP processors. Some central processing units are comprised of two components, an arithmetic unit and a logic unit. Other processors may have an arithmetic unit for calculating fixed-point operations and another AU for calculating floating-point computations.

Some PCs have a separate chip known as the numeric coprocessor. This coprocessor contains a floating-point unit for processing floating-point operands. The coprocessor increases the operating speed of the computer because of the coprocessor ability to perform computation faster and more efficiently. The redundancy associated with signed digit numbers offers the possibility of carry free addition. The redundancy provided in signed-digit representation allows for fast addition and subtraction because the sum or difference digit is a function of only the digits in two adjacent digit positions of the operands for a radix greater than 2, and 3 adjacent digit positions for a radix of 2.

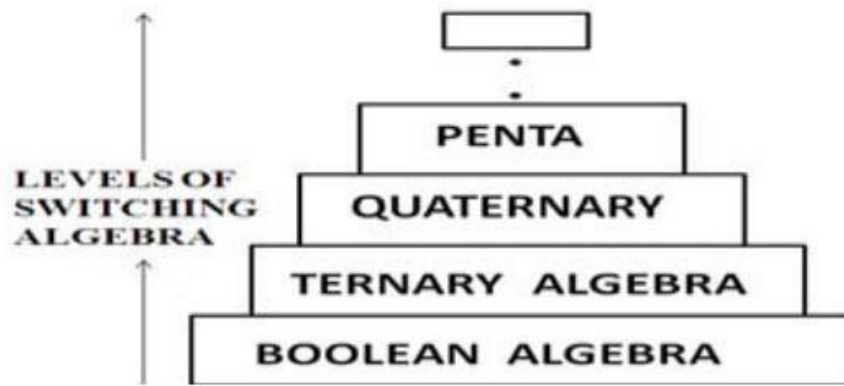
Thus, the add time for two redundant signed-digit numbers is a constant independent of the word length of the operands, which is the key to high speed computation. The advantage of carry free addition offered by QSD numbers is exploited in designing a fast adder circuit. Additionally adder designed with QSD number system has a regular layout which is suitable for VLSI implementation which is the great advantage over the BSD adder. An Algorithm for design of QSD adder is proposed. Binary signed-digit numbers are known to allow limited carry propagation with a somewhat more complex addition process requiring very large circuit for implementation.

A special higher radix based (quaternary) representation of binary signed digit numbers not only allows carry-free addition and borrow-free subtraction but also offers other important advantages such as simplicity in logic and higher storage density.

## 1.2 MOTIVATION

The major challenges in VLSI design are reducing the area of chip and increasing speed of the circuit. Reducing area can be achieved by optimization techniques and number of instructions executed per second increases as speed increases. The performance of a digital system depends upon performance of adders. Adders are also act as basic building blocks for all arithmetic circuits for example DSP processors.

Binary adders are easy to implement because of logic levels involved in it '0' and '1', but they have their own limitations in the area of circuit complexity and chip area which ultimately increases propagation delay of the circuit. Is it time to move beyond zeroes and ones? This is the title of Bernard Cole article's published in 2003 on the official site of the Embedded Development Community. The conclusion is "I think that the economics of semiconductor manufacturing now is forcing us to move beyond zero and one. Shouldn't we also take another look at multi-valued logic?"



**Fig 1.1 Levels of Switching Algebra**

This very thought brought many researches to work upon multi-valued logic to bring a new era of technology. Many authors have directed their efforts to the implementation of Multi-Valued logic looking for benefit from all advantages it possess over the binary logic. It is possible for ternary logic to achieve simplicity and energy efficiency in digital design since the logic reduces the complexity of interconnects and chip area, in turn, reducing the chip delay.

### 1.3 OVERVIEW OF NUMBER SYSTEMS

Humans are speaking to one another in a particular language made of words and letters. While we type words and letters in the computer, the computer does not understand the words and letters. Rather, those words and letters are translated into numbers. It means that computers “talk” and understand in numbers. Although many students know the decimal (base 10) system, and are very comfortable with performing operations using this system, it is too important for students to understand that the decimal system is not the only system.

By studying other number systems such as binary (base 2) quaternary (base 4), octal (base 8), and hexadecimal (base 16) and so forth, students will gain a better understanding of how number systems work in general.

#### 1.3.1 DIGITS

Before understanding the number systems and the conversion concepts of numbers from one number system to another, the digit of a number system must be understood. The first digit in any numbering system is always a zero. For example, a base 2 (binary) numbers contains 2 digits: 0 and 1, a base 3 (ternary) numbers contains 3 digits: 0, 1 and 2, a base 4 (quaternary) numbers contains 4 digits: 0 through 3 and so forth. Note that a base 10 (decimal) numbers does not contain the digit 10, similarly base 16 numbers does not contain a digit 16. Same is the case for the other number systems. Once the digits of a number system are understood, larger numbers can be constructed using positional notation or place-value notation method.

According to the positional notation method, in decimal number the first right most digits (integer) have a unit's position. Further, to the left of the units position is the ten's position, the position to the left of the ten's position is the hundred's position and so forth. Here, the units position has a weight of 100, or 1; the tens position has a weight

of 101, or 10; and the hundreds position has a weight of 102, or 100.

The exponential powers of the positions are critical for understanding numbers in other numbering systems. Remember the position to the left of the radix point is always the unit's position in any number system. For example the position to the left of the binary point is always 20, or 1; the position to the left of the senary point is always 60, or 1; the position to the left of the octal point is always 80 or 1 and so on. The position to the left of the unit's position is always the number whose base is raised to the first power; i.e. 21, 61, 81 and so on. These concepts can be extended to each and every number system.

### Number Representation

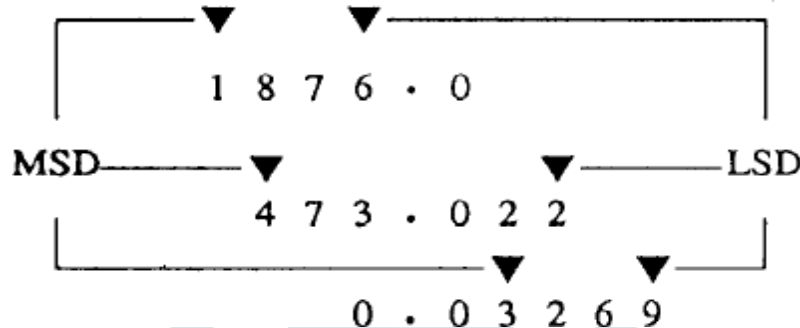
A number in any base system can be represented in a generalized format as follows:  $N = A_n B_n + A_{n-1} B_{n-1} + \dots + A_1 B_1 + A_0 B_0$ ,

Where N = Number, B=Base, A= any digit in that base

### 1.3.2 MOST SIGNIFICANT DIGIT (MSD) AND LEAST SIGNIFICANT DIGIT(LSD)

The MSD in a number is the digit that has the greatest effect on that number, while The LSD in a number is the digit that has the least effect on that number.

Look at the following examples:



You can easily see that a change in the MSD will increase or decrease the value of the number in the greatest amount, while changes in the LSD will have the smallest effect on the value.

### 1.3.3 DECIMAL NUMBER SYSTEM

The decimal number system is known as international system of numbers. It is also called base ten or occasionally denary number system. It has ten as its base. It is the numerical base most widely used by modern civilization. Decimal notation often refers to a base-10 positional notation however; it can also be used more generally to refer to non positional systems. Positional decimal systems include a zero and use symbols (called digits) for the ten values (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) to represent any number, no matter how large or how small.

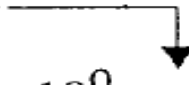
Let's examine the decimal (base 10) value of 427.5. You know that this value is four hundredtwenty-seven and one half. Now examine the position of each number:

**427.5**

This represents 1/2 unit This represents 7 units

This represents 20 units

This represents 400 units

|                                                                                                |                                   |        |   |           |
|------------------------------------------------------------------------------------------------|-----------------------------------|--------|---|-----------|
| Radix Point  |                                   |        |   |           |
| $10^2$                                                                                         | $10^1$                            | $10^0$ | . | $10^{-1}$ |
| 4                                                                                              | 2                                 | 7      | . | 5         |
| <hr/>                                                                                          |                                   |        |   |           |
| $10^2$                                                                                         | $= 4 \times 100, \text{ or } 400$ |        |   |           |
| $10^1$                                                                                         | $= 2 \times 10, \text{ or } 20$   |        |   |           |
| $10^0$                                                                                         | $= 7 \times 1, \text{ or } 7$     |        |   |           |
| $10^{-1}$                                                                                      | $= 5 \times .1, \text{ or } .5$   |        |   |           |

Each digit has its own value (weight) as described in the above figure. Now let's look at the value of the base 10 number 427.5 with the positional notation line graph

### 1.3.4 BINARY NUMBER SYSTEM

The number system with base (or radix) 2, is known as the binary number system. Only two symbols are used to represent numbers in this system and these are 0 and 1, these are known as bits. It is a positional system i.e. every position is assigned a specific weight. Moreover, it has two parts the Integral part or integers and the fractional part or fractions, set a part by a radix point.

For example (1101.101)<sub>2</sub> In binary number system the left-most bit is known as most significant bit (MSB) and the right-most bit is known as the least significant bit (LSB), similar to decimal number system. The following graph shows the position and the power of the base (2 in this case):

... 2<sup>3</sup> 2<sup>2</sup> 2<sup>1</sup> 2<sup>0</sup> 2<sup>-1</sup> 2<sup>-2</sup> 2<sup>-3</sup> .....

The arithmetic operations such as addition, subtraction, multiplication and division of decimal numbers can be also performed on binary numbers. Also binary arithmetic is much simpler than decimal arithmetic because here only two digits, 0 and 1 are involved.

### 1.3.5 QUATERNARY NUMBER SYSTEM

The number system with base (or radix) 4, is known as the quaternary number system. Only four symbols are used to represent numbers in this system and these are 0, 1, 2 and 3. It is also a positional system i.e. every position is assigned a specific weight. Moreover, it has two parts the Integral part or integers and the fractional part or fractions, set a part by a radix point.

For example (121.133)<sub>4</sub> In binary number system the left-most digit is known as most significant digit (MSB) &



the right–most digit is known as the least significant digit (LSB). The following graph shows the position and the power of the base (4 in this case):

... 43 42 41 40. 4-1 4-2 4-3 ...

The arithmetic operations such as addition, subtraction, multiplication and division of decimal numbers can be also performed on quaternary numbers. Quaternary numbers are used in the representation of 2D Hilbert curves, while many of the Chumashan languages originally used a base4 counting system, in which the names for numbers were structured according to multiples of 4 and 16.

## INTRODUCTION

Modern computers are based on binary number system (radix =2). It has two logical states '0' and '1'. In such system, '1' plus '1' is '0' with carry '1' (i.e. 1+1=10). This carry should have to add with another '1', as a result further carry '1' generates. This creates the delay problem in computer circuits. So to get rid of this carry formation again and again signed digit is essential. In high-speed arithmetical calculation, carry free adders improves the operational performance.

Binary logic is restricted to only two logical states; Multi-Valued Logic (MVL) replaces these with finite and infinite numbers of values. Multi-valued logic is a higher radix ( $R > 2$ ) logic system. Non- binary data requires less physical storage space than binary data [2-4]. Depending upon the radix number  $R$ , the number system are named as ternary ( $R = 3$ ), quaternary ( $R = 4$ ) etc. Ternary logic is based on ternary number system. They can further be divided into two groups; symmetric ternary

{1, 0, 1} and ordinary ternary {0, 1, 2}. Both groups are important in logical and arithmetical operations [5-8]. Quaternary logic is based on radix-4 number system.

