FPGA Implementation of Double Precision Floating Point Square Root with BIST Capability

Thesis report submitted towards the partial fulfillment of requirements for the award of the degree of

Master of Technology
in
VLSI Design & CAD

Submitted by

Padala Nandeeswara Rao
Roll No. 60761012

Under the Guidance of

Mrs. Manu Bansal
Senior Lecturer, ECED

Department of Electronics and Communication Engineering
THAPAR UNIVERSITY
PATIALA-147004, INDIA
JULY-2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"FPGA Implementation of Double Precision Floating Point Square Root with BIST Capability"** in partial fulfillment of the requirement for the award of degree of M.Tech (VLSI Design & CAD) at Electronics and communication Department of Thapar University, Patiala, is an authentic record of my own carried out under the supervision of Mrs. Manu Bansal, Senior Lecturer, ECED.

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

Date: 13-07-2009

Padala Nandeeswara Rao
Roll. No. 60761012

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Manu Bansal

Mrs. Manu Bansal
Senior Lecturer
ECED, Thapar University

Counter signed by:

(Dr. A. K. Chatterjee)
Professor & Head
Electronics and communication
Engineering Department,
Thapar University, Patiala.

(Dr. R. K Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

ACKNOWLEDGEMENT

To discover, analyze and to present something new is to venturing into an untrodden path towards an unexplored destination which is an arduous adventure unless one gets a true torchbearer to show the way. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various people. Words are often too less to reveals one's deep regards. I take this opportunity to express my profound sense of gratitude and respect to all those who helped me through the duration of this thesis. I found in my revered guide **Mrs. Manu Bansal, Senior Lecturer, ECED,** Thapar University, Patiala, under whose guidance I had the privilege to complete this thesis. I wish to express my deep gratitude towards her for providing individual guidance and support throughout the thesis work.

I express my heartfelt gratitude towards **Mrs. Alpana Agarwal, Assistant Professor & PG Coordinator, ECED,** Thapar University and **Mr. B. K. Hemant, Project Faculty, (VLSI Design & CAD Lab),** Thapar University for their valuable guidance, encouragement, inspiration and the enthusiasm with which they solved my difficulties.

I convey my sincere thanks to **Dr. A. K. Chatterjee, Professor and Head of Electronics & Communication Department,** Thapar University, Patiala for his encouragement and cooperation.

I would also like to thank all staff members and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of my thesis.

My greatest thanks are to all who wished me success especially my parents Bangaru papa and Tatabbayi, for making this all possible. Above all I render my gratitude to the Almighty who bestowed self-confidence, ability and strength in me to complete this work for not letting me down in the time of crisis and showing me the silver lining in the dark clouds. I do not find enough words with which I can express my feelings of thanks to my dear friends for their help, inspiration and moral support which went a long way in successful completion of the present study.

Padala Nandeeswara Rao

ABSTRACT

The scope of this work includes study of VERILOG language for computing arithmetic and logical operations and functions suited for hardware implementation, and after that before implementing on FPGA it has capability to test them, i.e. having capability of Built in Self Test. The Double precision floating point square root unit is coded in VERILOG and validated through extensive simulation. This is structured so that they provide the required performance i.e. speed and gate count as well as latency. This VERILOG code is then synthesized by XST (Xilinx synthesis Technology) tool to generate the gate level net list that can be implemented on the FPGA Spartan 3E. The implementations of these designs show that their performances are comparable to, and sometimes higher than, the performances of non-iterative designs based on high radix numbers. The design achieves the Throughput (MFLOPS) 60.16, Latency 9.671ns, and consumes the total X-Power 81(mw). The pipelining of these iterative designs target high throughput computations encountered in some space application, which can be used in doing test calculations like for square root, sine, cosine, etc. which can be utilized in scientific calculations and the main application is the squarer root floating-point unit (FPU) contained in math coprocessor. BIST is beneficial in many ways. BIST can provide at speed, in system testing of the Circuit-Under Test (CUT). This is crucial to the quality component of testing. In addition, BIST overcome pin limitations due to packaging make efficient use of available extra chip area. All these benefits are plentiful motivations for BIST.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BIST	Built-In Self-Test
BILBO	Built-In-Logic Block
CLB	Combinational Logic Blocks
CUT	Circuit under Test
CAD	Computer Aided Design
CA	Cellular Automata
DFT	Design for Testability
FMA	Failure Mode Analysis
FSM	Finite State Machine
FP	Floating Point
FPGA	Field Programmable Gate Arrays
ISE	Integrated System Environment
IOB	Input Output Blocks
ICT	In Circuit Testing
KFLOPS	Kilo-Floating-Point operations Per Second
LFSR	Linear Feedback Shift Register
LBIST	Logical BIST
MFLOPS	Millions of Floating Point operations Per Second
MBIST	Memory BIST
NGC	Native Generic Circuit
NCD	Native Circuit Description
PAR	Place and Route
PCB	Printed Circuit Board
SQRT	Square Root
TPG	Test-Pattern-Generator
VHDL	VHSIC Hardware Description Language
XST	Xilinx Synthesis Technology

CHAPTER 1
INTRODUCTION

1.1 Need of double precision floating point square root unit

Square root operation is hard to implement on FPGAs because of the complexity of the algorithms. In space applications, many systems require advanced digital signal processing (DSP) algorithms to address their mission needs. In particular, an entire class of satellite subsystems, such as payload processing, data-handling, communications, guidance, navigation, and control, rely on applications of DSP techniques. As these systems evolve, the amount of data which needs to be processed increases significantly. For instance, on-board processing of Synthesis Aperture Radar (SAR) is considered as one of the most demanding applications of DSP techniques. In fact, SAR instruments can produce raw radar data at high continuous rates which can surpass by many DSP processors. Since previous generations of DSP processors were unable to process raw data at these rates, data had to be down linked to an earth station instead of being processed in orbit. Increasingly, newer space applications require on-board processing of data without any recourse to computational support from ground stations. For some time, work has been under way at various facilities of NASA and the Air Force Research Lab to enable space vehicles formation flying through the development and deployment of space borne differential Global Positioning System (GPS) at TFLOP rates [1].

It is expected that this technology will result in swarms of spacecrafts flying as a virtual platform to gather significantly more and better scientific data in a totally autonomous fashion. Such autonomy requires minimal or no support from ground stations. To support these new applications, high performance, low power, computing devices which can produce high throughput floating point computations are needed. To meet these numerical requirements, many of these applications rely on the IEEE 754-1985 binary floating point standard. In fact, many DSP computations are centred on calculations involving floating point numbers. These numbers are used in double precision bit widths in order to accommodate the range and precisions required by these DSP algorithms. While most DSP applications can be implemented on DSP processors, there are computing scenarios in which a multitude of DSP processors, initially thought to be necessary to

provide the required throughput, fails to deliver such throughput. In fact, duplication of processors does not necessarily lead to a speedup in computation as is known in the parallel processing community. Only recently, FPGAs have begun to capture the attention of the DSP community as an alternative implementation technology capable of delivering significant speedups for compute-intensive DSP algorithms. Despite dire predictions at each step in technology evolution, FPGA densities continue to double approximately every 18 months as predicted by Moore's Law. With today's deep submicron technology, it is now possible to deliver multi-million gate FPGAs, which can implement complete systems integrated entirely on a single device. Although FPGAs are still lagging behind many ASICs in terms of raw performance, they have nevertheless crossed a gate density threshold that is usually seen in DSP processors.

Starting with earlier FPGA chips, many designers have realized that it is beneficial to implement double precision floating point computations on these chips due to their mapping versatility and reconfigurability. Since then, various mapping efforts of floating point computations on FPGAs have realized performances that are steadily surpassing the performance of those computations on commodity CPUs. What is attractive about FPGAs is their diverse catalo of embedded architectural features specifically optimized for arithmetic operations. For example, Xilinx FPGAs embed carry-chains along their CLB columns designed to speed up addition with narrow operands. These features can be exploited to support efficient floating point operations. However, exploiting these features require arithmetic algorithms, such as low-radix digit recurrence algorithms, that can be easily mapped on these structures.

In fact, careful pipelining and mapping of these algorithms on specific FPGA architectures can yield easily testable implementations which can produce throughputs that are comparable to the throughputs seen in high-radix algorithms. Although their area overhead is marginally higher than high radix implementations in many cases, low-radix digit recurring algorithms can easily reach the 100 MFLOPS mark on some chips. In double precision floating point units are all IEEE 754 compliant based on the sequential designs presented in [1]. The pipelining of this unit is based on unrolling the iterations of the digit recurrence computation and optimizing the operation within unrolled iteration. These units are implemented to take advantage of the fine-grain resources, such as LUTs

and carry chains, available in most FPGAs. Such common-denominator implementations can be ported to any FPGA architecture since most possess these fine grain resources. To this end, care has been taken not to use any advanced architectural features, such as Block RAMs available in recent FPGA chips. Furthermore, no specially designed cores have been integrated into our designs.

1.2Objectives

In general, the programming objectives of the floating-point applications fall into the following categories.

- Accuracy: The application produces that results that are close to the correct result.
- Performance: The application produces the most efficient code possible.
- Reproducibility and portability: The application produces results that are consistent across different runs, different set of build options, different compilers, different platforms, and different architecture.
- Latency: The application produces a single output with in less time.
- Throughput: The application produces more number of tasks that can be completed per unit time.
- Area: This application produces the less number of flip-flops and slices.

1.3 Organization of This Thesis

The rest of the thesis is organized as follows:

Chapter 2. Give the general background of the thesis, which includes the basic architectures and functionality of the double precision floating point square root.

Chapter 3. Describes the different modules of the block diagram, which is divided in order to show the basic functionality of the hardware. These modules are coded using simulation software ISE (Integrated Software Environment) and ModelSim.

Chapter 4. Gives the basic description of the design methodology and design flow of FPGA, how the design is implemented on FPGA (SPARTAN 3E).After the proper simulation, the design is synthesized and translated to structural architecture and then

perform post translate simulations in order to ensure proper functioning of the desire. The design is then mapped to existing slices of FPGA and the post mapped module is simulated. The post-map doesn't include routing delays. After post mapped simulation the design is routed and a post-route simulation model with appropriate routing delays is generated to be simulated on the HDL simulator. Introduce the Basic Concepts of Logic BIST, design flow of BIST. Describe Test Pattern Generation and Output Response Analysis Techniques, Various BIST Timing Control Diagrams.

Chapter 5. Give the Results and Conclusion of the thesis work, in terms of various results obtained during its implementation. It also shares the future prospective of Double Precision Floating Point Square root.

1.4 Tools used

The tools used in the thesis are as follows:

Simulation Software:

- ISE 9.2i (Integrated system environment) has been used for synthesis and implementation.
- ModelSim 6.1e has been used for modelling and simulation.

Hardware used:

Xilinx Spartan 3E (Family), XC3S5000 (Device), FG320 (Package) FPGA device.

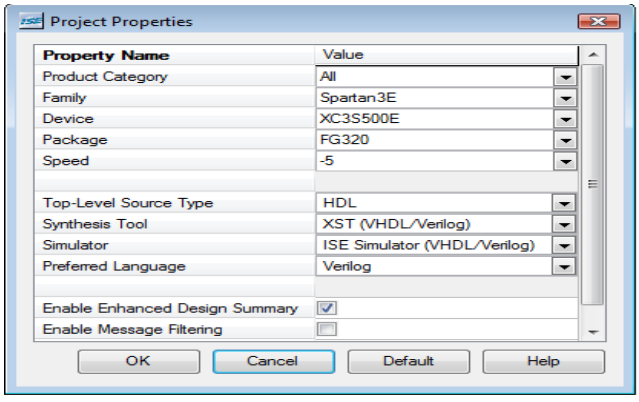


Figure 1.1: Xilinx Common Configuring and Programming Setups.

Tool used HDL (Top Level source Type), XST-VHDL/VERILOG (Synthesis Tool), ISE Simulator -VHDL/VERILOG (simulator), and VERILOG (Preferred Language).

CHAPTER 2
LITERATURE SURVEY

2.1 Square-rooting methods

The function $\sqrt{z}$ is the most important elementary function. Since square rooting is widely used in many applications, the IEEE floating point standard specifies square-rooting as a basic arithmetic operation is shown below [3].

2.1.1 The pencil-and-paper algorithm

Unlike multiplication and division, for which the pencil-and-paper algorithms are widely taught and used, square-rooting by hand, appears to have fallen prey to the five-dollar calculator. Since shift/subtract methods for computing $\sqrt{z}$, are derived directly from the ancient manual algorithm, we begin by describing the pencil-and-paper algorithm for square-rooting. Notation for our discussion of square root algorithms:

z	Radicand	$z_{2k-1}z_{2k-2}\cdots z_3z_2z_1z_0$
q	Square root	$q_{k-1}q_{k-2}\cdots q_1q_0$
s	Remainder, $z - q^2$	$s_k s_{k-1}s_{k-2}\cdots s_1s_0$

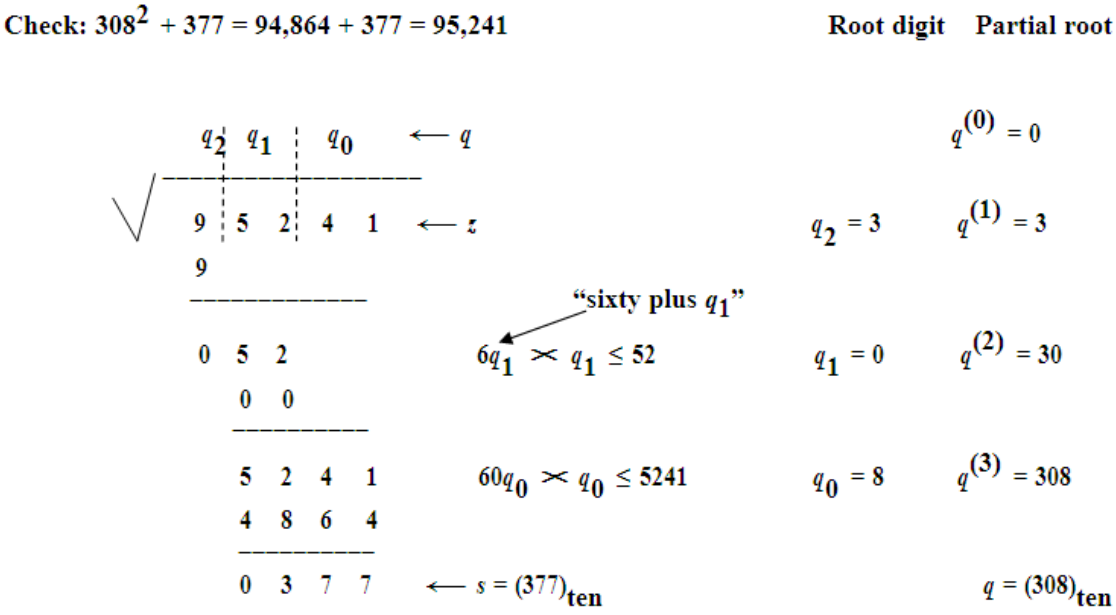


Figure 2.1: Pencil-and-paper algorithm to extract the square root of a decimal integer.

Remainder range, $0 \le s \le 2q$ (k + 1 digits)

Justification: $s \geq 2q + 1$ would lead to $z = q^2 + s \geq (q + 1)^2$

From the example in Fig.2.2, we can abstract the dot notation representation of binary square-rooting. The radicand z and the root q are shown at the top. Each of the following four rows of dots corresponds to the product of the product of the next root digit q_{k-i-1} and a number obtained by appending $0q_{k-i-1}$ to the right end of the partial root $q^{(i)}$.Thus, since the root digit z are in {0,1}, the problem of binary square-rooting reduces to subtracting a set of numbers, each being 1 or a shifted version of $(q^{(i)}\,01)_{two}$, from the radicand z.

2.1.2 Example of binary square-rooting

The figure 2.2 shows represent the binary square root using pencil-and-paper algorithm.

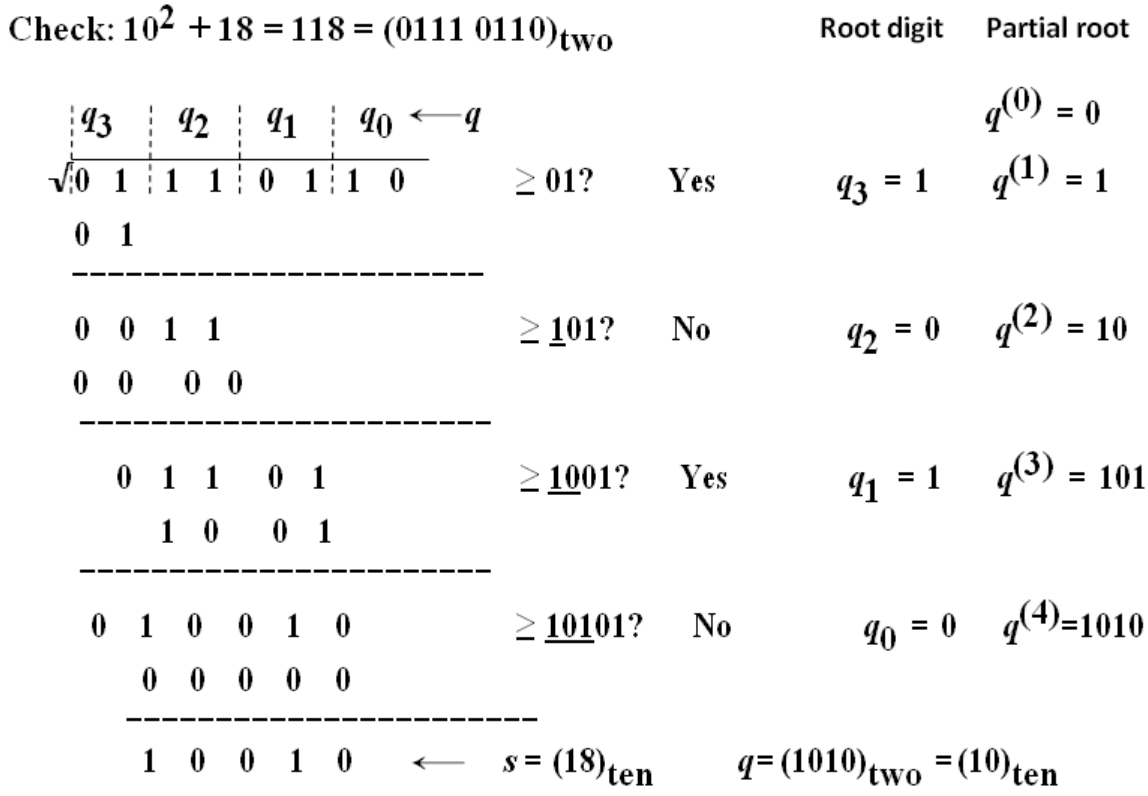


Figure 2.2: Extracting the square root of a decimal integer using the pencil-and - paper algorithm.

