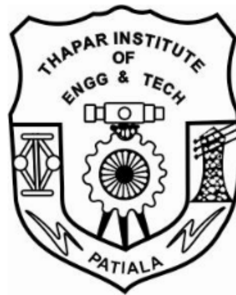
A High Speed Arithmetic Logic Unit using ROMs

THESIS

Submitted in partial fulfillment of the requirements for the award of the degree of

Master of Technology
in
VLSI Design and CAD



Submitted By
Anish Lal
Regn. No. 6040402

Under the supervision of

Mr. Deepak Agarwal
Senior Engineer
LG Electronics Limited

Mrs. Alpana Agarwal
Assistant Professor
TIET,Patiala

DEPARTMENT OF ELECTRONIC AND COMMUNICATION ENGINEERING
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA(PUNJAB) – 147004
June 2006

preamble

Do you not know ?
Have you not heard ?
The Lord is the everlasting God,
the Creator of the ends of the earth.
He will not grow tired or weary,
and his understanding no one can fathom.
He gives strength to the weary
and increases the power of the weak.
Even youths grow tired and weary,
and young men stumble and fall;
but those who hope in the Lord
will renew their strength.
They will soar on wings like eagles;
they will run and not grow weary,
they will walk and not be faint.

Isaiah 40:28-31

The fear of the Lord is the
beginning of wisdom,
and knowledge of the Holy One
is understanding.

Proverbs 9:10

Abstract

There is a flurry of activities going on in designing high speed circuits. Area, speed and power dissipation are crucial parameters for competitive advantage. Fixed point operations like multiplication is an operation essential to signal processing applications such as Fast Fourier Transform (FFT) algorithms, digital filter implementation. Scientific and engineering applications demand exceptionally high floating point performance. For such stringent requirements, design of fast, precise and efficient Arithmetic Logic Unit (ALU) is the goal of every design engineer.

In this thesis work, a novel design of ALU is presented. The design makes use of two read only memories (ROMs) for the purpose of addition and multiplication and thereby can be used for other operations like subtraction and division. An indigenous algorithm is developed to implement the design. ModelSim SE 5.5e and Xilinx ISE 6.1i tools are used for simulation and generating synthesis report. The synthesis report of the proposed design compares device utilization and timing summary of the design implanted on Field Programmable Gate Array (FPGA) and Complex Programmable Logic Devices (CPLD).

Certificate

I **Anish Lal**, hereby certify that the work which is being presented in the thesis entitled, **"A High Speed Arithmetic Logic Unit using ROMs"**, by me in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design and CAD submitted in Department of Electronics and Communication Engineering, Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Mrs. Alpana Agarwal and Mr. Deepak Agarwal.

The matter presented in this thesis has not been submitted in any other University / Institute for the award of any degree.

Anish Lal
Roll No. 6040402

This is to certify that the above statement made by the candidate is correct and true to the best of our knowledge.

Deepak Agarwal
Senior Engineer
LG Electronics Limited
Date……

Alpana Agarwal
Assistant Professor
TIET,Patiala
Date…….

Countersigned by

Head, ECED
T.I.E.T, Patiala
Date…………

Dean of Academic Affairs
T.I.E.T, Patiala
Date…………

Acknowledgement

"THE ESSENCE OF ALL BEAUTIFUL ART,

ALL GREAT ART, IS GRATITUDE"

I first thank my **LORD SHIVA** for graciously giving me his strength and guidance in my life. I owe him everything and I attribute all praise and glory to him. I would love to pay my gratitude to every bud and flower of the garden of my success.

As **Claude Bernard** says, ***" Art is I, Science is We ",*** which summarizes many truths about the importance of being a team during a scientific study. Related to this fact, I would like to thank the following persons that contributed to my thesis work. I am modestly bowing my head in the feet of divine Lord to thank them all for their love, support and sacrifices.

I can not thank **Dr. R.S Kaler** (Head of Department) enough for his role in this work. Thanks, Dear Sir, for giving me the opportunity and a learning platform in **ECED** to tackle this work in the first place. Your creative instincts and feel for the user's needs are reflected through out the work. Your dedication and constant encouragement inspired me to work hard.

Although the Knowledge like electricity pervades everywhere, yet the Teacher is the point where it shines as light. I am extremely lucky to have an opportunity to blossom under the supervision and guidance of **Mrs. Alpana Agarwal** who helped me to grow like Phoenix out of the ashes of my shortcomings and failures. To me, she is not mere a person but a current of love who treaded with me through thick and thin.

I am indebted to **Mr. Kulbir Singh** (Head of Electronics Club Society) for his vision and motivation in guiding me to explore my potential in every sphere of life. He is the ocean of support who uplifted my consciousness by touching the highest recesses of my heart. No word in this universe can help me to acknowledge his role as a Comforter.

I am profoundly grateful to **Mr. Sanjay Batish**. He has always been by my side, ready to talk and to help in finding ways to overcome problems in my work. Beyond that,

he is an inextinguishable source of encouragement when things did not go so well.

I humbly and earnestly would like to thank my batch mates in **TIET**, who acted as a building material in the citadel of my success. I would like to acknowledge the contribution of **Hitesh** and **Narender** (my junior). Thanks dear for your always opportune and heartfelt help and feedback. From deep inside, thanks to **Ghanshyam Sangar** for fun times and technical discussions. I especially thank **Varinder Deepak** for his advice and for sharing the joys and woes of **ALU.**

I would like to thank the faculties of **ECED**, for technical suggestions as well as great collegial working atmosphere. The whole staff of **ECED** was highly cordial and helpful in all possible ways. It would have really difficult for me to have proceeded without their constant encouragement.

Family is the anchor to make anyone stand tall in the crowd. A boat held to its moorings will see the floods pass by; but detached of its moorings, may not survive the flood. The support of all the members of family enthused me to work even while facing the Blues. I take pride of myself in being the son of ideal parents who sacrificed their little joys to bring me to the realization of my dreams and their hopes.

I am laying before you the result of numerous sleepless nights, strenuous hard work, determined and gritty solutions of the fights with my own self but everything is possible only with Almighty's grace as:

"What you make me know,
That alone I know,
What you make me see,
That alone I see."

Anish Lal

Table of Contents

Chapter No.	Title	Page No.
	Preamble	**i**
	Abstract	**ii**
	Certificate	**iii**
	Acknowledgement	**iv-v**
	Table of Contents	**vi-viii**
	List of Figures	**ix-x**
	List of Tables	**xi**
	List of Flow Charts	**xii**
1.	**Introduction**	**1-5**
	1.1 General	1
	1.2 Read Only Memory (ROM)	1
	1.3 Programmable Logic Devices.	2
	1.3.1 Field Programmable Gate Arrays(FPGAs)	2
	1.3.2 Complex Programmable Logic Devices(CPLDs)	2
	1.4 Number Representation	2
	1.5 Formulation of Problem	3
	1.6 Objective of Thesis	4
	1.5 Organization of the Thesis	5
2.	**IEEE Standard for Binary Floating Point Arithmetic**	**6-13**
	2.1 IEEE Standard	6
	2.2 Floating Point Format	6
	2.2.1 Single precision, Double Precision & Quad Precision Format	7
	2.2.2 The Fraction Field	8
	2.2.3 The Exponent Field	9
	2.2.4 Normalized and Denormalized Values	9
	2.2.5 Infinity, Not-a-Number, Zeros	10

2.3 Exception Conditions 11
2.3.1 Inexact Result 12
2.3.2 Underflow Conditions 12
2.3.3 Invalid Operation Conditions 13
2.3.4 Division by Zero Condition 13

3. Programmable Logic Devices 14-25

3.1 Programmable Logic 14
3.2 Fixed Logic vs Programmable Logic 14
3.3 Design Flow 15
3.3.1 Design Entry 16
3.3.2 Simulation 16
3.3.3 Synthesis 16
3.3.4 Implementation 17
3.3.5 Programming 17
3.4 CPLDs vs. FPGAs 17
3.4.1 PLD Market overview 18
3.4.2 CPLD and FPGA Architecture 19
3.4.3 CPLD and FPGA Interconnect Structure 20
3.4.3.1 Performance Predictability 21
3.4.4.2 In-System Performance 21
3.5 Logic Utilization 22
3.6 Summary 24

4. Building Blocks of ALU 26-33

4.1 Types of Adder 26
4.1.1 Shift and Parallel Add Multiplier. 26
4.1.2 Ripple Carry Array Multiplier 27
4.1.3 Carry Skip Adder 28
4.1.4 Carry Select Adder 29
4.2 Types of Multiplier 30
4.2.1 Ripple Carry Adder 30
4.2.2 Carry Lookahead Adder 30
4.2.3 Carry Save Array Multiplier 32

5. Algorithm, Flow Charts and Results	**34-69**
5.1 Algorithm	34
5.2 Flow charts	38
5.3 Results And Discussion	49
5.4 Synthesis Report	50
5.5 Simulation Result	52
6. Conclusion and Future Scope	**70**
References	**71**
Annexures	**74-78**
HDL Synthesis Report	74
Glossary of Terms	75
List of Publications	78

List of Figures

Figure No.	List	Page No.
2.1	IEEE Single precision Format	7
2.2	IEEE Double Precision Format	7
2.3	Quad Precision Format	8
2.4	IEEE Single Precision Format Example	9
3.1	Design Flow	15
3.2	Programmable Logic Market	18
3.3	Logic Design Application	19
3.4	FPGA vs CPLD Routing Scheme	20
3.5	CPLD vs FPGA Performance	22
3.6	CPLD vs FPGA Performance	22
3.7	Logic Utilization Register-Intensive Design	23
3.8	Combinatorial-Intensive Design	23
3.9	CPLD vs. FPGA Capacity	24
3.10	Four Bit Ripple carry adder	25
4.1	Four Bit Ripple Carry Adder	26
4.2	Four bit Lookahead Adders linked by ripple carry propagation	27
4.3	Carry Skip Adder	28
4.4	Carry Select Adder	29
4.5	Shift and Parallel Add Multiplier	30
4.6	Circuit of 4*4 Ripple Array Multiplier	31
4.7 (a)- (b)	Blocks in Ripple Array Multiplier	32
4.8	Carry Save Array Multiplier	32
4.9	Cell for CSA Multiplier	33
5.1	Shows the Simulation Result of Fixed Point 15 Bit Adder.	52
5.2	Shows the Simulation Result of Fixed Point 15 Bit Subtractor	53
5.3	Shows the Simulation Result of Fixed Point 7 Bit Multiplier.	54
5.4	Shows the Simulation Result of Fixed Point 7 Bit Divider.	55
5.5	Shows the Simulation result of Floating Point 32 Bit Adder.	56
5.6	Shows the Simulation Result of Floating Point 32 Bit Subtractor.	57

5.7	Shows the Simulation Result of Floating Point 32 Bit Multiplier.	58
5.8	Shows the Simulation Result of Floating Point 32 Bit Divider.	59
5.9	Shows the Simulation Result of Logical AND Operation.	60
5.10	Shows the Simulation Result of Logical OR Operation.	61
5.11	Shows the Simulation Result of Logical NOR Operation.	62
5.12	Shows the Simulation Result of Logical NAND Operation.	63
5.13	Shows the Simulation Result of Logical XOR Operation.	64
5.14	Shows the Simulation Result of Logical NOT Operation.	65
5.15	Shows the Simulation Result of Circular shift left operation.	66
5.16	Shows the Simulation Result of Circular shift right operation.	67
5.17	Shows the Simulation Result of Carry Lookahead Adder.	68
5.18	Shows the Simulation Result of Carry save array Multiplier.	69

List of Tables

Table No.	List	Page No.
Table 2.1	IEEE Single Precision Value Summary (Decimal)	10
Table 2.2	IEEE Single Precision Value Summary (Binary)	11
Table 2.3	Approximate Maximum Representable Values	12
Table 3.1	CPLD and FPGA Features	18
Table 5.1	Synthesis Report Summary of proposed adder and multiplier design along with the carry lookahead adder and carry save array multiplier	50
Table 5.2	Timing Report Summary of 7 Bit ALU on CPLD and FPGA	50
Table 5.2	Synthesis Report Summary of proposed ALU Implemented on FPGA	51

List of Flow Charts

Flow Chart No.	Name	Page No.
1.	High Speed ALU	38
2.	Control Unit	39
3.	Addition of two 15 Bit Numbers.	41
4.	Subtraction of two 15 Bit Numbers.	42
5.	Multiplication of two 7 Bit Numbers.	43
6.	Division of two 7 Bit Numbers.	44
7.	Addition and Subtraction of two Floating Point Numbers	45
8.	Division of two 22 Bit Numbers.	47
9.	Multiplication and Division of two Floating Point Numbers.	48

Chapter 1
Introduction

1.1 General

The arithmetic and logic unit performs both the arithmetic and the logic operations on the data supplied to it. Although many functions can be performed by ALU, the basic arithmetic operations – addition, subtraction, multiplication, and division – continue to be "bread and butter" operations [12,17]. The other basic operations like shifting, logical multiplication and logical addition are also performed by ALU.

Scientific and engineering applications demand exceptionally high floating-point performance which in turn requires high speed floating-point ALUs and multipliers to reduce executing time. In recent years, a number of high speed floating-point execution units have been presented [2,20,23]. In the floating point ALU the exponent of the larger operand is selected as the common exponent and the fraction of the operand with the smaller exponent is shifted to the right by the alignment shifter. The addition/subtraction of the fraction of the larger exponent operand and the right shifted fraction, as well as normalization, IEEE rounding, and correction of the common exponent are performed.

1.2 Read Only Memory (ROM)

A ROM has a set of input address lines and a set of outputs. The number of addressable entries in the ROM determines the number of address lines: if the ROM contains 2^m addressable entries, called the height, then there are m input lines. The number of bits in each addressable entry is equal to the number of output bits and is sometimes called the width of the ROM [12,17]. The total number of bits in the ROM is equal to the height times the width. The height and width are sometimes collectively referred to as the shape of the ROM. A ROM can encode a collection of logic functions directly from the truth table. The entries in the input portion of the truth table represent the addresses of the entries in the ROM, while the contents of the output portion of the truth table constitute the contents of the ROM.

1.3 Programmable Logic Devices [PLD]

A Programmable Logic Device, is a device whose logic characteristic can be changed and manipulated or stored through programming.

1.3.1 Field Programmable Gate Arrays(FPGAs)

FPGAs are array of logic blocks which can be linked together to form complex logic implementations. They are separated into two categories Fine Grained and Coarse Grained.

Fine Grained [13] being made up of sea of gates or transistors or small micro cells, while Coarse Grained [9] being made up of bigger macro cells which are often made up of flip-flops and Look Up Tables which make up the combinational logic functions. These are RAM based devices i.e. these devices loose their configuration when power is switched off. Hence they have to be configured every time when power is applied.

1.3.2 Complex Programmable Logic Devices (CPLDs)

CPLDs are made up of smaller common macro cells, which are programmable. CPLD's consists of multiple PAL like function block that can be interconnected through a switch matrix [29,30]. These are EPROM based devices (Flash) i.e. these devices store their configuration even when the power is switched off. Hence they need not to be configured every time when power is applied.

1.4 Number Representation

Most of computations need real numbers as an essential category in any real world calculations [17]. Real numbers are not finite; therefore no finite representation method is capable of representing all real numbers, even within a small range. Thus, most real values will have to be represented in an approximate manner [23]. Various methods can be used for this representation.

- Fixed-point number systems: offer limited range and/or precision. Computations must be "scaled" to ensure that values remain represent able and that they do not lose much precision.
- Rational number systems: approximate a real value by the ratio of two integers,

and lead to difficult arithmetic operations.

- Floating-point number systems: represent numbers in 3 fields (sign, exponent and mantissa), and is the most common approach which will be discussed in detail shortly.

1.5 Formulation of problem

Shailesh shah and H.Klar et al [25] proposed an algorithm for the array multiplier. As both multiplicand and multiplier may be positive or negative 2's complement number system is used to represent them and care for sign bit extension must be taken [19]. This algorithm which eliminates the need for the common sign bit extension in addition of partial products is implemented. This results in a reduction of circuit area as well as the capacitive load of the intermediate sum and carry sign bit signals decreases, which leads to an appropriate reduction of delay. A.D Booth and O.Macsorley [1] framed algorithm for Booth Multiplier and this algorithm is one of the most popular techniques to reduce the number of partial products to be added while multiplying two numbers. If 3-bit recoding (Radix-4) is used the number of partial products is reduced by half. This is a great saving in terms of silicon area and also speed as number of stages to be added is reduced to half compared to normal add and shift multiplication. C.S. Wallace suggested the way for Wallace Tree Multiplier [7]. Wallace introduced a very important iterative realization of parallel multiplier. In Wallace tree architecture, all the bits of all of the partial products in each column are added together by a set of counters in parallel without propagating any carries. Another set of counters then reduces this new matrix and so on, until a two-row matrix is generated. Then, a fast adder is used at the end to produce the final result. E.Abu-Shama [8] presented a multiplier architecture. The generation of all partial products of the multiplication can be done in parallel. The parallelism in generating the partial product is realized by ANDing the first (LSB) of the multiplier with the multiplicand bits. The second partial product is achieved by ANDing the second multiplier bit with the multiplicand bits proceeded by a zero. The third partial product is achieved by ANDing the third multiplier bit with the multiplicand bits proceeded by double zeros, and so on. Douglas J. Smith [25] lays the idea of Shift and Parallel Add Multiplier. In this all the partial products are obtained using AND gate or using mux

whose select line be multiplier x, and then inputs are multiplicand or zeros. These partial products are shifted left according the position of the multiplier bit. Now the sum of two partial products (pp) is taken in parallel using two pairs at a time [15]. This type of multiplier circuits can use carry lookahead adder to propagate the adder result fast. This multiplier circuit is pure combinatorial and can be easily pipelined to use in high performance system by using registers in front of adders. Thus, 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 [21,22]. But a serious inspection shows that multiplier construction has the following characteristics

- A number of Input/Output pins are needed.
- A number of AND gates and flip-flops are needed.
- A number of adders are needed to add partial products.
- Irregular structure of some multipliers creates difficulty in mapping to available FPGA routing resources.
- Long carry propagation path leads to long delay time.
- More hardware requirement in combinational multiplier for implementing floating point multiplication.

1.6 Objective of Thesis

Thus, to circumvent the problems associated with the multiplier construction discussed above which is also common to other building blocks of arithmetic logic unit, an indigenous design of arithmetic logic unit is proposed. The task of designing ALU and writing codes in VHDL is categorized in to following sub-tasks.

- To realize ROM in the designing process.

- To devise a mechanism to select particular operations.
- To realize fixed point number in building blocks of ALU.
- To incorporate floating point number in the design.
- To simulate the design on ModelSim SE 5.5e tool.
- To verify the results on FPGA and CPLD trainer kit.
- To synthesize the design on Xilinix ISE 6.1i by selecting device family Spartan 2E and XC9500 for FPGA and CPLD respectively.
- To compare proposed adder and subtractor with the conventional counter parts.
- To perform comparison of the FPGA and CPLD based on the synthesis report of proposed design.

1.7 Organization of the Thesis

The thesis work is organized in six chapters. Chapter-1, serves as a global introduction. This chapter revisits the ROM and programmable logic devices. Fixed point format and floating point format are also introduced in this chapter. A short and compact introduction of the IEEE standard for floating point arithmetic is provided in Chapter-2. This chapter introduces the single precision ,double precision and quad precision format. Chapter-3, deals with programmable logic devices and their usefulness in the design of high speed circuits. The main focus is on CPLD and FPGA. A short description of features of CPLD and FPGA are provided that can serve as a product bulletin. Brief overview of the different types of adders and multipliers are provided in Chapter-4.The main goal of this chapter is to provide an intuitive understanding of the operation of different building blocks (adders, multipliers) of the ALU.Chapter-5, supplies the algorithm and flowchart for the proposed high speed arithmetic logic unit. This chapter also presents the simulation results of different building blocks of ALU on ModelSim SE 5.5e Tool. The synthesis reports are generated on Xilinx ISE 6.1i Tool. The results are summarized in tables which can serve as an artifacts of the future design.Chapter-6, spends considerable focus on the analysis of the results to arrive at a conclusion. This chapter paves the path for future modifications in the present design. The major possible areas of further development are enumerated to felicitate advancement in the design.

Chapter 2

IEEE Standard for Binary Floating Point Arithmetic

Until the 1980s floating point number formats varied one computer family to the next, making it difficult to transport programs between different computers without encountering small but significant differences in such areas as exceptions. To deal with this problem, the Institute of Electrical and Electronics Engineers (IEEE) sponsored a standard format for 32-bit and larger floating point numbers, known as IEEE 754 standard [10]. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most UNIX platforms. This chapter gives a brief overview of IEEE floating point standards and its representation.