In quaternary system, the positive integer set {0, 1, 2, 3} is called ordinary quaternary digit (OQD) and the set of both positive and negative integer {3, 2, 1, 0, 1, 2, 3} is called quaternary signed digit (QSD), where 3= -3, 2= -2, 1= -1. In signed digit representation QSD number can be written as:

$$X = \sum_{i=0}^{n-1} x_i \cdot 4^i ; \text{ where } x_i \in \{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$$

### 1.3.6 SIGNED DIGIT NUMBER

Signed digit number representations are prefixed with a – (minus) sign to indicate that they are negative numbers. Signed digit numbers used to accomplish fast addition, subtraction, multiplication and division of integers because it can eliminate carry.

$$\begin{aligned} (1122)_2 &= 1 * 2^3 + \bar{1} * 2^2 - 2 * 2^1 + 1 * 2^0 \\ &= 8 + 4 - 4 + 2 \\ &= 10 \end{aligned}$$

Signed digit representation is essentially required for carry free arithmetic operation. As such, binary to quaternary signed digit conversion is very much interesting and required topics. This is called 'radix conversion'. The importance of radix conversion is shown in Flexogram. Quaternary is the base-4 numeral system. It uses the digits 0, 1, 2 and 3 to represent any real number. It shares with all fixed-radix numeral systems many properties, such as the ability to represent any real number with a canonical representation (almost unique) and the characteristics of the representations of rational numbers and irrational numbers.

## 1.4 RELATION TO BINARY

As with the octal and hexadecimal numeral systems, quaternary has a special relation to the binary numeral system. Each radix 4, 8 and 16 is a power of 2, so the conversion to and from binary is implemented by matching each digit with 2, 3 or 4 binary digits, or bits. For example, in base 4,

$$3021_4 = 11\ 00\ 10\ 01\ 00_2.$$

Although octal and hexadecimal are widely used in computing and computer programming in the discussion and analysis of binary arithmetic and logic, quaternary does not enjoy the same status. By analogy with bit, a quaternary digit is sometimes called a crumb.



Figure 1.2: The Importance of Radix Conversion

## 1.5 HOW DO BASE 4 WORK?

Bases have to do with how you write numbers in a number system, and how the place values work in that system. Let's start with the system you already know. We usually work in base 10. In base 10, the place values are ones, tens, hundreds, thousands and so on. So when we see a number like 437, it really means four hundreds, 3 tens and 7 ones.' We understand that to be worth 'four hundred and thirty seven'. The place values are: In base 10, ones is  $10^0$ , tens is  $10^1$ , hundreds

is  $10^2$ , thousands is  $10^3$  and so on. When we start to count in base 10, we can write as, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Each of those stands for how many ones we have. The number 8 means 8 ones, or  $8 * 1$ . But when we go past 9 to the number 10, we don't have a single digit that stands for '10 ones.' So instead, we use a two-digit number, 10, which stands for '1 ten and 0 ones.' Once we get to 99, we have reached 9 tens and 9 ones. Going past that, we move to a three-digit number, 100, which means '1 hundred, 0 tens and 0 ones. It's kind of hard to think about this, because your brain just does it without thinking about it, but that's what's really going on. So what happens in base 4? The place values are again given by raising 4 to powers. Determined by raising the base to powers.

$$4^0 = 1$$

$$4^1 = 4$$

$$4^2 = 16$$

$$4^3 = 64$$

So, the number 23 in base 4 is NOT worth twenty three. It's only twenty three in base 10, where it means 2 tens and 3 ones. In base 4, 23 (which is read as two-three) means 2 fours and 3 ones. So it has a value of  $2*4 + 3*1$  or  $8 + 3$  or 11. Now think about how we count in base 4. We start with 1, 2, 3. But there is no digit '4' to use--the number 4 is written '1 four and 0 ones,' so it's 10. I know this may be confusing, but here are the numbers from 1 to 10 in base 4:

Binary logic is restricted to only two logical states. Multi-Valued Logic (MVL) replaces these with finite and infinite numbers of values. Multi-valued logic is a higher radix ( $R > 2$ ) logic system. Non-binary data requires less physical storage space than binary data. Depending upon the radix number  $R$ , the number system are named as ternary ( $R = 3$ ), quaternary ( $R = 4$ ) etc.

Ternary logic is based on ternary number system. Quaternary logic is based on Quaternary number system. Quaternary is the base 4 redundant number system. The degree of redundancy usually increases with the increase of the radix. The signed digit number system allows us to implement parallel arithmetic by using redundancy. QSD numbers are the Signed Digit numbers with the digit set as:  $\{-3, -2, -1, 0, 1, 2, 3\}$  respectively.

## 1.6 QSD NUMBER REPRESENTATION

In general, a signed-digit decimal number  $D$  can be represented in terms of an  $n$  digit quaternary signed digit number as  $\sum_{i=0}^{n-1} x_i 4^i$  Where  $x_i$  can be any value from the set  $\{-3, -2, -1, 0, 1, 2, 3\}$  for producing an appropriate decimal representation. A QSD number can be represented in Binary in  $2^b$  (2 bit) notation for unsigned QSD number.

For digital implementation, QSD numbers are represented using 3-bit 2's complement notation. AQSD negative



number is the QSD complement of the QSD positive number. For example, using the primes to denote complementation, we have  $3 = -3$ ,  $2 = -2$  and  $1 = -1$ .

### 1.6.1 COMPARISON OF QSD WITH BSD

It offers the advantage of reduced circuit complexity both in terms of transistor count and interconnections. QSD number uses 25% less space than BSD to store number. QSD numbers store 25% storage compared to BSD. To represent a numeric value  $N$   $\lceil \log_4 N \rceil$  number of QSD digits and  $3 \lceil \log_4 N \rceil$  binary bits are required while for the same  $\log_2 N$  BSD digits and  $2 \lceil \log_2 N \rceil$  binary bits are required in BSD representation. Ratio of number of bits in QSD to BSD representation for an arbitrary number  $N$  is,

$$3 \lceil \log_4 N \rceil / 2 \lceil \log_2 N \rceil$$

This roughly equals to  $\frac{3}{4}$ . Therefore, QSD saves  $\frac{1}{4}$  of the storage used by BSD. The computation speed and circuit complexity increases as the number of computation steps decreases. The computation speed mainly depends on the number of bits required to represent a number, since the less the number of bit's the easy is the computation. In general the number of bits required by a QSD number system is less when compared to BSD number system, which in turn results in better speeds and performance.

### 1.6.2 ADVANTAGES OF QSD NUMBER SYSTEM

The main advantage of Quaternary logic is that it reduces the number of required computation steps for developing digital design. Furthermore memory, control unit, and processor can be carried out faster if the Quaternary logic is easily employed and memory utilization also less than binary. These advantages have been shown to be useful for the design of Quaternary computers, for digital filtering. Quaternary representation admits sign convention also.

- Quaternary logic is mainly applied in new transforms for encoding and more efficient for Compression, error correction, and state assignment, representation of discrete information and in automatic telephony.
- Quaternary logic also offers greater utilization of transmission channels because of the higher. Information content carried by every line. It gives exact and more efficient error detection and correction codes and possesses potentially higher density of information storage.
- We can achieve a carry free arithmetic operation by using higher radix number system such as QSD (Quaternary Signed Digit).
- Signed digit number system has redundancy associated with it. The redundancy provided in signed digit number system offers the possibility of carry free arithmetic operations which in terms allows for faster processing. In signed digit representation of the system the add time for two redundant signed digit numbers is a constant independent of the word length of the operands which is the key to high speed computation. Binary signed digit numbers allows limited carry propagation with a more complex addition process which requires very large circuit for implementation.
- A higher radix based representation of binary signed digit numbers such as quaternary allows carry free arithmetic operations as well as it offers the important advantage of logic simplicity and storage density.

## 1.7 OBJECTIVE

The objective is to design carry free adder using QSD number system to achieve fast addition with the help of Verilog which integrates novel design of high speed QSD adder and multiplier for higher input bit sequences. The programming objective of the VLSI Implementation of fast addition using QSD number system into the following categories

- QSD Adder Unit
- QSD Sub tractor Unit
- QSD Multiplication Unit
- QSD Division Unit
- Synthesis Reports
- Physical designing

## CHAPTER- 2 LITERATURE SURVEY

### 2.1 INTRODUCTION

Binary Signed Digit Numbers are known to allow limited carry propagation with more complex addition process. Arithmetic operations such as addition, subtraction and multiplication still suffer from known problems including limited number of bits, propagation delay, and circuit Complexity. Some of the limitations of this system are computational speed which limits formation and propagation of carry especially as the number of bits increases. Therefore it provides large complexity and low storage density. To construct combinational logic circuits that performs binary addition.

A binary signed digit representation of an integer  $k \in [0, 2^n - 1]$  is a base-2 representation denoted by  $(K_n, K_{n-1}, \dots, K_0)$  BSD where  $k_i \in \{-1, 0, 1\}$ . We will call the  $K$  is signed bits, or  $s$  bits. An integer can have several BSD representations. For example,  $k = (9)_{10}$  can be written as  $(01001)$  BSD among other possibilities. Among the possible BSD representations of an integer there are two unique representations: one is conventional the binary representation where there are no 1s and the other is the non-adjacent form (NAF).

The NAF of an integer can be generated using different methods. Recently, algorithms that generate a random BSD representation of an integer have been proposed. The original purpose of the algorithms has been to provide protection against differential side-channel attacks by randomly changing the BSD representation of the secret key of elliptic curve cryptosystems. Subsequent work has however shown that randomly changing BSD representation of the secret key alone is not sufficient. Tanay Chattopadhyay and Tamal Sarkar described that Quaternary Signed Digit (QSD) Number System has radix 4. “+3” is the Maximum digit and “-3” is the minimum digit in QSD Number System. In QSDNS 3 bits are required to represent a single digit, among these 2 bits are used to represent magnitude (0,1,2,3), 1 bit is used to represent sign (+ve/-ve).

To convert  $n$ -bit binary data to its equivalent  $q$ -digit QSD data, we have to convert this  $n$ -bit binary data into  $3q$ -bit

binary data. To achieve the target, we have to split the 3rd, 5th, 7th bit.... i.e. odd bit (from the LSB to MSB) into two portions. But we cannot split the MSB. If the odd bit is 1 then, it is split into 1 & 0 and if it is 0 then, it is split into 0 & 0. So we have to split the binary data  $(q-1)$  times (as example, for conversion of 2-bit quaternary number, the splitting is 1 time; for converting 3-digit quaternary number the split is 2-times and so on). In each such splitting one extra bit is generated.

So, the required binary bits for conversion to its QSD equivalent  $(n) = (\text{Total numbers of bits generated after divisions}) - (\text{extra bit generated due to splitting})$ .  $n = 2q+1$  Sachin Dubey<sup>1</sup>, Reena Rani<sup>2</sup>, Saroj Kumari<sup>3</sup>, Neelam Sharma<sup>4</sup> Member, IEEE Electronics & Communication Engineering Department Hindustan Institute of Technology & Management, Keetham, Agra, U.P, India, described that QSD adders are perfectly suitable for high speed operations. QSD number system radix is 4 whereas the binary number system has radix 2. QSD, each digit can be represented by a number from -3 to 3. QSD number system requires a different set of prime modulo based logic elements for each arithmetic operation.

Using a quaternary Signed Digit number system one may perform carry free addition, borrow free subtraction and multiplication. Carry free addition and other operations on a large number of digits such as 64, 128, or more can be implemented with constant delay and less complexity.

## 2.2 MULTIPLIERS

Multipliers play 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, regularity of layout and hence less area or even combination of them in one multiplier thus making them suitable for various high speed, low power and compact VLSI implementation. 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 number of partial products to be added, Modified Booth algorithm is one of the most popular algorithms. To achieve speed improvements Wallace Tree algorithm can be used to reduce the number of sequential adding stages. Further by combining both Modified Booth algorithm and Wallace Tree technique we can see advantage of both algorithms in one multiplier. However with increasing parallelism, the amount of shifts between the partial products and intermediate sums to be added will increase which may result in reduced speed, increase in silicon area due to irregularity of structure and also increased power consumption due to increase in interconnect resulting from complex routing.

On the other hand “serial-parallel” multipliers compromise speed to achieve better performance for area and power consumption. The selection of a parallel or serial multiplier actually depends on the nature of application. In this lecture we introduce the multiplication algorithms and architecture and compare them in terms of speed, area, power and combination of these metrics.