2.1.3 Restoring shift/SUBTRACT algorithm

											Root digit	Parital root
z (radicand = 118/64)			0	1 .	1	1	0	1	1	0(118/64)		
$S^{(0)} = z - 1$	0	0	0 .	1	1	0	1	1	0		$q_0 = 1$	$q^{(0)} = 1.$
$2s^{(0)}$	0	0	1 .	1	0	1	1	0	0			
$-[2 \times (1.) + 2^{-1}]$			1	0 .	1							
$s^{(1)}$	1	1	1 .	0	0	1	1	0	0		$q_{-1} = 0$	$q^{(1)} = 1.0$
$s^{(1)} = 2s^{(0)}$ Restore	0	0	1 .	1	0	1	1	0	0			
$2s^{(1)}$	0	1	1 .	0	1	1	0	0	0			
$-[2 \times (1.0) + 2^{-2}]$			1	0 .	0	1						
$S^{(2)}$	0	0	1 .	0	0	1	0	0	0		$q_{-2} = 1$	$q^{(2)} = 1.01$
$2s^{(2)}$	0	1	0 .	0	1	0	0	0	0			
$-[2 \times (1.01) + 2^{-3}]$			1	0 .	1	0	1					
$s^{(3)}$	1	1	1 .	1	0	1	0	0	0		$q_{-3} = 0$	$q^{(3)} = 1.01$
$s^{(3)} = 2s^{(2)}$ Restore		0	1	0 .	0	1	0	0	0	0		
$2s(3)$	1	0	0 .	1	0	0	0	0	0			
$-[2 \times (1.010) + 2^{-4}]$			1	0 .	1	0	0	1				
$s^{(4)}$	0	0	1 .	1	1	1	1	0	0		$q_{-4} = 1$	$q^{(4)} = 1.0101$
$2s^{(4)}$	0	1	1 .	1	1	1	0	0	0			
$-[2 \times (1.0101) + 2^{-5}]$			1	0 .	1	0	1	0	1			
$s^{(5)}$	0	0	1 .	0	0	1	1	1	0		$q_{-5} = 1$	$q^{(5)} = 1.01011$
$2s^{(5)}$	0	1	0 .	0	1	1	1	0	0			
$-[2 \times (1.01011) + 2^{-6}]$			1	0 .	1	0	1	1	0	1		
$s^{(6)}$	1	1	1 .	1	0	1	1	1	1		$q_{-6} = 0$	$q^{(6)} = 1.010110$
$s^{(6)} = 2s^{(5)}$ Restore	0	1	0 .	0	1	1	1	0	0			(156/64)
s (remainder = 156/64)			0 .	0	0	0	0	1	0	0 1 1 1 0		0(156/64²)
q (root = 86/64)			1.	0	1	0	1	1	0			(86/64)

Figure2.3: Example of sequential binary square-rooting by means of the restoring Algorithm.

We formulate our shift/subtract algorithms for a radicand in the range $1 \leq z < 4$ corresponding to the significant of a follows:

z Radicand	$z_1 z_0 \cdot z_{-1} z_{-2} \cdots z_{-l}$	$1 \leq z < 4$
q Square-root	$q_{-1} q_{-2} \cdots q_{-l}$	$1 \leq q < 2$
s Scaled remainder	$s_1 s_0 \cdot s_{-1} s_{-2} \cdots s_{-l}$	$0 \leq s < 4$

An example of binary restoring square-rooting using the preceding recurrence is shown in Figure 2.3, where we have provided three whole digits, plus the required six fractional digits, for representing the partial remainders two whole digits are required given that the partial remainders, as well as the radicand z, are in [0,4) .The third whole digit is needed to accommodate the extra bit that results from shifting the partial remainders $s^{(j-1)}$ to the left to form $2^{(j-1)}$.This bit also acts as the sign bit for the trial difference.

2.1.4 Restoring square-rooting

In fractional square-rooting, the remainder is usually of no interest. To properly round the square root, we can produce an extra digit ($q_{-l-1}=0$) or to round up ($q_{-l-1}=1$). The midway case, (i.e., $q_{-l-1}=1$with only 0s to its right), is impossible (why?), so we don't even have to test the remainder for 0.

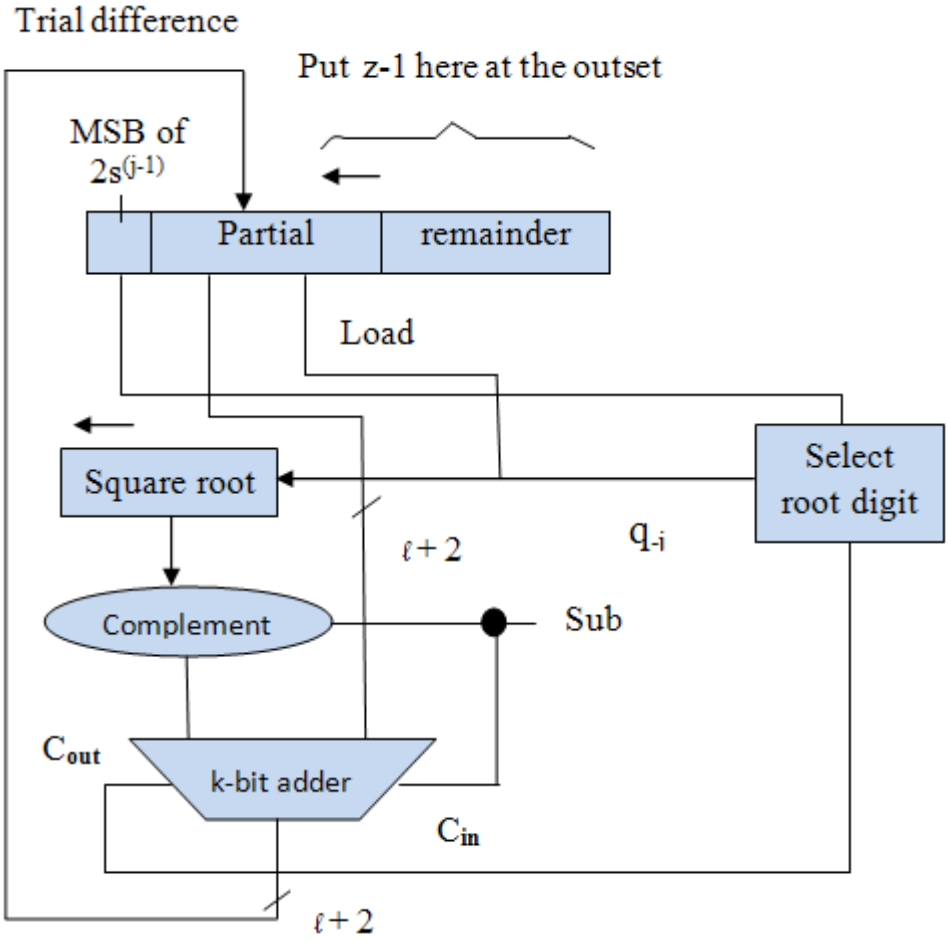
The preceding algorithm, which is similar to restoring division, is quite naturally called restoring square-rooting. An example of binary restoring square-rooting using the preceding recurrence is shown in Figure 2.3, where we have provided three whole digits, plus the required six fractional digits, for representing the partial remainders. Two whole digits are required given that the partial remainders, as well as the radicand z, are in [0,4).the third whole digit is need to accommodate the extra bit that results from shifting the partial remainder $s^{(i-1)}$ to the left to form $2s^{(i-1)}$.This bit also acts as the sign bit for the trial difference.

For the Example of Figure 2.3, an extra iteration produces q-7 = 1. So the root must be rounded up to $q = (1.010111)_2 = 87/64$. To check that the rounded-up value is closer to the actual root than the truncated version, we note that:

$$118/64 = (87/64)2 - 17/642$$

Thus, the rounded-up value yields a remainder with a smaller magnitude.

The figure 2.4 shows the hardware implementation of restoring square-



rooter.

Figure 2.4: Sequential shift/subtract restoring square-rooter.

2.1.5 Single precision floating point square root

Y. Li and W. Chu[4] presented a non-restoring square root algorithm and two very simple single precision floating point square root implementations based on the algorithm on FPGAs. One is low-cost iterative implementation that uses a traditional adder/subtractor. The operation latency is 25 clock cycles and the issue rate is 24 clock cycles. The other is high-throughput pipelined implementation that uses multiple adder/subtractions. The operation latency is 15 clock cycles and the issue rate is one clock cycle. It means that the pipelined implementation is capable of accepting a square root instruction on every clock cycle. Square root is basic operations in computer graphics and scientific calculation applications. The operations on integer numbers and floating point

numbers have been implemented with standard VLSI circuits. Some of them have been implemented on FPGAs based custom computing machines.

Shirazi *et al* [5] presented FPGA implementations of 18-bit floating point adder/subtractor, multiplier, and divider. Louca *et al* [6] presented FPGA implementations of single precision floating point adder/subtractor and multiplier. Because of the complexity of the square root algorithms, the square root operation is hard to implement on FPGAs for custom computing machines. Two single precision square root implementations on FPGAs, based on a non-restoring algorithm, were presented. The iterative implementation requires few areas, and the pipelined implementation achieves high performance at a reasonable cost. Both the implementations are very simple, which can be readily appreciated. The modern multi-issued processors require multiple dedicated, fully-pipelined functional units to exploit instruction level parallelism; hence the simplicity of the functional units becomes an important issue. The proposed implementations are shown to be suitable for designing a fully pipelined dedicated floating point unit on FPGAs.

2.2 IEEE-754-1985 standard for Binary Floating-Point Arithmetic

It is intended that an implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. [7].

2.2.1 Floating Point Numbers

It was quite common that machine from different vendors have different word length and unique floating point formats. This caused many problems; especially in the porting of programs between differences machines (design). So there was a need for standard. Finally the IEEE has developed a standard for the representation of floating point numbers. The main objective is to make numerical program predictable and completely portable in the sense of producing identical results when run on different machines.

The IEEE – 754 floating point standard formerly names ANSI/IEEE standard-754 was introduced to solve these problems. Another main objective for this standard is that an implementation of a floating-point system confirming to this standard can be realized in software, entirely in hardware in or any combination of software in hardware. The standard specifies two formats for floating point numbers basic (single precision), and extended (double precision), it also specifies basic operations for both formats which are addition, subtraction, multiplication, divisions, Logical operations, and comparison of operations. There was also a need for different conversion, such as integer to floating point, basic to extended and vice versa.

The format also describes the different floating-point exceptions and their handling, including not number (NaN). The standard has adopted the hidden approach (implicit approach) to save one bit in the representation this increases the precision and takes less space. This can be done by always representing the floating-point number in normalized form, starting with 1 in the MSB of the mantissa, so finally this one is omitted when we store this number. Denormals or renormalized values are defined as numbers without a hidden 1 and with the smallest possible exponent. These numbers were provided to decrease the effect in assignment of underflow, lead to high speed and low cost.

In this these only double precision format of the standard is discussed. In this format 1 bit is assigned for sign bit, 11 bits are assigned exponent and 52 bits for significant with 1 implicit bit.

Floating point representation is based on exponential (or scientific) notation. In exponential notation, a nonzero real number x is expressed in decimal as

$$x = \pm S \times 10^{E}, \qquad \text{Where } 1 \leq S < 1 \qquad (2.1)$$

and *E* is an integer. The numbers S and *E* are called the *significant* and the *exponent,* respectively. For example, the exponential representation of 365.25 is 3.6525 x 10^{2}, and the exponential representation of 0.00036525 is 3.6525 x 10^{-4}. It is always possible to satisfy the requirement that $1 < S < 10$, as *S* can be obtained from *x* by repeatedly multiplying or dividing by 10, decrementing or incrementing the exponent *E* accordingly. We can imagine that the *decimal point floats* to the position immediately after the first nonzero digit in the decimal expansion of the number—hence the name floating point. For representation on the computer, we prefer base 2 to base 10, so we write a nonzero number

x in the form[9]

$$x = \pm S \times 2^{E}, \qquad \text{Where } 1 \leq S < 2, \tag{2.2}$$

Consequently, the binary expansion of the significand is

$$S = (b_0 \ .b_1 \ b_2 \ b_3 \ \ldots)_2 \qquad \text{With } b_0 = 1 \tag{2.3}$$

For example, the number 11/2 is expressed as

$$\frac{11}{2} = (1.011)^{2} \times 2^{2} \tag{2.4}$$

Now it is the *binary point* that *floats* to the position after the first nonzero bit in the binary expansion of x, changing the exponent *E* accordingly. Of course, this is not possible if the number *x* is zero, but at present we are considering only the nonzero case. Since b_0 is 1, we may write

$$S = (b_0 \ .b_1 \ b_2 \ b_3 \ \ldots)_2 \tag{2.5}$$

The bits following the binary point are called the *fractional* part of the significand.

2.2.2 Floating Point Representation

All of the numeric formats are fixed-point formats, in practice they only represent integers, not fractions, however a general purpose computer must be able to work with fractions with varying number of bits or it will not be very useful. So it is desirable to allow a computer to use fractions for certain applications. This is the purpose of Floating-Point Numbers, Floating point format is very similar to scientific notation, a number expressed in scientific notation has a sign, a fraction Significant(also called Mantissa), and an Exponent. For example the number -1234.5678 could be expressed as -1.2345678×10^{3}, where the sign is negative the Significant is 1.2345678, and the exponent is 3. This example uses the base 10 but any base can be used to represent floating point numbers. Computers generally use base two to represent floating point numbers.

One disadvantage of scientific notation is that most numbers can be expressed in many different ways. For example $-1234.5678 = -1.2345678 \times 10^{3} = -1234567.8 \times 10^{-3}$, as well as several other representations. Computers are more efficient and have much simple hardware, if each number must be normalized i.e. each number's significant is fraction with no leading Zeros. For example, 1.0×10^{-9} and 10.0×10^{-9} are not correctly. Another value that requires a special representation is Not a Number or NaN. This value represents

the result of illegal operation, such as $\infty \div \infty$ or taking the square root of a negative number. Also

NaN can be assigned to uninitialized variables.

Fixed-Point numbers have a range that is limited by the significant digits used to represent the number. In scientific applications it is often necessary to deal with numbers that are very large or very small. Instead of using the fixed-point representation, which would require many significant digits, it is better to use the floating-point representation in which numbers are represented by a mantissa comprising the significant digits and an exponent of the radix R. the format is *Mantissa* × *R Exponent*.

The numbers are often normalized, such that the radix point is placed to the right of the first nonzero digit, as 6.31×10^{-28}. Binary floating point representation has been standardized by the Institute of Electrical and Electronic Engineers (IEEE) [8].

Computers represent real values in a form similar to that of scientific notation. Consider the value 1.23 x 10^4.The number has a sign (+ in this case). The significand (1.23) is written with one non-zero digit to the left of the decimal point. The base (radix) is 10 and exponent (an integer value) is 4. It too must have a sign. There are standards which define what the representation means, so that across computers there will be consistency. Note that this is not the only way to represent floating point numbers, it is just the IEEE standard way of doing it.

The representation has three fields:

S	E	F

- S is one bit representing the sign of the number
- E is an 8-bit biased integer representing the exponent
- F is an unsigned integer

The decimal value represented is:

$(-1)^S \times F^e \times 2$ Where e = E – bias and f = (F/ (2^n)) + 1 (2.6)

- For single precision representation (the emphasis in this class) n = 23, bias = 127.
- For double precision representation (a 64-bit representation) n = 52 (there are 52

bits for the mantissa field) bias = 1023 (there are 11 bits for the exponent field)

Now, what does all this mean?

- S, E, F all represents fields within a representation. Each is just a bunch of bits.
- S is just a sign bit. 0 for positive, 1 for negative. This is the sign of the number.
- E is an exponent field. The E field is a biased-127 integer representation.

So, the true exponent represented is (E - bias), the radix for the number is ALWAYS 2.

Note: Computers that did not use this representation,

Like those built before the standard, did not always use a radix of 2. For example, some IBM machines had radix of 16. F is the mantissa (significand). It is in a somewhat modified form. There are 23 bits available for the mantissa. It turns out that if floating point numbers are always stored in their normalized form, then the leading bit (the one on the left, or MSB) is always a 1. So, why store it at all? It gets put back into the number (giving 24 bits of precision for the mantissa) for any calculation, but we only have to store 23 bits. This MSB is called the hidden bit.

Format	$\mathbf{E_{min}}$	$\mathbf{E_{max}}$	$\mathbf{N_{min}}$	$\mathbf{N_{max}}$
Single	-126	127	$2^{-126} \approx 1.2 \times 10^{-38}$	$\approx 2^{128} \approx 3.4 \times 10^{38}$
Double	-1022	1023	$2^{-1022} \approx 1.2 \times 10^{-308}$	$\approx 2^{1024} \approx 1.2 \times 10^{308}$

Table 2.1: Range of IEEE Floating Point Format.

M. L. Overton [9] say that equations (2.2) and (2.3) is the *normalized* representation of x, and the process of obtaining it is called *normalization.* To store normalized numbers, we divide the computer word into three fields to represent the sign, the exponent *E*, and the significand S, respectively. A 32-bit word could be divided into fields as follows: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand. The sign bit is 0 for positive numbers and 1 for negative numbers. Since the exponent field is 8 bits, it can be used to represent exponents *E* between -128 and 127 (for example, using 2's complement, though this is not the way it is normally done). The 23 significand bits can be used to store

the first 23 bits after the binary point in the binary expansion of S, namely, $b_1 \ldots b_{23}$. It is not necessary to store b_0 since we know it has the value 1. M. L. Overton [9] says that b_0 is a *hidden bit.* Of course, it might not be possible to store the number *x* with such a scheme, either because *E* is outside the permissible range -128 to 127 or because the bits b_{24}, $b_{25,}$ in the binary expansion of *S* are not all zero. A real number is called a *floating point number* if it can be stored *exactly* on the computer using the given floating point representation scheme.

Using this idea, the number 11/2 would be stored as

0	ebits(2)	01100000000000000000000

and the number

$$71 = (1.000111)_2 \times 2^6$$

would be stored as

0	ebits(10)	00000000000000000000000

Now consider the much larger number

$$2^{71} = (1.000\ldots)_2 \times 2^{71}$$

This integer is much too large to store in a 32-bit word using the integer format. However, there is no difficulty representing it in floating point, using the representation

0	ebits(71)	00000000000000000000000

- Single precision representation is 32 bits.
- Double precision representation is 64 bits.

2.3 Definitions [9]

1. Biased exponent

The sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative.

2. Binary floating-point number

A bit-string characterized by three components: a sign, a signed exponent, and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent. In this standard a bit-string is not always distinguished from a number it may represent.

3. Exponent

The component of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

4. Fraction

The field of the significand that lies to the right of its implied binary point.

5. Denormalized Number

A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

6. Destination

The location for the result of a binary unary operation. A destination may be either explicitly designated by the user or implicitly supplied by the system (that is, intermediate results in sub expressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's precision as well as the operand's values.

7. Mode

A variable that a user may set, sense, save, and restore to control the execution of subsequent arithmetic operations. The default mode is the mode that a program can assume to be in effect unless an explicitly contrary statement is included in either the program or its specification. The following mode shall be implemented.

1. Rounding to control the direction of rounding errors.
2. In certain implementations, rounding precision, to shorten the precision of results.
3. The implementer may, at his option, implement the following modes: traps disabled or enabled, to handle exceptions.

8. Normal Number

A nonzero number that is finite and not subnormal.

9. Radix

The base for the representation of floating-point numbers. Radix produce more than one bit, in radix 4 represents 2 bits, in radix 8 represents 3 bits etc.

10. Significand

The component of a floating-point number that consist of a leading digit to the left of its implied radix point and a fraction field to the right.

11. Result

The digit string (usually representing a number) that is delivered to the destination.

12. Status flag

A variable that may be take two states, set and clear. A user may clear a flag, copy it, or restore it to a previous state. When set, a status flag may contain additional system-dependent information, possibly inaccessible to some users. The operations of this standard may as a side effect set some of the following flags:

Inexact result, underflow, overflow, and divide by zero, and invalid operation.