2.1 IEEE Standard

The IEEE standard was approved in 1985. Its main purpose is to define specification for representing and manipulating floating point values so that programs written on one IEEE conforming machine can be moved to another machine with predictable results. In addition, the standard is specifically designed to make it easier for the programmers to write useful and robust floating point programs. The standard defines the following:

1) Format for representing floating point numbers.
2) Representation for special values (for example infinity, very small value, not a number, zero).
3) Exception conditions (when they occur, and what happen when they occur).
4) Four rounding modes.

2.2 Floating Point Format

The IEEE standard specifies four formats for representing floating point values.

1) Single precision
2) Double precision
3) Single extended precision (optional)

4) Double extended precision (optional), , double extended precision is sometimes called as quad precision.

2.2.1 Single precision, Double Precision and Quad Precision Format

Single precision, double precision and quad precision values consist of three fields: sign bit, exponent and fraction. The sign bit reflects the algebraic sign of the value. The exponent represents an integer value that is a power to which 2 is raised. The fraction, also called the significand, represents a value between 1.0 and 2.0 (for normalized values). The result of the exponent expression is multiplied by the fraction to yield the actual numerical value.

The only difference among the single-precision, double-precision, and quad-precision formats is the number of bits allocated for the exponent and fraction. Figure 2-1, Figure 2-2, and Figure 2-3 show the number of bits allocated in each format.

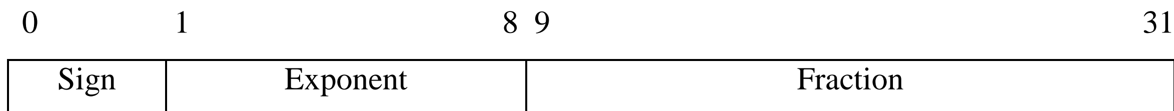


Figure-2.1 IEEE Single Precision Format

The single-precision format is 32 bits long: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fraction. The double-precision format is sometimes divided conceptually into two 32-bit words. The word containing the sign bit, the exponent field, and the first portion of the fraction field is referred to as the most significant word. The other word, containing the last portion of the fraction, is called the least significant word.

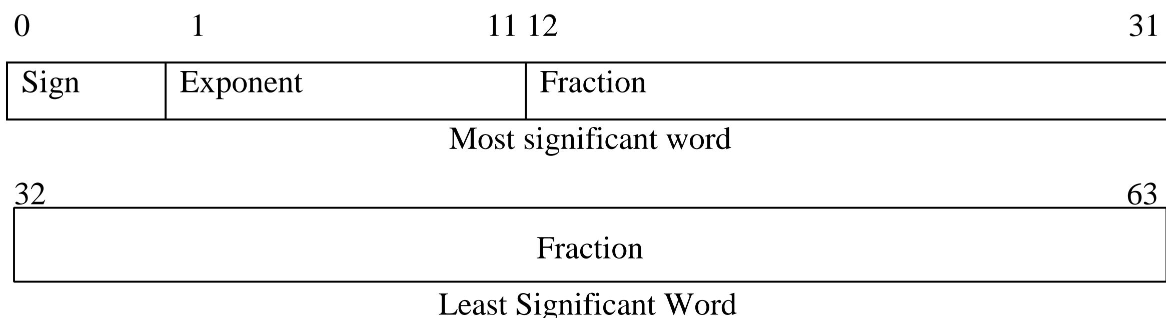


Figure -2.2 IEEE Double Precision Format

The quad-precision format is 128 bits long: 1 bit for the sign, 15 bits for the exponent, and 112 bits for the fraction. This format is divided conceptually into four 32-bit words: the most significant word, two middle words, and the least significant word.

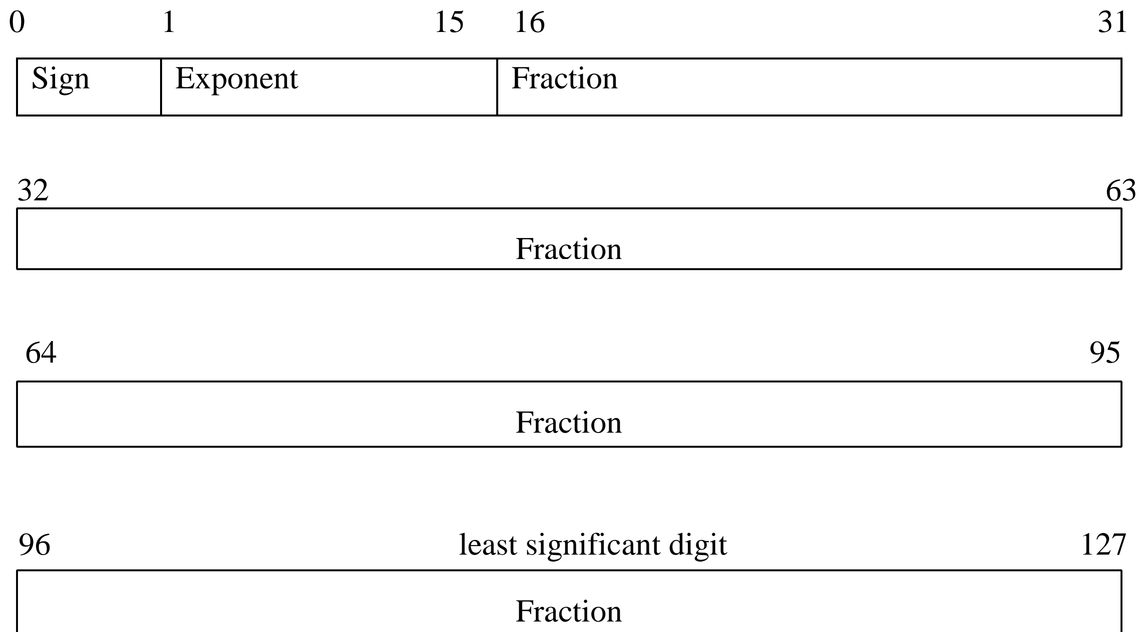


Figure-2.3 Quad Precision Format

2.2.2 The Fraction Field

For normalized values, the fraction represents a value greater than or equal to 1.0 and less than 2.0. Each bit in the fraction represents the value 2 raised to a negative power. For example, the first bit represents the value 2^{-1} (0.5), the second bit is 2^{-2} (0.25), and so on. The sum of 1.0 and the values represented by all these bits is the value of the fraction. The 1.0 in the sum corresponds to the zero th fraction bit, 2^0. Since this bit would always be set for a normalized value, it is not included in the actual format, but it is implied. It is sometimes referred to as the fraction implicit bit or hidden bit. For example, if the 23 bits in the fraction field of a single-precision numbers are

011 0100 0000 0000 0000 0000

and the exponent field is not all 1's or all 0's, the fraction value is

$$1.0 + 2^{-2} + 2^{-3} + 2^{-5} = 1.0 + 0.25 + 0.125 + .03125 = 1.40625$$

The 1.0 represents the fraction implicit bit, and the exponents of −2, −3, and −5 indicate that the second, third, and fifth bits of the fraction field are set.

2.2.3 The Exponent Field

The exponent field uses a biased representation. This means that the value represented by the exponent field is the value in the exponent field interpreted as an unsigned integer minus a constant value (the bias). The purpose of the bias is to allow all exponent calculations to be performed using unsigned arithmetic. For single-precision formats, the bias is 127; for double-precision formats, it is 1023; for quad-precision formats, it is 16383. The Floating point format example is, 6.0 is represented in single precision format as shown in Figure 2.4.

0	100 0000 1	100 0000 0000 0000 0000 0000

Figure-2.4 IEEE Single Precision Format Example

The first bit is the sign bit. Because the sign bit is 0, the floating-point value is positive. The next eight bits make up the exponent. 1000 0001 equals 129, but the true value of the exponent is derived by subtracting the bias constant 127 from this value. So the true exponent value is 2. The fraction bits are 100 0000 0000 0000 0000 0000, which, when added to the implicit bit, equal 1 + 0.5, or 1.5.

In algebraic terms, a floating-point value is

$$(-1)^s \times M \times 2^{E-B}$$

Where S is the value of the sign bit, M is the fraction (with implicit bit), E is the exponent, and B is the bias. In our example, this would be $(-1)^0 \times 1.5 \times 2^2 = 1.5 \times 4.0 =$ 6.0.

2.2.4 Normalized and Denormalized Values

Values that are represented by a sign bit, a fraction, and an exponent whose bits are not all zeros and not all ones are called normalized values (also called normal values).

Because the value in the exponent field of a normalized value cannot be 0, the size of the exponent field determines the smallest value that can be represented in normalized format. For single-precision numbers, the largest-magnitude negative exponent is −126 (that is, 1 − 127); for double-precision numbers, it is −1022 (that is, 1 − 1023); for quad-precision numbers, it is −16382 (that is, 1 − 16383). Denormalized values (also called subnormal values) fill in the gap on the number line between the smallest-magnitude normalized value and zero. A denormalized value is represented by a zero exponent field and a nonzero fraction (if the fraction were also zero, the floating-point value would be zero). Denormalized number can be computed by interpreting the fraction as an integer and then multiplying this integer by 2^{-149} for single-precision numbers, by 2^{-1074} for double-precision numbers, and by 2^{-16494} for quad-precision numbers. The maximum fraction is always 2^{k}-1, where k is the number of bits in the fraction. Table 2.1, 2.2 summarizes how IEEE values are represented in binary.

2.2.5 Infinity, Not-a-Number, Zeros

- Infinity

Values that are larger in magnitude than the maximum-magnitude normalized values are approximated by special bit patterns that represent positive and negative infinity. According to the IEEE standard, infinities are represented by setting all the bits in the exponent field to 1 (value 255 for single-precision, 2047 for double-precision, 32767 for quad-precision) and setting the fraction bits to 0.

TABLE-2.1 IEEE Single Precision Value Summary (Decimal)

Value	Single Precision (Positive)	Single Precision Negative
Zero	0.0	0.0
Denormalized	1.4012985E -45 to 1.754942E -38	-1.4012985E -45 to -1.754942E -38
Normalized	1.754944E -38 to 3.4028235E +38	-1.1754944E -38 to -3.4028235E +38
Infinity	Not Applicable	Not Applicable
NaN	Not Applicable	Not Applicable

There are actually two infinity values, negative infinity if the sign bit is 1 and positive infinity if the sign bit is 0.

TABLE-2.2 IEEE Single Precision Value Summary (Binary)

Values	Exponent	Fraction	Hexadecimal Values (Single Precision)	
			Positive	Negative
Zero	All Zeros	All Zeros	0000 0000	8000 0000
Denormalized	All Zeros	Non Zero	0000 0001 to 007F FFFF	8000 0001 to 807F FFFF
Normalized	Neither all zeros nor all ones	Anything	0080 0000 to 7F7F FFFF	8080 0000 to FF7F FFFF
Infinity	All Ones	All Zeros	7F80 0001	FF80 0001
NaN	All Ones	Most Significant Bit 1	7FC0 0000 to 7FFF FFFF	FFC0 0000 to FFFF FFFF

- Not-a-Number (NaN)

A NaN (Not-a-Number) is a special IEEE representation for error values. A NaN can be

• The result of an invalid operation

• An undetermined value

NaNs are represented by setting all of the bits in the exponent to 1 and setting at least one of the bits in the fraction field to 1.

- Zeros

The IEEE standard defines both a positive zero and a negative zero. In both cases, the value is represented by setting all bits in the exponent and fraction to zero. The only difference, therefore, is that the sign bit is set for a negative zero.

2.3 Exception Conditions

The IEEE standard defines five exception conditions, also called exceptions:

- Inexact result
- Overflow
- Underflow
- Invalid operation
- Division by zero

The IEEE standard defines the conditions under which exceptions occur only for the basic operations.

2.3.1 Inexact Result (Rounding)

When a computer system cannot represent a number exactly, it must choose a nearby representable value. This is called rounding and it always produces an inexact result condition. Because most floating-point operations produce rounded (that is, inexact) results most of the time, the inexact result exception is not usually considered to be an error. There are four rounding modes. Whether system rounds to the lower or higher of two representable values depends upon the rounding mode. The available choices include an IEEE standard default rounding mode as well as three alternate modes. The default rounding mode is round to nearest. The four rounding modes are Round to Nearest, Round towards positive infinity, Round towards negative infinity, Round towards zero. For all but specifically designed numerical analysis applications, round to nearest is the best rounding mode. In round-to-nearest mode, the nearest representable value is never more than 1/2 ULP away from the exact result being rounded, so the error introduced from one operation by rounding is never more than 1/2 ULP. For the other rounding modes, the error is less than 1 ULP.

TABLE-2.3 Approximate Maximum Representable Values

	Largest Negative Value	Largest Positive Value
Single-Precision	-3.4E38	3.4E38
Double-Precision	-1.7E308	1.7E308
Quad-Precision	-1.2E4932	1.2E4932

2.3.2 Underflow Conditions

An underflow condition may occur when a floating-point operation attempts to produce a result that is smaller in magnitude than the smallest normalized value.

2.3.3 Invalid Operation Conditions

An invalid operation condition occurs whenever the system attempts to perform an operation that has no numerically meaningful interpretation. The following are invalid operations (also called operation errors, operand errors, or domain errors):

- Any operation on a signaling NaN
- Magnitude subtraction of infinities.
- Multiplication of zero by infinity.
- Division of zero by zero or division of infinity by infinity.

2.3.4 Division by Zero Condition

A division by zero condition occurs whenever the system attempts to divide a nonzero, finite value by zero.

Chapter 3
Programmable Logic Devices

3.1 Programmable Logic

In the world of digital electronics systems, there are three basic kinds of devices: memory, microprocessor and logic. Memory devices store random information such as the contents of a spreadsheet or a data base. Microprocessors execute software instructions to perform a wide variety of tasks such as running a word processing program or video game. Logic devices provide specific functions, including device to device interfacing, data communications, signal processing, data display, timing and control operations and almost every other functions a system must perform.

3.2 Fixed Logic versus Programmable Logic

Logic devices can be classified into two broad categories – fixed and programmable. As the name suggests, the circuits in a fixed logic device are permanent, they perform one function or set of functions – once manufactured, they can not be changed. On the other hand, Programmable logic devices (PLDs) are standard , off the shelf parts that offer customers a wide range of logic capacity, features, speed, and voltage characteristics and these devices can be changed at any time to perform any number of functions [3,27]. With fixed logic devices, the time required to go from design, to prototypes, to a final manufacturing run can take from several months to more than a year, depending on the complexity of the device. And, if the device does not work properly, or if the requirements change, a new design must be developed. The up-front work of designing and verifying fixed logic devices involves substantial "non recurring engineering" costs, or NRE. These NRE costs can run from a few hundred thousands to several millions dollars [2].

With programmable logic devices, designers use inexpensive software tools to quickly develop, simulate, and test their designs [28]. Then, a design can be quickly programmed into a device, and immediately tested in a live circuit. There are no NRE costs and the final design is completed much faster than that of a custom, fixed logic device.

Another key benefit of using PLDs is that during the design phase customers can change the circuitry as often as they want until the design operates to their satisfaction. That's because PLDs are based on re-writable memory technology- to change the design, the device is simply reprogrammed. Once the design is final, customers can go in to immediate production by simply programming as many PLDs as they need with the final software design file [26].

3.3 Design Flow

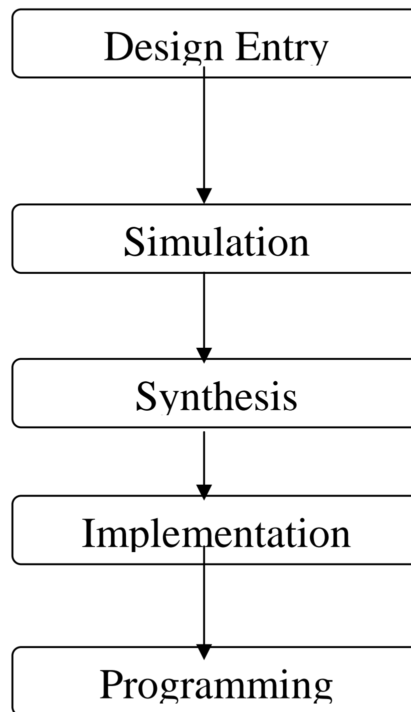


Figure-3.1 Design Flow

To start new design sequence is as follows

- DESIGN ENTRY
- SIMULATION
- SYNTHESIS
- IMPLEMENTATION
- PROGRAMMING

3.3.1 Design Entry

User can enter the design by three ways.

- HDL
- FSM
- SCHEMATIC

HDL (Hardware description language) can be written in two languages

- VHDL (very high speed integrated circuit) HDL
- VERILOG

In HDL there are, there are two parts in a programme

- ENTITY
- ARCHITECTURE

Entity- It is description of input and output.

Architecture-is functional behavior CODE of design. Once code has been completed, then syntax checks has to be performed and error free code has to be added in the project manager by using the option ADD TO PROJECT from PROJECT icon in the task Bar of HDL editor

3.3.2 Simulation

In simulation user can verify the functionality of his design by applying various input signal combination and observing the result the output results. The simulation is performed on gate level Netlist

3.3.3 Synthesis

It is the process which converts HDL CODE in to gate level circuit in the form of NET LIST. This process is Target Technology dependent and hence user must select proper DEVICE, FAMILY, PARTNUMBER and SPEED GRADE. Also user may select SYNTHESIS SETTINGS like CLOCK FREQUENCY,OPTIMIZATION FOR SPEED or AREA.

3.3.4 Implementation

Implementation is the process in which the design is passed through various stages by TRANSLATE,MAPPING,TIME ANALYSIS and BITSTREAM. For locking input and output signal to particular pins of the device user must write UCF(user constraint file) before implementation and guide the same file to implementation tool through the option set control files. Output of implementation is .jed for CPLD and .bit file for FPGA, which can be directly program in to target device.

3.3.5 Programming

This is the process by which user can physically download the design programming files from PC to Target device using programming cable.

- To programme CPLD, select Boundary scan (JTAG) mode
- To programme FPGA, select Boundary scan, slave serial mode or Master serial mode.

3.4 CPLDs vs. FPGAs

Introduction:- The high-capacity programmable logic device (HCPLD) market has expanded dramatically in recent years with the introduction of new devices and architectures. While these new devices offer greater design flexibility, design engineers must sift through all available devices to determine the best device for an application. Choosing the right device can lead to market success, while choosing the wrong device can result in major project setbacks.

To make an informed decision, designers need to understand the strengths and limitations of different HCPLDs, specifically complex programmable logic devices (CPLDs) and field-programmable gate arrays (FPGAs) [8,16]. The text presented here serve as a product information bulletin that provides guidelines on choosing the correct devices for design applications and discusses the following topics.

o PLD market overview

o CPLD vs. FPGA architecture

o CPLD vs. FPGA interconnect structures

o CPLD vs. FPGA process alternatives

o CPLD vs. FPGA development software

3.4.1 PLD Market Overview

The PLD market consists of low- and high-capacity devices. Low-capacity devices, called simple PLDs, typically contain fewer than 600 usable gates and include products such as PALs, GALs, and 22V10s. Simple PLDs are manufactured using CMOS technology offering EPROM, EEPROM, and FLASH memory elements.

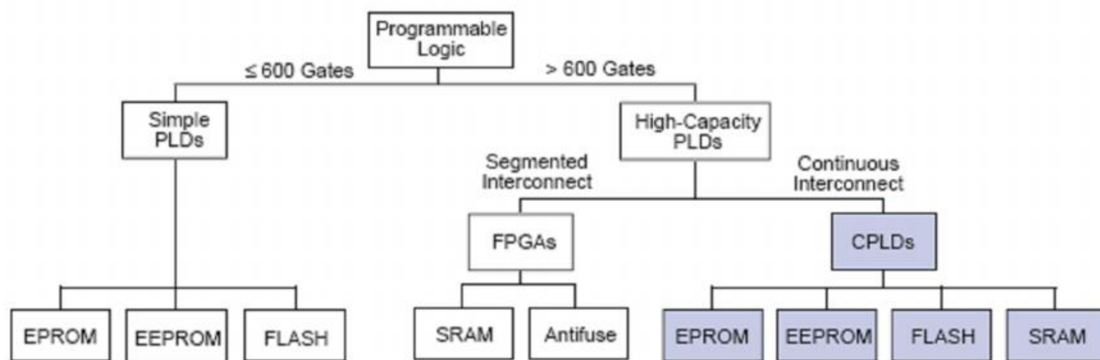


Figure-3.2 Programmable Logic Market

TABLE-3.2 CPLD and FPGA Features

Feature	FPGAs	CPLDs
Leading vendor	Xilinx	Altera
Density	Medium to high	Low to high
Interconnect structure	Segmented	Continuous
Timing	Variable/unpredictable	Fixed/predictable
CMOS options	SRAM, antifuse	EPROM, EEPROM, FLASH, SRAM
Device performance	Moderate	High
Device utilization	Moderate	High
Hand-routing required	Yes	No
Reprogrammability	Yes (SRAM only)	Yes
In-circuit reconfigurability	Yes (SRAM only)	Yes (SRAM only)
In-system programmability	No	Yes (FLASH, EEPROM only)
Compilation times	Slow	Fast
Logic synthesis	Yes (third-party only)	Yes

HCPLDs typically contain more than 600 usable gates, and include both CPLDs and

FPGAs. HCPLDs are manufactured using CMOS technology with EPROM, EEPROM, FLASH, SRAM, and antifuse options. HCPLDs can be differentiated by their interconnect structure: CPLDs use continuous interconnect structures, while FPGAs use segmented interconnect structures. See Figure 3.2.