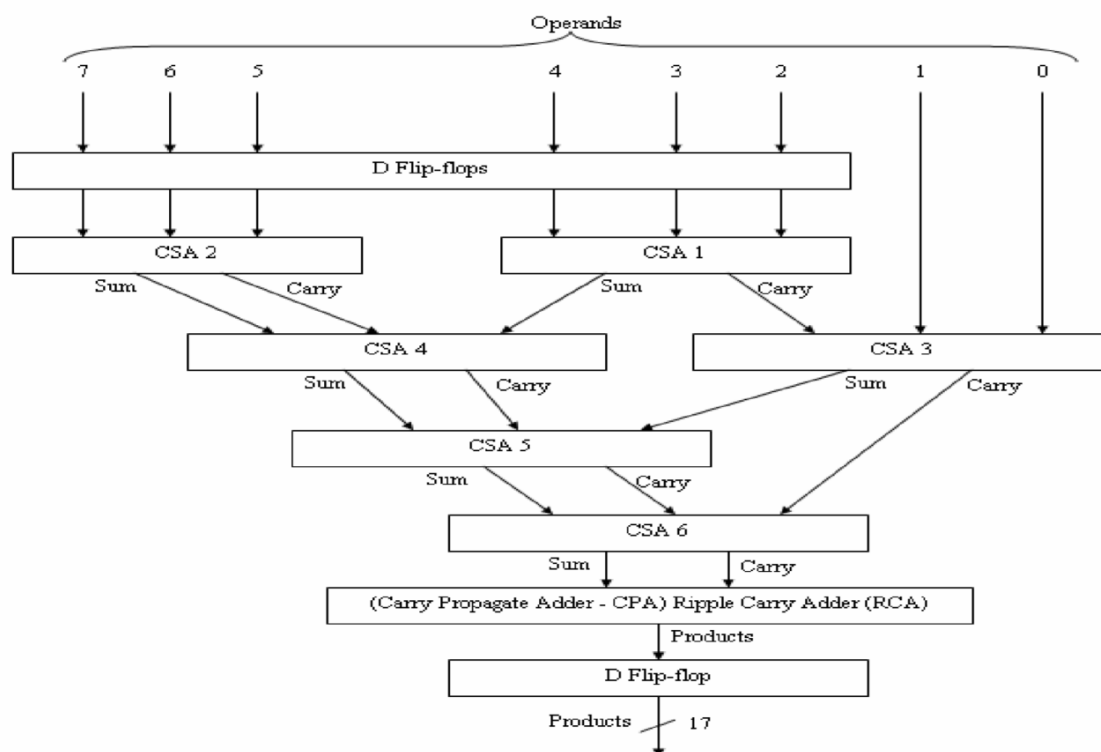
### 2.2.1 WALLACE TREE MULTIPLIER

The Wallace tree multiplier is considerably faster than a simple array multiplier because its height is logarithmic in word size, not linear. However, in addition to the large number of adders required, the Wallace tree's wiring is

much less regular and more complicated. As a result, Wallace trees are often avoided by designers, while design complexity is a concern to them. Wallace tree styles use a log-depth tree network for reduction.

Faster, but irregular, they trade ease of layout for speed. Wallace tree styles are generally avoided for low power applications, since excess of wiring is likely to consume extra power. While subsequently faster than Carry-save structure for large bit multipliers, the Wallace tree multiplier has the disadvantage of being very irregular, which complicates the task of coming with an efficient layout. The Wallace tree multiplier is a high speed multiplier. The summing of the partial product bits in parallel using a tree of carry-save adders became generally known as the “Wallace Tree”. Three step processes are used to multiply two numbers.

- Formation of bit products.
- Reduction of the bit product matrix into a two row matrix by means of a carry save adder.
- Summation of remaining two rows using a faster Carry Look Ahead Adder (CLA).



**Figure 2.1 Wallace Tree Block Diagram**

In order to design an **n-bit Wallace tree Multiplier** (Generic: =N) an algorithm was derived from the flow diagram developed below. The flow diagram below shows the intermediate state reductions of the multipliers are being done by Carry save adders and half adders while the final step additions being done by a Carry Look Ahead Adder. The flow diagram was done in Microsoft Excel sheet and Paint. After generating the flow diagram for 8-bit  $\times$  8-bit we generalized the algorithm for n-bit and hence we designed a GENERIC WALLACE TREE.

## 2.2.2 THE BOOTH'S MULTIPLIER

Booth multiplier can be used in different modes such as **radix-2**, **radix-4**, **radix-8** etc. But we decided to though Wallace Tree multipliers were faster than the traditional Carry Save method, it also was very irregular and hence

was complicated while drawing the Layouts. Slowly when multiplier bits gets beyond 32-bits large numbers of logic gates are required and hence also more interconnecting wires which makes chip design large and slows down operating speed use **Radix-4 Booth's Algorithm** because of number of Partial products is reduced to  $n/2$ .

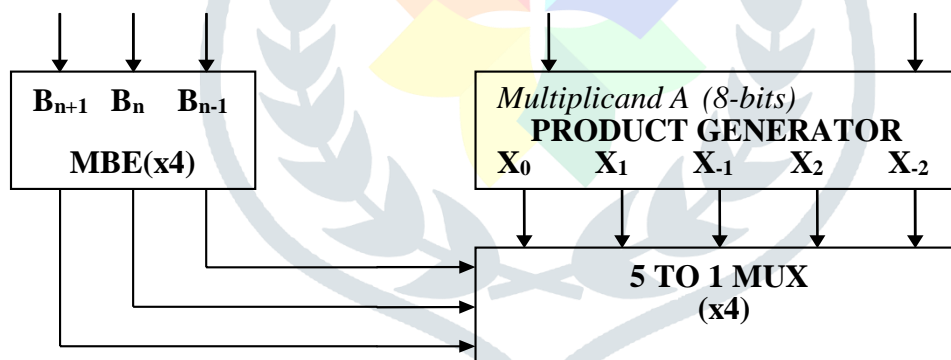
### 2.2.3 BOOTH MULTIPLICATION ALGORITHM (radix – 4)

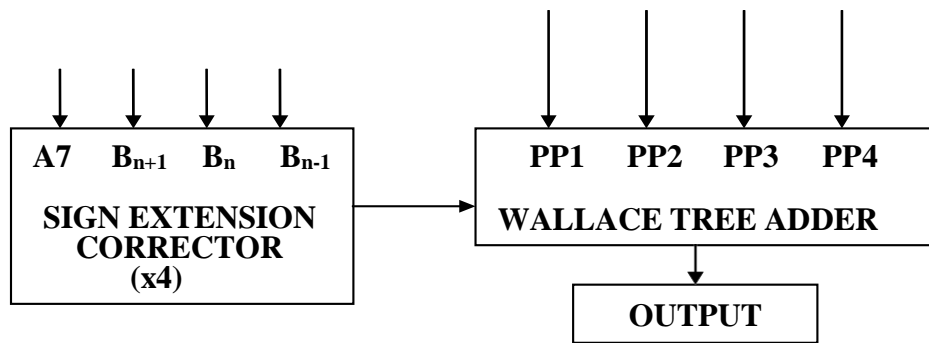
One of the solutions realizing high speed multipliers is to enhance parallelism which helps in decreasing the number of subsequent calculation stages. The Original version of Booth's multiplier (Radix – 2) had two drawbacks.

➤The number of add / subtract operations became variable and hence became inconvenient while designing Parallel multipliers.

➤The Algorithm becomes inefficient when there are isolated 1s .

These problems are overcome by using Radix 4 Booth's Algorithm which can scan strings of three bits with the algorithm given below. The design of Booth's multiplier in this project consists of four Modified Booth Encoded (MBE), four sign extension corrector, four partial product generators (comprises of 5:1 multiplexer) and finally a Wallace Tree Adder. This Booth multiplier technique is to increase speed by reducing the number of partial products by half. Since an 8bit booth multiplier is used in this project, so there are only four partial products that need to be added instead of eight partial products generated using conventional multiplier. The architecture design for the modified





Booths Algorithm used in this project is shown below.

**Figure 2.2 Architecture of designed Booth Multiplier in the Project.**

#### 2.2.4 MODIFIED BOOTH ENCODER (MBE)

Modified Booth encoding is most often used to avoid variable size partial product arrays. Before designing a MBE, the multiplier B has to be converted into a Radix-4 number by dividing them into three digits respectively according to Booth Encoder Table given afterwards. Prior to convert the multiplier, a zero is appended into the Least Significant Bit (LSB) of the multiplier. The figure above shows that the multiplier has been divided into four partitions and hence that mean four partial products will be generated using booth multiplier approach instead of eight partial products being generated using conventional multiplier.

$$Z_n = -2 * B_{n+1} + B_n + B_{n-1}$$

Lets take an example of converting an 8-bit number into a Radix-4 number. Let the number be **-36** = **1 1 0 1 1 1 0 0**. Now we have to append a '0' to the LSB. Hence the new number becomes a 9-digit number, that is **1 1 0 1 1 1 0 0 0**. This is now further encoded into Radix-4 numbers according to the following given table. Starting from right we have **0\*Multiplcand, -1\*Multiplcand, 2\*Multiplcand, -1\*Multiplcand**.

| B <sub>n+1</sub> | B <sub>n</sub> | B <sub>n-1</sub> | Z <sub>n</sub> | Partial Product | 1M | 2M | 3M |
|------------------|----------------|------------------|----------------|-----------------|----|----|----|
| 0                | 0              | 0                | 0              | 0               | 1  | 1  | 0  |
| 0                | 0              | 1                | 1              | 1×Multiplcand   | 0  | 1  | 0  |
| 0                | 1              | 0                | 1              | 1×Multiplcand   | 0  | 1  | 0  |
| 0                | 1              | 1                | 2              | 2×Multiplcand   | 1  | 0  | 0  |
| 1                | 0              | 0                | -2             | -2×Multiplcand  | 1  | 0  | 1  |
| 1                | 0              | 1                | -1             | -1×Multiplcand  | 0  | 1  | 1  |
| 1                | 1              | 0                | -1             | -1×Multiplcand  | 0  | 1  | 1  |
| 1                | 1              | 1                | 0              | 0               | 1  | 1  | 0  |

**Table 2.1**

**Modified Booth Encoder's table to generate M, 2M, 3M control signal**



Table 2.1 shows  $B_{n+1}$ ,  $B_n$  and  $B_{n-1}$  which are three bits wide binary numbers of the multiplier  $B_{in}$  which  $B_{n+1}$  is the most significant bit (MSB) and  $B_{n-1}$  is the least significant bit (LSB).  $Z_n$  is representing the Radix-4 number of the 3-bit binary multiplier number. For example, if the 3-bit multiplier value is “111”, so it means that multiplicand A will be 0. And it's the same for others either to multiply the multiplicand by -1, -2 and so on depending on 3 digit number. And thing to note is generated numbers are all of 9-bit.

From the table 2.1, the  $M$ ,  $2M$  and  $3M$  are the select control signals for the partial product generator. It will determine whether the multiplicand is multiplied by 0, -1, 2 or -2.  $M$  and  $2M$  are designed as an active low circuit which means if let's say the multiplicand should be multiplied by 1 then the  $M$  select signal will be set to low “0” whereas If the multiplicand should be multiplied by 2 then the  $2M$  select signal will be set to low “0”. The  $3M$  is representing the sign bit control signal and its active high circuit which means if the multiplicand should be multiplied by -1 or -2, then the sign,  $3M$  will be set to high “1”.

### 2.2.5 PARTIAL PRODUCT GENERATOR (PPG)

Partial product generator is the combination circuit of the product generator and the 5 to 1 MUX circuit. Product generator is designed to produce the product by multiplying the multiplicand A by 0, 1, -1, 2 or -2. A 5 to 1 MUX is designed to determine which product is chosen depending on the  $M$ ,  $2M$ ,  $3M$  control signal which is generated from the MBE. For product generator, multiply by zero means the multiplicand is multiplied by “0”. Multiply by “1” means the product still remains the same as the multiplicand value. Multiply by “-1” means that the product is the two's complement form of the number. Multiply by “-2” is to shift left one bit the two's complement of the multiplicand value and multiply by “2” means just shift left the multiplicand by one place.