13. Subnormal number

A nonzero floating-point number whose exponent is the precision's minimum and who's leading significant digit is zero.

14. User

Any person, hardware, or program not itself specified by this standard, having access

to and controlling those operations of the programming environment specified in this standard.

15. Precisions

This standard defined four floating - point precisions in two groups, basic and extended, each having two widths, single and double. The standard levels of implementation are distinguished by the combinations of precisions supported [10, 7].

16. Biased exponent

The sum of the exponent and a constant (bias) chosen to make the biased exponent's range.

17. Combinations of Formats

All implementations conforming to this standard shall support the single format Implementations should support the extended format corresponding to the widest basic format supported, and need not support any other extended format.

18. Extended formats

The standard also strongly recommends support for an extended format, with at least 15 bits available for the exponent and at least 63 bits for the fractional part of the significand. The Intel microprocessors implement arithmetic with the extended format in hardware, using 80-bit registers, with 1 bit for the sign, 15 bits for the exponent, and 64 bits for the significand. The leading bit of a normalized or subnormal number is not hidden as it is in the single and double formats, but is explicitly stored. Otherwise, the format is much the same as single and double. Other machines, such as the Spare microprocessor used in Sun workstations, implement extended precision arithmetic in software using 128 bits.

2.4 Single –precision Floating-Point Format

A 32-bit single format number X is divided as shown in Figure 2.5. The value V of X is inferred from its constituent fields thus

- If E=255 and F is nonzero, then V=NaN ("Not a number")
- If E=255 and F is zero and S is 1, then V=-Infinity
- If E=255 and F is zero and S is 0, then V=Infinity
- If 0<E<255 then $V=(-1)^S \times 2^{(E-127)} \times (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then $V= (-1)^S \times 2^{(-126)} \times (0.F)$ These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

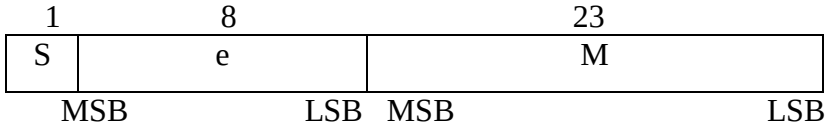


Figure 2.5: Single - Precision Floating - Point form.

$$\text{Value} = N = (-1)^S \times 2^{E-127} \times (1.M)$$

Msb: Most significant bit.

Lsb: Least significant bit.

Actual exponent is: $0 < E < 255$ and $e = E - 127$

Exponent: Excess 127 binary integers added.

±	$a_1\ a_2\ a_3 \ldots a_8$ is	$b_1\ b_2\ b_3 \ldots\ b_{23}$

If exponent bitstring $a_1 \ldots a_8$ is	**Then numerical value represented is**
$(00000000)2 = (0)_{10}$	$\pm (0.b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{-126}$

$(00000001)2 = (1)_{10}$ $(00000010)2 = (2)_{10}$ $(00000011)2 = (3)_{10}$ ↓ $(01111111)2 = (127)_{10}$ $(10000000)2 = (128)_{10}$ ↓ $(11111100)2 = (252)_{10}$ $(11111101)2 = (253)_{10}$ $(11111110)2 = (254)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{-126}$ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{-125}$ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{-124}$ ↓ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})2 \times 2^{0}$ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{1}$ ↓ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{125}$ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{126}$ $\pm (1.\ b_1\ b_2\ b_3 \ldots b_{23})_2 \times 2^{127}$
$(11111111)2 = (255)_{10}$	$\pm \infty$ if $b1 = \ldots = b_{23} = 0$,NAN Otherwise

Table 2.2: IEEE Single Precision Floating Point Format.

Mantissa: Sign + Magnitude, normalized binary significand with a hidden integer bit (1.M).

The above Figure 2.5 depicts the single precision format. The leftmost bit is the sign bit 0 for positive e and 1 for negative numbers. There is an 8-bit exponent field (E), and a 23-bit Mantissa field (M), the Exponent is write the radix 2.Because it is necessary to be able to represent both very large and very small numbers, the exponent can be either positive or negative. Instead of simply an 8-bit signed number as the Exponent, which would allow exponent values in the range -128 to -127, the IEEE standard specifies the exponent in excess -127 format. In this format the value 127 is added to the value of the actual Exponent so that

$$\text{Exponent} = E - 127$$

In this way E becomes a positive integer. This format is convenient for adding and subtracting floating point numbers because the first step in these operations involve comparing the exponent to determine whether the Mantissa must be appropriately shifted to add / subtract the significant bits. The range of E is 0 to 255. The extreme values of E =0 & E = 255 are taken to denote the exact Zero and infinity, respectively. Therefore the normal range of the Exponent is -126 to 127, which is represented by the values of E from 1 to 254.

The Mantissa is represented by using 23 bits. The IEEE standard calls for a normalized Mantissa, which means the most significant bit is always equal to 1. Thus it is

not necessary to include this bit explicitly in the Mantissa field. Therefore, if M is the bit vector in the Mantissa field, the actual value of the Mantissa is 1. M which gives a 24 bit Mantissa. Consequently the Floating-point format in above figure represents the number.

$$\text{Value} = \pm 1.M \times 2^{E-127}$$

The size of the mantissa field allows the representation of the numbers that have precision of about seven decimal digits. The Exponent field range of 2^{-126} to 2^{127} corresponds to about 10± 38 [11].

An example: Put the decimal number 64.2 into the IEEE standard single precision floating point representation.

First step:

Get a binary representation for 64.2 to do this, get unsigned binary representations for the stuff to the left and right of the decimal point separately.

64 is 1000000

.2 can be gotten using the algorithm:

.2 x 2 = 0.4 0

.4 x 2 = 0.8 0

.8 x 2 = 1.6 1

.6 x 2 = 1.2 1

.2 x 2 = 0.4 0 now this whole pattern (0011) repeats.

.4 x 2 = 0.8 0

.8 x 2 = 1.6 1

.6 x 2 = 1.2 1

So a binary representation for .2 is .001100110011. . . or .0011 (The bar over the top shows which bits repeat).

Putting the halves back together again: 64.2 are $1000000.00110011\overline{0011}...$

Second step:

Normalize the binary representation.(make it look like scientific notation).

$$1.000000\ 00110011. . . \text{ X } 2^6$$

Third step:

6 is the true exponent. For the standard form, it needs to be in 8-bit, biased-127 representation.

6
+ 127

133

133 in 8-bit, unsigned representation is 1000 0101.This is the bit pattern used for E in the standard form.

Fourth step:

The mantissa stored (F) is the stuff to the right of the radix point in the normalized form. We need 23 bits of it.

000000 00110011001100110

Put it all together (and include the correct sign bit):

S E F

0 10000101 00000000110011001100110

The values are often given in hex, so here it is

0100 0010 1000 0000 0110 0110 0110 0110

0x 4 2 8 0 6 6 6 6

2.5 Double –precision Floating-Point Format

A 64-bit double format number X us divided as shown in Figure 2.6. The value v of X is inferred from its constituent fields thus

- If E=2047 and F is nonzero, then V=NaN ("Not a number")
- If E=2047 and F is zero and S is 1, then V=-Infinity
- If E=2047 and F is zero and S is 0, then V=Infinity
- If $0 < E < 2047$ then $V=(-1)^{S} \times 2^{(E-1023)} \times (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If E=0 and F is nonzero, then $V=(-1)^{S} \times 2^{(-1022)} \times (0.F)$ These are "unnormalized" values.
- If E=0 and F is zero and S is 1, then V=-0
- If E=0 and F is zero and S is 0, then V=0

- S is the sign bit (same as for single precision)
- E is an 11-bit, biased-1023 integer for the exponent
- F is a 52-bit mantissa, using same method as single precision (hidden bit is not explicit in the representation)

Figure 2.6: Double-Precision Floating-Point format.

$$\text{Value} = N = (-1)^{s} \times 2^{E-1023} \times (1.M)$$

MSB: Most significant bit.

LSB: Least significant bit.

Actual exponent is: $0 < E < 2047$ and $e = E - 1023$.

Exponent: Excess 1023 binary integers added.

Mantissa: Sign + Magnitude, normalized binary significand with a hidden integer bit (1.M).

±	$a_1\ a_2\ a_3 \ldots a_{11}$ is	$b_1\ b_2\ b_3 \ldots b_{52}$

If exponent bitstring $a_1 \ldots a_{11}$ is	**Then numerical value represented is**
$(00000000000)2 = (0)_{10}$	$\pm (0.b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{-1022}$
$(00000000001)2 = (1)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{-1022}$
$(00000000010)2 = (2)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{-1021}$
$(00000000011)2 = (3)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{-124}$
↓	↓
$(01111111111)2 = (1023)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})2 \times 2^{0}$
$(10000000000)2 = (1024)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{1}$
↓	↓
$(11111111100)2 = (2044)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{1021}$
$(11111111101)2 = (2045)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{1022}$
$(11111111110)2 = (2046)_{10}$	$\pm (1.\ b_1\ b_2\ b_3 \ldots b_{52})_2 \times 2^{1023}$
$(11111111111)2 = (2047)_{10}$	$\pm \infty$ if $b1 = \ldots = b_{52} = 0$,NAN Otherwise

Table 2.3: IEEE Double Precision Floating Point Format.

The single format is not adequate for many applications, either because higher precision is desired or (less often) because a greater exponent range is needed. The IEEE standard specifies a second basic format, *double*, which uses a 64-bit double word. Details are shown in Table 2.3. The ideas are the same as before, only the field widths and exponent bias are different. Now the exponents range from E_{min} = -1022 to E_{max} = 1023, and the number of bits in the fraction field is 52. Numbers with no finite binary expansion, such as 1/10 or π, are represented more accurately with the double format than they are with the single format. The smallest positive normalized double format number is $N_{min} = 2^{-1022} \approx 2.2 \times 10^{-308}$ and $N_{max} = (2- 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{3}$.

2.6 Single versus Double –precision Floating-Point Format

The following Table 2.4 shows the layout for single (32-bit) and doubles (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets).

Format	Sign	Exponent	Fraction	Bias
Single Precision	1[31]	8[30-23]	23[22-00]	127
Double Precision	1[63]	11[62-52]	52[51-00]	1023

Table 2.4: Comparison of Single and double precision Format.

2.6.1 Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive and negative exponent. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 are stored in the exponent field. A stored value of 200 indicates an exponent (200-127), or 73. For reasons discussed later, exponent of -127(all 0's) and +128(all 1's) are reserved for special numbers. For double precision, the exponent fields are 11 bits, and have a bias of 1023.

2.6.2 Mantissa

The Mantissa, also known as the significant, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits [12]. To find out the

value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

$$8.00 \times 10^{0}$$

$$0.08 \times 10^{2}$$

$$8000 \times 10^{-3}$$

In order to maximize the quantity of represent able numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^{0}. A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1.f, where f is the field of fraction bits.

The IEEE standard requires that machines provide the single format. The double format is optional, but is provided by almost all computers that implement the standard, and we shall therefore *assume that the double format is always provided.* Support for the requirements may be provided by hardware or software, but almost all machines have hardware support for both the single and double formats. Because of its greater precision, the double format is preferred for most applications in scientific computing, though the single format provides an efficient way to store huge quantities of data.

Format	**Precision**	**Machine Epsilon**
Single	P = 24	$\in = 2^{-23} \approx 1.2 \times 10^{-7}$
Double	P = 53	$\in = 2^{-52} \approx 2.2 \times 10^{-16}$
Extended(Intel)	P = 64	$\in = 2^{-63} \approx 1.1 \times 10^{-19}$

Table 2.5: Precision of IEEE Floating Point Formats.

2.7 The Extended Format

The standard also strongly recommends support for an extended format, with at least 15 bits available for the exponent and at least 63 bits for the fractional part of the significand. The Intel microprocessors implement arithmetic with the extended format in hardware, using 80-bit registers, with 1 bit for the sign, 15 bits for the exponent, and 64 bits for the significand. The leading bit of a normalized or subnormal number is not hidden as it is in the single and double formats, but is explicitly stored. Otherwise, the format is much the same as single and double. Other machines, such as the Sparc microprocessor used in Sun workstations, implement extended precision arithmetic in software using 128 bits.

2.8 Special Values

IEEE reserves exponent field values of all 0's and all 1's to denote special values in the floating-point scheme [2].

2.8.1 Zero

As mentioned above, zero is not directly represent able in the straight format, due to the assumption of a leading 1(we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

2.8.2 Denormalized

If the exponent is all 0's, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(1)^s \times 0.f \times 2^{-1022.}$ From this we can interpret zero as a special type of denormalized number.

2.8.3 Infinity

The values +infinity and –infinity are denoted with an exponent of all 1's and a fraction of all 0's. The sign bit distinguishes between negative infinity and positive

infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

2.8.4 Not a Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1's and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN). A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined. A SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage. Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

2.8.5 Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Others operations are follows:

Operation	Result
N ÷ ±Infinity	0
±Infinity × ±Infinity	±Infinity
±nonzero ÷ 0	±Infinity
Infinity + Infinity	Infinity
±0 ÷ ±0	NaN
Inifinity - Inifinity	NaN
±Infinity ÷ ±Infinity	NaN
±Infinity × 0	NaN

Table 2.6: Special Operations.

2.9 Rounding

Arithmetic operations on floating-pint values compute results that cannot be

represented in the given amount of precision. There are many ways of rounding. Each rounding has correct uses and exists for different reasons. The goal in a computation is to have the computer round such that the end result is as correct as possible. Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception (2.10). Except for binary <---> decimal conversion (whose weaker conditions are specified in 2.11), every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section. The rounding modes affect all arithmetic operations except comparison and remainder. The rounding modes may affect the signs of zero sums (2.12), and do affect the thresholds beyond which overflow (2.13) and underflow (2.14) may be signaled [2].

2.10 Inexact

If the rounded results of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler [2].

2.11 Sign Bit

This standard does not interpret the sign of a NaN. Otherwise, the sign of a product or quotient is the exclusive or of the operands' signs, the sign of a sum, or of a difference $x - y$ regarded as a sum $x + (-y)$, differs from at most one of the addends signs, and the sign of the result of the round floating-point number to integral value operation is the sign of the operand. These rules shall apply even when operands or results are zero or infinite.

When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be +in all rounding modes except round toward -∞, in which mode that sign shall be '–'. However, $x + x = x - (-x)$ retains the same sign as x even when x is zero. Except that $\sqrt{-0}$ shall be -0, every valid square root shall have a positive sign.

2.12 Binary <---> Decimal Conversion

Conversion between decimal strings in at least one format and binary floating-point numbers in all supported basic formats shall be provided for numbers throughout the

Format	Decimal to Binary		Binary to Decimal	
	Max *M*	**Max *N***	**Max *M***	**Max *N***
Single	10^9-1	99	10^9-1	53
Double	l0^{17}-1	999	10^{17}-1	340

Table 2.7: Decimal Conversion Ranges.

ranges specified in Table 2.7. The integers *M* and *N* in Tables 2.7 and these are such that the decimal strings have values ±*M*, *10*±N. On input, trailing zeros shall be appended to or stripped from *M* (up to the limits specified in: Table 2.7) so as to minimize *N*. When the destination is a decimal string, its least significant digit should be located by format specifications for purposes of rounding.

2.13 Overflow

The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (Section 4) were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:

1) Round to nearest carries all overflows to ∞ with the sign of the intermediate result

2) Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result

3) Round toward -∞ carries positive overflows to the formats largest finite number, and carries negative overflows to -∞

4) Round toward +∞ carries negative overflows to the format's most negative finite number, and carries positive overflows to +∞ Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by 2^{∞} and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3 \times 2^{n-2}$ in the extended format, when *n* is the number of bits in the exponent field. Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that or a wider format, possibly with the exponent bias adjusted,

but rounded to the destination's precision. Trapped overflow on decimal to binary conversion shall deliver to the trap handler a result in the widest supported format, possibly with the exponent bias adjusted, but rounded to the destination's precision, when the result lies too far outside the range for the bias to be adjusted, a quiet NaN shall be delivered instead.

2.14 Underflow

Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between ±2*E* min which, because it is so tiny, may cause some other exception later such as overflow upon division. The other is extraordinary lorn of accuracy during the approximation of such tiny numbers by denormalized numbers. The implementer may choose how these events are detected, but shall detect these events in the same way for all operations.

2.14.1 After rounding

When a nonzero result computed as though the exponent range were unbounded would lie strictly between ±2*E* min

2.14.2 Before rounding

When a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm 2E_{\min}$. Loss of accuracy may be detected as either

2.14.3 A denormalization loss

When the delivered result differs from what would have been computed were exponent range unbounded.

2.14.4 An inexact result

When the delivered result differs from what would have been computed were both exponent range and precision unbounded (This is the condition called inexact in 2.10).

When an underflow trap is not implemented, or is not enabled (the default case), underflow shall be signaled (by way of the underflow flag) only when both tininess and loss of accuracy have been detected. The method for detecting tininess and loss of

accuracy does not affect the delivered result which might be zero, denormalized, or $\pm 2E_{\min}$. When an underflow trap has been implemented and is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. Trapped underflows on all operations except conversion shall deliver to the trap handler the result obtained by multiplying the infinitely precise result by 2a and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3 \times 2n$-2 in the extended format, where *n* is the number of bits in the exponent field 6. Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.

2.15 Round to Nearest

An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode they representable value nearest to the infinitely precise result shall be delivered, if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. However, an infinitely precise result with magnitude at least $2E_{\max}(2\text{-}2^{-p})$ shall round to ∞ with no change in sign, here $E_{\max}$ and p are determined by the destination format unless overridden by a rounding precision mode (2.17) [2].

2.16 Directed Rounding's

An implementation shall also provide three user-selectable directed rounding modes: round toward +∞, round toward -∞, and round toward 0. When rounding toward +∞ the result shall be the format's value (possibly +∞) closest to and no less than the infinitely precise result. When rounding toward +∞ the result shall be the format's value (possibly -∞) closest to and no greater than the infinitely precise result. When rounding toward 0 the result shall be the format's value closest to and no greater in magnitude than the infinitely precise result.

2.17 Rounding Precision

Normally, a result is rounded to the precision of its destination. However, some systems deliver results only to double or extended destinations. On such a system the user,

which may be a high-level language compiler, shall be able to specify that a result be rounded ins to single precision, though it may be stored in the double or extended format with its wider exponent range. Similarly, a system that delivers results only to double extended destinations shall permit the user to specify rounding to single or double precision. Note that to meet the specifications in 2.15, the result cannot suffer more than one rounding error [2].

2.18 Division by Zero

If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result, when no trap occurs, shall be a correctly signed ∞ (2.11).

2.19 Invalid Operation

The invalid operation exception is signaled if an operand is invalid for the operation on to be performed. The result, when the exception occurs without a trap, shall be a quiet NaN (2.8.5) provided the destination has a floating-point format. The invalid operations are

1) Any operation on a signaling NaN (2.8.5)
2) Addition or subtraction—magnitude subtraction of infinites such as, (+∞) + (-∞)
3) Multiplication -0 × ∞
4) Division -0/0 or ∞/∞
5) Remainder -*x* REM *y*, where *y* is zero or *x* is infinite
6) Square root if the operand is less than zero
7) Conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled
8) Comparison by way of predicates involving < or >, without? When the operands are *unordered.*

CHAPTER 3
SQUARE ROOT OPERATION

3.1 Introduction

Computation with floating point arithmetic is a necessary talk in many applications. Until the end of the 1980's floating point operations were mainly implemented as software emulation while hardware implementation was an option for mainstream general purpose microprocessors, due to the high cost of the hardware. At the present, all major microprocessors include hardware specific for handling floating point operations, but the advancements in reconfigurable logic in the mid-1990's, particularly in size and speed, allows for the implementation of multiple, high performance floating point units in Field Programmable Gate Arrays(FPGA).