3.4.2 CPLD vs. FPGA Architecture

CPLDs use several different varieties of CMOS technology and architectural alternatives to address the wide range of logic design applications. EPROM-,EEPROM-, and FLASH-based devices—such as the Altera Classic, MAX 5000, MAX 7000, MAX9000, and FLASH logic families—use a product-term architecture that is optimized for combinatorial-intensive logic designs. EPROM, EEPROM, and FLASH devices are reprogrammable and nonvolatile. SRAM-based CPLDs, such as the Altera FLEX8000 family, use a look-up table (LUT) architecture that is optimized for register-intensive designs. SRAM-based devices offer in-circuit reconfigurability for "on-the-fly" logic changes. Altera is the only CPLD vendor to offer all four major CMOS processes—EPROM, EEPROM, FLASH, and SRAM—to encompass the broadest range of logic design applications. In contrast, most FPGAs use SRAM or one-time-programmable antifuse memory elements. The granular architectures of FPGAs can implement a wide range of applications, but tend to be less efficient than CPLD architectures at implementing combinatorial-intensive logic designs.

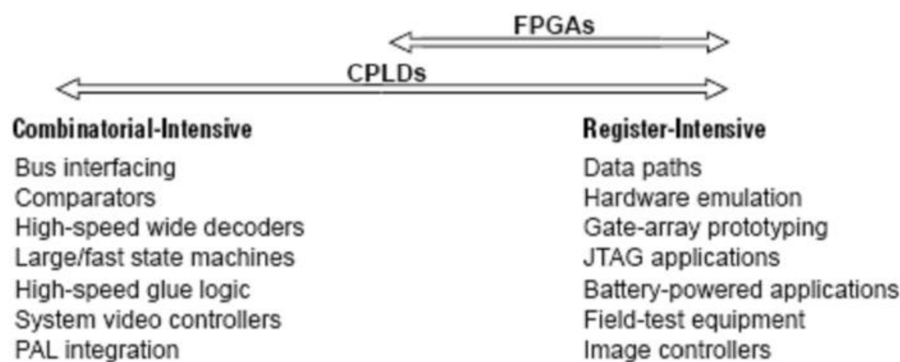


Figure-3.3 Logic Design Applications

While antifuse-based devices typically implement the same register-intensive logic designs as SRAM-based FPGAs, they cannot be erased or reconfigured. Whereas low-

cost, simple PLDs—such as 22V10s—may not require erasability and reconfigurability, these characteristics are essential for HCPLDs. Typically, large designs that contain several thousand gates require several design iterations. Computer simulation helps prevent design errors; however, designers often need to test hardware or incorporate unexpected changes to system specifications. With one-time-programmable antifuse elements, multiple design iterations can result in thousands of dollars of additional expenses in engineering time and device cost. Figure 3.3 shows the logic design applications best suited for CPLD and FPGA architectures.

3.4.3 CPLD vs. FPGA Interconnect Structure

CPLDs and FPGAs use different interconnect structures: CPLDs use a continuous interconnect structure, while FPGAs use a segmented interconnect structure. The continuous interconnect structure of CPLDs consists of metal lines of uniform length that traverse the entire length and width of the device. Since the resistance and capacitance of all interconnect paths is fixed, delays between any two logic cells in the device are predictable. This uniformity of performance across the device minimizes signal skew. The segmented interconnect structure of FPGAs consists of a matrix of metal segments that run throughout the device.

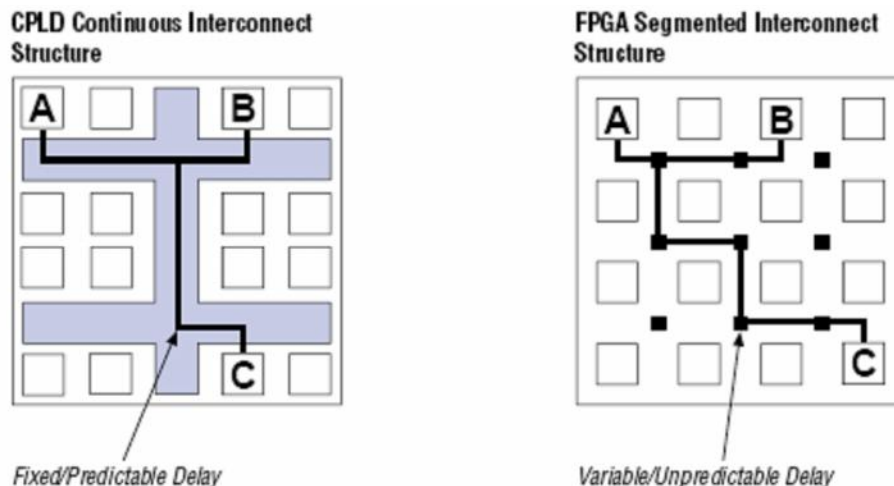


Figure-3.4 CPLD vs. FPGA Routing Scheme

Depending on the type of FPGA, either switch matrices or antifuses join the ends of these segments, allowing signals to travel between logic cells. The number of segments required to interconnect signals is neither constant nor predictable; therefore, delays

cannot be quantified until placement and routing has been completed. See Figure 3.4.

Interconnect structures affect the following device characteristics:

- o Performance predictability
- o In-system performance
- o Logic utilization

3.4.3.1 Performance Predictability

The constant delay between any two logic cells in a continuous interconnect structure allows designers to predict the performance of a CPLD design. The timing model for CPLDs contains all parameters necessary for a designer to predict the overall performance of a design, including interconnect delays.

The total interconnect delay in a segmented interconnect structure is directly proportional to the number of segments necessary to route a signal. It is impossible to know this number until the logic design has been completely placed and routed by the software. Even minor logic changes can require major routing changes, affecting the performance characteristics of the overall device.

3.4.3.2 In-System Performance

CPLDs are known for high in-system performance, which is a direct result of their continuous interconnect structure and efficient signal routing. CPLD delays are not cumulative; the delay the signal incurs is independent of the path the signal takes. Signals that must travel to different locations in the device arrive at their destinations with negligible signal skew. In contrast, FPGA segmented interconnect delays are cumulative. As the number of interconnect segments increases, the interconnect delay also increases. If interconnect segments for a direct path are already used, a signal must pass through a different path, i.e. more segments, to reach its destination. Therefore, no two signals can be guaranteed to arrive at the same time. Typically, one signal arrives after the other, which limits the performance of the design. Signal skew and performance degradation become more prevalent as more interconnect segments are used, inefficiently routing signals through the device. CPLDs offer higher performance than FPGAs of equivalent densities. Device performance is verified by the Programmable Electronics Performance Corporation (PREP), a consortium of PLD companies that has established standard benchmarks for measuring the performance and logic capacity of PLDs. Figure 3.5 shows

the average benchmark speed (ABS) of CPLDs and FPGAs with similar gate counts. ABS is calculated by averaging the mean internal and external benchmark results for all PREP benchmarks.

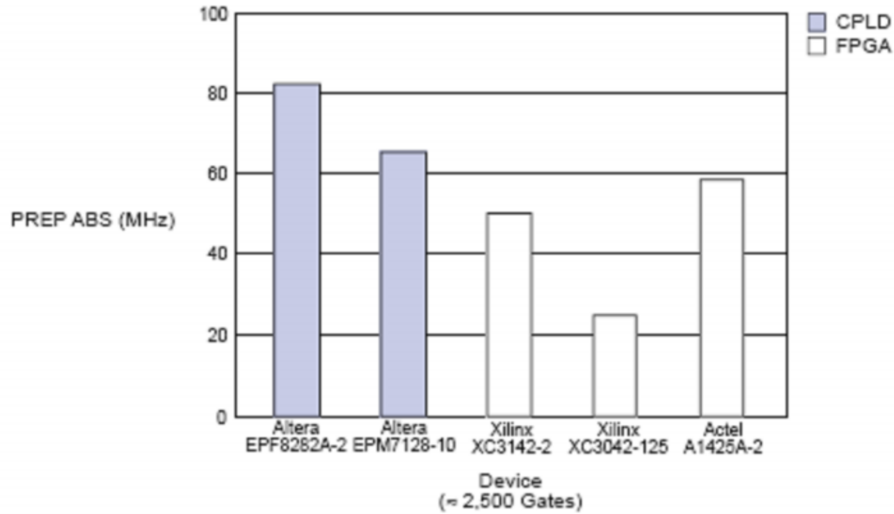


Figure-3.5 CPLD vs. FPGA Performance

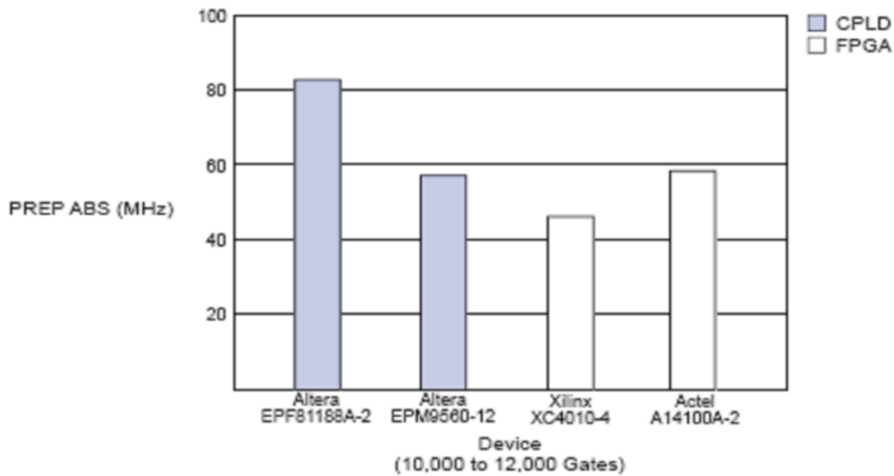


Figure-3.6 CPLD vs. FPGA Performance

3.5 Logic Utilization

The logic cells in most FPGA architectures have fine granularity; therefore, more logic cells are required to implement a function in an FPGA than in a CPLD. Figure 3.6. shows the number of logic cells necessary to implement a register-intensive design (16-bit up/down binary counter) and a combinatorial-intensive design (24-bit decoder) in each architecture.

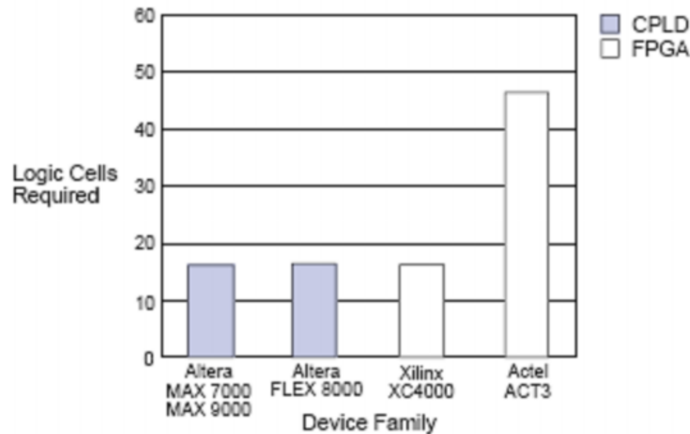


Figure-3.7 Logic Utilization Register-Intensive Design (16-Bit Up/Down Counter)

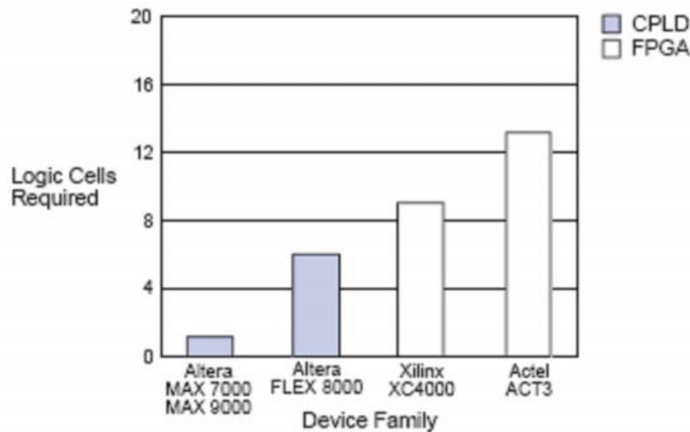
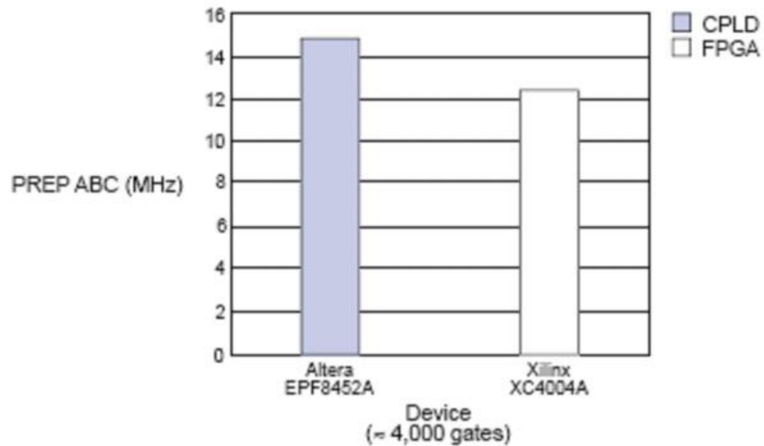


Figure-3.8 Combinatorial-Intensive Design (24-Bit Decoder)

Because the logic cell of an FPGA can contain only small portions of a design, a heavy burden is placed on its segmented interconnect structure. Therefore, any routing bottleneck adversely affects device utilization, as well as device performance, as shown in Figure 3.7 and Figure 3.8. earlier in the product information bulletin. As design complexity increases, the probability of routing conflicts also increases, ultimately leading to lower FPGA device utilization. Figure 3.9. shows the average benchmark capacity (ABC)—the average number of benchmark instances implemented in a device across all PREP benchmarks—of CPLDs and FPGAs with similar gate counts. These results illustrate the lower device utilization in devices with segmented interconnect structures.



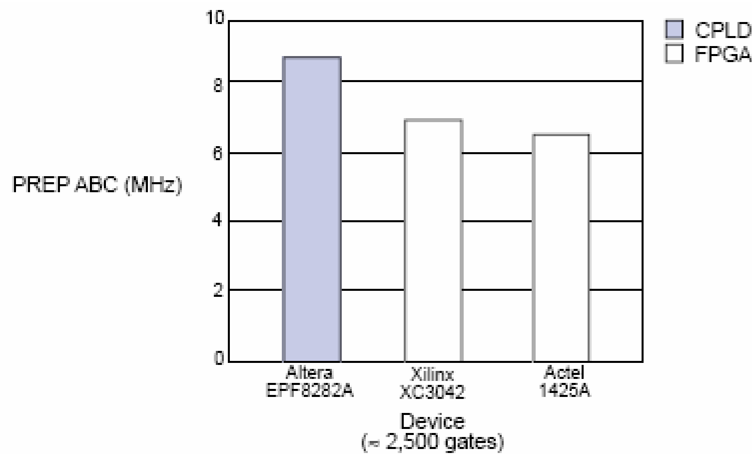


Figure-3.9 CPLD vs. FPGA Capacity

3.6 Summary

- FPGAs are RAM based. They need to be "downloaded" (configured) at each power-up. CPLDs are EEPROM based. They are active at power-up (i.e. as long as they've been programmed at least once...).
- CPLDs have a faster input-to-output timings than FPGAs (because of their coarse-grain architecture, one block of logic can hold a big equation), so are better suited for microprocessor decoding logic for example than FPGAs.
- FPGAs have special routing resources to implement efficiently binary counters and arithmetic functions (adders, comparators...) and RAM. CPLDs do not.

- FPGAs can contain very large digital designs, while CPLDs can contain small designs only.
- CPLDs use a continuous interconnect structure that offers predictable interconnect delays, high performance, and efficient logic utilization. Because this structure is metal-optimized, it enables CPLDs to take full advantage of advances in process technology, increasing performance and reducing customer costs. In contrast, a segmented interconnect structure, found in FPGAs, has many limitations, including unpredictable timing, slow performance, and poor device utilization.

Chapter 4
Building Blocks of ALU

The demand for high speed processing has been increasing as a result of expanding computer and signal processing applications. Higher throughput arithmetic operations are important to achieve the desired performance in many real-time signal and image processing applications [5,6]. The key arithmetic operations in such applications is multiplication, addition, division and subtraction [14,23,24]. The following section outlines different types of adders and multipliers.

4.1 Types of Adder

4.1.1 Ripple Carry Adder

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are half adders or full adders [11].

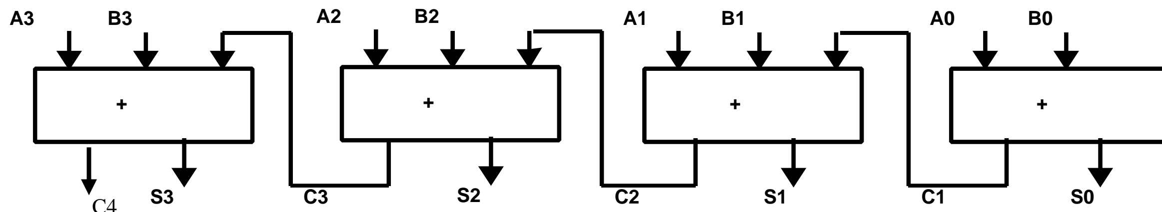


Figure-4.1 Four Bit Ripple Carry Adder

The principal problem in constructing an adder for n-bit numbers out of small pieces is propagating carries from one piece to the next. The most obvious way to solve this with a ripple carry adder, consisting of n full adders, illustrated in Figure 4.1, where each block consists of full adder. The carry out of one full adder is connected to the carry in of the adder for the next most significant bit. The carries ripple from the least significant bit (on the right) to the most significant bit.

In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. It takes two levels for first carry to

compute from a_0 and b_0. Then it takes 2n levels for n bits. The ripple carry adders are slowest but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way.

4.1.2 Carry Lookahead Adder

In this type of adder the carries are generated and propagated [18,11]. The name generate comes from the fact that stage I generates a carry of 1 independent of values of C_{i-1} if both x_i and y_i are 1. i.e. if xy=1. Stage I propagates c_{i-1}; that is it makes c_i=1 in response to last carry input.

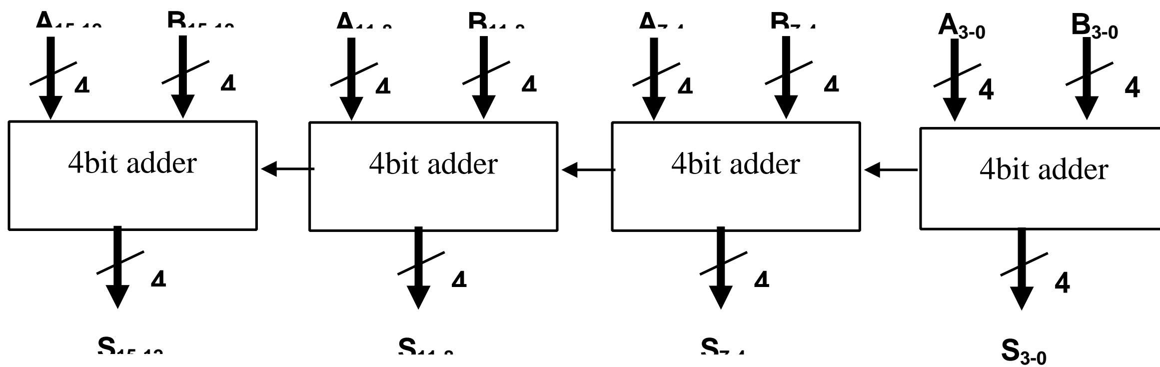


Figure-4.2 Four bit Lookahead Adders linked by Rripple Carry Propagation.