### 2.2.6 SIGN EXTENSION CORRECTOR

Sign Extension Corrector is designed to enhance the ability of the booth multiplier to multiply not only the unsigned number but as well as the signed number. As shown in Table 2.2 when bit 7 of the multiplicand A ( $A_7$ ) is zero (unsigned number) and  $B_{n+1}$  is equal to one, then sign E will have one value (become signed number for resulted partial product). It is the same when the bit 7 of the multiplicand A ( $A_7$ ) is one (signed number) and  $B_{n+1}$  is equal to zero, the sign E will have a new value. However when both the value of  $A_7$  and  $B_{n+1}$  are equal either to zero or one, the sign E will have a value zero (unsigned number). For the case when all three bits of the multiplier value  $B_{n+1}$ ,  $B_n$  and  $B_{n-1}$  are equal to zero or one, the sign E will directly have a zero value independent to the  $A_7$  value. The table for the Sign Extension Corrector is shown below.

TABLE 2.2 (A) Sign E when A&amp; is Zero

| A7 | Bn+1 | Bn | Bn-1 | E |
|----|------|----|------|---|
| 0  | 0    | 0  | 0    | 0 |
| 0  | 0    | 0  | 1    | 0 |
| 0  | 0    | 1  | 0    | 0 |
| 0  | 0    | 1  | 1    | 0 |
| 0  | 1    | 0  | 0    | 1 |
| 0  | 1    | 0  | 1    | 1 |
| 0  | 1    | 1  | 0    | 1 |
| 0  | 1    | 1  | 1    | 0 |

TABLE 2.2 (B) Sign E when A&amp; is One

| A7 | Bn+1 | Bn | Bn-1 | E |
|----|------|----|------|---|
| 1  | 0    | 0  | 0    | 0 |
| 1  | 0    | 0  | 1    | 1 |
| 1  | 0    | 1  | 0    | 1 |
| 1  | 0    | 1  | 1    | 1 |
| 1  | 1    | 0  | 0    | 0 |
| 1  | 1    | 0  | 1    | 0 |
| 1  | 1    | 1  | 0    | 0 |
| 1  | 1    | 1  | 1    | 0 |

## 2.3 EXISTED SYSTEM

### 2.3.1 Binary Extension Field $GF(2^m)$

A finite field is defined as a set of finite many elements, where addition and multiplication are the operations defined on the set. A binary extension field,  $GF(2^m)$ , is generated by a degree  $m$  irreducible polynomial,  $p(x) = x^m + p_{m-1}x^{m-1} + \dots + p_2x^2 + p_1x + 1$ , where  $p_i$  is either 0 or 1.  $p(x)$  also specifies a PB  $\{1, x, x^2, \dots, x^{m-1}\}$ . Each element of  $GF(2^m)$  can be represented as a polynomial of degree at most  $m - 1$  over  $GF(2^m)$  with respect to the PB. For instance, an element  $A$

$\in GF(2^m)$  can be expressed as

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_2x^2 + a_1x + a_0$$

(1)

With  $a_i \in \text{GF}(2)$ ,  $0 \leq i \leq m-1$ .

Multiplication of two field elements  $A(x)$  and  $B(x)$  of the

binary extension field can be given by

$$(x) = A(x) B(x) \bmod p(x).$$

### 2.3.2. Digit-Serial PB Multiplication

In digit-serial multiplication, the bits of one operand are divided into digits of size  $k$  while the bits of the other input operand are processed in parallel. Only one digit of the first operand is accessible in each clock cycle.

MSD digit-serial multiplication can be realized in several ways depending on when and where polynomial modular reduction is performed.

### 2.3.3. Power Dissipation for CMOS-Based Circuits

Power consumption in a CMOS-based design contains two major components: static power and dynamic power. For a CMOS-based design, dynamic power plays a dominant role in the total power consumption. Dynamic power consumption of a CMOS-based design

Switching activity,  $\alpha_i$ , denotes the probability of a  $0 \rightarrow 1$  transition during a clock period on the output node of cell  $i$ .  $C_{Li}$  represents total load capacitance at the output of cell  $i$  and  $P$  is a variable independent of switching activity and load capacitance, but related to clock frequency of the circuit, and supply voltage.

The second term,  $P_{\text{internal}}$ , in (4) is the total internal power obtained by summing over all cells. The internal power of each cell  $i$  ( $P_{\text{internal}i}$ ) is the power consumed within the cell because of the charging and discharging of internal nodes capacitances of a cell and short-circuit current. During the transitions of the input signals and the output node for a short period of time, both pull-up and pull-down paths in the CMOS cell conduct and the current flows from  $V_{DD}$  to GND. The current is called short-circuit current. It can be concluded that  $P_{\text{internal}}$  is also a function of switching activity and

correlates positively with the switching activity. As can be seen in (4), dynamic power ( $P_{\text{dynamic}}$ ) can be reduced by lowering  $P_{\text{switching}}$  or  $P_{\text{internal}}$  or the both.

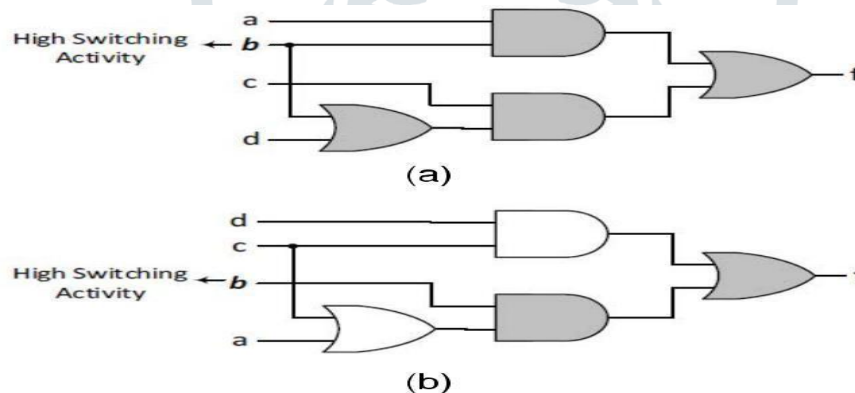
In this paper, we have utilized a factoring technique, which is explained later, to reduce  $P_{\text{switching}}$  by minimizing the switching activities ( $\alpha_i$ s). We have also reduced  $P_{\text{internal}}$  through logic gate substitution in which gates with larger number of internal nodes are replaced by gates with smaller number of internal nodes.

### 2.3.4. Low-Power Techniques for Dynamic Power Reduction

Factoring is an effective method to reduce power consumption applicable at both architecture and gate level. This method reduces  $P_{\text{switching}}$  by reducing the logic depth, which is connected to the nets with high switching activity [38] to reduce the switching activity of the circuit.

Fig. 1 shows a simple gate level example of using factoring to reduce the switching activity of a small circuit. Both circuits shown in Fig. 1 realize function  $f = ab + cb + cd$ . Assume that input  $b$  has higher switching activity compared with the three other inputs,  $a$ ,  $c$ , and  $d$ .

In Fig, input  $b$  with high switching activity propagates through two gates at the first stage, which results in larger number of high activity nets, and as a result, it causes higher switching activity in the whole circuit. While in the circuit shown in Fig, input  $b$  propagates through one gate at the second stage, and thus, it results in lower number of high activity nets. Therefore, the circuit shown in Fig has lower switching activity, and thus, it has lower  $P_{\text{switching}}$  compared with the circuit presented in Fig. Logic gate substitution reduces  $P_{\text{internal}}$  by replacing the gates with higher internal power with those that consume lower internal power.



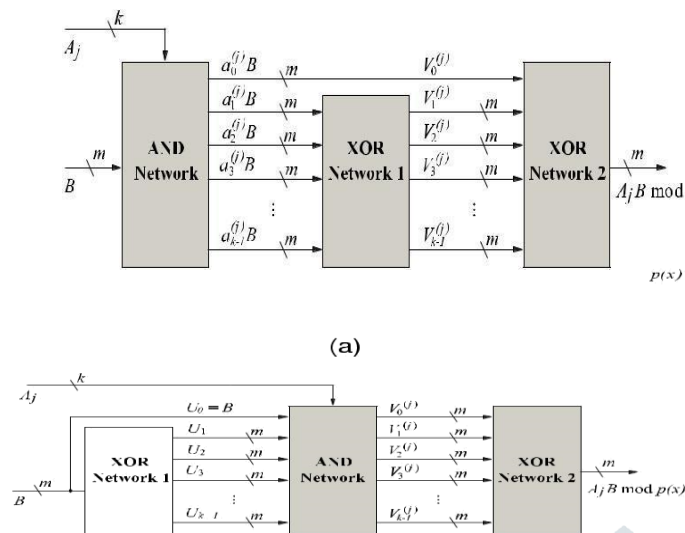
**Fig 2.3 Existed Low Power Design of a Digital Serial Multiplier in  $GF(2^m)$**

In this section, we present a factoring-based circuit design for a digit-serial PB multiplier in  $GF(2^m)$  that reduces  $P_{\text{switching}}$  effectively. A logic gate substitution technique is also presented that reduces  $P_{\text{internal}}$  by using gates with lower internal power consumption. Gate count of the proposed digit-serial PB multiplier is also optimized.

### 2.3.5. Multiplier Architecture

An architecture diagram for the proposed digit-serial PB multiplier in  $GF(2^m)$ . There are three modules, namely,  $k \times m$  multiplier, constant multiplier, and field adder.

1)  $k \times m$  multiplier takes one operand  $B$  of  $m$ -bit and the other operand  $A_j$  of  $k$ -bit. Note that  $A_j$  changes for different clock cycles  $j$ . Thus, it has higher switching activity compared with operand  $B$ . A straightforward realization of this module was used. For the comparison purpose, it is given in Algorithm 1. Note that a modification to this algorithm using a factoring method is proposed. The three steps in Algorithm 1 are, respectively, realized with the circuit blocks from left to right.



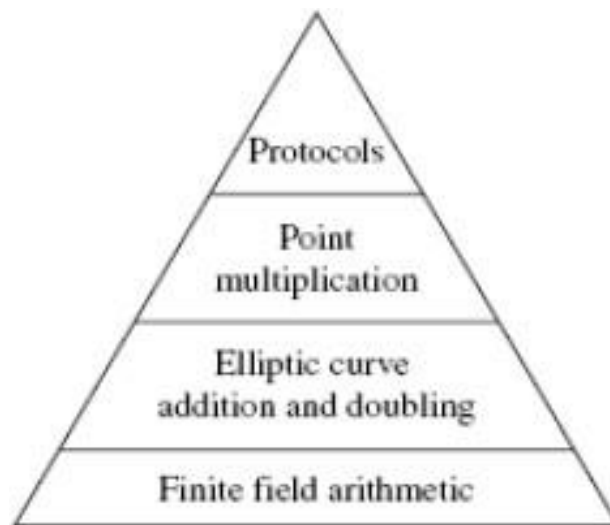
| $k \times m$ multiplier | $P_{dynamic}$                             |                                          |
|-------------------------|-------------------------------------------|------------------------------------------|
|                         | $P_{switching}$<br>( $\mu W$ ) (Relative) | $P_{internal}$<br>( $\mu W$ ) (Relative) |
| without factoring       | 110 (120.4%)                              | 111 (101.8%)                             |
| with factoring          | 91.4 (100%)                               | 109 (100%)                               |

- 2) Constant multiplier module realizes multiplication between a field element and the constant  $x^k$ .
  - 3) Field adder module implements finite field addition using  $m$  two-input XOR gates formed as a one-layer network.
- Note that  $k \times m$  multiplier is the most complex module among the three modules. In fact, it takes majority of system complexity in terms of gate count. By experiment, we also found that its power consumption is much higher than all the other modules combined.<sup>2</sup> In the following, we will propose a low-power design of the  $k \times m$  multiplier using factoring and logic gate substitution methods. Complexity optimization of this module is also presented.

## CHAPTER-3 PROPOSED SYSTEM

The operation of cryptographic protocol is point multiplication. The implementation of point multiplication is separated into three distinct layers (1) finite field arithmetic (2) elliptic curve point addition (3) point multiplication technique.