3.2 Unpacking

The unpacking block separates the 64 bits of each number, *A* and *B*, into the sign bit S which is the most significant bit, the exponent *E* which is the next significant 11 bits, and the mantissa *M* which is the least significant 52 bits. The mantissa is converted to an explicit fraction by concatenating the hidden 1 bit. The biased exponent and the input number are used to determine whether the input number is NaN (not a number), infinity, zero, or neither of these [2]. If any of the three first conditions is true, the flag *F* is set and computation can stop. Otherwise, S_A, E_A and M_A are fed to the next appropriate blocks.

3.3 Double Precision Square Root

As Figure 3.1 shows, the square root unit takes as input a 64-bit number *A* and outputs its square root *R* as a 64-bit number.

3.3.1 Exponent Calculation

This block computes the resulting exponent of the square root based on the biased exponent:

Input: E_A 11-bit exponent

Output: E_c 11-bit exponent

F_s shift flag

1: F_s = 0;

2: **if** (E_A is even)

3: E_c = (E_A + 1022) / 2;

4: F_s = 1;

5: **else if** (E_A is odd)

6: E_c = (E_A + 1023) / 2;

7: **else**

8: $display ("Invalid Number");

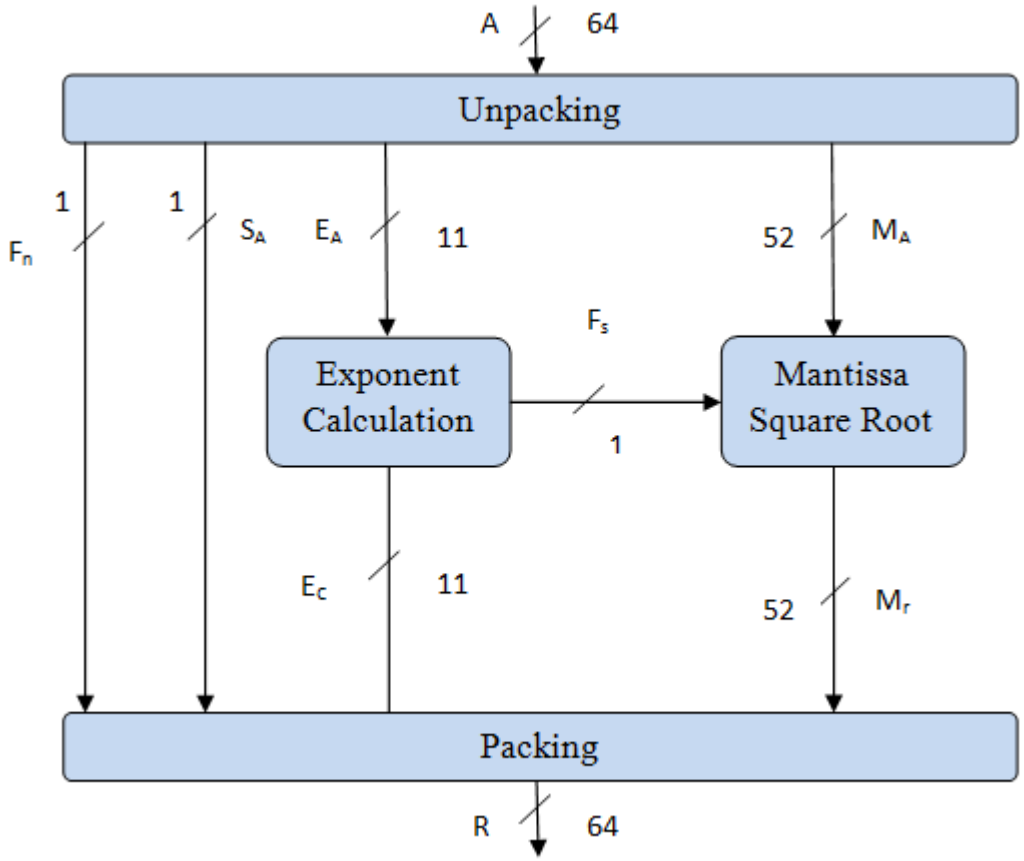


Figure 3.1: Double Precision Floating Point Square Root Unit.

If the biased exponent is even, the biased exponent is added to 1022 and divided by two. In addition, shift flag is set to indicate that the mantissa should be shifted to the left by 1 bit before computing its square root. Note that before shifting, the mantissa bits are stored in

the 53-bit register M_A as 1.xxxx… After left shifting, M_A contents become 1x.xxx… If the biased exponent is odd, the biased exponent is added to 1023 and divided by two [1].

3.3.2 Mantissa Square Root

The block, which computes the square root of the mantissa, uses the following iterative approach based on two 57-bit registers, *X* and *T*, and a 55-bit register M_r.

Input: M_A 53-bit mantissa

F_s 1-bit shift flag

Output: M_r 55-bit mantissa

```
1: Mr = 000…0;
2: T = 01000…0;
3: Initial
4:   begin
5:       for (i = 0; i < 55; i = i + 1)
6:           if (i = =0)
7:               if (Fs == 1)
8:                   X = MA / 2;
9:               else
10:                  X = MA;
11:          else
12:              if (X >= T)
13:                  X = 2 * (X – T);
14:                  T = T [(56-i)..0] << 1;
15:                  T [56-i] = 1;
16:              else
17:                  X = 2 * X;
18:                  T = T [(56-i)...0] >> 1;
19:                  T [56-i] = 0;
21:  end
22: Mr = T [56...2];
```

In the first iteration, the contents of M_A are shifted to the right and stored in *X* if the shift flag is on as shown in lines 6, 7, and 8. Otherwise, they are stored in X as shown in

line 10. In other iterations, *X* is compared to *T* as shown in line 12. If it is greater or equal to *T*, the contents of *X* are subtracted from those of *T*, shifted to the left, and stored in *X* as shown in line 13. The contents of *T* are shifted to the left by one bit starting from the current pointer position as shown in line 14. Then, a 1 is inserted in the current bit position bit in *T* as shown in line 15. Note that in each iteration, a pointer points to the current bit that will be replaced in *T*. If *X* is less than *T*, its contents are shifted to the left and stored in *X* as shown in line 17. The contents of *T* are shifted to the right by one bit starting from the current pointer position as shown in line 18. Then, a 0 is inserted in the current bit position bit in *T* as shown in line 19. After the last iteration, the contents of *T*, with the exception of the two least significant bits, are copied to M_r as shown in line 22. The computation of the mantissa's square root takes 55 clock cycles to complete [1].

3.3.3 Packing

The packing block concatenates from left to right the sign (*S*), the 11-bit exponent (E_c), and the 53-bit mantissa (M_r).

3.4 Implementation of Double Precision Square Root using Verilog code

This block represents the implementation of Double Precision Floating Point Square root. There are seven ports namely, data input (D_IN), clock (CLK), ready to accept output (RDY_OUT), valid input value (VAL_IN), data output (D_OUT), ready to accept input (RDY_IN), valid output value (VAL_OUT), all signals are active high.

The algorithm used is iterative and will take 110 clock cycles for double precision. The sign bit is simply passed through from input to output, so if given negative input, this code will produce a negative output corresponding to -1 × sqrt (|input|).

The representation has seven ports:

1) D_IN (63:0): It is the input of double precision floating point square root.
2) CLK : It is the input clock.
3) RDY_OUT: It is the output wire, this input will give high when asserted, and downstream logic is ready to accept output of module.
4) VAL_IN: It is the input value, the module will only accept input when RDY_IN

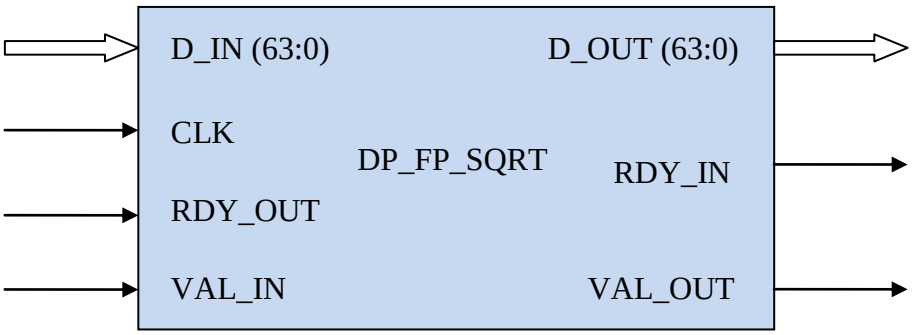


Figure 3.2: Pin Diagram of Square root.

is low, then VAL_IN should remain asserted until RDY_IN goes high. VAL_IN and RDY_IN go high. VAL_IN and RDY_IN must both be asserted for on clock cycle for computation to begin.

5) D_OUT (63:0): It is the output register of double precision floating point square root.
6) RDY_IN: It is the output wire, this output high when module is ready to accept input.
7) VAL_OUT: It is the output register, VAL_OUT is asserted when the output is valid. VAL_OUT will remain asserted and D_OUT will persist until VAL_OUT and RDY_OUT have both been asserted for one clock cycle.

This code was designed using synchronous resets, for use in FPGA's. Both numerical accuracy and performance of the double precision versions of this code have been verified in a Xilinx Spartan 3E XC3s500 at 50 MHZ.

CHAPTER 4
FPGA AND BIST IMPLEMENTATION

4.1 Introduction to FPGA

A field-programmable gate array (FPGA) is a semiconductor device that can be configured by the customer or designer after manufacturing hence the name "field-programmable". Field-programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. FPGAs are programmed using a logic circuit diagram or a source code in a hardware description language (HDL) to specify how the chip will work. They can be used to implement any logical function that an application-specific integrated circuit (ASIC) could perform, but the ability to update the functionality after shipping offers advantages for many applications. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" somewhat like a one-chip programmable breadboard. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Field-programmable gate arrays (FPGAs) are truly revolutionary devices that blend the benefits of both hardware and software. They implement circuits just like hardware, providing huge power, area, and performance benefits over software, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However, unlike in ASICs, these computations are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times [13].

FPGAs are being incorporated as central processing elements in many applications such as consumer electronics, automotive, image/video processing, military/aerospace, base-stations, networking/ communications, supercomputing and wireless applications.

4.1.1 FPGA Technology Trends

- General trend is bigger and faster.
- This is being achieved by increases in device density through ever smaller fabrication process technology.
- New generations of FPGAs are geared towards implementing entire systems on a single device.
- Features such as RAM, dedicated arithmetic hardware, clock management and transceivers are available in addition to the main programmable logic.
- FPGAs are also available with embedded processors (embedded in silicon or as cores within the programmable logic fabric).

4.2 FPGA Implementation

The FPGA that is used for the implementation of the circuit is the Xilinx Spartan 3E (Family), XC3S5000 (Device), FG320 (Package) FPGA device. The working environment/tool for the design is the Xilinx ISE 9.2i is used for FPGA Design flow of verilog code as shown in figure 4.1.

4.2.1 Overview of FPGA Design Flow

As the FPGA architecture evolves and its complexity increases, CAD software has become more mature as well. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit stream to program FPGA chips. A typical FPGA design flow includes the steps and components shown in Fig. 4.1. Inputs to the design flow typically include the HDL specification of the design, design constraints, and specification of target FPGA devices [13]. We further elaborate on these components of the design input in the following:

Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained.

The second design input component is the choice of FPGA device. Each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost,

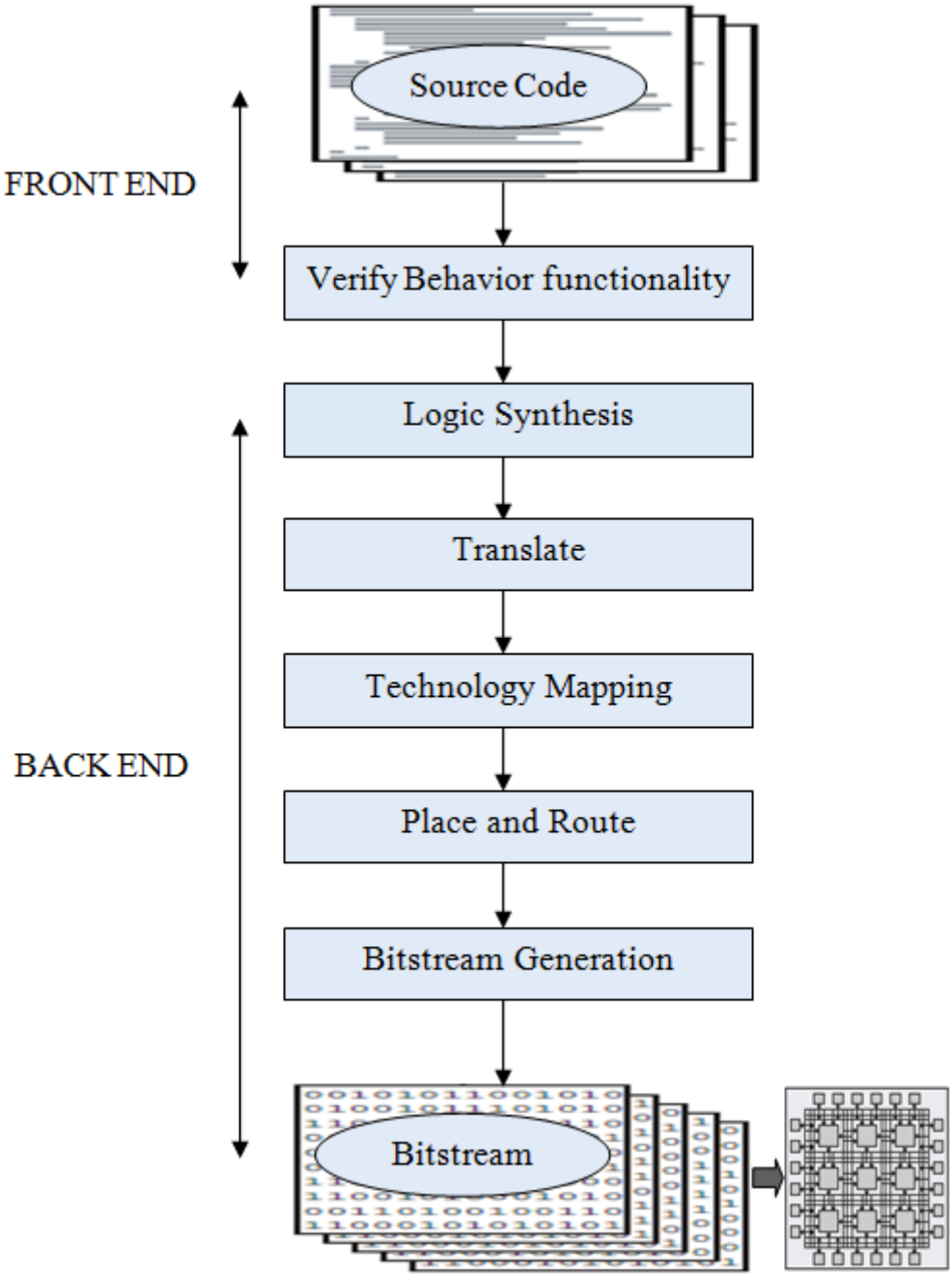


Figure 4.1: FPGA Design Flow.

and power tradeoffs. The selection of target device may be an iterative process. The designer may start with a small (low capacity) device with a nominal speed-grade. But, if synthesis effort fails to map the design into the target device, the designer has to upgrade to

a high-capacity device. Similarly, if the synthesis result fails to meet the operating frequency, he has to upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase in some cases by 50% or even by 100%. This clearly underscores the need to have better synthesis tools, since their quality directly impacts the performance and cost of FPGA designs [14].

4.2.2 Design Entity

The basic architecture of the system is designed in this step which is coded in a Hardware description Language like VERILOG or VHDL. A design is described in VERILOG using the concept of a design module. A design module is split into two parts, each of which is called a design unit in VERILOG. The module declaration represents the external interface to the design module. The module internals represents the internal description of the design module-its behavior, its structure, or a mixture of both. Take square root double precision floating point unit in VERILOG. Consider the square root double precision floating point unit as a single chip package, it will have 64 input pins and 64 output pins; keeping no concern with power and ground pins in modeling square root double precision floating point unit design.

4.2.3 Behavioral Simulation

After the design phase, create a test bench waveform containing input stimulus to verify the functionality of the verilog code module using a simulation software i.e. Xilinx ISE 9.2i for different inputs to generate outputs and if it verifies then proceed further, otherwise modifications and necessary corrections will be done in the HDL code. This is called as the behavioral simulation.

4.2.4 Design Synthesis

After the correct simulations results, the design is then synthesized. During synthesis, the Xilinx ISE tool does the following operations:

a) HDL Compilation: The tool compiles all the sub-modules in the main module if any and then checks the syntax of the code written for the design.

b) Design Hierarchy Analysis: Analysis the hierarchy of the design.

c) HDL Synthesis: The process which translates VHDL or Verilog code into a device

netlist formate, i.e. a complete circuit with logical elements such as Multiplexer, Adder/subtractors, counters, registers, flip flops Latches, Comparators, XORs, tristate buffers, decoders, etc. for the design. If the design contains more than one sub designs, ex. to implement a processor, we need a CPU as one design element and RAM as another and so on, and then the synthesis process generates netlist for each design element. Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected. The resulting netlist is saved to an NGC (Native Generic Circuit) file (for Xilinx® Synthesis Technology (XST)).

d) Advanced HDL Synthesis: Low Level synthesis: The blocks synthesized in the HDL synthesis and the Advanced HDL synthesis are further defined in terms of the low level blocks such as buffers, lookup tables. It also optimizes the logic entities in the design by eliminating the redundant logic, if any. The tool then generates a 'netlist' file (NGC file) and then optimizes it. The final netlist output file has an extension of .ngc. This NGC file contains both the design data and the constraints. The optimization goal can be pre-specified to be the faster speed of operation or the minimum area of implementation before running this process. The level optimization effort can also be specified. The higher the effort, the more optimized is the design but higher effort can also be specified. The higher the effort, the more optimized is the design but higher effort requires larger CPU time (i.e. the design time) because multiple optimization algorithms are tried to get the best result for the target architecture.

4.2.5 Design Implementation

The design implementation process consists of the following sub processes:

1. Translation: The Translate process combines all the input netlists and constraints to a logic design file. This information is saved as a NGD (Native Generic Database) file. This can be done using NGD Build program and .ngd file describes the logical design reduced to the Xilinx device primitive cells. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE,

Constraint Editor Etc.

2. Mapping: The Map process is run after the Translate process is complete. Map process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

3. Place and Route: Place and Route **(**PAR) program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Example if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may affect some other constraint. So tradeoff between all the constraints is taken account by the place and route process .The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consist the routing information.

4. Bitstream Generation: The collection of binary data used to program the reconfigurable logic device is most commonly referred to as a "bitstream," although this is somewhat misleading because the data are no more bit oriented than that of an instruction set processor and there is generally no "streaming." While in an instruction set processor the configuration data are in fact continuously streamed into the internal units, they are typically loaded into the reconfigurable logic device only once during an initial setup phase.