Carry Generate Gi = Ai Bi

Carry Propagate Pi = Ai xor Bi

Sum and Carry can be reexpressed in terms of generate/propagate:

Si = Ai xor Bi xor Ci = Pi xor Ci

Ci+1 = Ai Bi + Ai Ci + Bi Ci

= Ai Bi + Ci (Ai + Bi)

= Ai Bi + Ci (Ai xor Bi)

= Gi + Ci Pi

Reexpress the carry logic for each of the bits:

C1 = G0 + P0 C0

C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0

C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0

C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 C0

Four 4 bit adders formed in carry lookahead circuit can be combined in form shown in Figure 4.2. This design represents compromise between a 16 stage ripple carry adder, which is cheap but slow, and a single stag 16 bit carry lookahead adder, which is fast, expensive and impractical because of the complexity of its carry generation logic. The circuit of Figure 4.2 effectively combines sets of four inputs into groups that are added via carry lookahead; the resulted computed by various groups are then linked via ripple carries.

4.1.3 Carry Skip Adder

Carry-skip designs are intended to offer an alternative between ripple-carry and carry-lookahead implementations [11]. These designs compute the carry-propagation terms for several blocks in parallel, permitting the carry to skip over adjacent blocks if the carry from previous block are available.

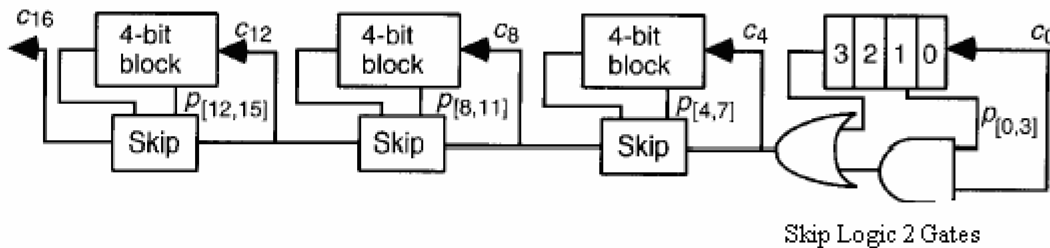


Figure-4.3 Carry Skip Adder

The motivation for this adder comes from the examining equation for P and G for the carry lookahead adder [18]. Computing P is much simpler than computing G and carry skip adder just computes P's. Such a adder is shown in Figure 4.3, where carry being rippling from each stage. If any block generates a carry, then a carry out of a block will be true, even though the carryin of the block may not be correct yet. If at the start of each

add operation the carryin to each block is 0, then no spurious carry out is generated. Once the carryout of the least significant block is generated, it not only feeds to next block but is also fed through the AND gate with P signal from the next block. If the carry out and P signal both are true, then carry skips the second block and is ready to feed into third block, and so on.

4.1.4 Carry Select Adder

Carry-select addition functions by performing two additions on groups of multiple bits in parallel, one that assumes a carryin of 0, one a carryin of 1. Once the true carry is known, the correct sum is selected [18]. An example of such a design is shown in Figure 4.4.

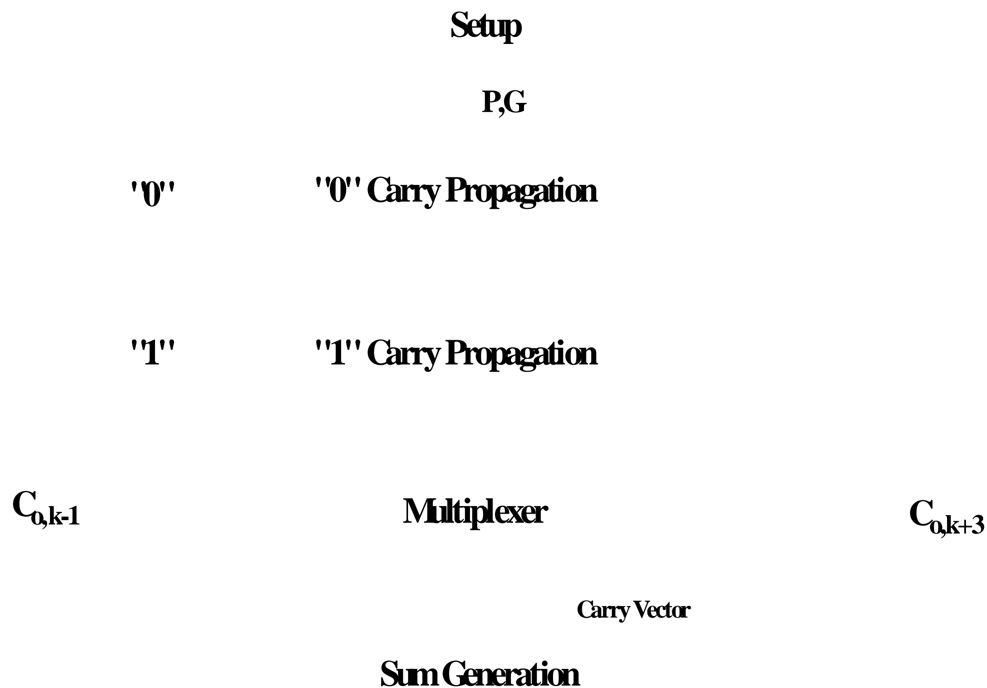


Figure-4.4 Simple Carry Select Adder

From a circuit point of view, this means that two carry paths are implemented. When out $C_{0,k-1}$ finally settles, either the result of 0 or the 1 path is selected by the multiplexer,

which can be performed with a minimal delay. The hardware overhead of the carry select adder is restricted to an additional carry path and a multiplexer. A full carry select adder is now constructed by chaining a number of equal length adders stages.

4.2 Types of Multiplier

4.2.1 Shift and Parallel Add Multiplier

In this all the partial products are obtained using AND gate or using mux whose select line be multiplier x, and then inputs are multiplicand or zeros. These partial products are shifted left according the position of the multiplier bit. Now the sum of two partial products (pp) is taken in parallel using two pairs at a time [16,11].

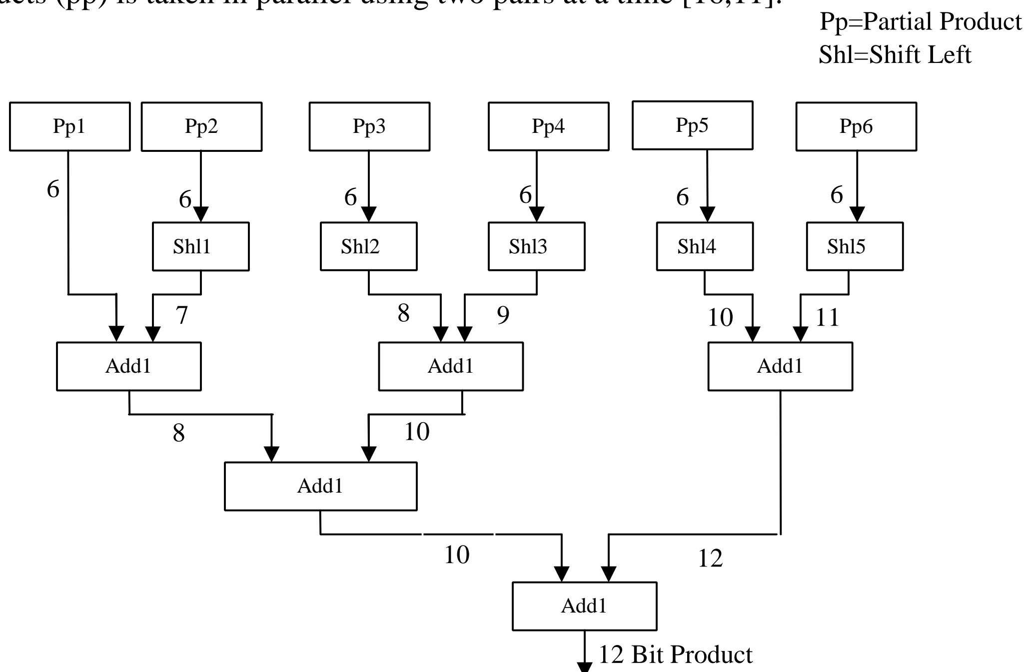


Figure-4.5 Shift and Parallel Add Multiplier

This type of multiplier circuits can use carry lookahead adder to propagate the adder result fast. This multiplier circuit is pure combinatorial as shown in Figure 4.5 and can be easily pipelined to use in high performance system by using registers in front of adders.

`

4.2.2 Ripple Carry Array Multiplier

Advances in VLSI technologies have made it possible to build combinational circuits that perform n * n multiplication for fairly large values of n [4]. These are composed of arrays

of simple combinational elements, each of which implements an add/subtract-and-shift operation for small slices of the multiplication operands. Suppose that two binary numbers X=$x_{n-1} \ldots x_0$ and Y= $y_{n-1} \ldots y_0$ are to be multiplied. For simplicity assume that X and Y are unsigned integers. The product P=X*Y can therefore can be expressed as

$$P = \sum_{i=0}^{n-1} 2^i x_i y \qquad \ldots\text{Eq.4.1}$$

corresponding to the bit by bit multiplication style of Figure 4.6. Now this equation can be written as

$$P = \sum_{i=0}^{n-1} 2^i \sum_{j=0}^{n-1} x_i y_j 2^j \qquad ..\text{Eq.4.2}$$

Each of the n^2 1-bit product terms $x_i y_j$ appearing above can be computed by a two input AND gate. Hence the terms summed according to the Eq 4.2 by an array of n(n-1) one bit full adders as shown in Figure 4.6. This circuit is kind of two dimensional ripple carry adder. A ripple carry array multiplier (also called row ripple form) is an unrolled embodiment of the classic shift-add multiplication algorithm. In this type of multiplier the delay is proportional to the N bit. The Figure 4.9 here has shown below uses the two types of blocks for the row1 and rest of rows.

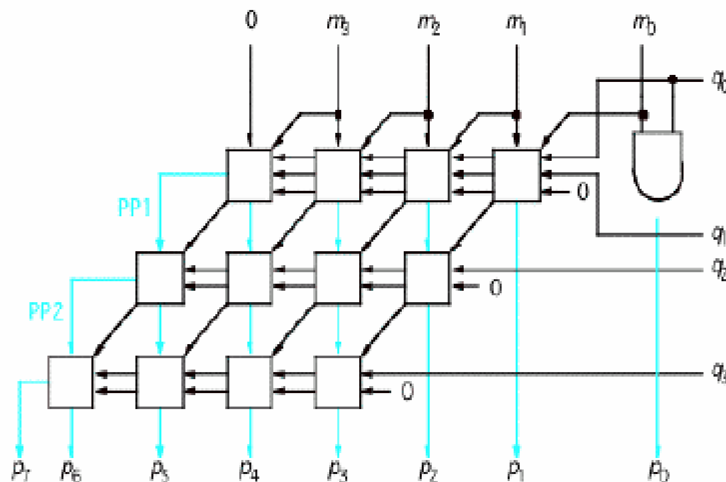


Figure-4.6 Circuit for 4*4 Ripple Array Multiplier

FA denotes the full adder circuit. The carry out of the each block is rippled into the next

block. This is slower as carry is rippled into each block, despite the $x_i y_i$ is computed in the next rows. The delay is approximately equal 2 * n. An improved type of this multiplier is Carry save array multiplier.

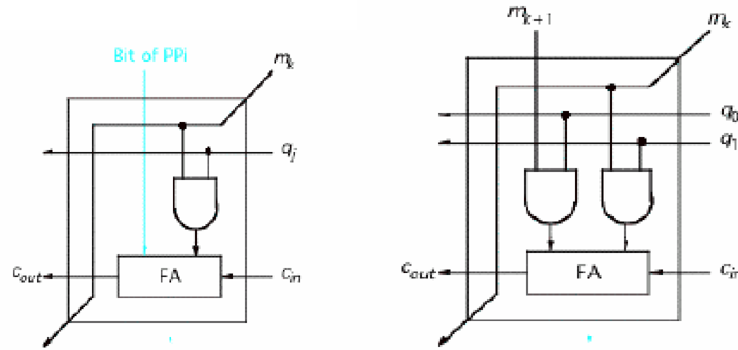


Figure-4.7(a) Block in To p row **Figure-4.7(b) Block in Bottom two Row**

4.2.3 Carry Save Array Multiplier

This type of multiplier are really fast because here ripple of carry is through the column rather than row wise. Secondly there is regular routing pattern. The Figure 4.8 shows the structural architecture to build CSA multiplier. The AND gate and add functions of the array can be combined into the single component (cell) shown in Figure 4.9. This cell

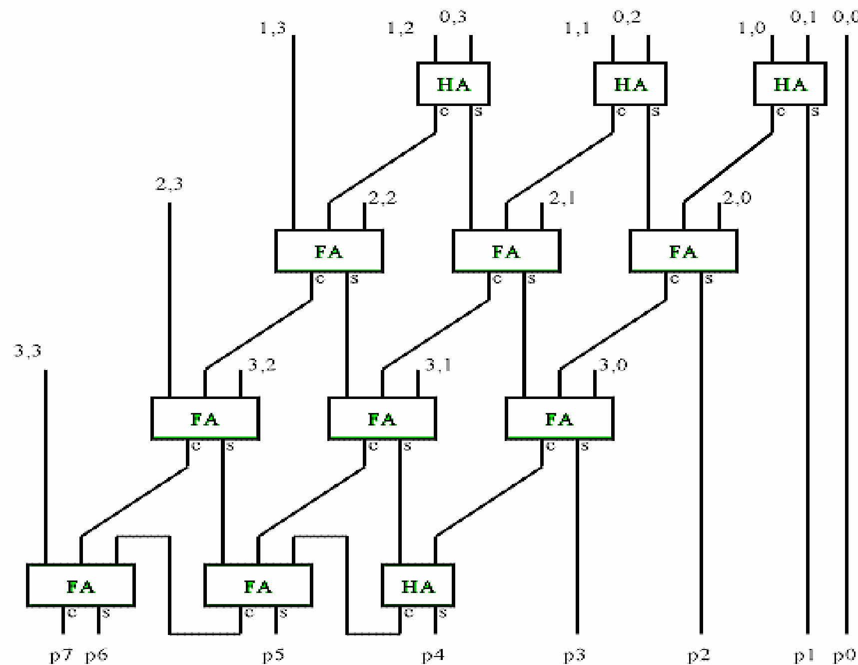


Figure-4.8 Carry Save Array Multiplier

now realizes the arithmetic expression

c_{out} s = a plus b plus xy

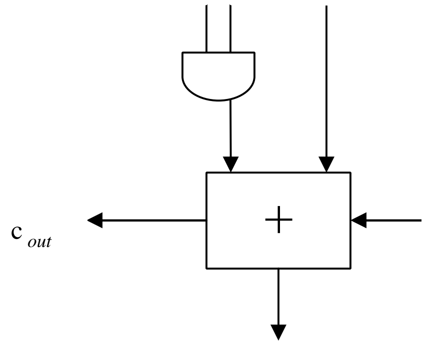


Figure-4.9 Cell for CSA Multiplier

An n*n multiplier can be built using n^2 copies of this cell as the sole component. Some cells on the periphery of array have inputs set to 0 or 1, effectively reducing their operation from a plus b *plus* xy to a *plus* b (half adder). The multiplication time for this multiplier is less than that of a ripple carry array multiplier.

Chapter 5

Algorithm, Flow Chart and Results

5.1 Algorithm

To implement the design, and indigenous algorithm is devised. Algorithm realization steps are summarized below:

(1) Design a ROM (127 x 4) for Addition.
(2) Design a ROM (64 x 6) for Multiplication.
(3) Devise a control mechanism by having 4 bit binary number(say d1) to select the various operands.

(i) When d1 = "0000" Add two 15 bit number

(a) Initialize I to 0 and S to '0'
(b) Provide three bits (I to I+2) of augend, three bits of addend and S (carry) to the ROM. Store the output of the ROM in Z (I to I +3).
(c) I = I + 3 , S = Z (I)
Repeat i(b) and i(c)until I < 16.

(ii) When d1 = "0010" Subtract two 15 bit number

(a) Initialize I to 0 and S to '1'
(b) Provide three bits (I to I+2) of augend, complement three bits of addend and S (carry) to the ROM. Store the output of the ROM in Z (I to I +3).
(c) I = I + 3 , S = Z (I)
Repeat ii(b) and ii(c)until I < 16.
(d) Design a 24 bit adder and subtractor in accordance with step i(a) to i(c) and step ii(a) to ii(c).
(e) Design a 8 bit subtractor in accordance with step ii(a) to ii(c).

(iii) When d1 = "0100" Multiply two 7 bit numbers.

(a) Multiplication is done with the aid of ROM for multiplier.
(b) Multiply multiplicand to multiplier$(0)_8$ to get the first partial product.
(c) Multiply multiplicand to multiplier$(1)_8$ to get the second partial product.
(d) Multiply multiplicand to multiplier$(2)_2$ to get the third partial product

(e) Addition of partial products is done with the aid of ROM used in Step (i).

(iv) When d1 = "0101" Divide two 7 bit numbers.

(a) A variable (V_1) of double size of dividend is taken and store the dividend in LSB of variable, while the MSB of variable consists of all Zeroes.
(b) Take variable V_3 of 8 bit which has first MSB as zero and divisor is stored in the remaining bits.
(c) Initialize I = 0
(d) Logical shift left of V_1 by one bit is performed.
(e) Take variable V_2 of 8 bit and store '1' in MSB followed by first 7 bit of V_1 .
(f) Apply $V_2 = V_2 - V_3$.
(g) Check MSB of V_2. If it is '1' then LSB of V_1 is '1' and store the 7 bit LSB of V_2 in 7 bit MSB of V_1

$I = I + 1;$ Repeat iv (a) to iv (g) until $I < 7$.

(v) When d1 = "0110" Rotate left shift by one bit.

(a) Rotate left by one bit; and result is stored to the output register.

(vi) When d1 = "0111" Rotate right shift by one bit.

(a) Rotate right by one bit; and result is stored to the output register.

(vii) When d1 = "1000" Bitwise logical AND of two 15 bit numbers.

(a) Logical AND is performed by ANDing every bit of first input with the corresponding bit of second input.

(viii) When d1 = "1001" Bitwise logical OR of two 15 bit numbers.

(a) Logical OR is performed by taking logical OR of every bit of first input with the corresponding bit of second bit

(ix) When d1 = "1010" Bitwise logical XOR of two 15 bit numbers.

(a) Logical XOR is performed by taking logical XOR of every bit of first input with the corresponding bit of second bit

(x) When d1 = "1011" Bitwise logical NOR of two 15 bit numbers.

(a) Logical NOR is performed by taking logical NOR of every bit of first input with the corresponding bit of second bit

(xi) When d1 = "1100" Bitwise logical NAND of two 15 bit numbers.

(a) Logical NAND is performed by taking logical NAND of every bit of first input with the corresponding bit of second bit

(xii) When d1 = "0001" Add two 32 bit single precision floating point number.

(a) Take two 32 bit number Z_1 and Z_2.
(b) $Z_1(0)$ and $Z_2(0)$ are sign bit denoted by signa and signb.
Z_1(1 to 8) and Z_2(1 to 8) are exponent bits denoted by expa and expb.
Z_1(9 to 31) and Z_2(9 to 31) are fraction bits denoted by fracta and fractb.
(c) Take a variable V_1 and V_2 of 32 bit each.
Compare expa and expb
If expa > expb
$V_1 = Z_1$
$V_2 = Z_2$
Else if expb > expa
$V_1 = Z_2$
$V_2 = Z_1$
Else Compare fracta and fractb
If fracta > fractb
$V_1 = Z_1$
$V_2 = Z_2$
Else fractb > fracta
$V_1 = Z_2$
$V_2 = Z_1$

(d) Compare signa and signb
If signa = signb and operation is add then sign of result is signa.
If signa /= signb and operation is subtract then sign of result is signa.
If signa = signb and operation is subtract then sign of result is Sign of V_1 .
If signa /= signb and operation is add then sign of result is Sign of V_1.
(e) Subtract the expa and expb and store the result. And convert into integer form in I_1.
(f) Identify the smaller exponent and logical shift right the fraction part of the number by I_1 after appending '1' at the MSB of fraction part.
(g) Add the fraction parts based on the operation in accordance with step i(a) to i(c) and step ii(a) to ii(c).
(h) Logical shift the result by one bit and decrease the exponent by 1
Repeat step (h) until MSB of result = '1'

(xiii)When d1 ="0011" Subtract two 32 bit single precision floating point number.
Go to xii(a) to xii(f)
(a) Subtract the fraction parts based on the operation in accordance with step i(a) to i(c) and step ii(a) to ii(c).
Go to xii(h)

(xiv) When d1 = "1110" Divide two single precision floating point number.

(a) A variable (V_1) of double size of dividend is taken and store the dividend in LSB of variables, while the rest bit of variable consists of all Zeroes.

(b) Take variable V_3 of 24 bit which has MSB as zero and divisor is stored in the remaining bits.

(c) Initialize I = 0

(d) Logical shift left of V_1 by one bit is performed.