Finite field arithmetic can be designed into any hardware implementation accelerator for finite field arithmetic to perform the higher level functions of elliptic curve point arithmetic. Along with program and data memory, the three components are arithmetic logic unit (AU), an arithmetic unit controller (AUC) and a main controller.

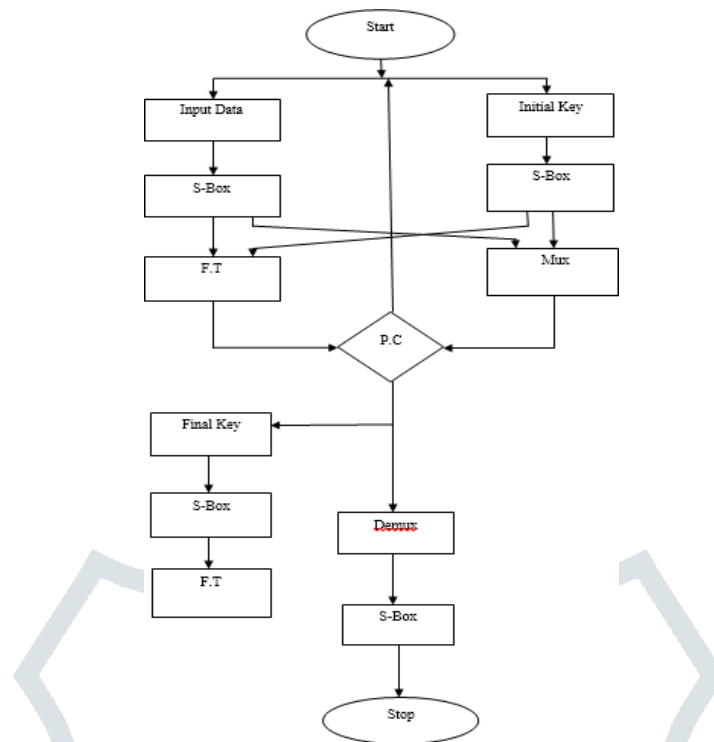


**FIG. 3.1 HIERARCHY OF OPERATIONS IN CRYPTOGRAPHY**

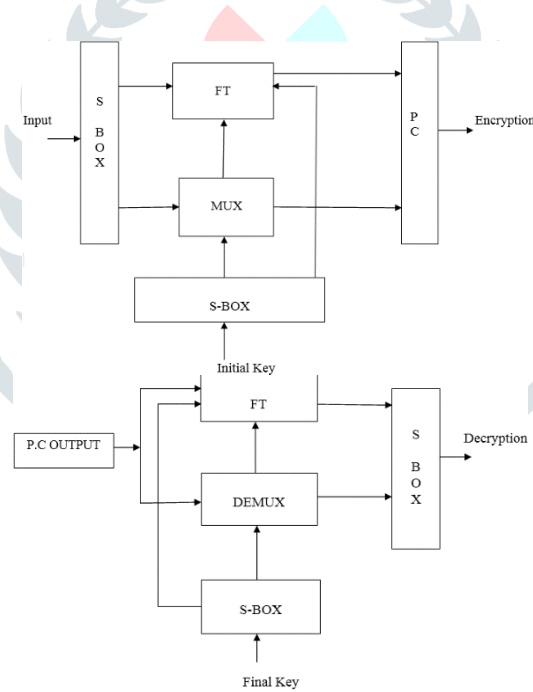
The function of AU is to perform the basic field operation of addition, squaring, multiplication, and inversion, and it is controlled by the AUC. The function of AUC is to execute the elliptic curve operation of point addition and doubling. The micro controller coordinates and executes the method chosen for point multiplication and interacts with the lost system.

In the proposed system the total input bits are converted into s-box and initial key is converted into s-box. The two s-boxes are created at a time and then the data from two s-boxes are given to F.T and mux. The mux consisting of finite field arithmetic, elliptic curve point addition and point multiplication technique. The F.T and mux outputs are mapped to parseval's checks, the checking operation of errors is over come in this parseval's check block and the finalized output encryption data is send to decryption block.





**FIG. 3.2 ALGORITHM**



The decryption block takes P.C outputs and they are given to I.F.T and de-mux. The total data is done the inverse operations of mux and encrypted F.T block. The total data from de-mux and I.F.Tis given to s-box to store the decrypted data.

we proposed an efficient VLSI architecture for advanced encryption standard design methodology in order to provide a high-speed and effective cryptographic operation. High-performance and fast implementation of proposed multiplication is applied to cryptographic systems. The internal multiplier consists of three stages of operations to focuses on final result. In this paper, we propose efficient and high speed architectures to implement cryptography using proposed multiplier.

Cryptography is the operation in wireless communication between transmissions and receiving of data, the secured data is communicated in an unsecured channel between transmitter and receiver with high security. The total proposal is done in XILINX 14.7 with Spartan 3E family.

## **CHAPTER-4 SOFTWARE USED**

### **XILINX SOFTWARE**

#### **4.1 INTRODUCTION TO VERILOG HDL**

In lab, we will be using a hardware description language (HDL) called Verilog. Writing in Verilog lets us focus on the high-level behavior of the hardware we are trying to describe rather than the low-level behavior of every single logic gate.

## 4.2 Design Flow

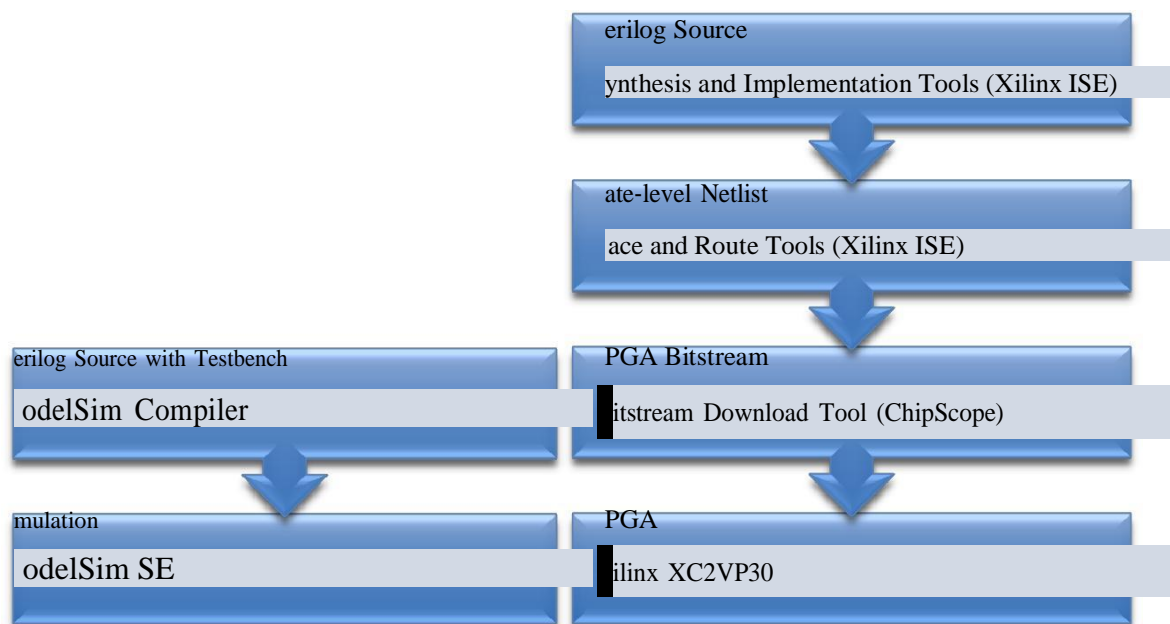


Fig 4.1 Simulation flow (left) and synthesis flow (right)

The design of a digital circuit using Verilog primarily follows two design flows. First, we feed our Verilog source files into a simulation tool, as shown by the diagram on the left. The simulation tool simulates in software the actual behavior of the hardware circuit for certain input conditions, which we describe in a test bench. Because compiling our Verilog for the simulation tool is relatively fast, we primarily use simulation tools when we are testing our design. When we are confident that design is correct, we then use a hardware synthesis tool to turn our high level Verilog code to a low level gate net list. A mapping tool then maps the netlist to the applicable resources on the device we are targeting in our case, a field programmable grid array (FPGA). Finally, we download a bit stream describing the way the FPGA should be reconfigured onto the FPGA, resulting in an actual digital circuit.

## 4.3 Philosophy

Verilog has C-like syntax. However, it is philosophically different than most programming languages since it is used to describe hardware rather than software. In particular verilog statements are concurrent in nature; except for code between begin and end blocks, there is no defined order in which they execute. In comparison, most languages like C consist of statements that are executed sequentially; the first line in main () is executed first, followed by the line after that, and so on. Synthesizable Verilog code is eventually mapped to actual hardware gates. Compiled C code, on the other hand, is mapped to some bits in storage that a CPU may or may not execute.

## 4.4 Synthesizable Combinational Verilog Syntax

### 4.4.1 Modules

The basic building block of Verilog is the module statement. It is somewhat analogous to defining a function in C:

```
module <module_name>(<input_list>, <output_list>); input    <input_list>;
output <output_list>;

endmodule
```

Here is a module that takes in three inputs: two 5-bit operands called a and b, and an enable input called en. The module's name is comparator.

```
module comparator(a, b, en, a_gt_b); input [4:0] a, b;

    input en; output a_gt_b;

endmodule
```

In this state, the module just does nothing, for two reasons. First, there is no code in the body of the module—both the inputs and outputs are dangling. Secondly, defining a module in and of itself does nothing (unless it is the top level module). We need to create an instance of a module in our design to actually use it.

### 4.4.2 Instantiating Modules

We can include an instance of a module within another module using the following syntax:

```
<module_name><instance_name>(<port_list>);
```

For example, to instantiate a comparator module with the name comparator1, input wires in1, in2, and en, and an output wire gt, we could write:

```
comparator comparator1 (in1, in2, en, gt);
```

This instantiation depends on the ordering of the ports in the comparator module. There is an alternate syntax for instantiating modules which does not depend on port ordering, and is thus usually vastly preferred. The syntax is:

```
<module_name><instance_name>(.<port_name>(ioname), ...);
```

Continuing from the last example, we could instead write:

```
comparator comparator1(.b(in2), .a(in1), .en(en),
.a_gt_b(gt));
```

Notice that although we switched the order of ports b and a in this example, the instantiation will still work because we have named which ports we are connecting to.

### 4.4.3 Comments

Comments in Verilog are exactly the same as in C.

```
// This is a comment /* Multi-line
comment */
```

### 4.4.4 Numerical Literals

Many modules will contain numerical literals. In Verilog, numerical literals are unsigned 32-bit numbers by default, but in this class you should probably get into the habit of declaring the width of each numerical literal. This leads to less guesswork when, for example, you concatenate a wire and a numerical literal together (as shown later). Here are a few example numerical literals:

```
/* General syntax: <bits>'<base><number>
where<base> is generally b, d, or h */

ire [2:0]      3'b111;           3    it binary
ire  :0]      5'd31;            it decimal
ire  1:0] c    32'hdeadbeef; //    2 bit hexadecimal
```

### 4.4.5 Constants

We can use `define to define global constants in our code (like the #define preprocessor directive in C). Note that unlike C, when referencing the constant, we need to append a back tick to the front

of the constant: e.g., in our case we had to use `FRI instead of FRI. Also, do not append a semicolon to the `define statement.

```

define RED      b00  DON'T add a semicolon to these
define WHITE    b01  statements, just as with C's #define
define BLUE     b10

wire [1:0] color1 = RED;
wire [1:0] color2 = WHITE;
wire [1:0] color3 = BLUE;

```

#### 4.4.6 Wires

To start with, we will declare two kinds of data types in our modules: wires and registers. You can think of wires as modeling physical wires—you can connect them either to another wire, an input or output port on another module, or to a constant logical value. To declare a wire, we use the wire statement:

```

wire      a_wire;
wire [1:0] two_bit_wire;
wire [4:0] five_bit_wire;

```

We then use the assign statement to connect them to something else. Assuming that we are in a module that takes a two bit input named two\_bit\_input, we could do the following:

```

assign two_bit_wire = two_bit_input;
// Connect a_wire to the lowest bit of two_bit_wire assign a_wire = two_bit_wire[0];

/* {} is concatenation – 3 MSB will be 101, 2 LSB will be connected to two_bit_wire */ assign five_bit_wire =
{3'b101, two_bit_wire};

1.4.2 This is an error! You cannot assign a wire twice!
1.4.3 assign a_wire = 1'b1;

```

Note that these are continuous assignments. That means that in the previous example, whenever the input two\_bit\_input changes, so do the values of two\_bit\_wire, a\_wire, and five\_bit\_wire. There is no “order” by which



they change—the changes occur at the same time. This is also why you cannot assign the same wire twice in the same module—a wire cannot be driven by two different signals at the same time. This is what we mean when saying Verilog is “naturally concurrent.”