5. Functional Simulation: Post-Translate (functional) simulation can be performed prior to mapping of the design. This simulation process allows the user to verify that the design has been synthesized correctly and any differences due to the lower level of abstraction can be identified.

6. Static timing analysis: Three types of static timing analysis can be performed that are:

(i) Post-fit Static timing analysis: The timing results of the Post-fit process can be

analyzed. The Analyze Post-Fit Static timing process opens the timing Analyzer window, which interactively select timing paths in design for tracing.

(ii) Post-Map Static Timing Analysis: Analyze the timing results of the Map process. Post-Map timing reports can be very useful in evaluating timing performance. Although route delays are not accounted for the logic delays can provide valuable information about the design. If logic delays account for a significant portion (>50%) of the total allowable delay of a path, the path may not be able to meet your timing requirements when routing delays are added. Routing delays typically account for 45% to 65% of the total path delays. By identifying problem paths, redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path. If logic-only-delays account for much less (35%) than the total allowable delay for a path or timing constraint, then the place-and-route software can use very low placement effort levels. In these cases, reducing effort levels allow to decrease runtimes while still meeting performance requirements.

(iii) Post Place and Route Static Timing Analysis: Aanalyze the timing results of the Post-Place and Route process. Post-PAR timing reports incorporate all delays to provide a comprehensive timing summary. If a placed and routed design has met all of timing constraints, then proceed by creating configuration data and downloading a device. On the other hand, identify problems and the timing reports, try fixing the problems by increasing the placer effort level, using re-entrant routing, or using multi-pass place and route. Redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the paths.

(iv) Timing Simulation: Perform Post-Place and Route simulation after the design has been and routed. After the design has been through all of the Xilinx implementation tools, a timing simulation netlist can be created. This simulation process allows to see how the design will behave in the circuit. Before performing this simulation it will benefit to create a test bench or test fixture to apply stimulus to the design. After this a .ncd file will be created that is used for the generation of power results of the design.

4.3 Procedure for Implementing the Design in FPGA

The Spartan-3E Starter Kit board highlights the unique features of the Spartan-3E FPGA family and provides a convenient development board for embedded processing applications. The board highlights these features:

Spartan-3E FPGA specific features

- Parallel NOR Flash configuration
- MultiBoot FPGA configuration from Parallel NOR Flash PROM
- SPI serial Flash configuration

Embedded development

- MicroBlaze 32-bit embedded RISC processor
- PicoBlaze 8-bit embedded controller
- DDR memory interfaces

Implement the design and verify that it meets the timing constraints after check syntax and synthesis.

4.3.1 Implementing the Design

1. Select the lcd_fp_sqrt source file in the Sources window.
2. Open the Design Summary by double-clicking the View Design Summary process in the Processes tab.
3. Double-click the Implement Design process to view the Translate and Map Place & Route in the Processes tab.
4. Notice that after Implementation is complete, the Implementation processes have a Green check mark next to them indicating that they completed successfully without Errors or Warnings.
5. Locate the Performance Summary table near the bottom of the Design Summary.
6. Click the All Constraints Met link in the Timing Constraints field to view the Timing Constraints report. Verify that the design meets the specified timing requirements.
7. Close the Design Summary.

4.3.2 Assigning Pin Location Constraints

1. Verify that lcd_fp_sqrt is selected in the Sources window.

2. Double-click the Floorplan Area/IO/Logic - Post Synthesis process found in the User Constraints process group. The Xilinx Pinout and Area Constraints Editor (PACE) opens.

3. Select the Package View tab.

4. In the Design Object List window, enter a pin location for each pin in the Location column using the following information:

- CLK input port connect to FPGA pin C9
- RDY_OUT input port connect to FPGA pin L13
- VAL_IN input port connect to FPGA pin L14
- SF_CE0 input port connect to FPGA pin D16
- LCD_RS input port connect to FPGA pin L18
- LCD_RW input port connect to FPGA pin L17
- LCD_E input port connect to FPGA pin M18
- LCD_4 input port connect to FPGA pin R15
- LCD_5 input port connect to FPGA pin R16
- LCD_6 input port connect to FPGA pin P17
- LCD_7 input port connect to FPGA pin M15

5. Select File to Save. You are prompted to select the bus delimiter type based on the Synthesis tool you are using. Select XST Default and click OK.

6. Close PACE.

Notice that the Implement Design processes have an orange question mark next to them, indicating they are out-of-date with one or more of the design files. This is because the UCF File has been modified.

4.3.3 Download Design to the Spartan-3E Demo Board

This is the last step in the design verification process. This section provides simple Instructions for downloading the counter design to the Spartan-3 Starter Kit demo board.

1. Connect the 5V DC power cable to the power input on the demo board (J4).

2. Connect the download cable between the PC and demo board (J7).

3. Select Implementation from the drop-down list in the Sources window.

4. Select lcd_fp_sqrt in the Sources window.

5. In the Process window, double-click the Configure Target Device process.

6. The Xilinx webtalk Dialog box may open during this process. Click Decline.

7. In the Welcome dialog box, select Configure devices using Boundary-Scan (JTAG).

8. Verify that automatically connect to a cable and identify Boundary-Scan chain is selected.

9. Click Finish.

10. If you get a message saying that there are two devices found, click OK to continue. The devices connected to the JTAG chain on the board will be detected and displayed in the impact window.

11. The Assign New Configuration File dialog box appears. To assign a configuration file to the XC3S500E device in the JTAG chain, select the counter.bit file and click Open.

12. If you get a Warning message, click OK.

13. Select Bypass to skip any remaining devices.

14. Right-click on the XC3S500E device image, and select Program. The Programming Properties dialog box opens.

15. Click OK to program the device.

When programming is complete, the Program Succeeded message is displayed. On the board, LCD Display the output of lcd_fp_sqrt.

4.4 LCD and KEYBOARD Interfacing

A programming file is generated by running the Generate Programming File process. This process can be run after the FPGA design has been completely routed. The Generate Programming File process runs BitGen, the Xilinx bitstream generation program, to produce a bitstream (.BIT) or (.ISC) file for Xilinx device configuration. The FPGA device is then configured with the .bit file using the JTAG boundary scan method. After the Spartan device is configured for the intended design, then its working is verified by applying different inputs.

4.4.1 LCD Interfacing in FPGA

The figure 4.2 shows represent the implementation of LCD Interfacing of Double Precision Floating Point Square root.

The representation has eleven ports:

1) CLK: It is the input clock.
2) RDY_OUT: It is the input wire of double precision floating point square root, it is ready to accept output.
3) VAL_IN: It is the input wire of double precision floating point square root, it is valid input value.
4) LCD_E: It is the output register, Read/Write Enable Pulse '0' for Disabled. '1'for Read/Write operation enabled. FPGA pin number is M18.

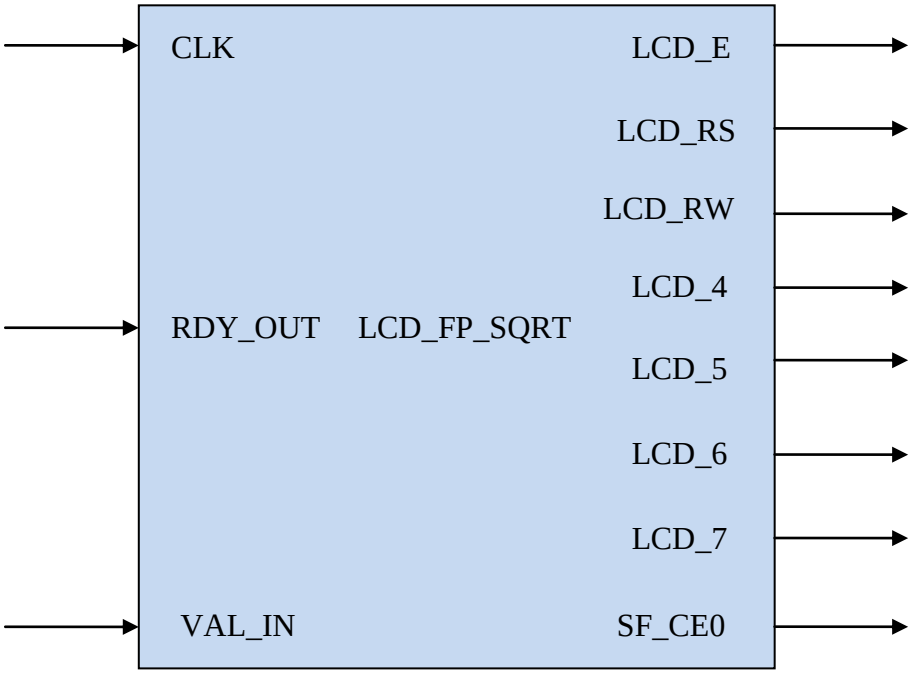


Figure 4.2: Pin Diagram of LCD Interfacing of Double Precision Floating Point SQRT.

5) LCD_RS: It is the output register, Register Select '0' for Instruction register during write operations, Busy Flash during read operations. '1' for Data for read or writes operations, FPGA pin number is L18.
6) LCD_RW: It is the output register, Read/Write Control '0' for WRITE, LCD accepts data. '1' for READ, LCD Presents data. FPGA pin number is L17.
7) LCD_4: It is the output register. Data bit is DB4 and FPGA pin number is R15.
8) LCD_5: It is the output register. Data bit is DB5 and FPGA pin number is R16.
9) LCD_6: It is the output register. Data bit is DB6 and FPGA pin number is P17.
10) LCD_7: It is the output register. Data bit is DB7 and FPGA pin number is M15.

11) SF_CE0: It is the output register, when the Strata Flash memory is disabled (SF_CE0 = High), then the FPGA application has full read/write access to the LCD.

4.4.2 Keyboard Interfacing in FPGA

The figure 4.3 shows represent the implementation of Keyboard Interfacing of

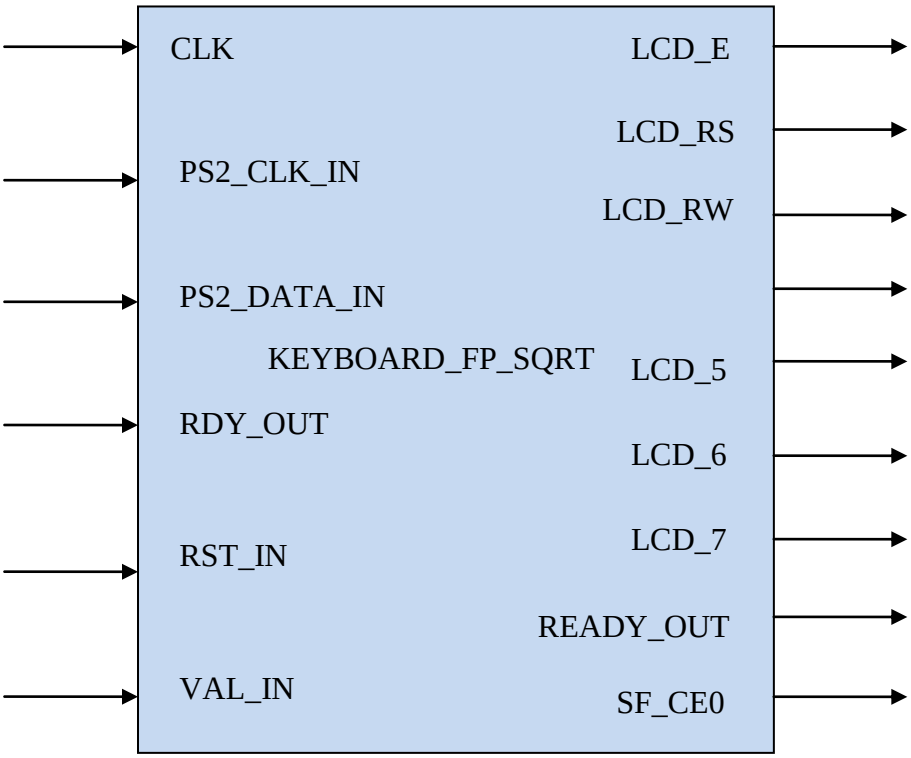


Figure 4.3: Pin Diagram of Keyboard Interfacing of Double Precision FP SQRT.

Double Precision Floating Point Square root. There are fifteen ports, some of the ports used in keyboard interfacing these are all LCD interfacing ports rest of the ports are discussed as follows.

The representation has rest of the four ports:

1) PS2_CLK_IN: It is the input wire of PS/2 Clock line. FPGA pin number is G14.
2) PSE_DATA_IN: It is the input wire of PS/2 Data line, FPGA pin number is G13.
3) RST_IN: It is the input wire of Asynchronous reset.
4) READY_OUT: It is the output register, this output high when module is output ready.

4.5 Built-In Self-Test

The basic idea of BIST, in its most simple form, is to design a circuit so that the circuit can test itself and determine whether it is “good” or “bad” (fault-free or faulty, respectively). This typically requires that additional circuitry and functionality be incorporated into the design of the circuit to facilitate the self-testing feature. This additional functionality must be capable of generating test patterns as well as providing a mechanism to determine if the output responses of the circuit under test (CUT) to the test patterns correspond to that of a fault-free circuit. One of the first definitions of BIST (referred to as self-verification at the time).

Current trends in semiconductor technologies, as well as in design methodologies, readily indicate that the ever-increasing degree of integration of devices on a single substrate continuously demands more efforts in achieving zero-defect designs. Clearly, this ultimate quality goal cannot be met without including testability as a design objective. Although the process of integration, strongly supported by CAD tools, has already led to an improved quality of integrated circuits, adding testability to a number of criteria considered during design, such as performance, area, power, manufacturability, etc., may significantly enhance the reliability of products and their overall quality.

Testability, although difficult to define and quantify because of the many different factors affecting costs and quality of testing, reflects ability of the circuit's tests to detect, and possibly locate, failures causing malfunctioning of the circuit. As the number and kind of faults that may occur depends on the type of device and a technology used to fabricate it, evaluation of test quality can be a difficult and often computationally intensive process.

An attractive alternative to the classical testing scenario, where test patterns are applied from an external tester, is built-in self-test (BIST). In BIST, an additional "on-chip" circuitry is included to generate test vectors, evaluate test responses, and control the test. Random, or in fact pseudo-random, patterns can be generated by simple circuits, and test responses can be compacted into a short statistic by calculating a signature. This signature, 1obtained from the CUT, can be subsequently compared with a fault-free signature. BIST has revolutionized the way the integrated circuits can be tested. It reduces the cost of manufacturing testing by shortening the test application time, minimizing the amount of test data stored, and lowering the cost of testing equipment. Its implementation

can result in a reduction of the product development cycle and cost, as well as a reduction of the cost of system maintenance. The latter benefits may have a dramatic impact on the economics of testing. It follows from the fact that built-in test circuitry can test chips, boards, and the entire system virtually without very expensive external automatic test equipment. The ability to run tests at different levels of the system's hierarchy significantly simplifies diagnostic testing, which in turn improves troubleshooting procedures and sanity checks during assembly, integration, and field service. Since the BIST hardware is an integral part of the chip, BIST, in principle, could allow for at-speed testing, thus covering faults affecting circuit timing characteristics.

The basic BIST objectives are often expressed with respect to test-pattern generation and test-response compaction. It is expected that appending BIST circuitry to the circuit under test will result in high fault coverage, short test application time, small volume of test data, and compatibility with the assumed DFT methodology. High fault coverage in BIST can be achieved only if all faults of interest are detected and their effects are retained in the final signature after compaction. Numerous test generation and test-response compaction techniques have been proposed in the open literature and are used in industrial practice as implementation platforms to cope with these objectives for various types of failures, errors, and a variety of test scenarios. In the following subsections we will outline several schemes used in different BIST environments. They have gained a wide acceptance by BIST practitioners, and their superiority over non- BIST approaches ensures a successful applicability of BIST in current and future technologies.

Clearly, the use of BIST is also associated with certain costs. Additional silicon area is required for the test hardware to perform test-pattern generation and test-response compaction. Some performance degradation may be introduced due to the presence of multiplexers needed to apply the test patterns in the test mode. Some testing equipment may still be needed to test the BIST hardware and to carry out the parametric testing. BIST also requires more rigid design. In particular, unknown states are not allowed since they can produce unknown signatures [15].

4.5.1 The Economic case for BIST

These are some chip-level, testability problems include:

1) A very high and still increasing logic-to-pin ratio, which points out a highly unbalanced relationship between a limited number of input/output ports and unprecedently complex semiconductor devices which are accessible only through these terminals,
2) A circuit complexity which continues to grow as new submicron technologies offer higher densities and speeds,
3) An increasingly long test-pattern generation and test application time; it has been repeatedly reported that functional and random tests for general class of circuits containing memory elements have very low fault coverage; in the case of deterministic patterns, an extraordinary amount of processing time might be required to generate a test vector, and then it may take a large number of clock cycles to excite a fault and propagate it to primary outputs,
4) A prohibitively large volume of test data that must be kept by testing equipment,
5) An inability to perform at-speed testing through external testing equipment,
6) Incomplete knowledge of the gate level structure as designers are separated from the level of implementation by automated synthesis tools,
7) Lack of methods and metrics to measure the completeness of employed testing schemes,
8) Difficulties in finding skilled resources.

4.5.2 Complexity

One unfortunate property of large VLSI circuits is that testing cannot be easily partitioned. Consider two cascaded devices. There is frequently no simple way to obtain tests for the complete system from tests for the complete system from tests for the individual parts. In fact, even though each part is fully testable and has a test set that gives 100% struck-fault coverage, the cascaded connection of the two parts will often have untestable and redundant hardware and much lower struck-fault coverage. In order words, testing is global problem. It is well known that there is no simple way to create tests for an entire circuit board (PCB) from tests for the chips on the board. For design the test development effort, BIST provides a way to hierarchically decompose the electronic system-under-test, so this allows sub-assemblies to be first run through a BIST cycle. Finally, if there are no board faults, then the entire system can be run through a BIST

cycle. As an example, consider a system containing boards, which in turn contain chips. For a chip test, the system sends a control signal to the PCB, which then activates self-test on the desired chip, and sends the test result back to the system. BIST efficiently tests embedded components and interconnect, thus reducing the burden on system-level test, which now only needs to verify the synergy among the functional components. When faults occur, the BIST hardware should be designed to indicate via an error signal or bus which sub-assembly is faulty. This greatly reduces repair costs.

4.6 Test Generation Problems

It is difficult to carry a test stimulus involving hundreds of chip input through many layers of circuitry to the chip-under-test, and the convey the test result back through the many circuit layers to an observable point. BIST localizes testing, which eliminates these problems.

4.7 Test Application Problems

In the past, in-circuit testing [17] used a bed of nails fixture customized for the PCB under test. The Bed-of-nails tester applied stimuli to the solder balls on the back of the PCB where the component leads were soldered to the PCB. Power was applied only to the component under test-all others in the PCB were left unpowered. It was effective for chip diagnosis and board wiring so it is not helpful in system-level diagnosis. Also, surface mount technology components are often mounted densely on both sides of the board, and the PCB wire pitch is also too small for accurate probing of the back of the board by the bed-of-nails tester. Therefore, ICT is no longer a solution. BIST, however, solves theses problems by eliminating expensive ATE, and BIST also lets us use the same tests virtually unlimited circuits access via test points designed into the circuits through scan chains, resulting in an electronic bed-of-nails. Another advantage of BIST is that the testing capability grows with the VLSI technology, whereas with external testing, the capability always lags behind the VLSI technology capability. Logic agates transistors are relatively cheap compared to the labor needed to develop test programs, the cost of automatic test equipment, and the cost of real time for the test to be run on production chips with ATE.