(e) Take variable V_2 of 24 bit and store '1' in MSB followed by first 23 bit of V_1 .

(f) Apply $V_2 = V_2 - V_3$.

(g) Check MSB of V_2. If it is '1' then more LSB of V_1 is '1' and store the 23 bit LSB of V_2 in 23 bit MSB of V_1

I = I + 1 ;

Repeat xiv (a) to xiv (g) until I < 24

(h) Compare Signa and Signb

If signa = signb

Sign of the result is positive

Else

Sign of the result is negative.

(i) In case of division, subtract the exponent and divide the fraction part using ROM and $(127)_{10}$ is added in the exponent result.

(j) Normalize the result as in xii (h).

(xv) When d1 = "1111" Multiply two single precision floating point number.

Go to xiv (h) .

(a) In case of multiplication, add the exponent and multiply the fraction part using ROM and $(127)_{10}$ is subtracted in the exponent result.

Go to xiv (j) .

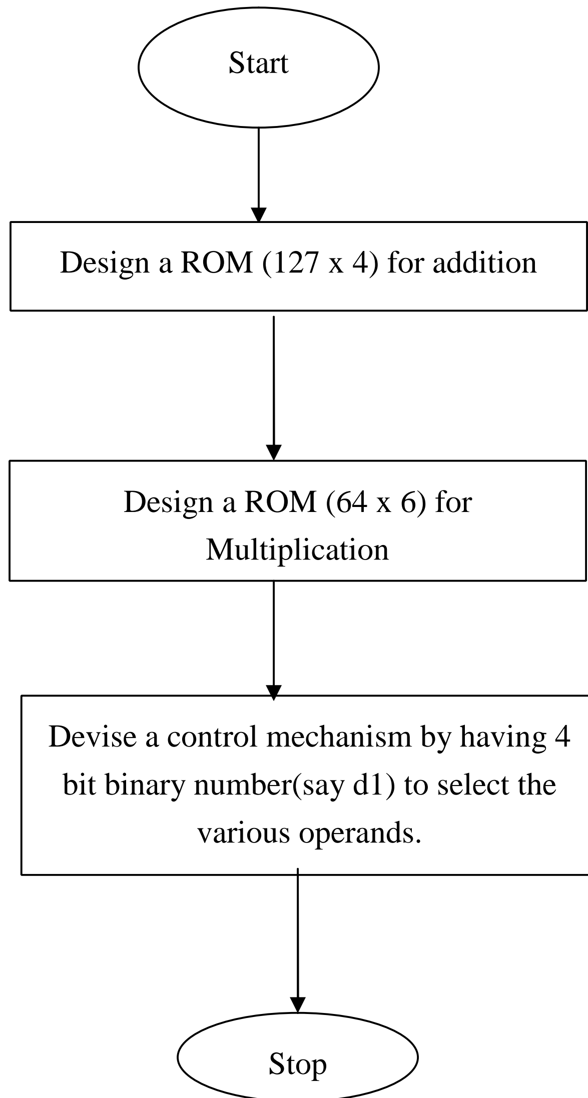
(xvi) When d1 = "1101" Bitwise logical NOT of a 15 bit number.

STOP

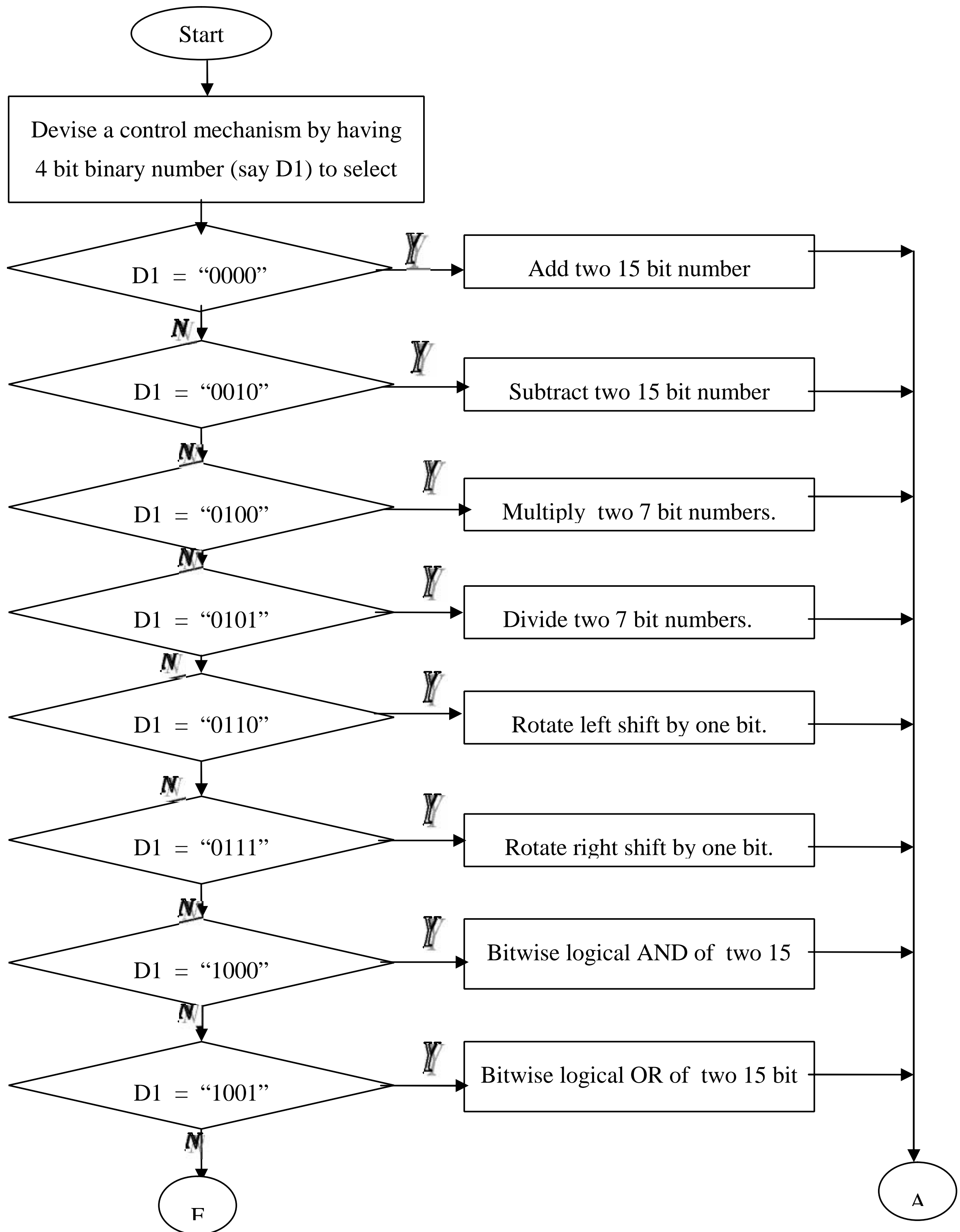
5.2 Flow Chart

The implementation steps for realization of the proposed design are described in the following flow charts. Flow charts for different building blocks of ALU are drawn separately both for the fixed and floating type numbers.

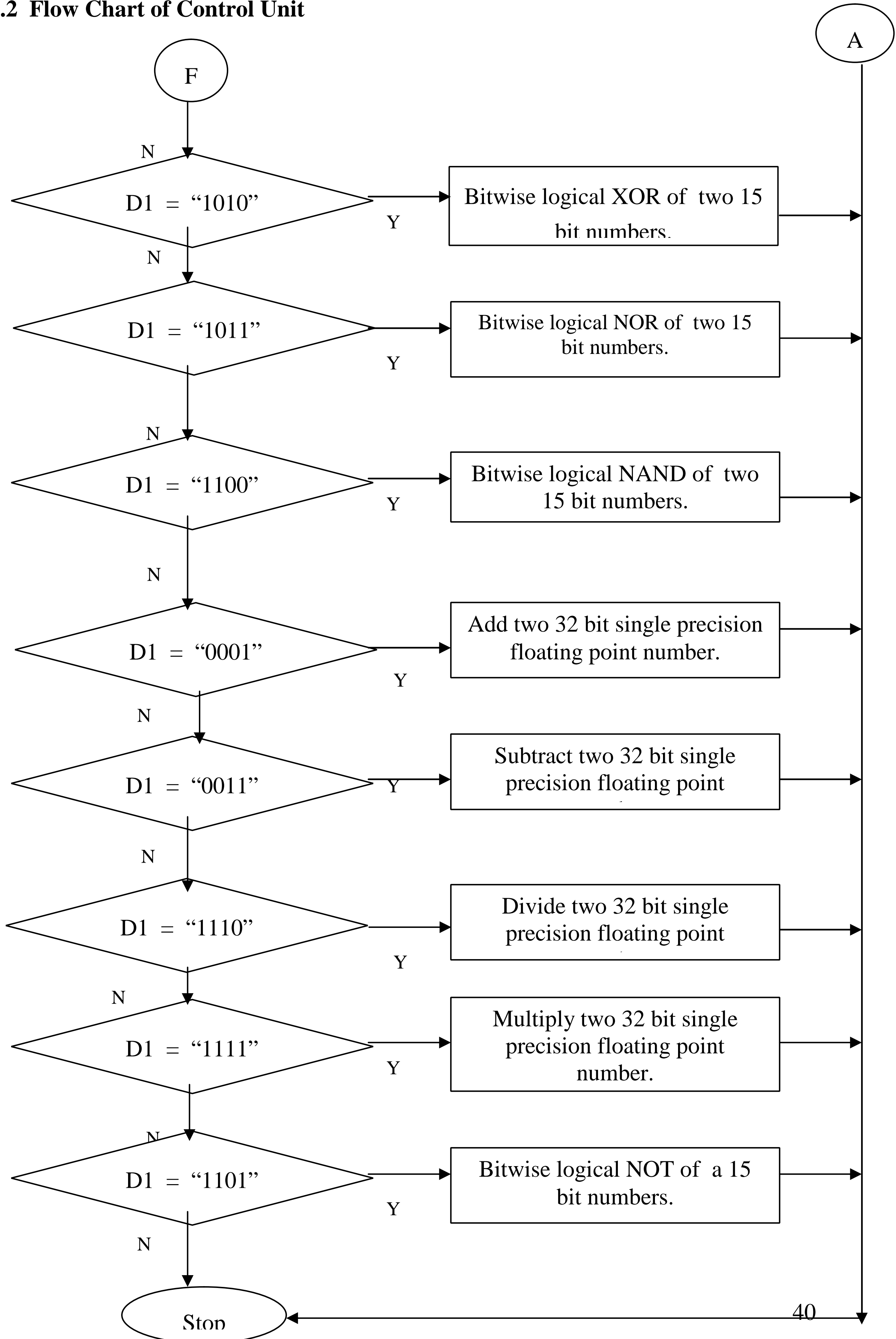
5.2.1 Flow Chart of High Speed ALU



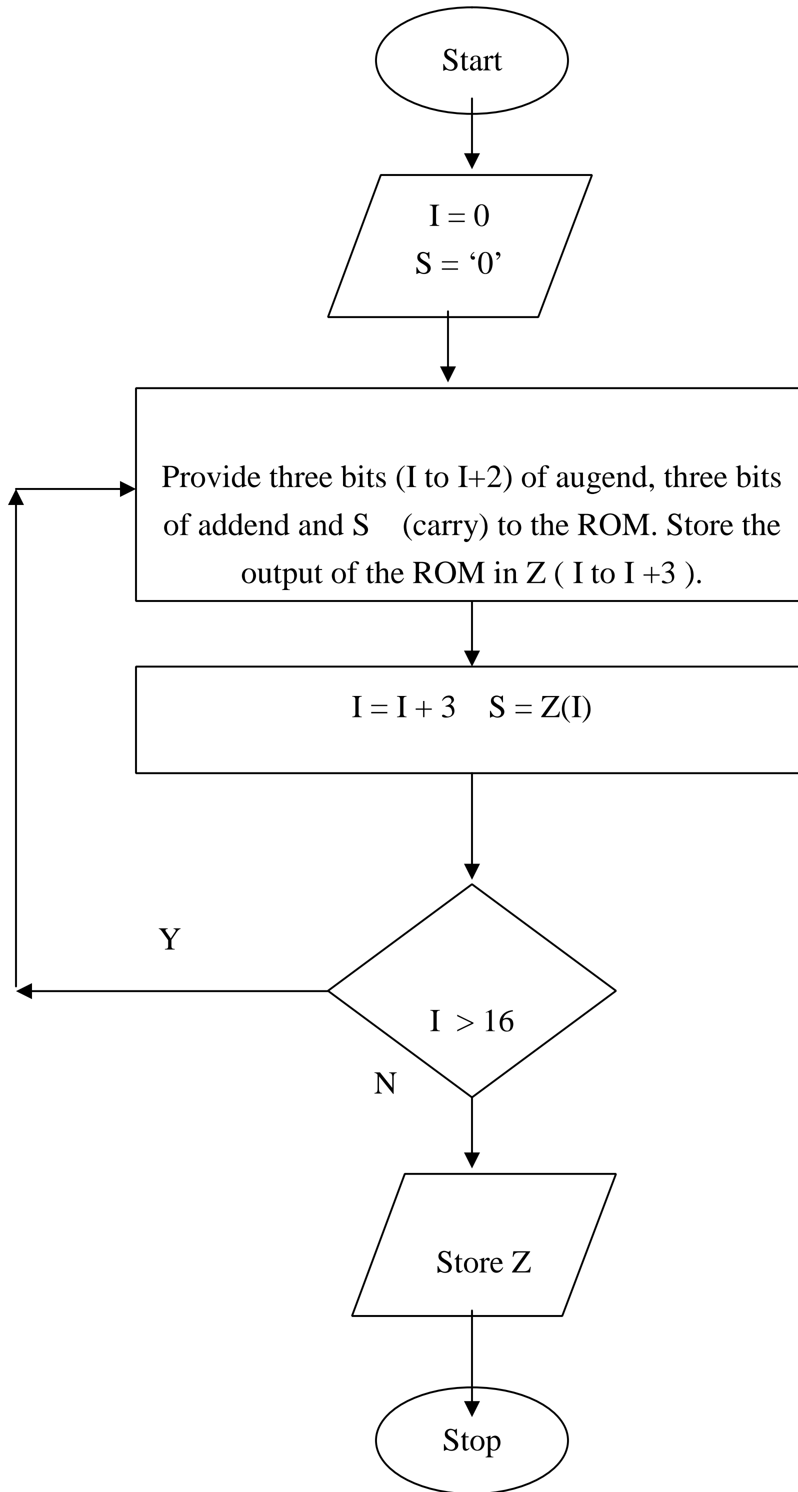
5.2.2 Flow Chart of Control Unit (Continue…)



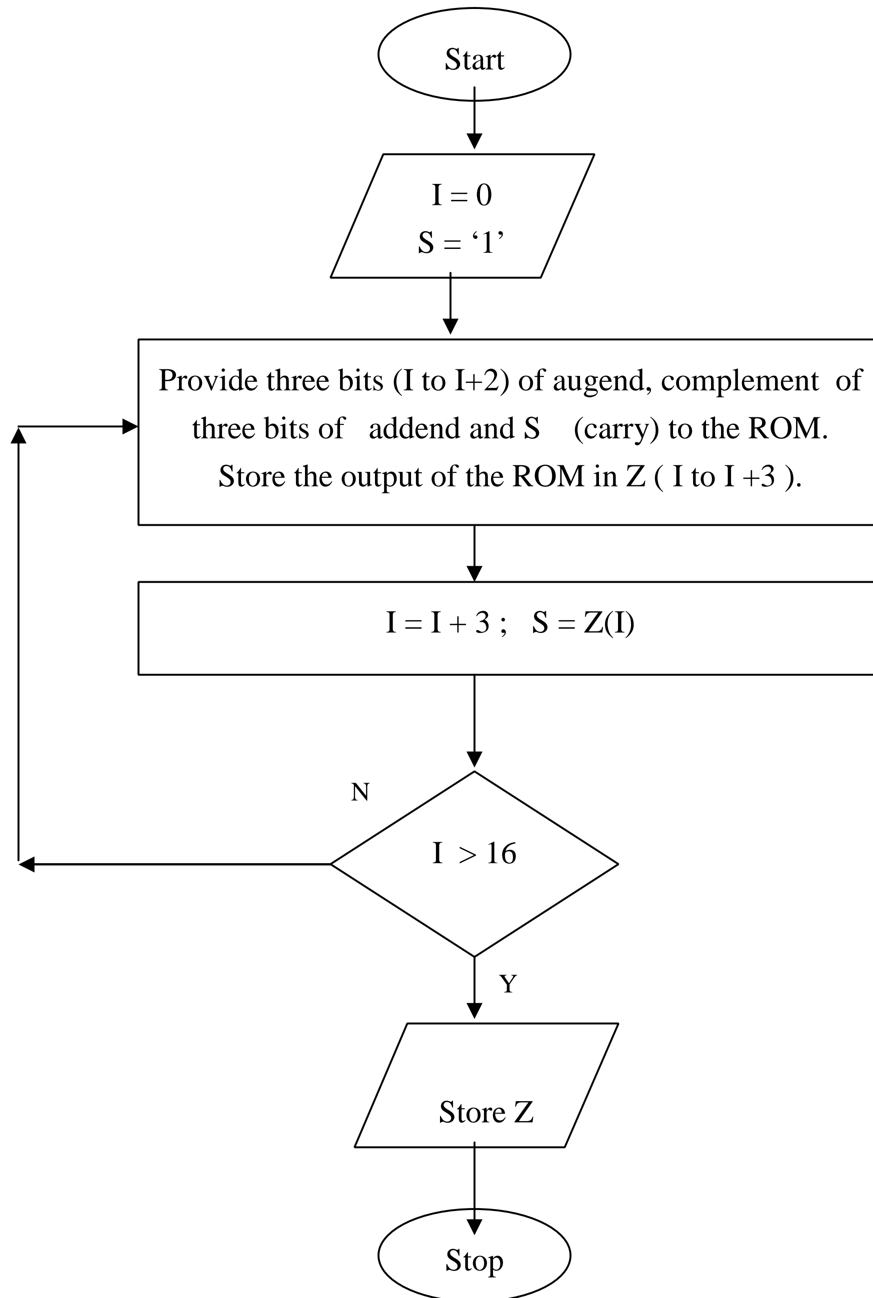
5.2.2 Flow Chart of Control Unit



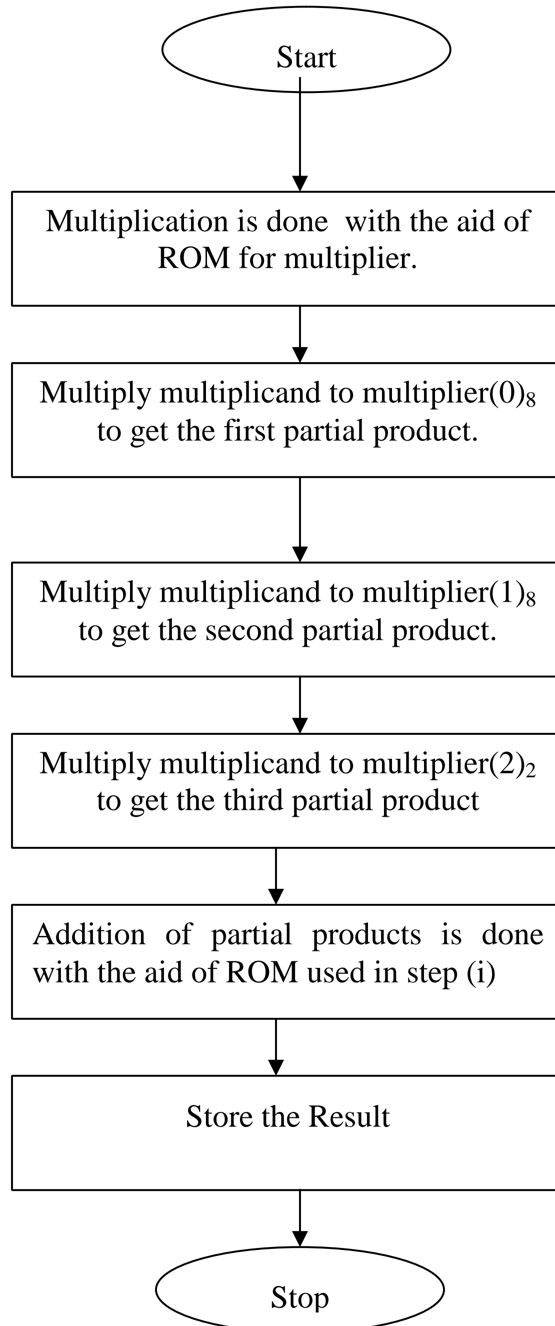
5.2.3 Flow Chart of Addition of two 15 Bit Numbers