Finally, there is a shortcut that is sometimes used to declare and assign a wire at the same time:

```
// Declares gnd, and assigns it to 0 wire gnd = 1'b0;
```

#### 4.4.7 Registers

The other data type we will use is register. Despite the name, registers do not imply memory. They are simply a language construct denoting variables that are on the left hand side of an always block (and in simulation code, initial and forever blocks). You declare registers, like wires, at the top level of a module, but you *use* them within always blocks. You cannot assign registers values at the top level of a module, and you cannot assign wires while inside an always block.

#### 4.4.8 Always Blocks

Always blocks are blocks which model behavior that occurs repeatedly based on a sensitivity list. Whenever a signal in the sensitivity list changes values, the statements in the always block will be run sequentially in the simulator. In terms of actual hardware, the synthesis tool will synthesize circuits that are logically equivalent to the statements within the always block.

In the degenerate case, a register in an always statement acts like a wire data type, as in this simple module:

```
module bitwise_not(a_in, a_out);
```

```
input [1:0] a_in; output [1:0] a_out;
```

```
/* Declare the 2-bit output a_out as a register, since it is used on the LHS of an always block */ reg [1:0] a_out;
```

```
// better to use always @* – see next example always @(a_in) begin a_out = ~a_in; // out = bitwise not of in end
```

```
endmodule
```

So whenever the input `a_in` changes, the code within the `always` block is evaluated—`a_out` takes the value of `a_in`. It is as if we declared `a_out` to be a wire, and assigned it to be `~a_in`.

#### 4.4.9 Initial Blocks

Initial and forever blocks are like `always` blocks in that the statements within an initial block execute in order when triggered. Also, only registers are allowed on the left hand side of an initial block. However, while an `always` block executes every time a condition changes, initial blocks are executed once—at the beginning of the program. The following code sets `opcode`, `op_a`, and `op_b` to 0, 10, and 20 respectively at `t=0` in the simulation, and then changes those values to 2, 10, and 20 respectively at `t=5` in the simulation:

```
reg [2:0] opcode;
reg [4:0] op_a, op_b;

initial begin opcode = 3'b000; op_a = 5'd10; op_b = 5'd20;

#5 opcode = 3'b010; op_a
```

In this state, the module just does nothing, for two reasons. First, there is no code in the body of the module—both the inputs and outputs are dangling. Secondly, defining a module in and of itself does nothing (unless it is the top level module). We need to create an instance of a module in our design to actually use it.

#### 4.5 PROCEDURE

This XILINX software is used for creating, synthesizing, simulating, implementing, and downloading a simple VHDL design using the XILINX Project Navigator.

##### 4.5.1 Create a New Project

Create a new ISE project which will target the FPGA device on the Spartan-3 Startup Kit demo board.

To create a new project:

- Select File > New Project... The New Project Wizard appears.
- Type tutorial in the Project Name field.
- Enter or browse to a location (directory path) for the new project. A tutorial subdirectory is created automatically.
- Verify that HDL is selected from the Top-Level Source Type list.
- Click Next to move to the device properties page.
- Fill in the properties in the table as shown below fig:

◆ Product Category: All

- ◆ Family: Spartan3A and Spartan3AN
  - ◆ Device: XC3S50A
  - ◆ Package: TQ144
  - ◆ Speed Grade: -5
  - ◆ Top-Level Source Type: HDL
  - ◆ Synthesis Tool: XST (VHDL/Verilog)
  - ◆ Simulator: ISIM (VHDL/Verilog)
  - ◆ Preferred Language: Verilog (or VHDL)
  - ◆ Verify that Enable Enhanced Design Summary is selected.
- Leave the default values in the remaining fields.

### Project Settings

Specify device and project properties.  
Select the device and design flow for the project

| Property Name                          | Value                    |
|----------------------------------------|--------------------------|
| Evaluation Development Board           | None Specified           |
| Product Category                       | All                      |
| <b>Family</b>                          | Spartan3A and Spartan3AN |
| Device                                 | XC3S50A                  |
| Package                                | TQ144                    |
| Speed                                  | -5                       |
| Top-Level Source Type                  | HDL                      |
| Synthesis Tool                         | XST (VHDL/Verilog)       |
| Simulator                              | ISim (VHDL/Verilog)      |
| Preferred Language                     | Verilog                  |
| Property Specification in Project File | Store all values         |
| Manual Compile Order                   | <input type="checkbox"/> |
| VHDL Source Analysis Standard          | VHDL-93                  |
| Enable Message Filtering               | <input type="checkbox"/> |

More Info      Next      Cancel

**Fig. 4.2: Project Device Properties**

- Click Next to proceed to the Create New Source window in the New Project Wizard. At the end of the next section, your new project will be complete.

### 4.5.2 Starting the Project Navigator

Next, create a new source by selecting File

New Project. The following “New Project” window appears

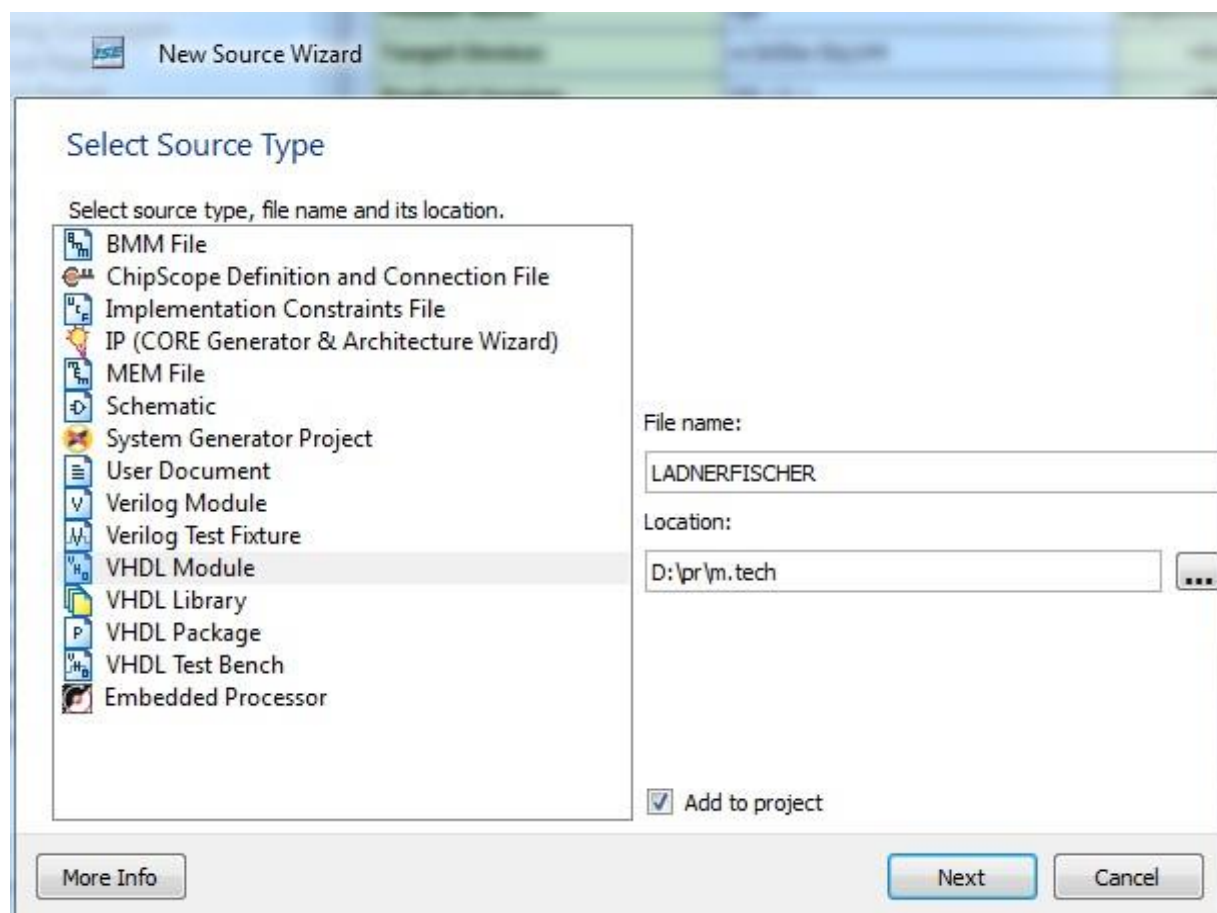
In this section, you will create the top-level HDL file for your design. Determine the language that you wish to use for the tutorial. Then, continue either to the “Creating a VHDL Source” section below, or skip to the “Creating a Verilog Source” section.

#### Creating a VHDL Source

Create a VHDL source file for the project as follows:

1. Click the New Source button in the New Project Wizard.
2. Select VHDL Module as the source type.
3. Type in the file name counter as shown below fig.
4. Verify that the Add to project checkbox is selected.
5. Click Next.





**Fig. 4.3: create new files window**

6. Declare the ports for the counter design by filling in the port information as shown below fig.4.4

**Define Module**

Specify ports for module.

Entity name: LADNERFISCHER

Architecture name: Behavioral

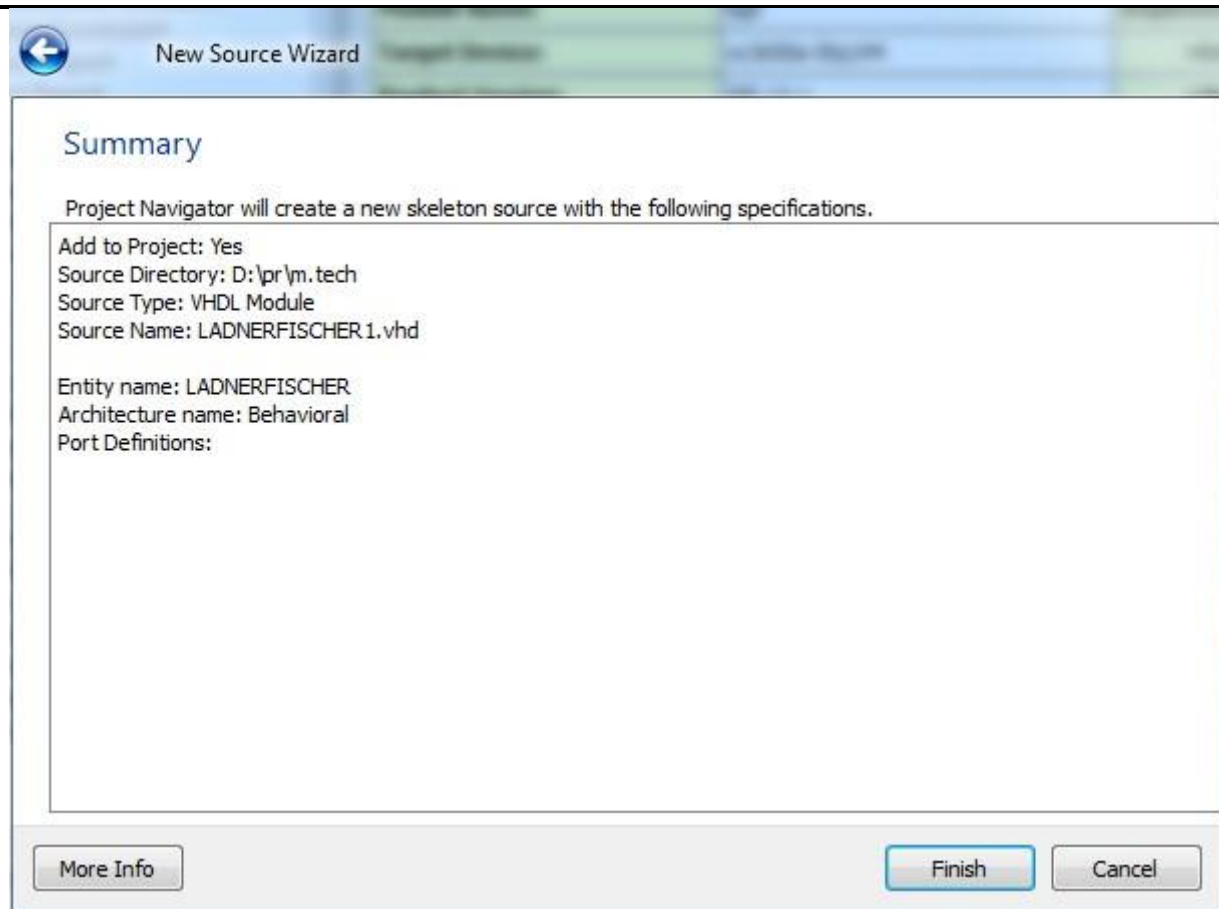
| Port Name | Direction | Bus                      | MSB | LSB |
|-----------|-----------|--------------------------|-----|-----|
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |
|           | in        | <input type="checkbox"/> |     |     |

More Info      Next      Cancel

**Fig. 4.4: declaring ports**

7. Click Next, then Finish in the New Source Wizard - Summary dialog box to complete the new source file template shown below fig. 4.5.