An additional benefit of BIST is lower test development cost, because BIST can be automatically added to a circuit with a CAD tool. Also, BIST generally provides 90 to 95% fault coverage and even 99% in exceptional cases. The test engineer used no longer worry and about back driving problem in-circuit test(where electrical stimuli provided to the middle of the circuitry damage outputs of logic gates), or how much memory is available in ATE.

Level	Design & Test	Fabrication	Prod. Test	Maintenance Test	Diagnosis & Repair	Service interruption
CHIPS	+/-	+	-			
BOARDS	+/-	+	-		-	
SYSTEMS	+/-	+	-	-	-	-

Table 4.1: Built-in-Self-testing costs.

'+': Cost increases; '-': Cost reduction; '+/-': Cost increase

Table 4.1 [15, 18] shows the relative BIST cost at the chip, board, and system levels of packing. BIST always requires added circuit hardware for a test controller to operate the testing process, design for testability hardware in the circuit to improve fault coverage during BIST, a hardware pattern generator to generate test-patterns algorithmically during testing, and some form of hardware response compacter to compact the circuit response during testing. We see an increase in fabrication costs at all three levels of circuit packing. The BIST cost is frequently measured in terms of the added logic gates are declining, because hardware continues to become cheaper, but the relative costs of added long wires for test mode control are not really decreasing. This cost can also include added circuit delay, due to the extra device loads and delays from the test hardware. This may require a slight increase in the clock rate, and additional electrical adjustment to the design. Also the test hardware can consume extra power, which is an additional cost. Since the BIST circuitry used chip area. BIST feasibility for a system must be evaluated using benefit-cost analysis, in the context of assessing total life cycle costs.

4.8 Random Logic BIST

4.8.1 Definitions

1. BILBO

Built-In-logic Block (BILBO) is a bank of circuit flop-flops with added testing hardware, which can be configured to make the flip-flop behave like a scan chain, a linear feedback shift register (LFSR) pattern generator, an LFSR – based response compacter, or merely as D flip-flops.

2. Concurrent Testing

A testing process that detects faults during normal system operation.

3. Exhaustive Testing

A BIST approach in which all 2^n possible patterns are applied to n circuit inputs.

4. Irreducible Polynomial

A Boolean polynomial that cannot be factored.

5. LFRS

Linear Feedback shift Register (LFRS) is hardware that generates an exhaustive or pseudo-random pattern sequence of test pattern, and can also be used as a response compacter.

6. Non -Concurrent testing

A testing process that require suspension of normal system operation to test for faults.

7. Primitive Polynomial

A primitive Boolean polynomial p(x) has the property that we can compute increasing powers of x modulo p(x), and obtain all possible non zero polynomials of degree less than p(x). We compute the remainders of x/p(x), x^2/p(x), and so on. A Primitive polynomial defines a mathematical number system that has the properties of a

mathematical field.

8. Pseudo exhaustive testing

A BIST approach in which a circuit having n primary inputs (PIs) is broken into smaller, overlapping blocks, each with < n inputs. Each of the smaller blocks is tested exhaustively.

9. Pseudo random testing

A BIST pattern generator that produces, via an algorithm, a subset of all possible tests that has most of the properties of randomly-generated tests. The random patterns must have statistically high enough fault coverage to ensure a good test.

4.9 BIST Process

Figure 4.4 shows the BIST system hierarchy and all three levels of packing

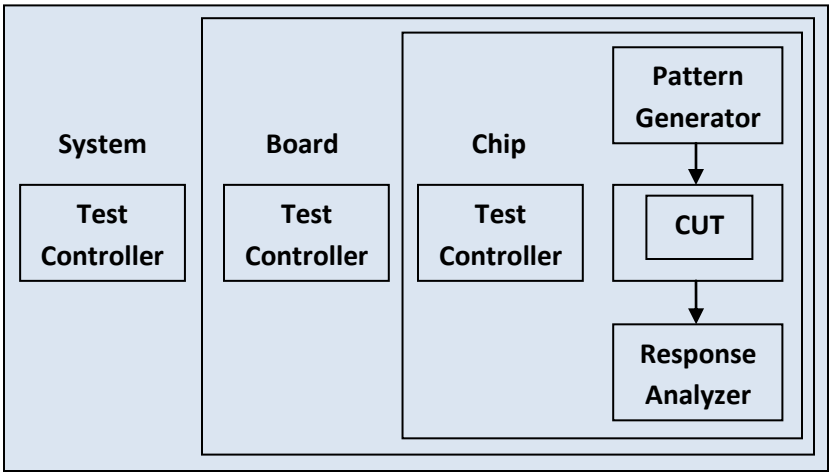


Figure 4.4: BIST Hierarchy.

mentioned earlier. The System has several PCB's, each of which, in turn, has multiple chips, the system test controller can activate self test simultaneously on all PCB's. Each test controller on each PCB can activate self-test on all chips on the PCB's. The Test controller on chip executes self test for that chip, and then transmits the results to the PCB Test Controller, which accumulates test results from all chips on the board and sends the results to the system test controller. The system Test controller uses all of these results to

isolate faulty chips and boards [15, 18].

System diagnosis is effective only if the self-test procedures are through. For BIST, fault coverage is a major issue. Other issues are chip area overhead, its impact on chip yield, the cost of the additional chip pins required for test, the performance penalty in terms of added circuit delay, and extra power requirement. For BIST, the test engineer frequently, but not always, modifies the chip logic to make all latches and flip-flops controllable, perhaps by using the scan technique [15, 18].

4.10 Types of Test Patterns

There are several classes of test patterns. As a result, TPGs are sometimes classified according to the class of test patterns they produce [11]. These classes of test patterns are described below along with examples of the types of circuits that can be used in a BIST environment to generate these test patterns [19].

- **Deterministic test patterns** are developed to detect specific faults and/or structural defects for a given CUT. An example of hardware for applying deterministic vectors would include a ROM with a counter for addressing the ROM, as was used in the simple BIST design.
- **Algorithmic test patterns** are similar to deterministic patterns in that they are specific to a given CUT and are developed to detect specific fault models in the CUT. However, because of the repetition and/or sequence typically associated with algorithmic test patterns, the hardware for generating algorithmic vectors is usually a finite state machine (FSM). There is considerable applicability of this test pattern generation approach to BIST for regular structures such as RAMs.
- **Exhaustive test patterns** produce every possible combination of input test patterns. This is easy to see in the case of an *N*-input combinational logic circuit where an *N*-bit counter produces all possible 2^N test patterns and will detect all detectable gate-level stuck-at faults as well as all detectable wired-AND/OR and dominant bridging faults in the combinational logic circuit without the need for fault simulation. This counter-based approach will not detect all possible transistor-level faults or delay faults since those faults require specific ordering of vectors as well as the ability to repeat certain test vectors within the vector set. As a result,

fault simulation will be required to determine the fault coverage for transistor and delay faults. Exhaustive test patterns are not practical for large *N*.

- **Pseudo-exhaustive test patterns** are an alternative to exhaustive test patterns. In this case, each partitioned combinational logic subcircuit will be exhaustively tested; each *K*-input subcircuit receives all 2^Kpossible patterns, where $K < N$, while the outputs of the subcircuit are observed during the testing sequence [20]. As a result, this approach is more practical for large *N* as long as *K* is not large. Like exhaustive test patterns all detectable gate-level stuck-at faults and bridging faults within the combinational logic sub circuits are guaranteed to be detected without the need for fault simulation. Bridging faults between the sub circuits are not guaranteed detection, along with transistor and delay faults, requiring fault simulation to determine fault coverage. Candidate hardware for this type of test pattern generation for BIST includes counters, Linear Feedback Shift Registers (LFSRs), and Cellular Automata (CA).
- **Pseudo-random test patterns** are the most commonly produced patterns by TPG hardware found in BIST applications. The primary hardware for producing these test patterns are LFSRs and CA. Pseudo-random test patterns have properties similar to those of random pattern sequences but the sequences are repeatable.
- **Weighted pseudo-random test patterns** are good for circuits that contain random pattern resistant faults. This type of test pattern generation uses an LFSR or CA to generate pseudo-random test patterns and then filters the patterns with combinations of AND/NAND gates or OR/NOR gates to produce more logic 0s or logic 1s in the test patterns applied to the CUT.
- **Random test patterns** have frequently been used for external functional testing of microprocessors [21] as well as in ATPG software [22]. But the generation of truly random patterns for a BIST application is of little value since these test patterns cannot be repeated and since the fault coverage obtained would be different from one execution of the BIST sequence to the next.

The types of test patterns described above are not mutually exclusive. For example, pseudo-random test patterns may also be pseudo-exhaustive. In some BIST applications, multiple types of test patterns are used; pseudo-random test patterns may be used in

conjunction with deterministic test patterns. The subsequent sections address test pattern generation from the hardware standpoint, in the context of BIST, with emphasis on minimizing area overhead and maximizing performance. In just about all TPG implementations, a very important capability is initialization of the TPG to some known starting place in the test pattern sequence; unless otherwise specified, we will assume that a reset or preset capability is incorporated into all of the TPGs discussed in the subsequent sections.

4.11 Testing and Fault Simulation

The mechanics of testing, as illustrated in Figure 4.5, are similar at all levels of testing, including design verification [19, 23]. A set of input stimuli is applied to a circuit

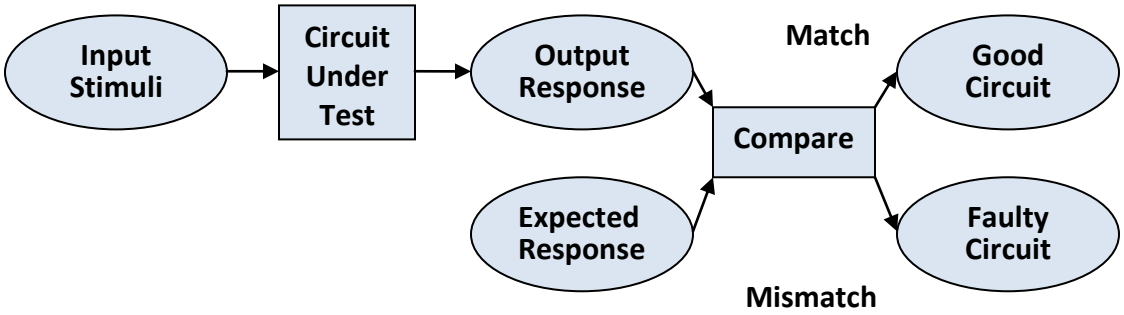


Figure 4.5: BIST Architecture [19].

and the output response of that circuit is compared to the known good output response, or expected response, to determine if the circuit is "good" or "faulty". During the design verification process, the input stimuli are the design verification vectors intended to determine whether the design functions properly, according to requirements and specifications.

The design verification vectors (often produced by test benches) are applied to the hardware description language (HDL) abstraction of the circuit in the simulation environment to produce the output responses of the design to the input stimuli. The comparison of the output response to the expected response is usually done by the designer (perhaps via the test bench) with the expected response being a mental collection of the designer's interpretation of the requirements and specifications. Alternative approaches include computer-aided design (CAD) based techniques such as formal verification or

automatic test pattern generation (ATPG) based methods [24]. A "faulty" circuit in this case would indicate that there is either a design error or a problem with the design verification vectors that prevented manipulation of the circuit in the way the designer had originally intended. A "good" circuit, on the other hand, does not mean that the design is free of errors. This requires sufficient design verification be performed to obtain a high level of confidence that the design is error free (or at least that there is a relatively low probability of any remaining design error). Obtaining the "warm, fuzzy feeling" that sufficient design verification has been obtained is a difficult step for the designer. While the ultimate determination is whether or not the design works in the system, fault simulation can provide a rough quantitative measure of the level of design verification much earlier in the design process. Fault simulation also provides valuable information on portions of the design that need further design verification (as we shall discuss shortly).

The design verification vectors are often used as functional vectors during manufacturing testing. The test machine, also referred to as the automatic test equipment (ATE), applies the test vectors to the fabricated circuit and compares the output responses to the expected responses obtained from the design verification simulation environment for the fault-free (and hopefully, design error-free) circuit. A "faulty" circuit is now considered to be a circuit with manufacturing defects. In the case of a VLSI device, the chip may be discarded or it may be investigated by failure mode analysis (FMA) for yield enhancement [25]. In the case of a PCB, FMA may be performed for yield enhancement or the PCB may undergo further testing for fault location and repair. A "good" circuit is assumed to be defect-free, but this assumption is only as good as the quality of the tests being applied to the manufactured design. Once again, fault simulation provides a quantitative measure of the quality of the set of tests.

Fault simulation also follows the basic testing flow illustrated in Figure 4.4 with one exception. In addition to the input test vectors, a list of faults that are to be emulated in the CUT is also applied to the fault simulation environment [22], The fault simulator emulates each fault (or set of faults) in the list and applies the test vectors. Each output response is then compared to the associated expected response obtained from a fault-free circuit simulation. If a mismatch is obtained for any test vector, the faulty circuit has produced an erroneous response compared to the fault-free circuit and the fault (or set of faults) is

considered to be *detected.* As soon as a fault is detected, most fault simulators will stop simulating that fault (this is often referred to as *fault dropping*) and restart simulation at the beginning of the set of test vectors with the next fault in the list being emulated. If there are no mismatches in the output response for the complete set of test vectors, then the fault is *undetected.* At this point, the next fault (or set of faults) is emulated in the CUT and the simulation restarts at the beginning of the set of test vectors. Once the fault simulation is complete (all faults have been emulated), the fault coverage can be determined for the set of test vectors. The fault coverage, *FC*, is a quantitative measure of the effectiveness of the set of test vectors in detecting faults, and in its most basic form is given by:

$$\mathrm{FC} = \frac{D}{T} \tag{4.1}$$

Where *D* is the number of detected faults and *T* is the total number of faults in the fault list. Obviously 100% fault coverage is desired but in practice, fault coverage in excess of 95% is considered by some companies to be acceptable for manufacturing testing. For design verification, the fault coverage can not only give the designer a rough quantitative measure of how well the design has been exercised, but the undetected fault list can provide valuable information on those subcircuits that have not been exercised as thoroughly as other subcircuits. For example, the fault naming convention in most fault simulators maintains the hierarchical naming convention of the design. Therefore, the designer can quickly scan the undetected fault list to determine the subcircuit(s) with the most undetected faults and target those subcircuit(s) for additional design verification. While this is a valuable tool for the designer in obtaining feedback on the quality of design verification, it should be remembered that fault coverage only gives the designer a measure of how well the circuit has been exercised and not a guarantee the circuit is free of design errors.

Logic and timing simulation during the design verification process can take considerable time for large circuits. Fault simulation time will take even longer since each fault or set of faults must be simulated. Fault simulation is not complete until all faults in the fault list have been emulated and simulated with the set of test vectors. Therefore, fault simulation time is not only a function of the size of the circuit and the number of test

vectors to be simulated, but also a function of the number of faults to be simulated and the type of fault simulator used. As a result, fault simulation time can be a major concern during the design and test development processes. There are a number of techniques used to improve fault simulation time. For example, like most test machines that stop as soon as a faulty device is detected (often referred to as *tripon- first-failure*) and move on to the next chip to be tested, most fault simulators employ fault dropping to stop simulating a fault once it is detected and continue to simulate only the undetected faults. The type of fault simulator used also makes a big difference in the fault simulation time [22]. For example, serial fault simulators are slow because they emulate only one fault in the circuit at a time. Parallel fault simulators significantly reduce the fault simulation time by taking advantage of the multiple bit computer words and the fact that simulation of a digital circuit is binary (0 and 1 logic values). As a result, each bit in a computer word used for the fault simulation can represent a different faulty circuit. In a 32-bit machine, a parallel fault simulator can emulate either 31 or 32 faults in parallel; 31 faults in simulators that emulate the fault-free circuit in one of the bit positions and 32 faults in simulators that compare the faulty circuit response with the stored results of a previous fault-free circuit simulation [26]. There are other types of fault simulators, including deductive and concurrent fault simulators, as well as hardware accelerators that also provide significant improvements in fault simulation time. The number of faults to be simulated is a major component in fault simulation time and is a function of the size of the CUT as well as the fault models.

4.12 BIST Response Compaction

During BIST, it is necessary to reduce the enormous number of circuit responses to a manageable size that can be stored on the chip. For example, consider a circuit testing with hardware pattern generator that computes 5 million test patterns during testing, and where there are 200 Pos. The total number of response will be 5, 000, 00 × 200 = 1, 000, 000, 00 bits. This amount of information cannot be economically stored on the CUT, so the circuit response must be compacted. Compaction is performed by merging all logic (across module boundaries) within a logic equivalence class and processing the resulting functions with classical logic synthesis and optimization tools.

4.12.1 Definitions

1. Aliasing

During circuit response compaction, because of the information loss, it is possible that a signature of a bad machine may the good machine signature, which is called aliasing. In such cases, a failing circuit will pass the testing process.

2. Compaction

A method of drastically reducing the number of bits in the original circuit response during testing in which some information is lost.

3. Compression

A method of reducing the number of bits in the original circuits response during testing in which no information is lost, so the original output sequence can be fully regenerated from the compressed sequence.

4. Signature

A statistical property of a circuit, usually a number computed for a circuit from its response during testing, with the property that faults in the circuit usually cause the signature to deviate from that of the good machine. To discourage their unauthorized copying, FPGA configurations can be watermarked with a special signature based on the circuit designer and the purchasing customer.

5. Transition Count Response Compaction

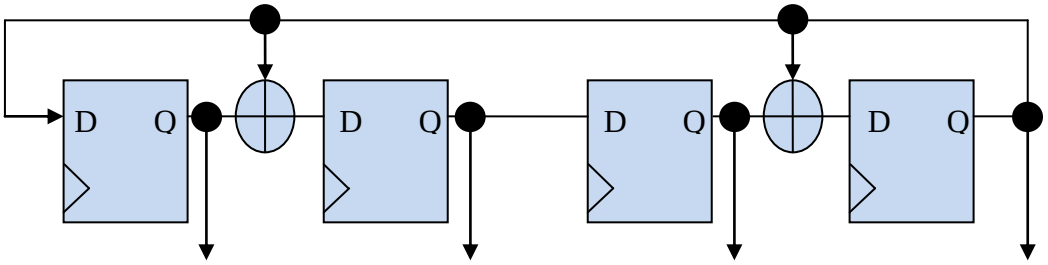
A method response compaction in which the number of transitions from 0 to 1 and 1 to 0 at circuit Pos are counted to create a testing signature.

6. Signature Analysis

A method of circuit response compaction during testing, whereby the entire good circuit response is compacted into a good machine signature. The actual circuit signature is generated during the testing process on the CUT, and then compared with the good machine signature to determine whether the CUT is faulty.