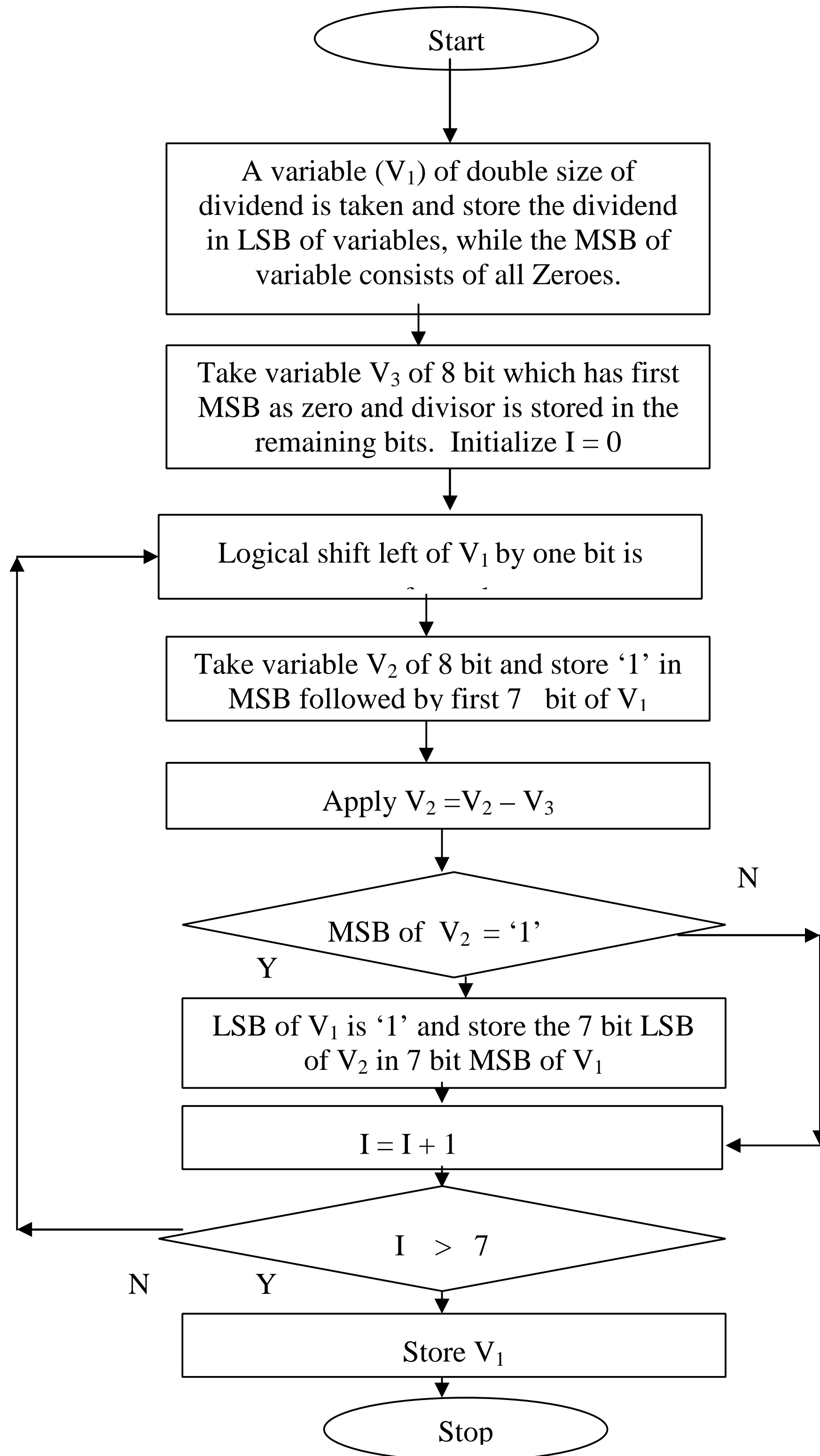
5.2.4 Flow Chart of Subtraction of two 15 Bit Numbers



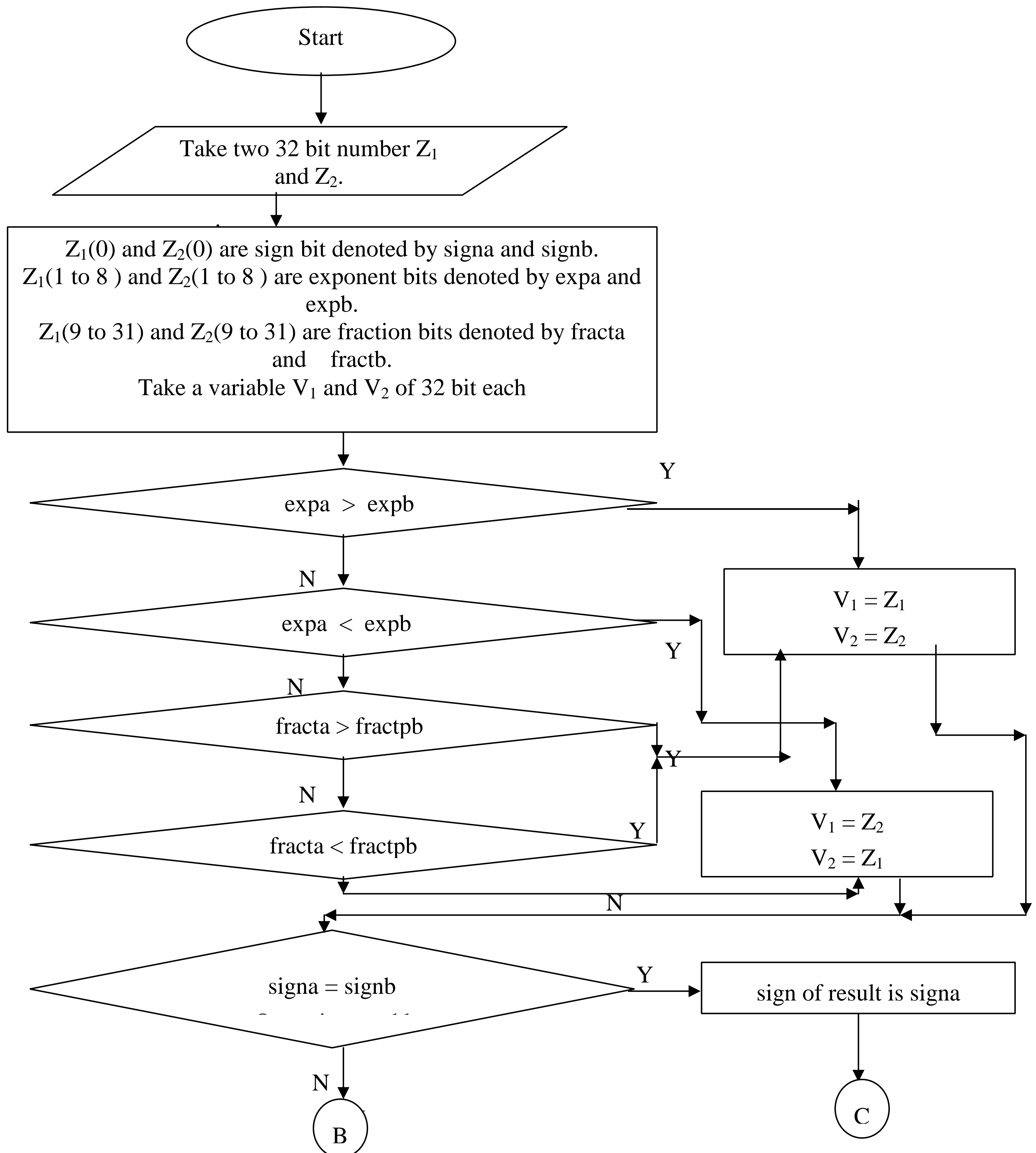
5.2.5 Flow Chart of Multiplication of two 7 Bit Numbers



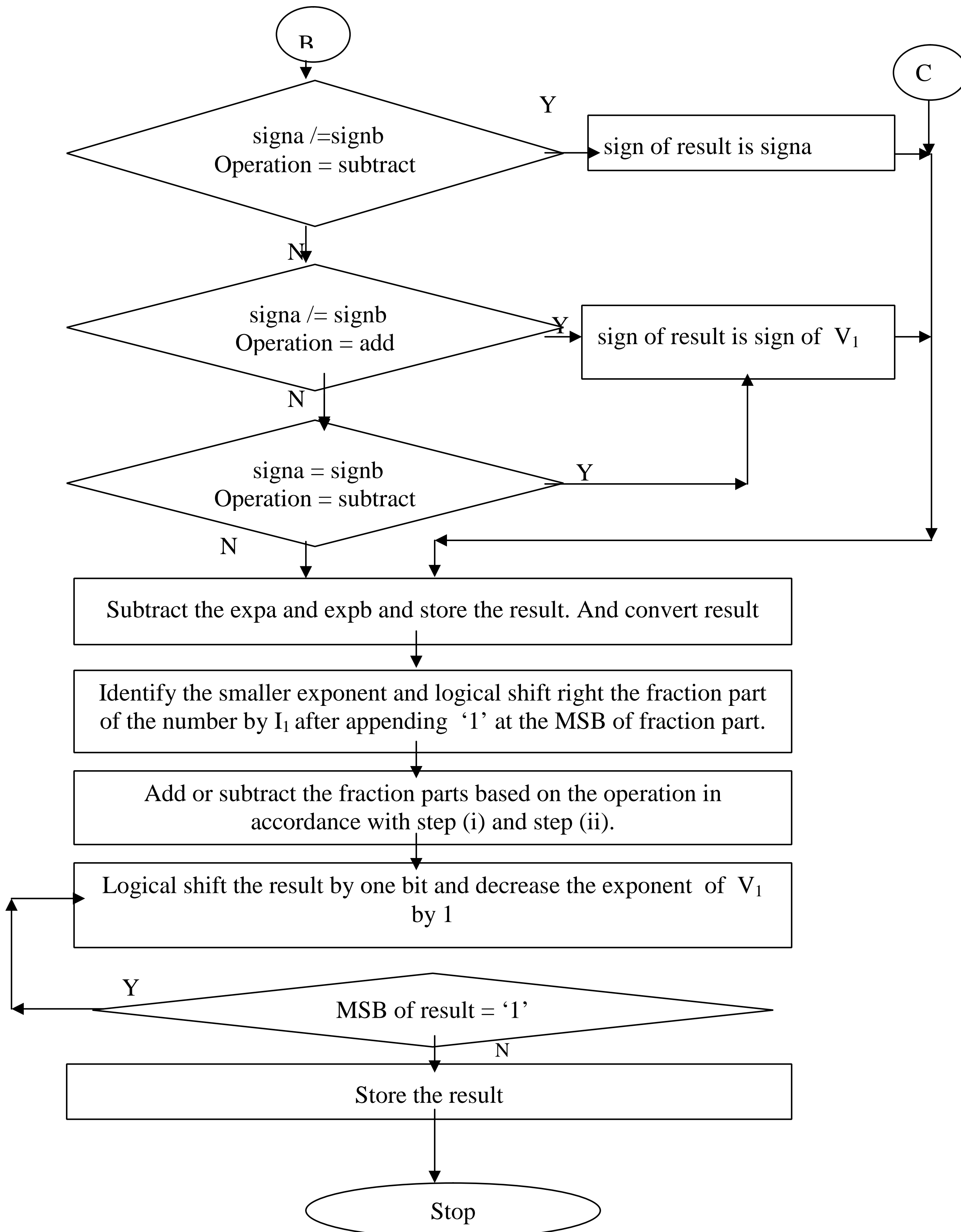
5.2.6 Flow Chart of Division of two 7 Bit Numbers



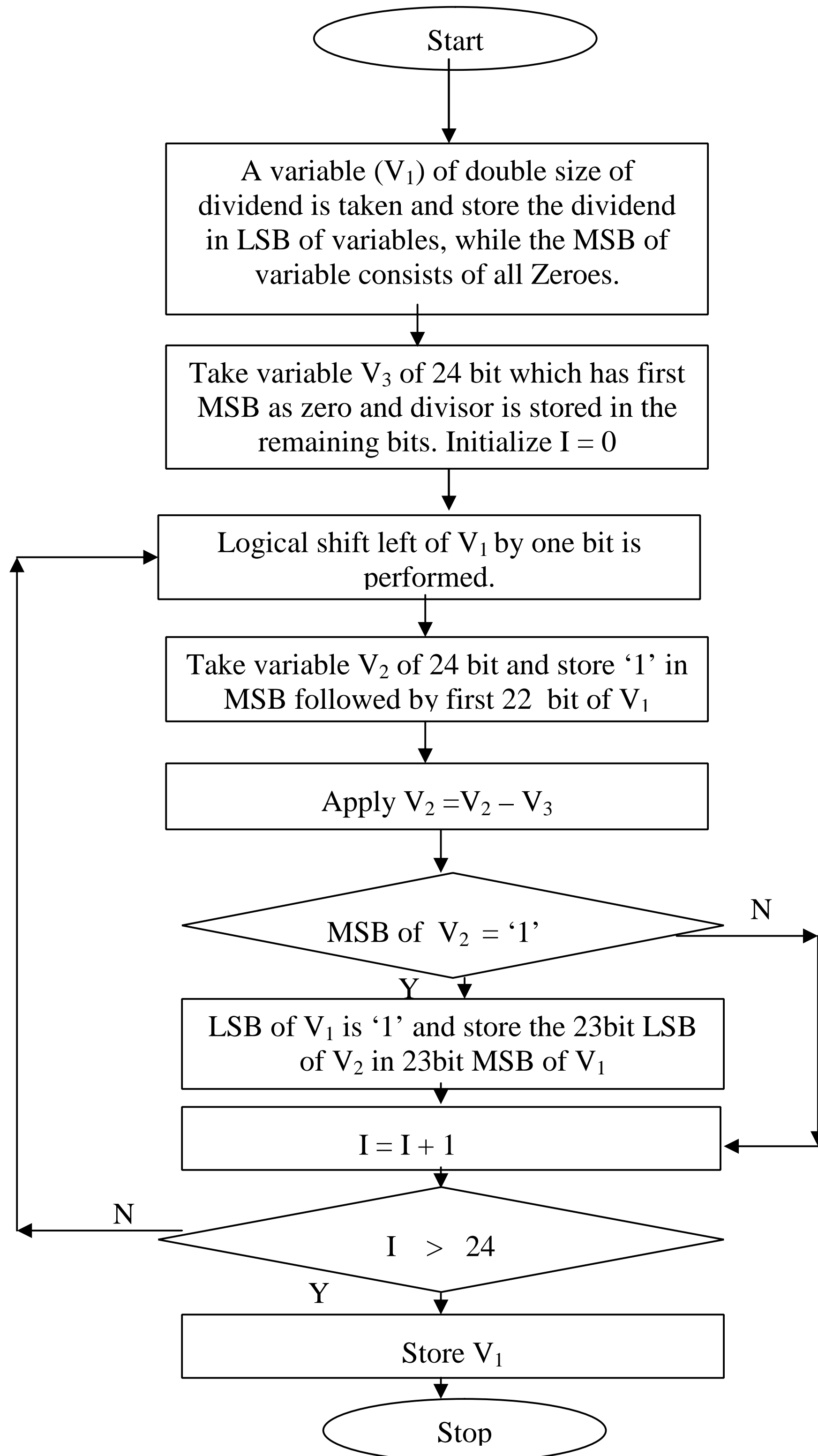
5.2.7 Flow Chart of Addition and Subtraction of two Floating Point Numbers(Continue…)



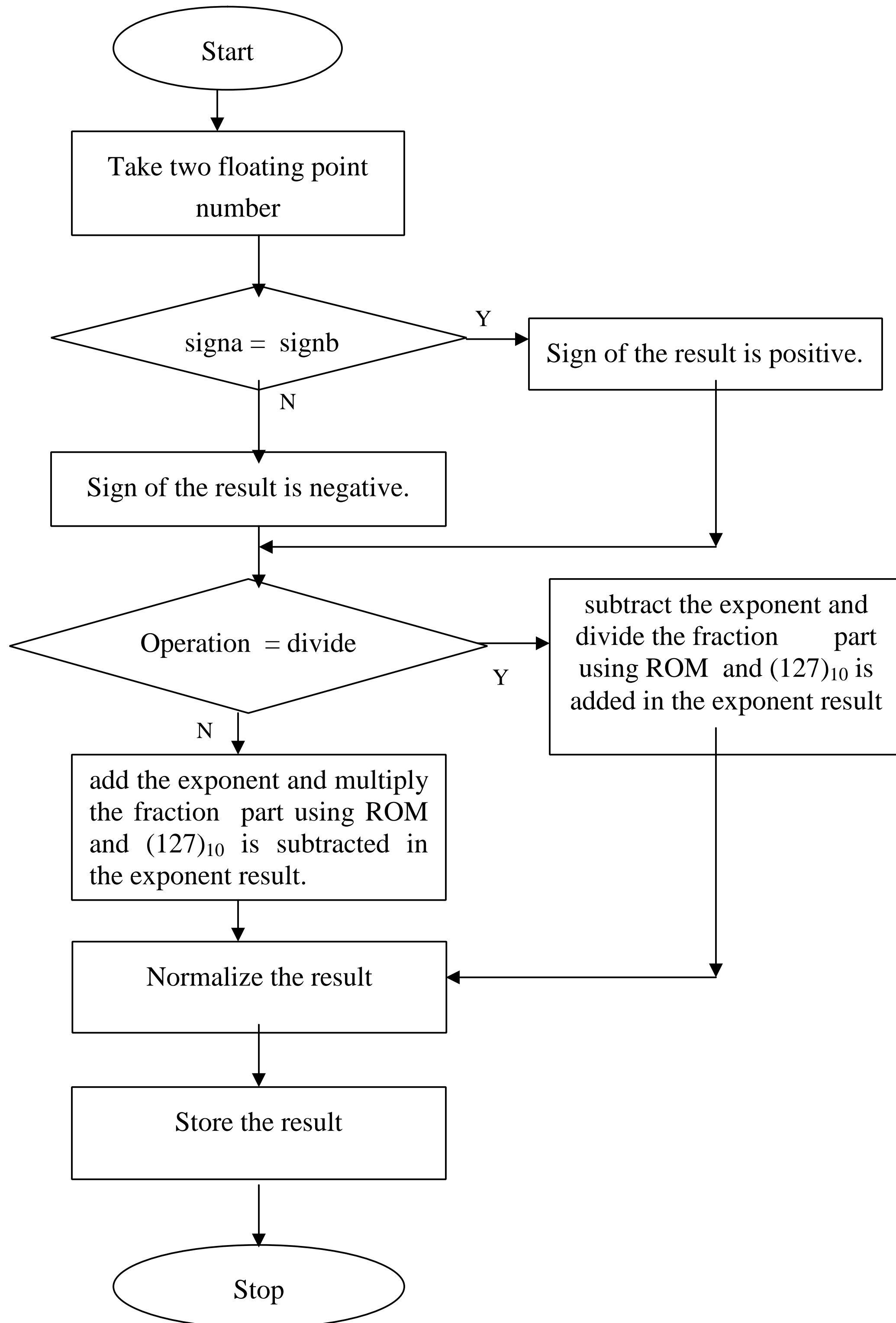
5.2.7 Flow Chart of Addition and Subtraction of two Floating Point Numbers



5.2.8 Flow Chart of Division of two 22 Bit Numbers



5.2.9 Flow Chart of Multiplication and Division of two Floating Point Numbers



5.3 Results & Discussion

All the results that are obtained are based on the PLD design flow. Firstly behavioral simulation model is generated, then the designs are synthesized and the synthesis report is generated. Carry lookahead adder and carry save array multiplier are coded in VHDL, so as to compare them with their counterparts in the proposed design. The tool used for all these is Xilinx ISE 6.1i. The Device family used is Spartan 2E in FPGA and XC9500 in CPLD.

Simulation results

a) Figure 5.1 shows the simulation result of fixed point 15 bit adder.
b) Figure 5.2 shows the simulation result of fixed point 15 bit subtractor.
c) Figure 5.3 shows the simulation result of fixed point 7 bit multiplier.
d) Figure 5.4 shows the simulation result of fixed point 7 bit divider.
e) Figure 5.5 shows the simulation result of floating point 32 bit adder.
f) Figure 5.6 shows the simulation result of floating point 32 bit subtractor.
g) Figure 5.7 shows the simulation result of floating point 32 bit multiplier.
h) Figure 5.8 shows the simulation result of floating point 32 bit divider.
i) Figure 5.9 shows the simulation result of logical AND operation.
j) Figure 5.10 shows the simulation result of logical OR operation.
k) Figure 5.11 shows the simulation result of logical NOR operation.
l) Figure 5.12 shows the simulation result of logical NAND operation.
m) Figure 5.13 shows the simulation result of logical XOR operation.
n) Figure 5.14 shows the simulation result of logical NOT operation.
o) Figure 5.15 shows the simulation result of circular shift left operation.
p) Figure 5.16 shows the simulation result of circular shift right operation.
q) Figure 5.17 shows the simulation result of carry lookahead adder.
r) Figure 5.18 shows the simulation result of carry save array multiplier.