**Fig. 4.5: Summary dialog box**

8. Click Next, then Next, then Finish.

The source file containing the entity/architecture pair displays in the Workspace, and the counter displays in the Source tab, as shown fig. 4.6.

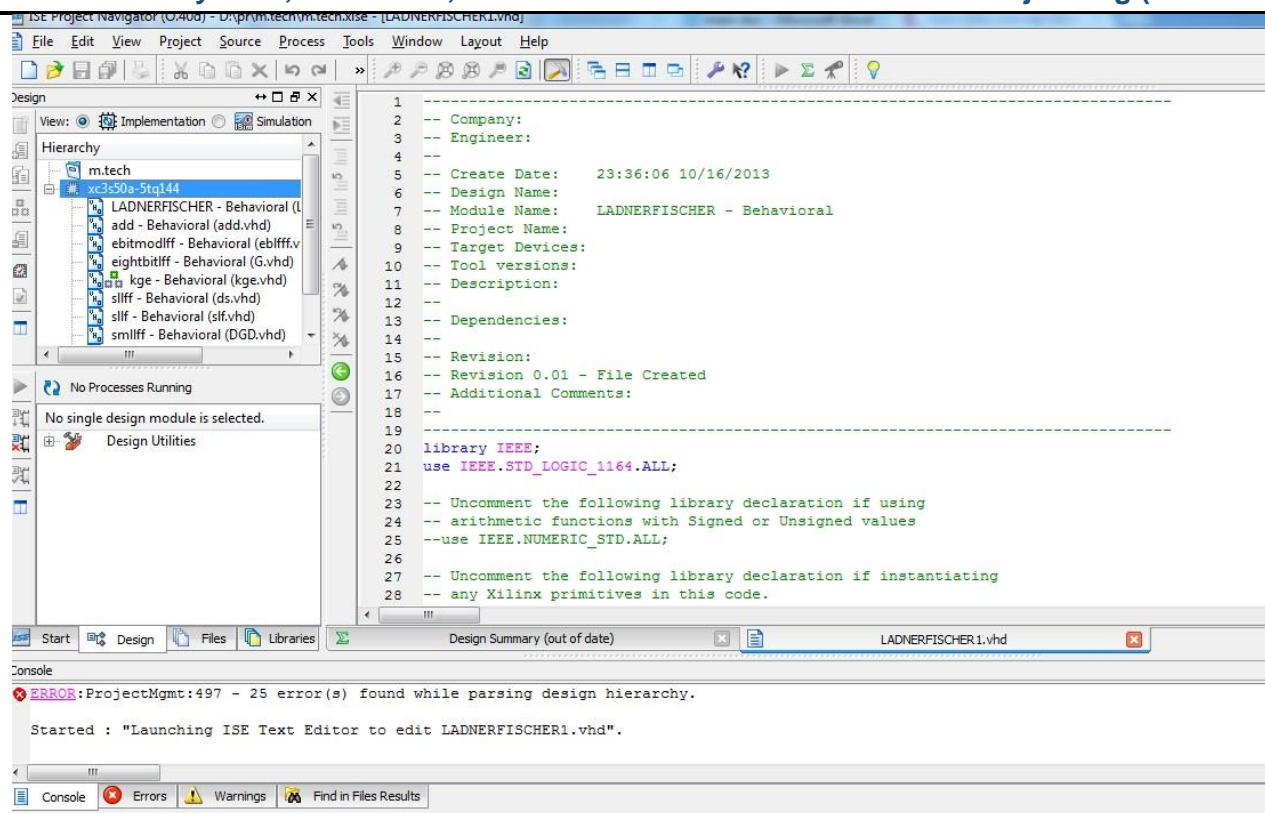
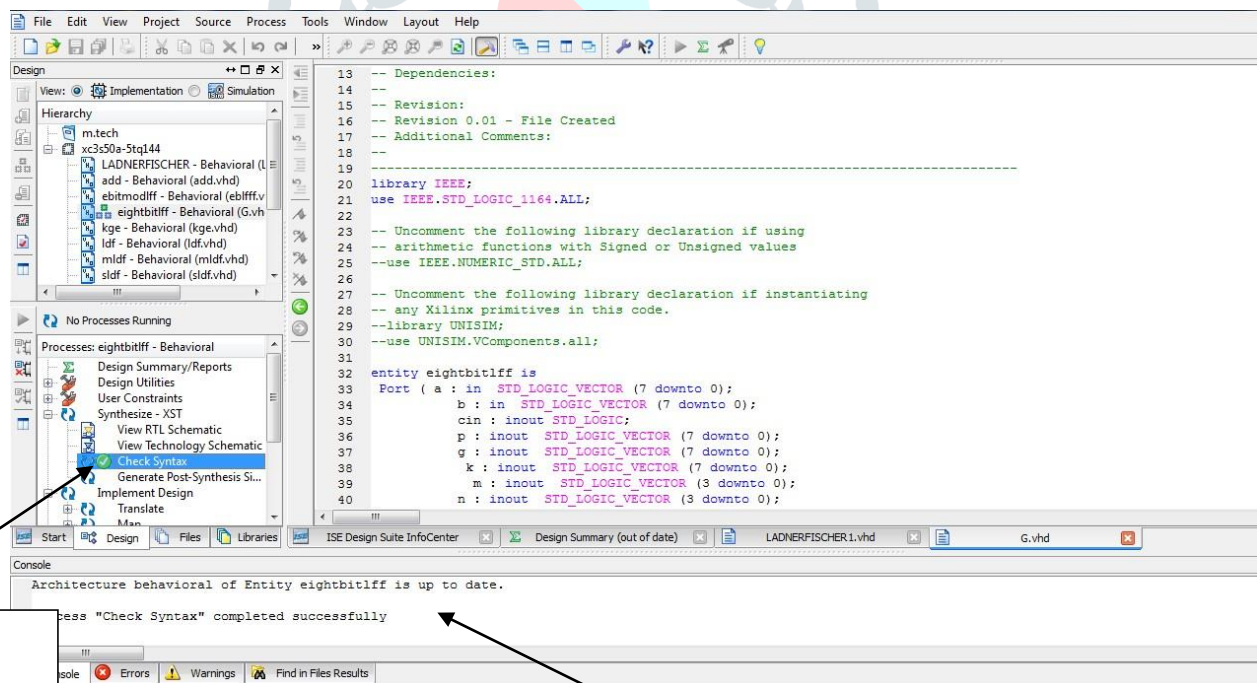


Fig. 4.6: New Project in ISE



A check mark indicates check syntax.

The "Process view" window displays messages to the user.

### 4.5.3 Checking Syntax and Simulating

Check the Syntax of the code that has just typed. Follow the instructions and on the sametime keep an eye on Fig. 4.7.

First, check that top file is selected (highlighted) under the Sources in Project window shown in Fig. 4.7. If it is not, then select it by left clicking on it. Next, locate the Processes for source window on the left of the screen (If the window is not visible, enable it by selecting view ☐ processes). On this window, locate the synthesize menu and expand it by pressing on the +. Locate the “Check Syntax” command under the synthesize menu. Left double-click on “CheckSyntax” and wait for the software to finish checking the code.

**Fig. 4.7: A check mark indicates check syntax**

### 4.5.4 Synthesizing

First, check that the top file is selected (highlighted) under the Sources in Project window, as shown for “DFF.vhd” in Figure. If it is not, then select it by left clicking on it. Next, synthesize the design by left double-clicking on “Synthesize XST”, located in the Process for Source window. If there are any errors, fix the errors and re-synthesize the corrected code again.

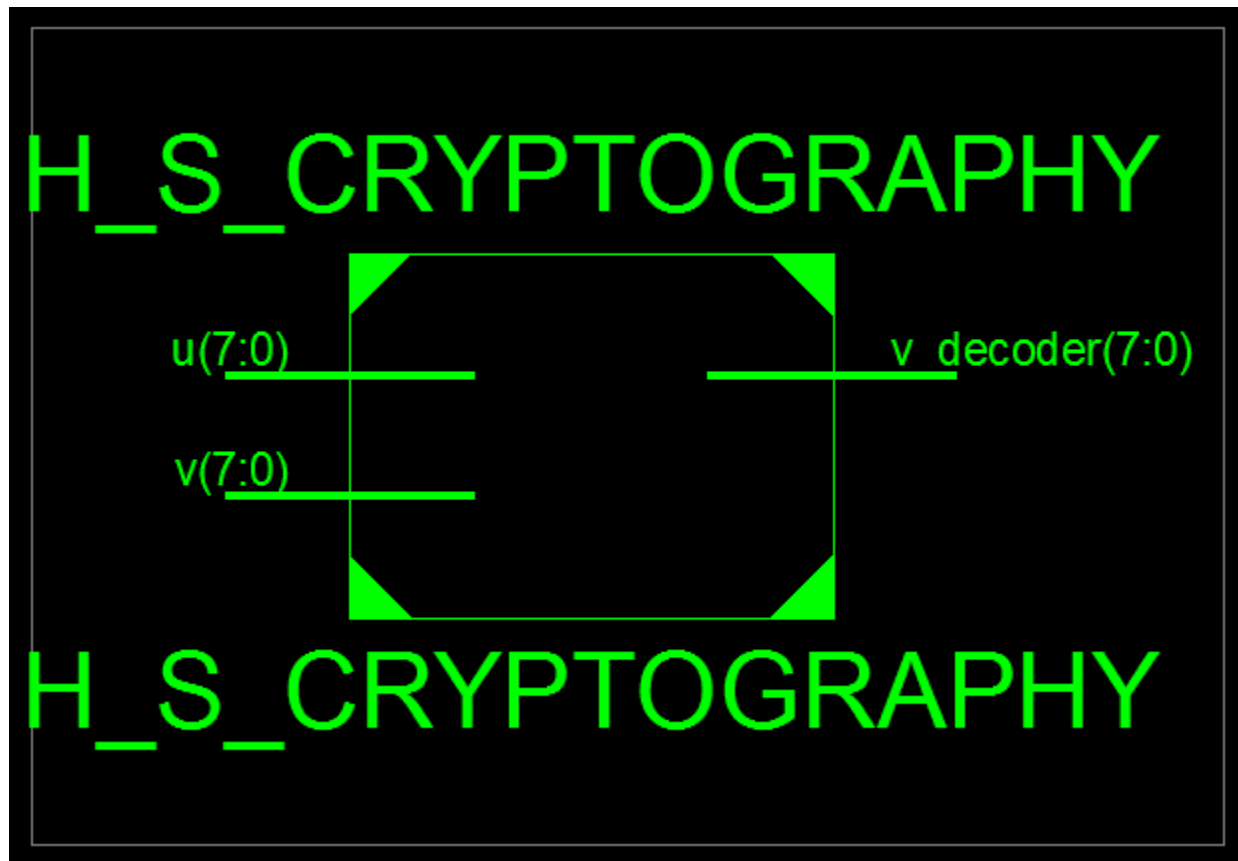
There are several options for synthesis that can change to optimize the design. Right click on “Synthesize XST” and click on properties. The main options that are interested are: Optimization Goal, Optimization Effort, and FSM Encoding Algorithm. The first two options can be found under the Synthesis Options tab. By selecting Speed in this option box, Xilinx will try to synthesize the code to produce faster design. By selecting Area, Xilinx will sacrifice speed and try to build the smallest design possible. The Optimization Effort option can be set to Normal or High, and tells Xilinx just how “hard to try” to get a design that is as fast as possible or as small as possible. Clicking on the HDL Options tab will find the FSM Encoding Algorithm at the top.