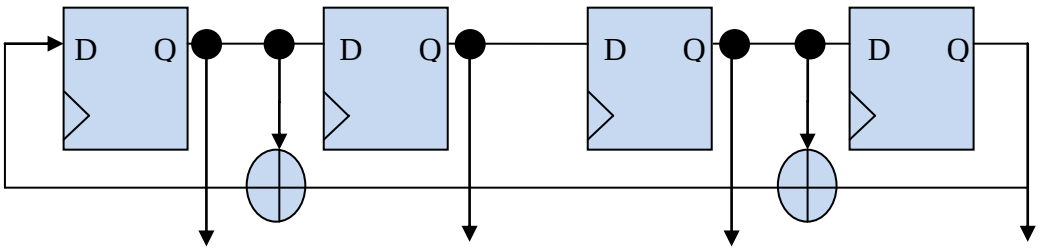
4.13 Linear Feedback Shift Registers

The LFSR is one of the most frequently used TPG implementations in BIST applications [27]. One reason for this is that an LFSR is more area efficient than a counter, requiring less combinational logic per flip-flop. There are two basic types of LFSR implementations, the internal feedback and external feedback LFSRs illustrated in Figure 4.6, internal feedback LFSRs are sometimes referred to as Type 1 LFSRs while external feedback LFSRs are referred to as type 2 LFSRs [27]. One problem with this terminology is remembering which is which, but a more serious problem is that the designations are often reversed in some papers where the external feedback LFSR is referred to as Type 1 and the internal feedback LFSR is referred to as Type 2 [28]. The internal and external feedback designations are unambiguous, avoid confusion and, more importantly, avoid



Binary test patterns

(a) Internal feedback LFSR.



Binary test patterns

(b) External feedback LFSR.

Figure 4.6: LFSR Implementations [19].

design errors. The external feedback LFSR in Figure 4.6(b) best illustrates the origin of the name of the circuit: a shift register with feedback paths that are linearly combined via the exclusive-OR gates. Internal and external feedback LFSRs are duals of each other [27].

Both implementations require the same amount of logic in terms of exclusive- OR gates and flip-flops. In the external feedback LFSR in Figure 4.6(b), there are two exclusive-OR gates in the worst case path from the output of the last flip-flop in the shift register to the input of the first flip-flop in the shift register. On the other hand, the internal feedback LFSR has, at most, one exclusive-OR gate in any path between flip-flops. Therefore, the internal feedback LFSR provides the implementation with the highest maximum operating frequency for use in high performance applications. The main advantage of external feedback LFSRs is the uniformity of the shift register; hence, there are some applications where external feedback is preferred. The sequence of test patterns produced at the outputs of the flip-flops in an LFSR is a function of the placement of the exclusive-OR gates in the feedback network.

4.14 Implementation of Double Precision Square Root using BIST

This block represents the implementation of Double Precision Floating Point Square root using BIST.

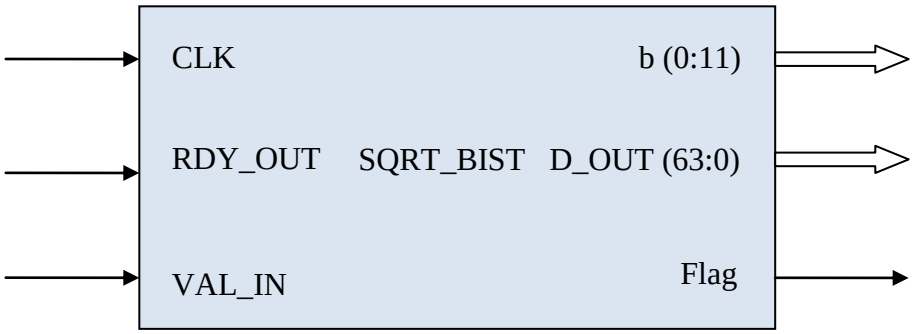


Figure 4.7: Pin Diagram of of Square Root using BIST.

There are six ports namely, data clock (CLK), ready to accept output (RDY_OUT), valid input value (VAL_IN), exponent output (b),data output (D_OUT), flag output(Flag), all signals are active high.

The representation has six ports:

1) CLK: It is the input clock.
2) RDY_OUT: It is the output wire, this input will give high when asserted, and downstream logic is ready to accept output of module.

3) VAL_IN: It is the input value, the module will only accept input when RDY_IN is low, then VAL_IN should remain asserted until RDY_IN goes high. VAL_IN and RDY_IN go high. VAL_IN and RDY_IN must both be asserted for on clock cycle for computation to begin.
4) b (0:11): It is the output of the double precision floating point square root exponent.
5) D_OUT (63:0): It is the output register of double precision floating point square root.
6) Flag: It is the output register, this output will give high when output response equal to expected response otherwise flag will zero.

CHAPTER 5
RESULTS AND CONCLUSION

5.1 Results

5.1.1 Simulation Result for Double Precision floating point Square root

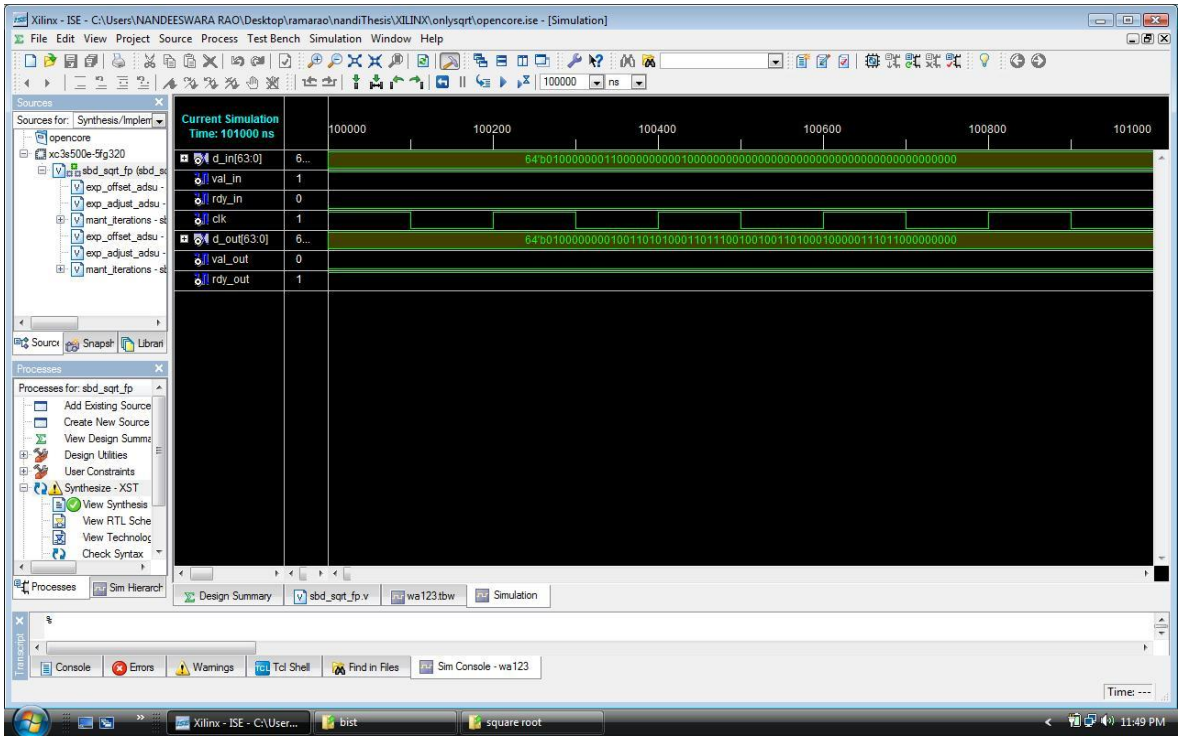


Figure 5.1: Output waveform of Double Precision floating point Square root.

Description:

Input: D_IN (63:0) = 128.12

0	10000000110	0000000010000000000000000000000000000000000000000000

Output: D_OUT (63:0) = 11.319…….

0	10000000010	0110101000111000001001001101000100000111011000000000

There are seven ports namely, data input (D_IN), clock (CLK), ready to accept output (RDY_OUT), valid input value (VAL_IN), data output (D_OUT), ready to accept input (RDY_IN), valid output value (VAL_OUT), all signals are active high.

These inputs and outputs are in IEEE format having 1-bit for sign, 11-bits for exponent and 52-bits for mantissa. The simulated results shown above are as explained Figure 5.1 and each port are as explained in Chapter 2.

5.1.2 Simulation Result for LCD Interfacing of Double Precision Floating Point Square Root

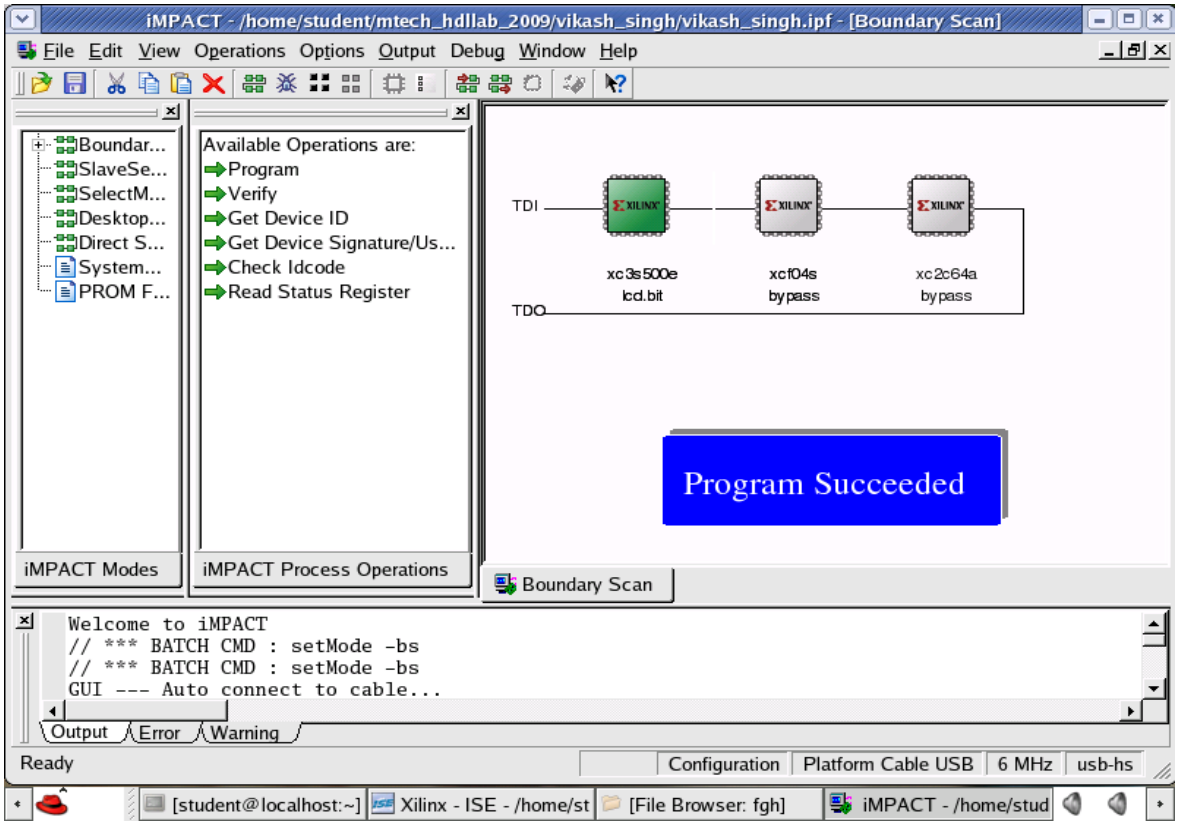


Figure 5.2: Assign New Configuration File on Spartan 3E kit.

As shown in figure 5.2, when programming is complete, the "Program Succeeded" message is displayed. The simulated results shown above are as explained Figure 5.2 and procedure of Interfacing on FPGA are as explained in Chapter 4.

As shown in figure 5.3, it shows that the output of the Double Precision Floating Point Square Root. Input is fixed in the programming of Double precision Floating Point Square Root unit then it generated output this output displayed on LCD, this programming required separated LCD interfacing programming to display an output on LCD. The Spartan® - 3E FPGA Starter Kit board prominently features a 2 – line by 16-characterliquid crystal display (LCD). Last 3 hexadimal enough to see the result of

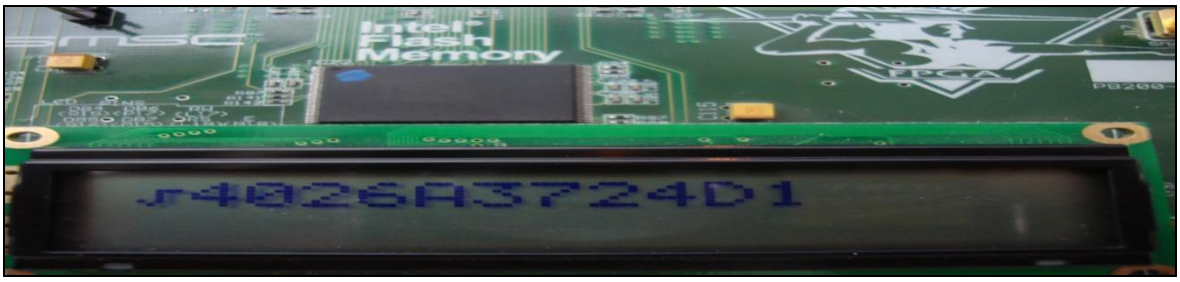


Figure 5.3: Display the output on LCD on Spartan 3E kit.

Double Precision Floating Point Square root, so output displayed the last 11 Hexadecimal (D_OUT (63:20)) on LCD.

Output: D_OUT (63:20) = 11.319……. $= \sqrt{}$ 4026A3724D1

5.1.3 Simulation Result for KEYBOARD Interfacing of Double Precision Floating Point Square Root

As shown in figure 5.4, when Keyboard Interfacing programming is completed,

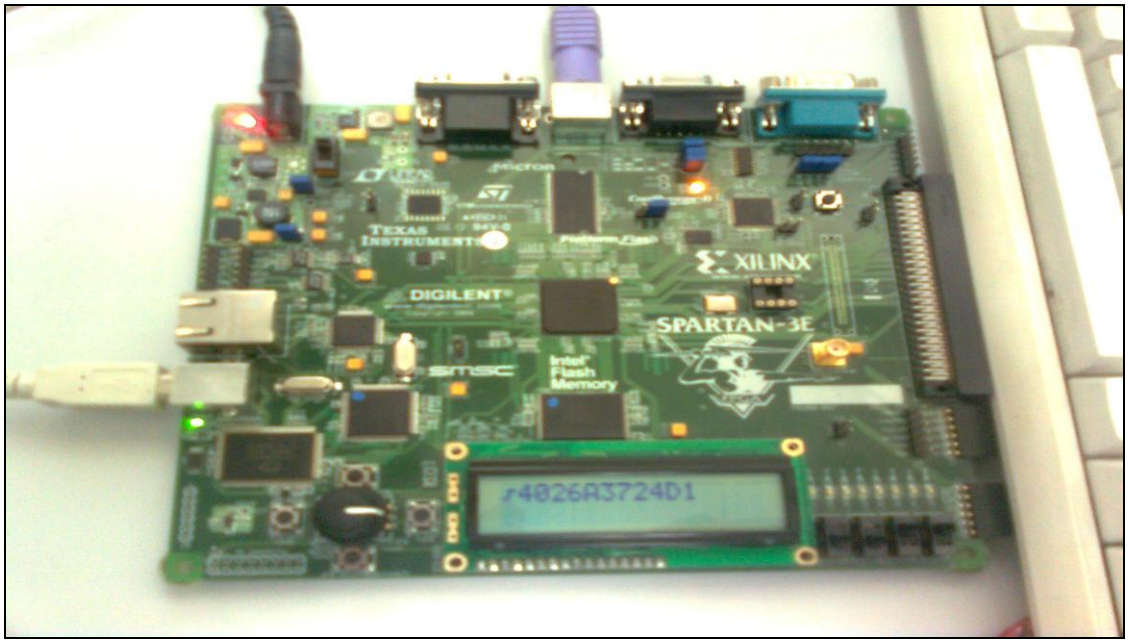


Figure 5.4: Keyboard Interfacing of Double Precision floating point Square Root Output displayed on LCD.

displayed the last 11 Hexadecimal (D_OUT (63:20)) on LCD.

5.1.4 Simulation Result for Double Precision floating point Square Root using BIST

There are six ports namely, data clock (CLK), ready to accept output (RDY_OUT), valid input value (VAL_IN), exponent output (b),data output (D_OUT), flag output(Flag), all signals are active high.

Output: Flag = '1'

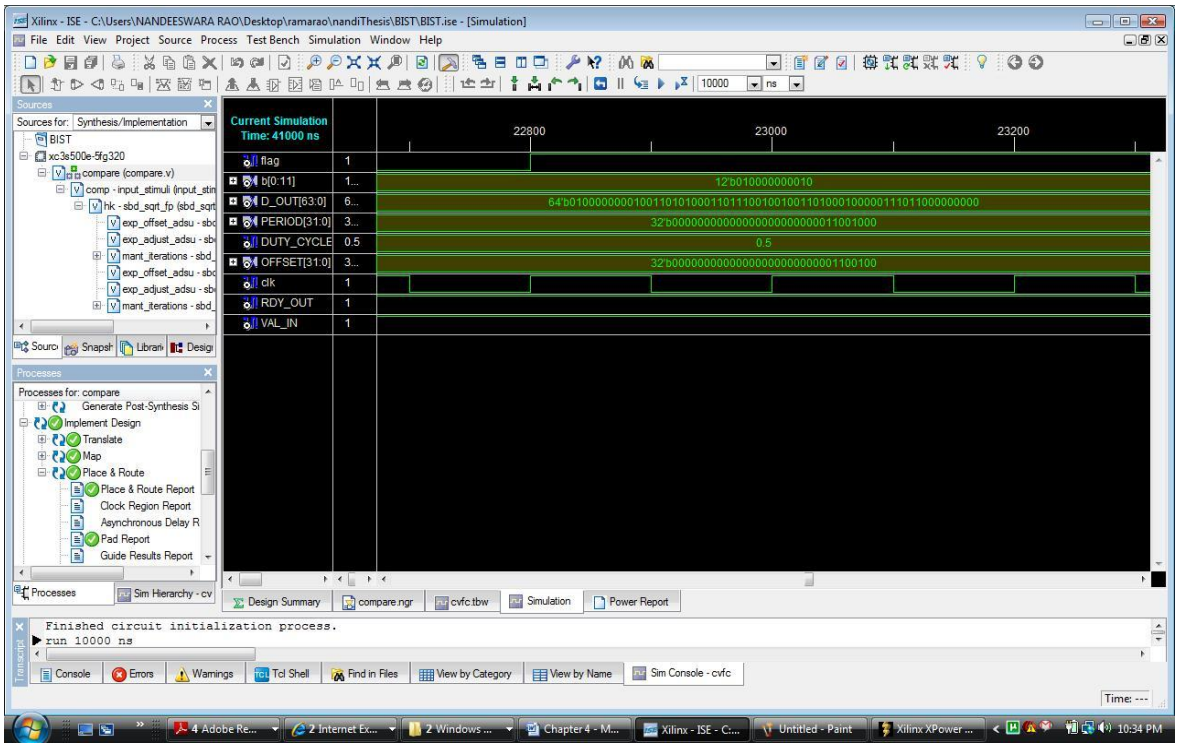


Figure 5.5: Output waveform of Double Precision floating point Square Root using BIST.

The output result of Double Precision floating point Square Root is fixed in BIST using verilog code[16], this output is compared with the output response, if output response equals expected response, flag will be high otherwise flag will be zero.

The simulated results shown above are as explained Figure 5.5 and each port are as explained in Chapter 4.

5.2 Experimental Results

The square root units were modeled in VERILOG[16], simulated in ModelSim 6.1, Synthesized using Xilinx ISE 9.1.Table 5.5 shows the implementation results of the units on the Spartan 3E XC3S500E chip.