The results thus, obtained show that the adders, multipliers, subtractors, divisors and shifters that have been implemented in VHDL, are functionally correct.

5.4 Synthesis Report

Synthesis Reports are generated on Xilinx ISE 6.1i Tool .The results thus, obtained are summarized in three tables.

TABLE 5.1 Synthesis Report Summary of proposed adder and multiplier design along with the carry lookahead adder and carry save array multiplier

	Proposed Adder	Proposed Multiplier	Carry lookahead Adder	Carry save array Multiplier
No. of Slices /768	43	109	27	35
No. of FFs/1536	16	14	17	14
No. of input LUT/1536	81	194	47	58
Min. Arrival Time Before CLK(ns)	11.917	33.494	21.561	40.134
Max CLK to O/P Delay(ns)	6.140	6.140	6.140	6.140

TABLE 5.2 Timing Report Summary of 7 Bit ALU Implemented on CPLD and FPGA

	FPGA	CPLD
Min. Arrival Time Before CLK(ns)/ Set up Time(ns)	11.017	29.500
Max CLK to O/P Delay(ns)/Hold Time(ns)	6.347	4.5
Max Frequency(MHz)	334.784	125

Table 5.1 provides the comparison between carry lookahead adder and carry save

array multiplier with the proposed adder and multiplier. It can be deduced from the table that proposed design overshadows the design of carry lookahead adder and carry save array multiplier in respect of speed.

TABLE 5.3 Synthesis Report Summary of proposed ALU Implemented on FPGA

	FPGA
No. of Slices	4078
No. of FFs	48
No. of input LUT	6494
Min. Arrival Time Before CLK(ns)	228.296
Max CLK to O/P Delay(ns)	6.680
Max. Frequency(MHz)	194.477

Table 5.2 outlines the Timing Report summary of the 7 bit fixed point arithmetic and logical units (based on proposed ALU) implemented both on FPGA and CPLD. It can be asserted that the implementation of the above units on FPGA offers higher performance than the CPLD as far as speed is concerned. Table 5.3 provides the synthesis report summary of the proposed ALU implemented on FPGA.