If the synthesizer generates any errors, then synthesis will fail and errors have to be fixed in the code before re-synthesize. If the synthesizer succeeds, then it is still important to check if the synthesizer has generated any warnings. Check for warnings (and errors) in the Synthesis Report.

After you have synthesized your design, double left-click on “View Synthesis Report.” The report should contain a summary of any errors or warnings that were generated. The report should not contain any latches as these can cause timing problems, when not properly used. If the report contains latch warnings, then it need to go over your code, and fix these latches, and re- synthesize the code again.

## CHAPTER-5 RESULTS

### 5.1 RTL SCHEMATIC



#### REPORT

Release 14.7 - xst P.20131013 (nt)

Copyright (c) 1995-2013 Xilinx, Inc. All rights reserved.

--> Parameter TMPDIR set to xst/projnav.tmp

Total REAL time to Xst completion: 0.00 secs Total CPU time to Xst completion: 0.12 secs

--> Parameter xsthdpdir set to xst

Total REAL time to Xst completion: 0.00 secs Total CPU time to Xst completion: 0.12 secs

--> Reading design: H\_S\_CRYPTOGRAPHY.prj

#### TABLE OF CONTENTS

- 1) Synthesis Options Summary
- 2) HDL Parsing
- 3) HDL Elaboration

## 4) HDL Synthesis

## 4.1) HDL Synthesis Report

## 5) Advanced HDL Synthesis

## 5.1) Advanced HDL Synthesis Report

## 6) Low Level Synthesis

## 7) Partition Report

## 8) Design Summary

## 8.1) Primitive and Black Box Usage

## 8.2) Device utilization summary

## 8.3) Partition Resource Summary

## 8.4) Timing Report

## 8.4.1) Clock Information

## 8.4.2) Asynchronous Control Signals Information

## 8.4.3) Timing Summary

## 8.4.4) Timing Details

## 8.4.5) Cross Clock Domains Report

## \* Synthesis Options Summary

## ---- Source Parameters

Input File Name: "H\_S\_CRYPTOGRAPHY.prj" Ignore Synthesis Constraint File : NO

## ---- Target Parameters

Output File Name: "H\_S\_CRYPTOGRAPHY"

Output Format: NGC

Target Device: xc6vlx75tl-1L-ff484

## ---- Source Options

Top Module Name: H\_S\_CRYPTOGRAPHY Automatic FSM Extraction : YES

FSM Encoding Algorithm : AutoSafe Implementation

: No FSM Style

: LUT

RAM Extraction : Yes RAM Style : Auto ROM Extraction

: Yes Shift Register Extraction

: YESROM Style : Auto

Resource Sharing : YES Asynchronous To Synchronous

: NOShift Register Minimum Size

2

Use DSP Block : Auto Automatic Register Balancing

: No

---- Target Options

LUT Combining : Auto

Reduce Control Sets : Auto

Add IO Buffers : YES

Global Maximum Fanout 100000

Add Generic Clock Buffer(BUFG)

32

Register Duplication : YES Optimize Instantiated Primitives : NOUse Clock Enable : Auto

Use Synchronous Set : Auto

Use Synchronous Reset : Auto Pack IO Registers into IOBs

: AutoEquivalent register Removal

: YES

---- General Options

Optimization Goal : Speed

Optimization Effort 1

Power Reduction : NO

Keep Hierarchy : No

Netlist Hierarchy : As\_Optimized

RTL Output : Yes

Global Optimization : AllClockNetsRead Cores

: YES

Write Timing Constraints : NOCross Clock Analysis : NO

Hierarchy Separator : /Bus Delimiter : &lt;&gt;



Case Specifier : MaintainSlice Utilization Ratio 100

BRAM Utilization Ratio 100

DSP48 Utilization Ratio 100

Auto BRAM Packing : NOSlice Utilization Ratio Delta 5

\* HDL Parsing \*

Analyzing Verilog file "C:\.Xilinx\SHARON\VXCV.v" into library workParsing module <ACS>.

Analyzing Verilog file "C:\.Xilinx\SHARON\VCXVXC.v" into library workParsing module <PM>.

Analyzing Verilog file "C:\.Xilinx\SHARON\NVCX.v" into library workParsing module <BMC>.

Analyzing Verilog file "C:\.Xilinx\SHARON\NNN.v" into library workParsing module <ASMU>.

Analyzing Verilog file "C:\.Xilinx\SHARON\CCCV.v" into library workParsing module <AACS>.

Analyzing Verilog file "C:\.Xilinx\SHARON\CC.v" into library workParsing module <APM>.

Analyzing Verilog file "C:\.Xilinx\SHARON\C.v" into library workParsing module <SMU>.

Analyzing Verilog file "C:\.Xilinx\SHARON\BB.v" into library workParsing module <ABMC>.

Analyzing Verilog file "C:\.Xilinx\SHARON\CXXVXC.v" into library work Parsing module <H\_S\_CRYPTOGRAPHY>.

Summary:no macro.

Unit <H\_S\_CRYPTOGRAPHY> synthesized.Synthesizing Unit <ABMC>.

Related source file is "C:\.Xilinx\SHARON\BB.v".Summary:no macro.

Unit <ABMC> synthesized.Synthesizing Unit <APM>.

Related source file is "C:\.Xilinx\SHARON\CC.v".

Summary: no macro.Unit <APM> synthesized.

Synthesizing Unit <SMU>.

Related source file is "C:\.Xilinx\SHARON\C.v".

WARNING:Xst:647 - Input <b> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.



Summary: no macro.Unit <SMU> synthesized.

HDL Synthesis ReportFound no macro

Advanced HDL Synthesis ReportFound no macro

\* Low Level Synthesis \*

Optimizing unit <H\_S\_CRYPTOGRAPHY> ...

Mapping all equations...

Building and optimizing final netlist ...

Found area constraint ratio of 100 (+ 5) on block H\_S\_CRYPTOGRAPHY, actual ratio is 0.Final Macro Processing

...

Final Register ReportFound no macro

\* Partition Report \*

Partition Implementation Status

No Partitions were found in this design.

\* Design Summary \*

Top Level Output File Name : H\_S\_CRYPTOGRAPHY.ngc

Primitive and Black Box Usage:

# IO Buffers 16

# IBUF 8

# OBUF 8

Device utilization summary:

Selected Device : 6vlx75t1ff484-11

Slice Logic Utilization:

Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 0

Number with an unused Flip Flop: 0 out of 0

Number with an unused LUT: 0 out of 0

Number of fully used LUT-FF pairs: 0 out of 0 Number of unique control sets: 0

IO Utilization:

Number of IOs: 24

-----

Number of bonded IOBs: 16 out of 240 6% Specific Feature Utilization:

Partition Resource Summary:

No Partitions were found in this design.

Timing Report

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.

FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

No clock signals found in this design

Asynchronous Control Signals Information:

No asynchronous control signals found in this design Timing Summary:

Speed Grade: -1

Minimum period: No path found

Minimum input arrival time before clock: No path found Maximum output required time after clock: No path found

Maximum combinational path delay: 0.405ns

Timing Details:

All values displayed in nanoseconds (ns)

Timing constraint: Default path analysis

Total number of paths / destination ports: 8 / 8

Delay: 0.405ns (Levels of Logic = 2)

Source: u<7> (PAD) Destination: v\_decoder<7> (PAD)

Data Path: u<7> to v\_decoder<7>

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

IBUF:I->O 1 0.003 0.399 u\_7\_IBUF (v\_decoder\_7\_OBUF) OBUF:I->O 0.003 v\_decoder\_7\_OBUF

(v\_decoder<7>)

Total 0.405ns (0.006ns logic, 0.399ns route)(1.5% logic, 98.5% route)

Cross Clock Domains Report:

Total REAL time to Xst completion: 5.00 secs Total CPU time to Xst completion: 4.96 secs Total memory usage is

208252 kilobytes Number of errors : 0 ( 0 filtered)

Number of warnings : 75 ( 0 filtered) Number of infos : 34 ( 0 filtered)

## 5.2 OUTPUT

| Name            | Value       | 1,999,995 ps | 1,999,996 ps | 1,999,997 ps     | 1,999,998 ps | 1,999,999 ps |
|-----------------|-------------|--------------|--------------|------------------|--------------|--------------|
| u[7:0]          | 00101010    |              |              | 00101010         |              |              |
| v[7:0]          | 01010011    |              |              | 01010011         |              |              |
| v_encoder[15:0] | 11010101000 |              |              | 1101010100000000 |              |              |
| v_decoder[7:0]  | 00101010    |              |              | 00101010         |              |              |
| d[7:0]          | 00000010    |              |              | 00000010         |              |              |
| e[7:0]          | 11010101    |              |              | 11010101         |              |              |
| f[7:0]          | 00000010    |              |              | 00000010         |              |              |
| g[7:0]          | 00000000    |              |              | 00000000         |              |              |
| h[7:0]          | 11010111    |              |              | 11010111         |              |              |
| i[7:0]          | 11111101    |              |              | 11111101         |              |              |
| dd[7:0]         | 01010000    |              |              | 01010000         |              |              |
| ee[7:0]         | 10101111    |              |              | 10101111         |              |              |
| ff[7:0]         | 01010000    |              |              | 01010000         |              |              |
| gg[7:0]         | 00000000    |              |              | 00000000         |              |              |
| hh[7:0]         | ZZZZZZ11    |              |              | ZZZZZZ11         |              |              |
| ii[7:0]         | ZZZZZZ11    |              |              | ZZZZZZ11         |              |              |

## CHAPTER - 6 CONCLUSION

Two new digital level SIPO finite fields multipliers using redundant representation have been proposed. For about 60% of the field sizes with in the practical range of ECC applications, the relationship between extension degree  $m$  and the size of the smallest cyclotomic field,  $(n)$ , in which  $F_2$  can be embedded is expressed as  $n=Tm+1$  for  $T$  even and greater than or equal to. In the case, a specific feature of redundant was used to alleviate the redundancy problem in this representation system. Numerical complexity comparison showed that both new architecture have the lowest delay cost compared with the existing RB architecture. VLSI implementation of the proposed architecture for binary extension field of 233 and three practical digital sizes in 65nm CMOS technology was also presented.

## CHAPTER – 7 REFERENCES

- [1] T.ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", IEEE trans. Inf. Theory, vol. 31, no. 4, pp. 469-472, sep-2006.
- [2] I.F.Blake, G. Seroussi, and N.P.Smart, Elliptic curves in cryptography (London mathematical society lecture note series). Cambridge, U.K:Cambridge Univ. press, 1990.
- [3] A.J.Menezes, p.c. van oorschot, and S.A. Vanstone, Handbook of applied cryptography (Discrete mathematical and its applications). Boca Raton, FL, USA: CRC Press, 1996.
- [4] T.IOTH and S.Tsujji, "A fast algorithm for computing multiplicative inverse in  $GF(2^m)$  using normal basis," Inf. Comput., vol. 78, no. 3, pp. 171-177, 1988.
- [5] E.D.Mastrovito, "VLSI architectures for computations in GF fields," ph.D. dissertation, Dept. Electr. Eng., Linköping Univ., Linköping, Sweden, 1991.