5.5 Simulation Result

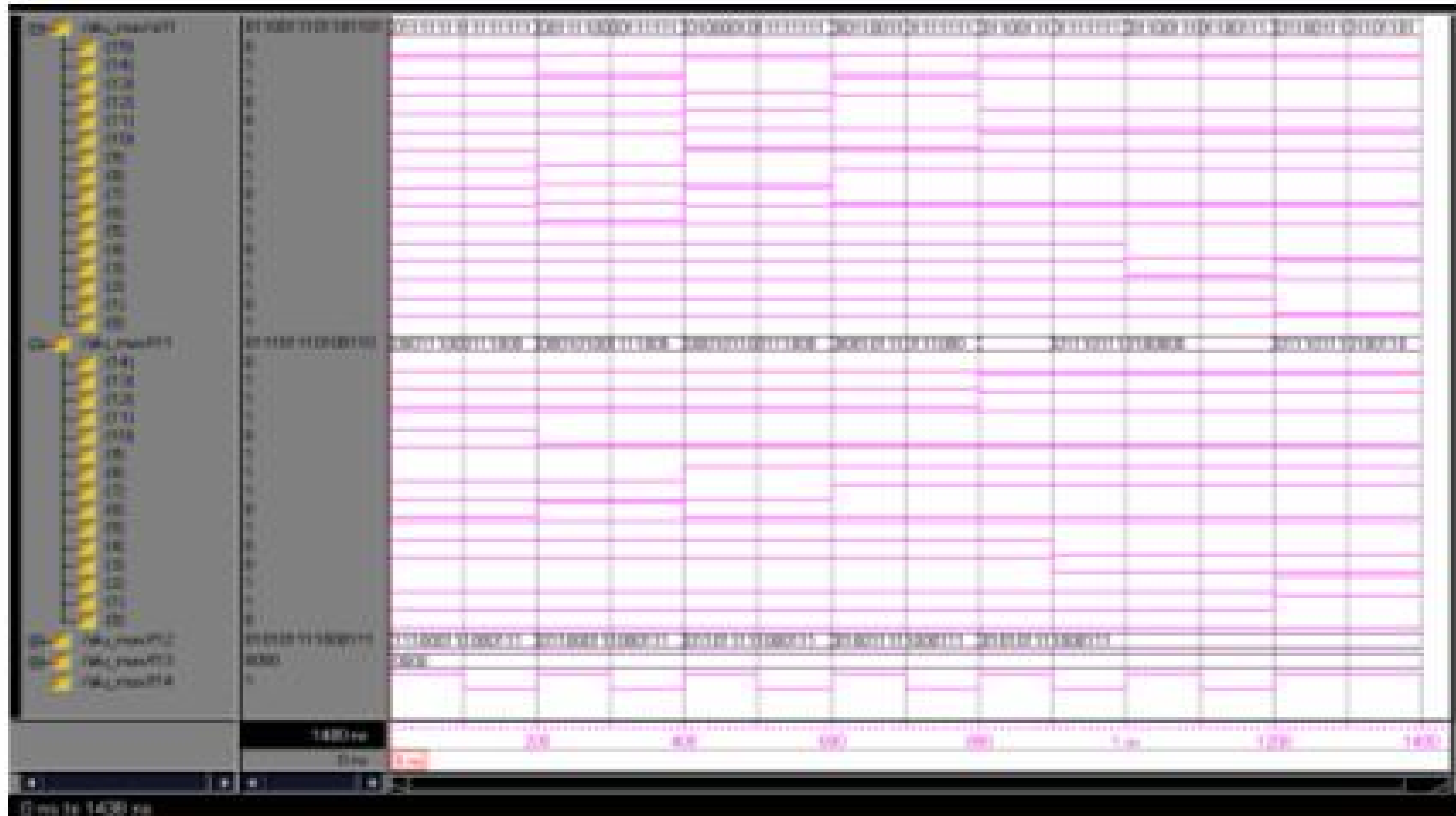


Figure -5.1 ADDITION OF TWO 15 BIT NUMBERS

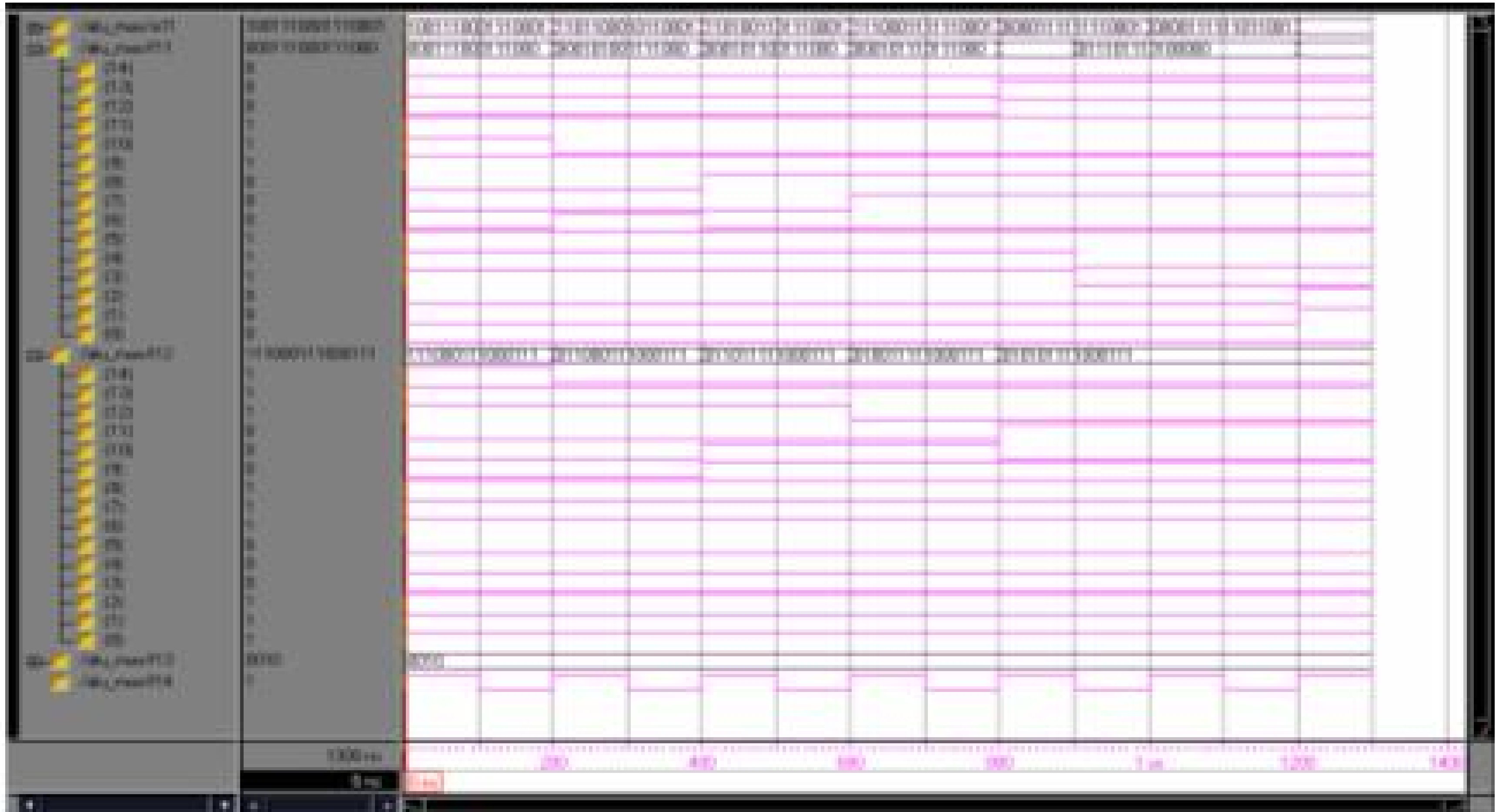


Figure -5.2 SUBTRACTION OF TWO 15 BIT NUMBERS

Figure -5.3 MULTIPLICATION OF TWO 7 BIT NUMBERS

Figure -5.4 DIVISION OF TWO 7 BIT NUMBERS

Figure -5.5 ADDITION OF TWO 32 BIT FLOATING POINT NUMBERS

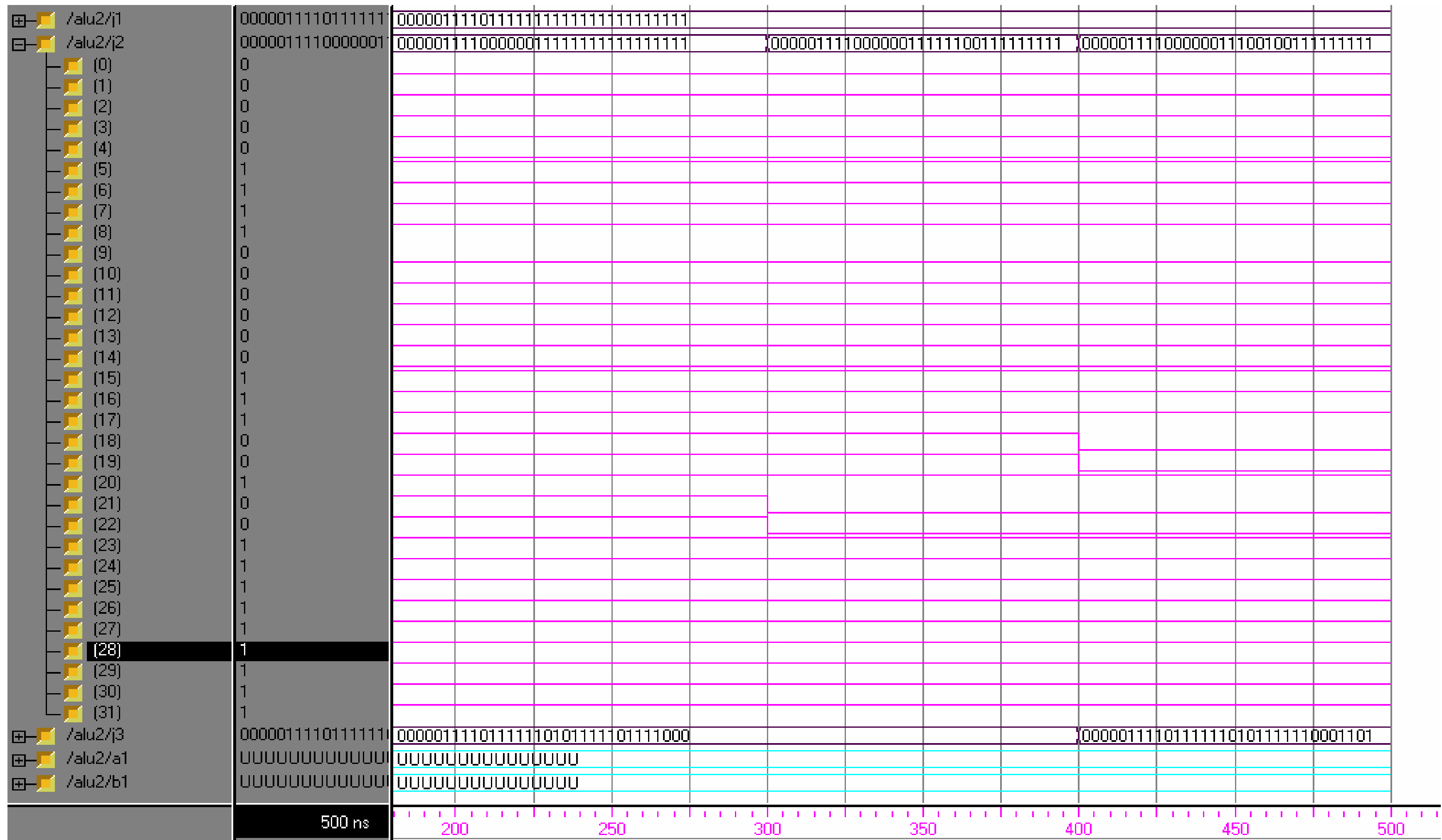


Figure -5.6 SUBTRACTION OF TWO 32 BIT FLOATING POINT NUMBERS

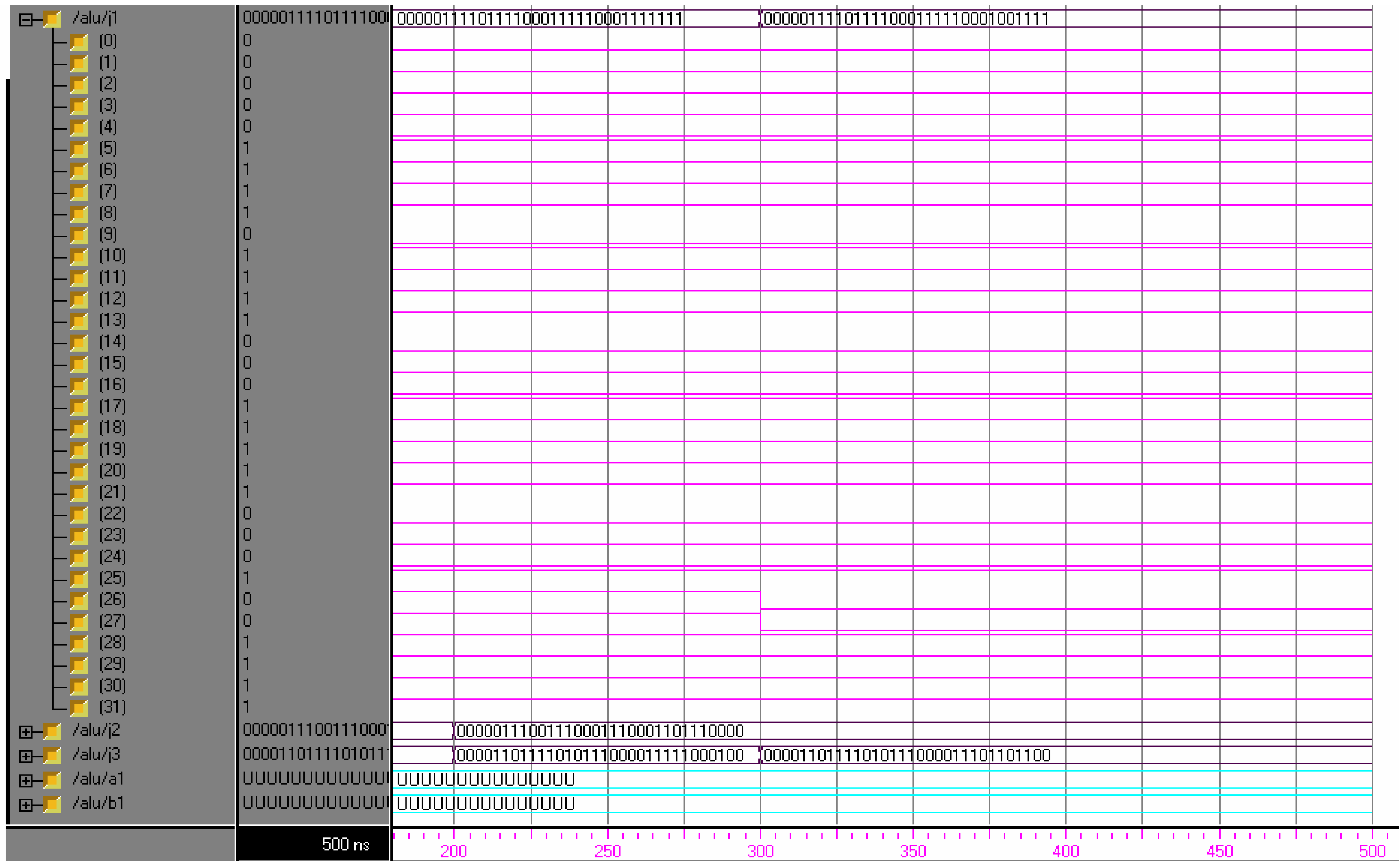


Figure -5.7 MULTIPLICATION OF TWO 32 BIT FLOATING POINT NUMBERS

Figure -5.8 DIVISION OF TWO 32 BIT FLOATING POINT NUMBERS

Figure -5.9 BITWISE LOGICAL AND OPERATION OF TWO 15 BIT NUMBERS

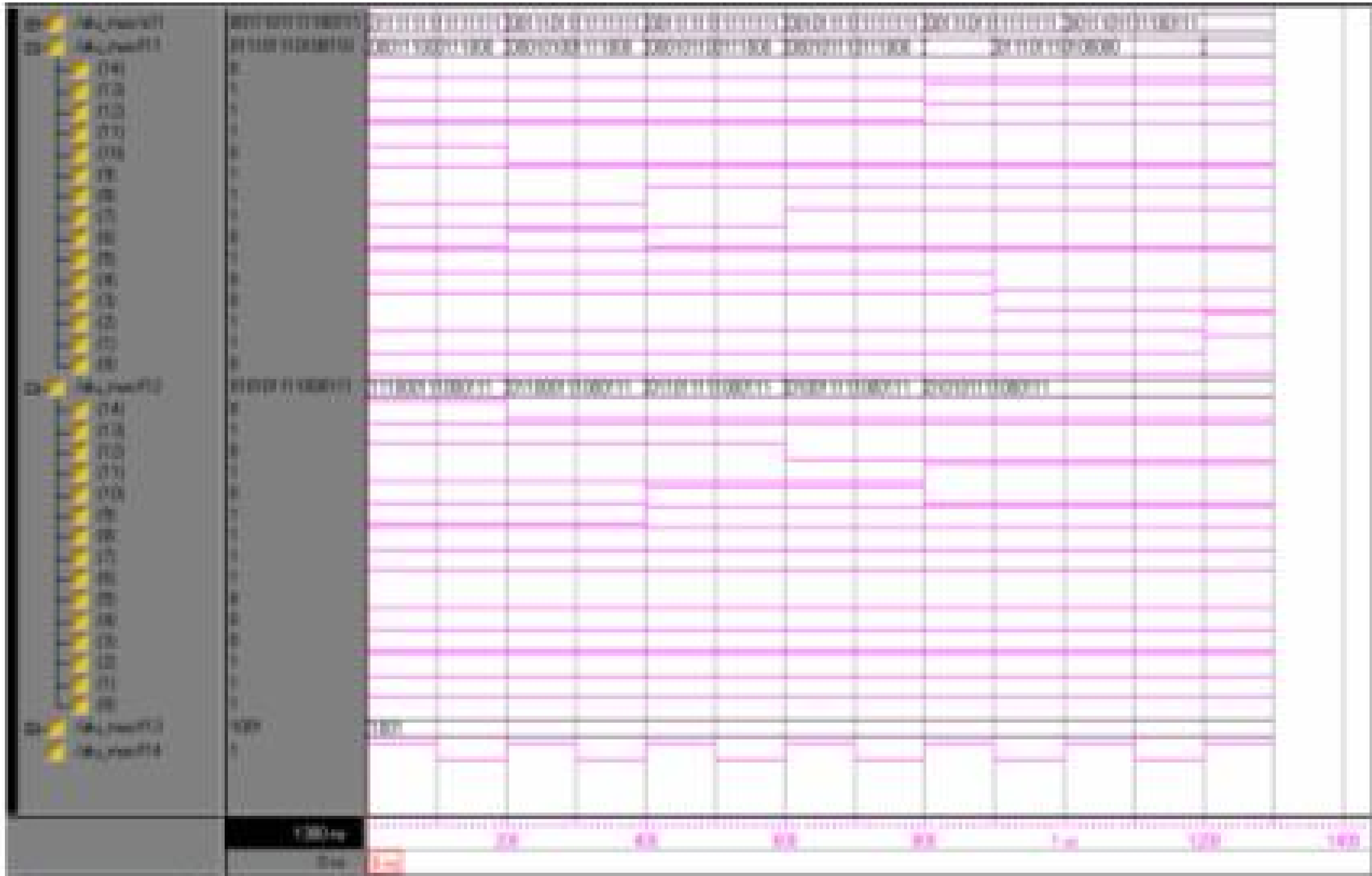


Figure -5.10 BITWISE LOGICAL OR OPERATION OF TWO 15 BIT NUMBERS

Figure -5.11 BITWISE LOGICAL NOR OPERATION OF TWO 15 BIT NUMBERS

Figure -5.12 BITWISE LOGICAL NAND OPERATION OF TWO 15 BIT NUMBERS

Figure -5.13 BITWISE LOGICAL XOR OPERATION OF TWO 15 BIT NUMBERS

Figure -5.14 BITWISE LOGICAL NOT OPERATION OF A 15 BIT NUMBER

Figure -5.15 CIRCULAR SHIFT LEFT

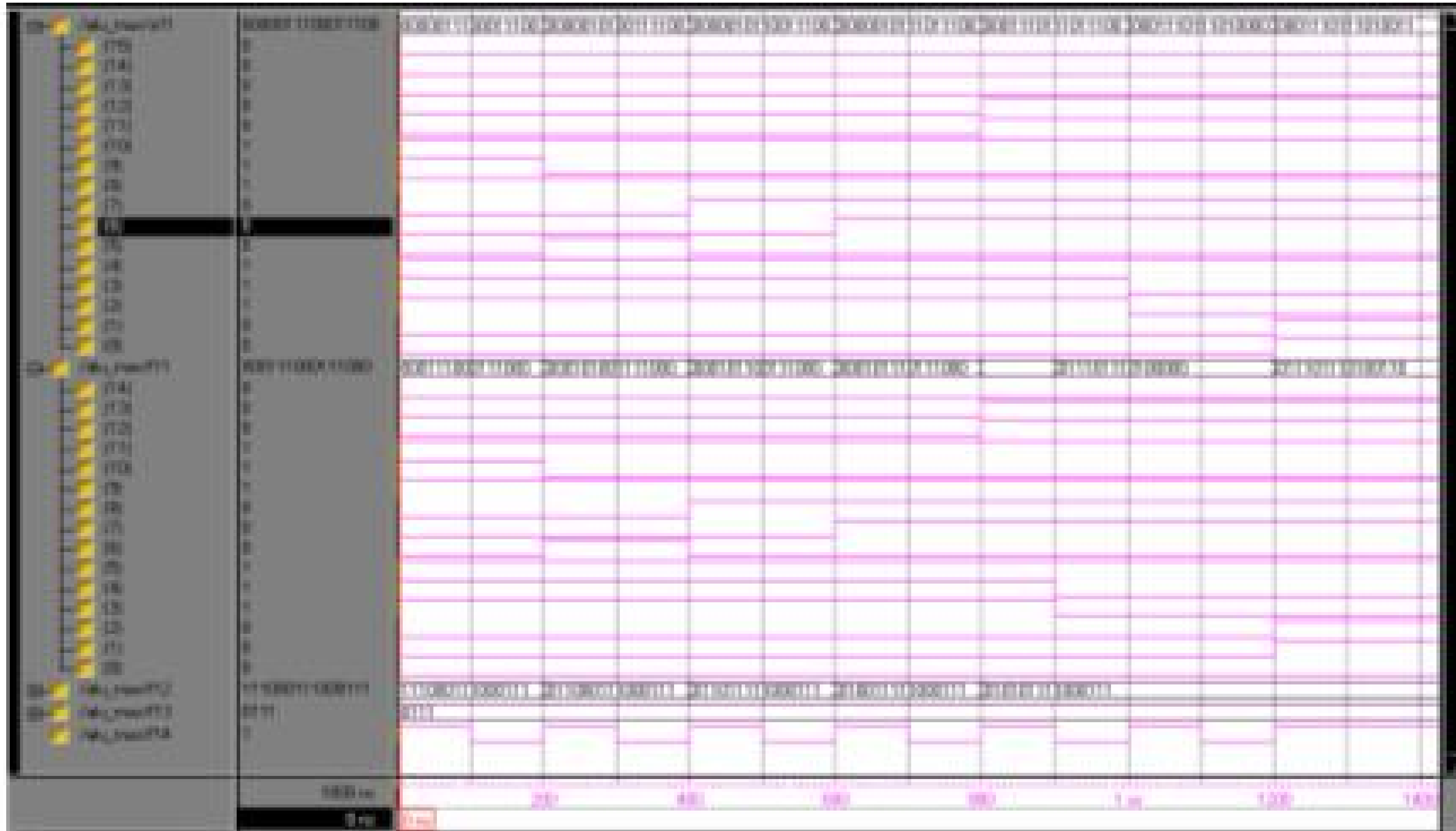


Figure -5.16 CIRCULAR SHIFT RIGHT

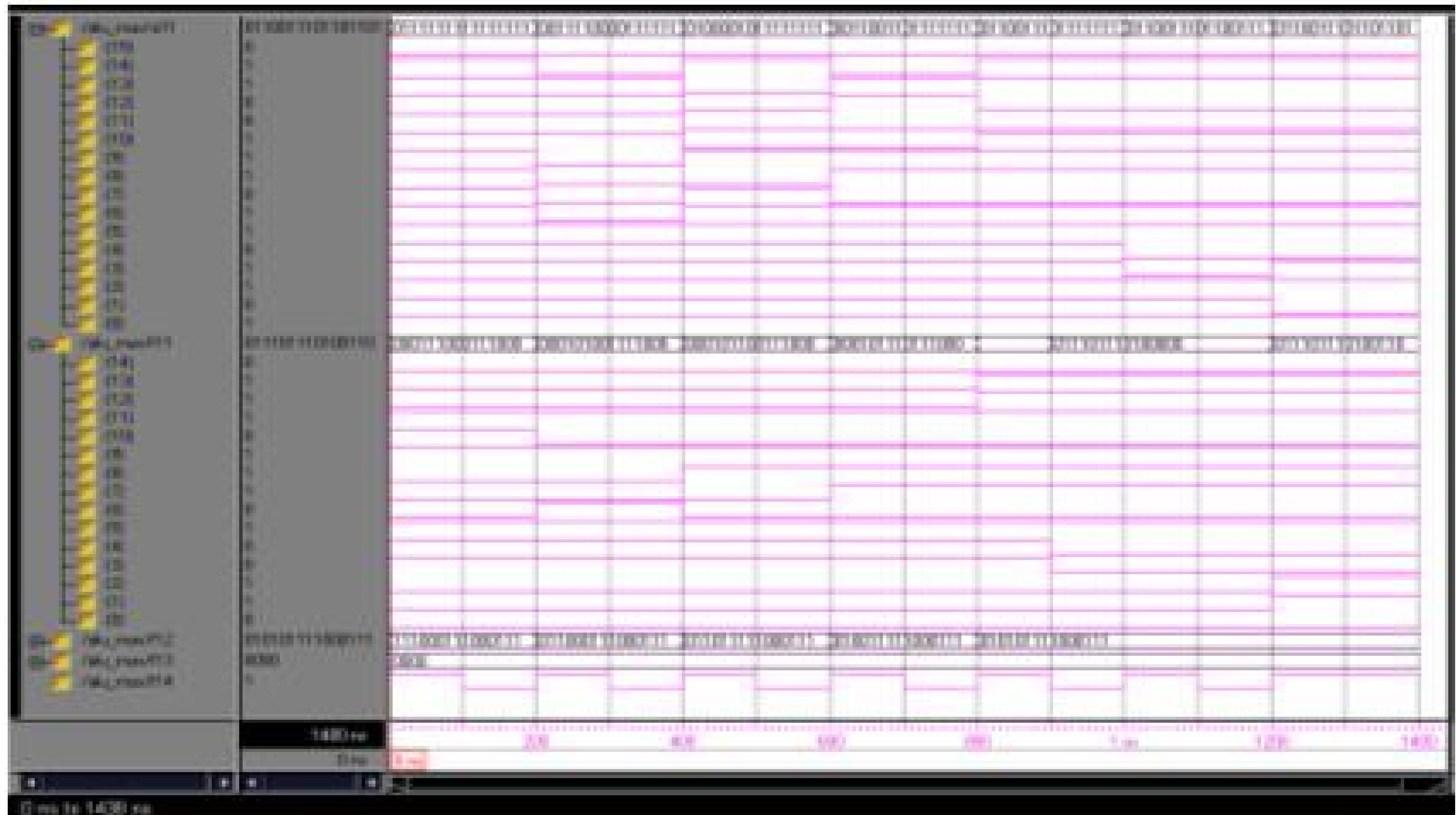


Figure -5.17 ADDITION OF TWO 15 BIT NUMBERS BY CARRY LOOKAHEAD ADDER

Figure -5.18 MULTIPLICATION OF TWO 7 BIT NUMBERS BY CARRY SAVE ARRAY MULTIPLIER

Chapter 6

Conclusion and Future Scope

A novel design of ALU containing two ROMs both for the purpose of addition and multiplication and thereafter for subtraction and division is presented in this thesis. An algorithm is devised in order to felicitate the writing of codes in VHDL. The VHDL coding synthesis issues play a vital role in the speed-area optimality because RTL schematic depends heavily on how we have coded in VHDL. The design is simulated on ModelSim SE 5.5e tool. The different building blocks of the proposed design are verified separately on FPGA and CPLD; results thus obtained are in compliance with the expected result. The design is synthesized by selecting device family spartan2E and XC9500 for FPGA and CPLD respectively. The FPGA differ from the CPLD in that, instead of solving the logic design by interconnecting logic gates, it uses a look up table method to resolve the particular logic requirement. It can be inferred from the synthesis report that FPGAs offer high performance than CPLD as far as speed is concerned. Comparison of the design of proposed adder and multiplier with the carry lookahead adder and carry save array multiplier proves the supremacy of the proposed design over the later.

There are some design issues that could be explored and incorporated in the present design by necessary modifications in future.

- Floating point formats such as double precision can also be used to realize more dynamic range and precision
- The control circuit can be modified to have more operations and to have a leading zeroes count for both the multiplicand and the multiplier, so it would reduce the computation time and power consumption by running less number of adders.
- The design only optimized speed, but others parameters like power dissipation and effect of temperature can be dealt with in future.
- It is worth while to compare results with full custom ASICs to visualize the effect of device technology on the area and performance cost of architectural feature support.

References

[1] A.D. Booth, "A signed binary multiplication technique", Quart. J. Math., vol. IV, part 2, 1951.

[2] Altera, "Programmable logic device family data book", 2000.

[3] B. Fagin *et al*, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on VLSI, vol 2, pp. 365-367,1994.

[4] V.G. Mobdzija, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach", IEEE Transactions on Computers, vol. 45, No, 3, pp. 294-305,1996.

[5] C. Nagendra, M.J. Irwin, and R.M. Owens, "Area-timepower tradeoffs in parallel adders", IEEE Transactions on Circuits and Systems, pp. 689–702, 1996.

[6] C. Nagendra, R.M. Owens, and M.J. Irwin, "Power-delay characteristics of CMOS adders", IEEE Transaction. on VLSI, pp. 377–381, 1994.

[7] C.S. Wallace, "A Suggestion for a Fast Multiplier," Transactions of IEEE Electronic Computers, vol. EC-13, pp. 14-17, 1964.

[8] E. Abu Shama, A. Elchoumi, S. Sayed, M. Bayoumi, "An Efficient Low Power Basic Cell for Adders," Proceeding of 38th Midwest Symposium on Circuit and Systems, Rio de Janeiro, Brazil, 1995.

[9] H. Lee, and M. Flynn, "Coarse Grained Carry Architecture for FPGA", Proceedings of the ACM/SIGDA International Symposium on FPGAs, Monterey, 2000.

[10] IEEE Standard for Binary Floating-point Arithmetic, ANSI/IEEE Standard No.754, 1988.

[11] J.M. Rabaey, "Digital Integrated Circuits", Pearson Education, pp. 559-596, 2005.

[12] J.P. Hayes, "Computer Architecture and Organization, Computer Organization", McGraw Hill, pp. 230-250, 1988.

[13] J. Rose *et al*, "Architecture of Field Programmable Gate Arrays," Proceedings of IEEE, vol 81, pp. 1013-1029,1993.

[14] K. Chapman, “Constant Coefficient Multipliers for the XC4000E”, Xilinx App. Note, XAPP 054, 1996.

[15] L. Dadda., “Some Schemes for Parallel Multipliers”, Alta Frequenza, vol. 34, pp. 349-356, 1965.

[16] Lucent Technologies Inc, “Create Multiply Accumulate Functions in ORCA FPGAs”, Article from Synario Design Automation, 1996.

[17] M. Morris Mano, “Digital Logic and Computer Design”, Prentice-Hall, pp. 434-446, 1979.

[18] O. Kwon *et al*, “A fast hybrid carry-lookahead/carry-select adder design,” Proceedings of 11th Great Lake Symposium on VLSI, pp. 149-152,2001.

[19] O. Salomon, J.M. Green, and H. Mar, “General Algorithms for a Simplified Addition of 2’s Complement Numbers”, IEEE *Journal* of *Solid-state Circuits,* vol. 30, No.7, pp. 839-844, July 1995.

[20] R.K. Montoye *et al.*, “Design of the IBM RISC System/6000 Floating-Point Execution Unit,” IBM J. Res. Develop, vol. 34, No. 1, pp. 59-70, 1990.

[21] S.D. Haynes *et al*, “Configurable Multiplier Blocks for use within a FPGA”,IEEE Transaction on Computers, vol . 3, No.1, pp. 638-639, 1998.

[22] S.D. Haynes *et al*, “Flexible Reconfigurable Multiplier Blocks Suitable for Enhancing the Architecture of FPGAs”, Proceedings of Custom Integrated Circuit Conference, San Diego, California, pp. 191-194, 1999.

[23] S. Oberman and M. Flynn, “Design Issues in Division and other Floating Point Operations”, IEEE Transactions on Computers, vol 46, pp. 154-161,1997.

[24] S. Oberman and M. Flynn, “Division algorithms and Implementation, ”IEEE Transactions on Computers, vol 46, pp. 833-854,1997.

[25] S. Shah *et al*, “Comparison of 32-bit multiplier for various performance measures,” Proceedings of the 12th International Conference on Microelectronics, Tehran, pp. 75-80, 2000.

[26] W. Kleitz, “Digital Electronics-A Practical Approach”,Pearson Prentice Hall, pp. 108-112, 2005.

[27] Xilinx Corporation Inc, “Programmable logic Databook”, 2000

[28] www.altera.com

[29] www.xess.com

[30] www.xilinx.com

HDL Synthesis Report

Macro Statistics

# ROMs	: 64
128x4-bit ROM	: 60
64x6-bit ROM	: 4
# Registers	: 18
1-bit register	: 16
32-bit register	: 1
16-bit register	: 1
# Multiplexers	: 3530
1-bit 16-to-1 multiplexer	: 6
2-to-1 multiplexer	: 3524
# Adders/Subtractors	: 150
3-bit adder	: 17
8-bit subtractor	: 100
24-bit subtractor	: 23
4-bit adder	: 9
8-bit adder	: 1
# Multipliers	: 1
23x23-bit multiplier	: 1
# Comparators	: 6
32-bit comparator greater	: 4
8-bit comparator greater	: 1
8-bit comparator less	: 1
# Xors	: 16
1-bit xor2	: 16

Glossary of Terms

The most important terminology used in this thesis work is defined below:

- ***Programmable Logic Device (PLD)-*** A general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs.
- ***PLA-*** A Programmable Logic Array (PLA) is a relatively small PLD that contains two levels of logic, an AND-plane and an OR-plane, where both levels are programmable.
- ***PAL -*** A Programmable Array Logic (PAL) is a relatively small PLD that has a programmable AND-plane followed by a fixed OR-plane.
- ***SPLD -*** Refers to any type of Simple PLD, usually either a PLA or PAL.
- ***CPLD-*** A more Complex PLD that consists of an arrangement of multiple SPLD-like blocks on a single chip.
- ***FPGA-*** A Field-Programmable Gate Array is a PLD featuring a general structure that allows very high logic capacity. Whereas CPLDs feature logic resources with a wide number of inputs (AND planes), FPGAs offer more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than CPLDs.
- ***HCPLDs-*** High-capacity PLDs: a single acronym that refers to both CPLDs and FPGAs.This term has been coined in trade literature for providing an easy way to refer to both types of devices.
- ***Interconnect-*** The wiring resources in a PLD.
- ***Programmable Switch-*** A user-programmable switch that can connect a logic element to an interconnect wire, or one interconnect wire to another.
- ***Logic Block-*** A relatively small circuit block that is replicated in an array in a PLD. When a circuit is implemented in a PLD, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block. The term logic block is mostly used in the context of FPGAs, but it could also refer to a block of circuitry in a CPLD.

- ***Logic Capacity-*** The amount of digital logic that can be mapped into a single PLD. This is usually measured in units of "equivalent number of gates in a traditional gate array". In other words, the capacity of an PLD is measured by the size of gate array that it is comparable to. In simpler terms, logic capacity can be thought of as "number of 2-input NAND gates".
- ***Logic Density-*** The amount of logic per unit area in a PLD.
- ***Speed-Performance-*** Measures the maximum operable speed of a circuit when implemented in an PLD. For combinational circuits, it is set by the longest delay through any path, and for sequential circuits it is the maximum clock frequency for which the circuit functions properly.
- ***EDIF(Electronic Design Interchange Format)***- A Standard representation format for describing electronic circuit, used to allow the interchange of circuit design information between EDA tools.
- ***Net List (Or Netlist)-*** A computer file (sometimes a printed listing) containing a list of the signals in an electronic design and all of the circuit elements (transistors, resistors, capacitors, ICs, etc.) connected to the signal in the design.
- ***RTL(Register Transfer Level or Register Transfer Logic)-*** A register level description of a digital electronic circuit. Register store intermediate information between clock cycles in a digital circuit, so an RTL description describes what intermediate information is stored, Where it is stored within the design, and how that information moves through the design as it operates.
- ***Simulation-*** Modeling of an electronic circuit (or any other physical system) using computer based algorithms and programming. Simulation can model designs at many levels of abstraction (system, gate, transistor, etc.). Simulation allows engineers to test designs without actually building them and thus can help in speeding the development of complex electronic system.
- ***Synthesis (also Logic Synthesis)*** – A computer process that transforms a circuit description from one level of abstraction to a lower level, usually towards some physical implementation. Synthesis is to hardware design what compilation is to software development.

- ***User Constraints File (UCF)*** – A user created ASCII file for storing timing constraints and location constraints for a design implementation. This is the file where the user has to give the constraints of pin locking according to the Trainer hardware. Pin locking is nothing but assigning the signals of the code to specific pins of Target Device.
- ***VHDL(VHSIC Hardware Description Language)-*** A hardware description language develop in the 1980s by IBM, Texas Instruments, and Intermetrics under US government contract for the Department of Defense's VHSIC (Very High Speed Integrated Circuit) programme.
- ***JTAG-*** Joint Test Action Group.

List of Publications

1. Anish Lal, “Logic Gates Through DNA Computing”, National Conference on Bioinformatics Computing (NCBC’05), TIET, Patiala, Punjab, March 18-19, 2005 (Accepted).
2. Anish Lal, “A Novel Design of High Speed Arithmetic Logic Unit using ROM**”,** National Conference on Emerging Principles and Practices of Computer Science and Information Technology (EPPCSIT’06), GNDE, Ludhiana, Punjab, August 18-19, 2006(Communicated).