5.2.1 Synthesis Report

The table 5.1 shows the Synthesis Results of Square Root and table 5.2 shows the Synthesis Results LCD Interfacing. The Square root Characteristics may be acceptable in some applications like typical desktop applications. However, high performance is crucial for many applications such as scientific computation, CAD tools

	Square Root
Clock Period(ns)	9.671
Clock Frequency(MHz)	103.405
Latency(ns)	9.671
Throughput(MFLOPS)	60.16
No. of Slices	325
Throughput/Area(KFLOPS/Slice)	185.10
No. of 4 I/P LUT's	475(5%)
Flip Flops	411
X-Power(mw)	81
Number of bonded IOBs	133 out of 232 (57%)
Total Equivalent Gate Count	6,960
Peak Memory usage(MB)	185

Table 5.1: Synthesis Results of Square Root

	LCD Interfacing
Clock Period(ns)	10.014
Clock Frequency(MHz)	99.860
Latency(ns)	10.014
Throughput(MFLOPS)	58.10
No. of Slices	263
Throughput/Area(KFLOPS/Slice)	220.91
No. of 4 I/P LUT's	477(5%)
Flip Flops	301
X-Power(mw)	83
Number of bonded IOBs	11 out of 232 (4%)
Total Equivalent Gate Count	6,351
Peak Memory usage(MB)	152

Table 5.2: Synthesis Results of LCD interfacing.

and 3D graphics rendering which have a higher frequency of floating-point square root operations.

	KEYBOARD Interfacing
Clock Period(ns)	9.671
Clock Frequency(MHz)	103.405
Latency(ns)	9.671
Throughput(MFLOPS)	60.16
No. of Slices	299
Throughput/Area(KFLOPS/Slice)	201.20
No. of 4 I/P LUT's	540(5%)
Flip Flops	335
X-Power(mw)	83
Number of bonded IOBs	15 out of 232 (6%)
Total Equivalent Gate Count	6,998
Peak Memory usage(MB)	142

Table 5.3: Synthesis Results of Keyboard Interfacing.

	BIST
Clock Period(ns)	9.671
Clock Frequency(MHz)	103.405
Latency(ns)	9.671
Throughput(MFLOPS)	60.16
No. of Slices	222
Throughput/Area(KFLOPS/Slice)	270.99
No. of 4 I/P LUT's	342(5%)
Flip Flops	291
X-Power(mw)	81
Number of bonded IOBs	79 out of 232 (34%)
Total Equivalent Gate Count	5,012
Peak Memory usage(MB)	183

Table 5.4: Synthesis Results of BIST.

5.2.2 Synthesis Results

This implementation is significantly low in area overhead measured in terms of LUT s shown in Table 5.5 although their throughput around the single MFLOP mark. In contrast, the double precision floating point square root unit implementations display Latency is significantly low and throughput is high. This shows that the square root implementation is highly efficient since their gain in performance is offset by a relatively low cost in area overhead.

Unit	**Family and Device**	**Clock Period (ns)**	**Clock Frequency (MHz)**	**Latency (ns)**	**Throughput (MFLOPS)**	**Slices**	**LUTs**	**Flip-Flops**
Square root Obtained results	*Spartan 3E and XC3S500E*	9.671	103.405	9.671	60.16	325	477(5%)	411
Square root Simulated results	*Vertex and XCV1000*	15.18	65.876	15.18	38.32	321	569(2%)	422

Table 5.5: Synthesis Results of the Square Root unit.

Square root[1]	*Vertex and XCV1000*	17.06	58.60	17.06	0.94	386	772(3%)	437

Table 5.6: Synthesis Results of the Square Root unit[1].

Calculation of throughput:

The Square Root implementation is presented two important criteria in Table 5.5,these are (Latency and Throughput) used to examine the performance of a system. Latency is the time required to complete one task, and throughput is the number of tasks that can be completed per unit time. The two are related but are not identical. Double precision floating point square root unit, sets the bit length Parameter to 64.The algorithm used is iterative and will take 110 clock cycles and produce a 64-bit output, clock frequency is 103.405 MHz (1/103.405 MHz) × 110 clocks Cycle = 1.06s (1.06 micro-seconds = time to process one block) 1s / 1.06 μs = how many blocks could be processed in 1 second = 940,045.4545 and each block is 64 bits.

64 bits × 940,045.4545s = 60,162,909.09 total bits.

Therefore throughput = 60.16 Mbits per second.

Calculation of Throughput per slice:

Hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve the routing congestion. This results in an increase in the number of CLB slices without a corresponding increase in logic gates. To achieve a more accurate measure of chip utilization, CLB slice count was chosen as the most reliable area measurement. Therefore, to measure the hardware resource cost associated with an implementation's resultant throughput, the Throughput per Slice (TPS) metric is used.

TPS = (Encryption Rate) / (# CLB Slices Used)

Here the encryption rate is 60.16Mb/ sec. The number of slices used is 325.

Therefore the Throughput per Slice (TPS) is calculated as

Throughput per Slice (TPS) = (60.16M b/sec) / 325

= 185.10 **Kb/sec/slice**

5.3 Conclusion

This thesis presents the iterative designs of double precision floating point square root unit. The whole design was captured entirely in VERILOG language using a bottom-up design and verification methodology. The implementation of this unit was achieved without taking advantages of any advanced architectural features available in high end FPGA chips or using any pre-designed architecture-specific arithmetic cores. The proposed VLSI implementation of the Square root reduces the covered latency (9.671ns), area and achieves a data throughput up to 60.16Mb/ sec.This experiments reveal that this design can produce performances that are comparable to, and in some case higher than, non-iterative designs based on number representations of higher radices. The iterative implementation requires less area, and the sequential implementation achieves high performance at a reasonable cost.

FUTURE ASPECTS

This project can put to various applications, which are described below:

The parts of Double precision floating point Square Root unit can be used to make dedicated hardware. Square Root unit have a complete arithmetic unit for any microprocessor for high speed and low power operation. When the program is executed on system, it will surely give the result but consume more power and time than a dedicated hardware. In case of dedicated hardware, instructions are directly implemented on hardware and they consume less power and are faster than microprocessor.

The model can also be implemented in depth to construct trigonometric like sine, cosine, etc. and logarithmic functions which can be utilized in scientific calculations and the main application is the Square Root floating-point unit (FPU) contained in the processor, which was formerly a separate external math coprocessor. When Intel introduced the 486DX, it included a built-in math coprocessor, and every processor built by Intel (and AMD and Cyrix, for that matter) since then includes a math coprocessor. Clear Speed, a small fables design firm, has developed the 64-bit CSX600, an extremely parallel math processor designed to be used in conjunction with high-end x86 CPUs to improve the performance of scientific applications. The processor contains 96 processing units running on a 250MHz clock, which allows it to dissipate only 5 watts under normal conditions and sustain a 25GFLOPS (50GFLOPS peak) performance level.

If designer wants to achieve higher performance then double precision floating point square root unit is shifted to "*extended format*" (128 bits).The Intel microprocessors implement arithmetic with the extended format in hardware, using 80-bit registers, with 1-bit for the sign, 15-bits for the exponent, and 64-bits for the significand. The leading bit of a normalized or subnormal number is not hidden as it is in the single and double formats, but is explicitly stored. Otherwise, the format is much the same as single and double precision. Other machines, such as the Sparc microprocessor used in Sun workstations, implement extended precision arithmetic in software using 128-bits. Consequently, computations with the extended format are fast on an Intel microprocessor but relatively slow on a Sparc. These can be possible to implement on FPGA device, such as the ***Xilinx,*** Spartan3E or Vertex.

REFERENCES

[1] A. J. Thakkar and A. Ejnioui, “Design and Implementation of Double Precision Floating Point Division and Square Root on FPGAs,” 0-7803-9546-8/2006 IEEE.

[2] IEEE Standard Board, "IEEE Standard for Binary Floating-point Arithmetic," The Institute for Electrical and Electronics Engineers, 1985.

[3] B. Parhami “Computer Arithmetic Algorithms and Hardware Designs,” Oxford University press, 2000.

[4] Y. Li and W. Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs," IEEE Symposium on Field Programmable Custom Computing Machines, 1997.

[5] N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machine (FCCM95), IEEE Computer Society Press, 1995. pp 155-162.

[6] L. Louca, T. A. Cook, W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM96), IEEE Computer Society Press, 1996. pp107-116.

[7] B. Koenemann, J. Mucha, and G. Zwiehoff, “Built-In Test for Complexity Digital Integrated Circuits,” IEEE Journal of Solid-State Circuits, vol. SC-15, no.3, pp 315-318, June 1980.

[8] A. K. Chandra, L. T. Kou, G. Markowsky, and S. Zakas, “On sets of Boolean n-

vectors with all K-projections Surjective," Acta Informatica, vol. 20, no.1, pp. 103-111, oct. 1983.

[9] M. L. Overton, "Numerical Computing with IEEE Floating Point Arithmetic," Society for Industrial and Applied Mathematics, 2001.

[10] A. N. D. Zamfirescus, "Floating Point Types for Synthesis", Alternative system concepts, Inc. 644 Emerson stuite 10, Palo Alto, CA USA.

[11] C. Dufaza and G. Cambon, "LFSR-Based deterministic and Pseudo-random Test Pattern Generator Structures," in proc. Of the European test conf., April 1991, pp 27-34.

[12] C. L. Chen, "Exhaustiven Test Pattern Generation Using Cyclic Codes, "IEEE Transactions on Computers," vol. C-37, no. 2, pp. 225-228, Feb. 1988.

[13] S. Hauck and A. DeHon, "Reconfigurable computing the theory and practice of FPGA-based computation, "Morgan Kaufmann, 2008.

[14] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," Foundations and Trends in Electronic Design Automation, now, 2006.

[15] J. Rajski and J. Tyszer, "Arithmetic Built-in Self-Test for Embedded Systems," Published by Prentice Hall PTR Upper Saddle River, New Jersey, 1998.

[16] J.BHASKER "Verilog HDL Synthesis A Practical Primer B. Parhami," Star Galaxy, 1998.

[17] M. D. Ercegovac, T. Lang, "Digital Arithmetic," Morgan Kaufmann, 2004.

[18] J.G.Udell Jr. and E.J. McCluskey,"Partial Hardware Partitioning:A New Pseudo-Exhaustive Test Implementation," in Proc. of the international Test Conf., Sept. 1988, pp

1000.

[19] C. E. Stroud, “A Designer’s Guide to Built-In Self-Test,” Kluwer Academic, 2002.

[20] E. McCluskey, “Verification Testing - A Pseudoexhaustive Test Technique,” IEEE Trans. on Computers, Vol. 33, No. 6, pp. 541-546, June 1984.

[21] H. Klug, “Microprocessor Testing by Instruction Sequences Derived from Random Patterns,” Proc. IEEE International Test Conf., 1987, pp. 73-80.

[22] M. Abramovici, M. Breuer and A. Friedman, “Digital Systems Testing and Testable Design”, Piscataway, New Jersey: IEEE Press, 1994.

[23] P.H. Bardell, W.H. McAnney, “Pseudorandom arrays for built-in tests,” IEEE Trans. Comput., vol. C-35, No. 7, pp. 653-658, 1986.

[24] D. Clarke and I. Lee, “Testing-Based Analysis of Real-Time System Models,” Proc. IEEE International Test Conf., 1996, pp. 894-903.

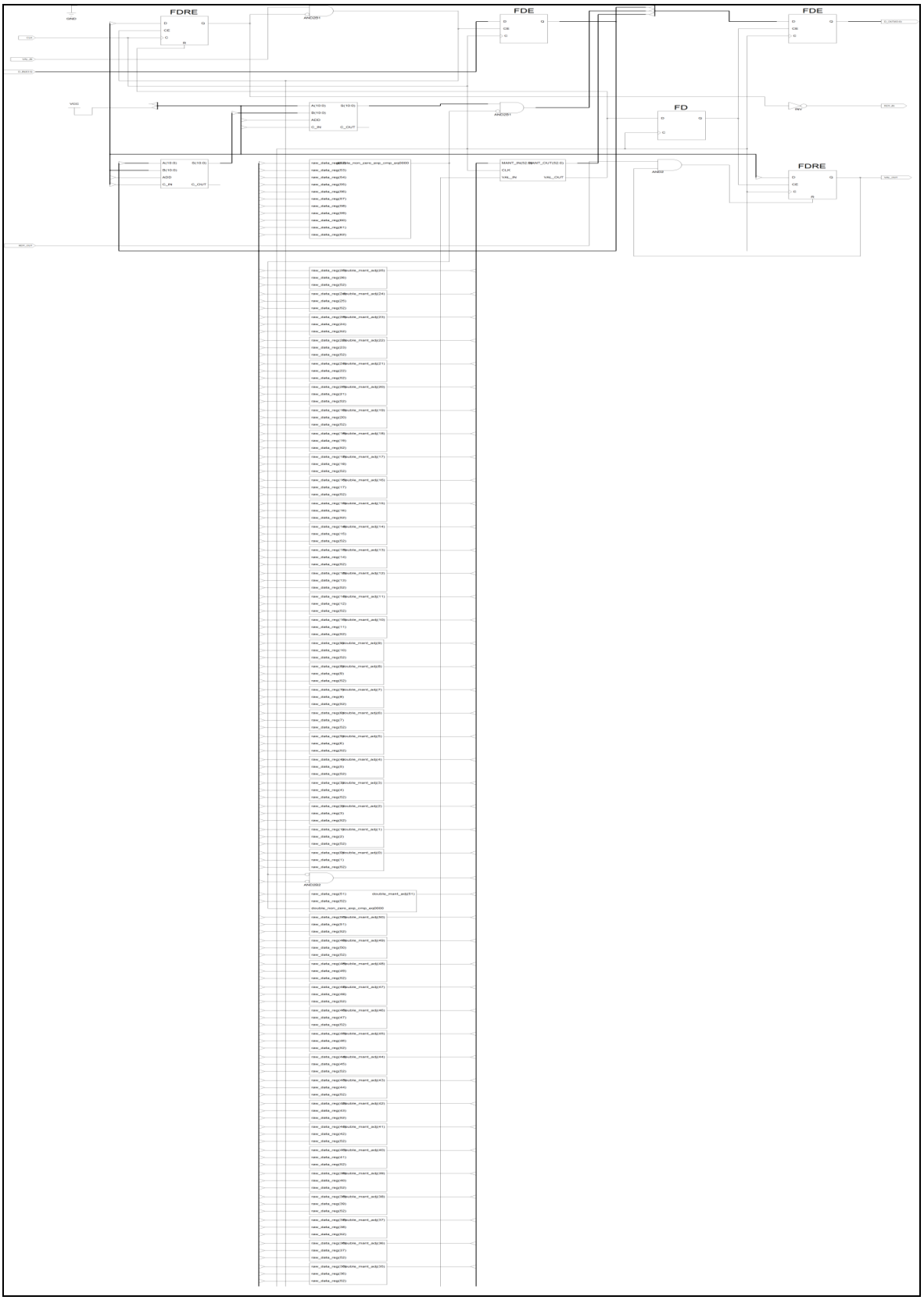
[25] M. Bushnell and V. Agrawal, “Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits,” Boston: Kluwer Academic Publishers, 2000.

[26] C. Stroud and C. Ryan, “Multiple Fault Simulation with Random and Cluster Fault Injection Capabilities,” Proc. IEEE International ASIC Conf., 1995, pp. 218-221.

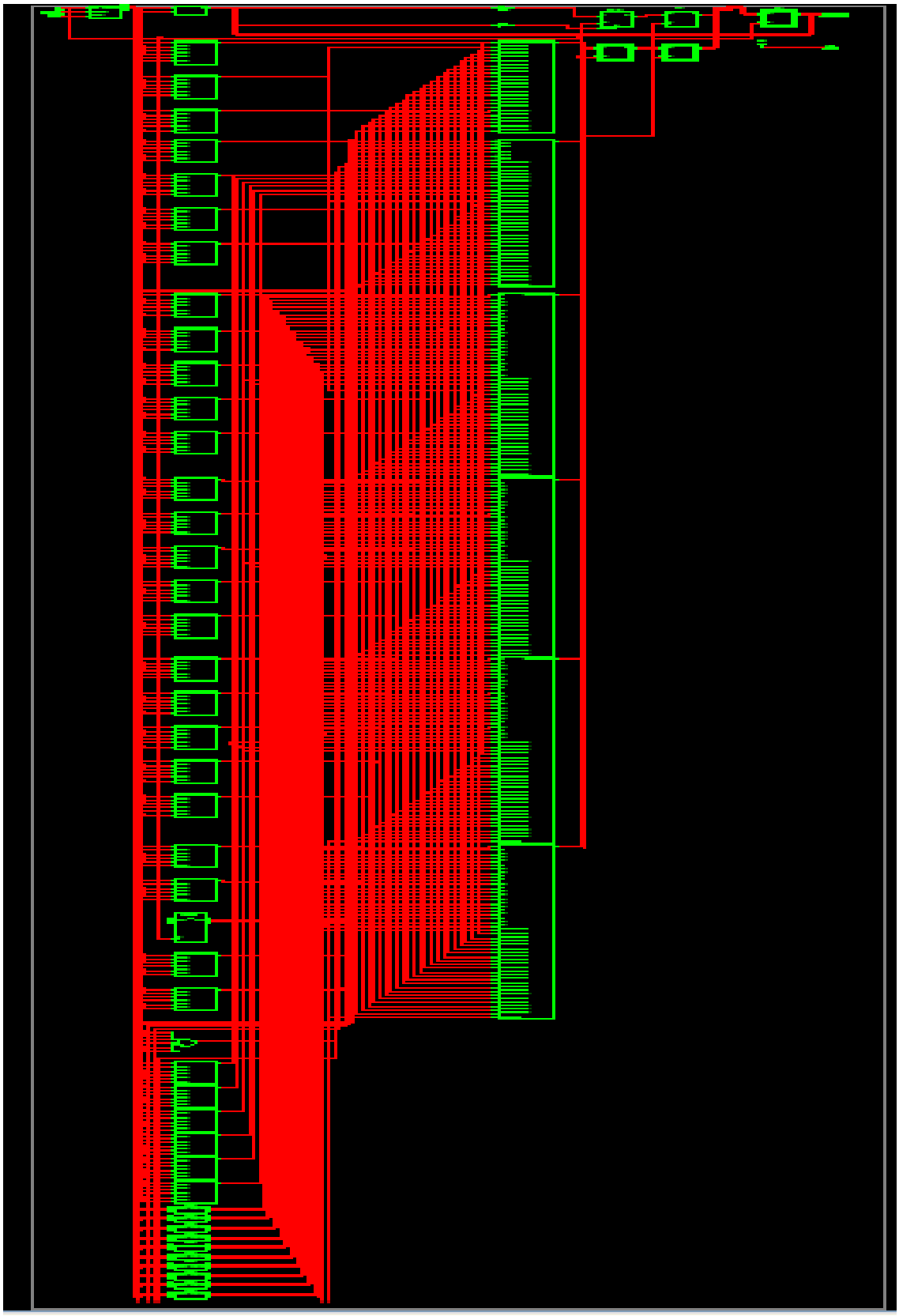
[27] P. Bardell, W. McAnney and J. Savir, “Built-In Self-Test for VLSI: Pseudorandom Sequences,” Somerset, New Jersey: John Wiley & Sons, 1987.

[28] J. Rajski and J. Tyszer, “Design of Phase Shifters for BIST Applications,” Proc. IEEE VLSI Test Symp., 1998, pp. 218-224.

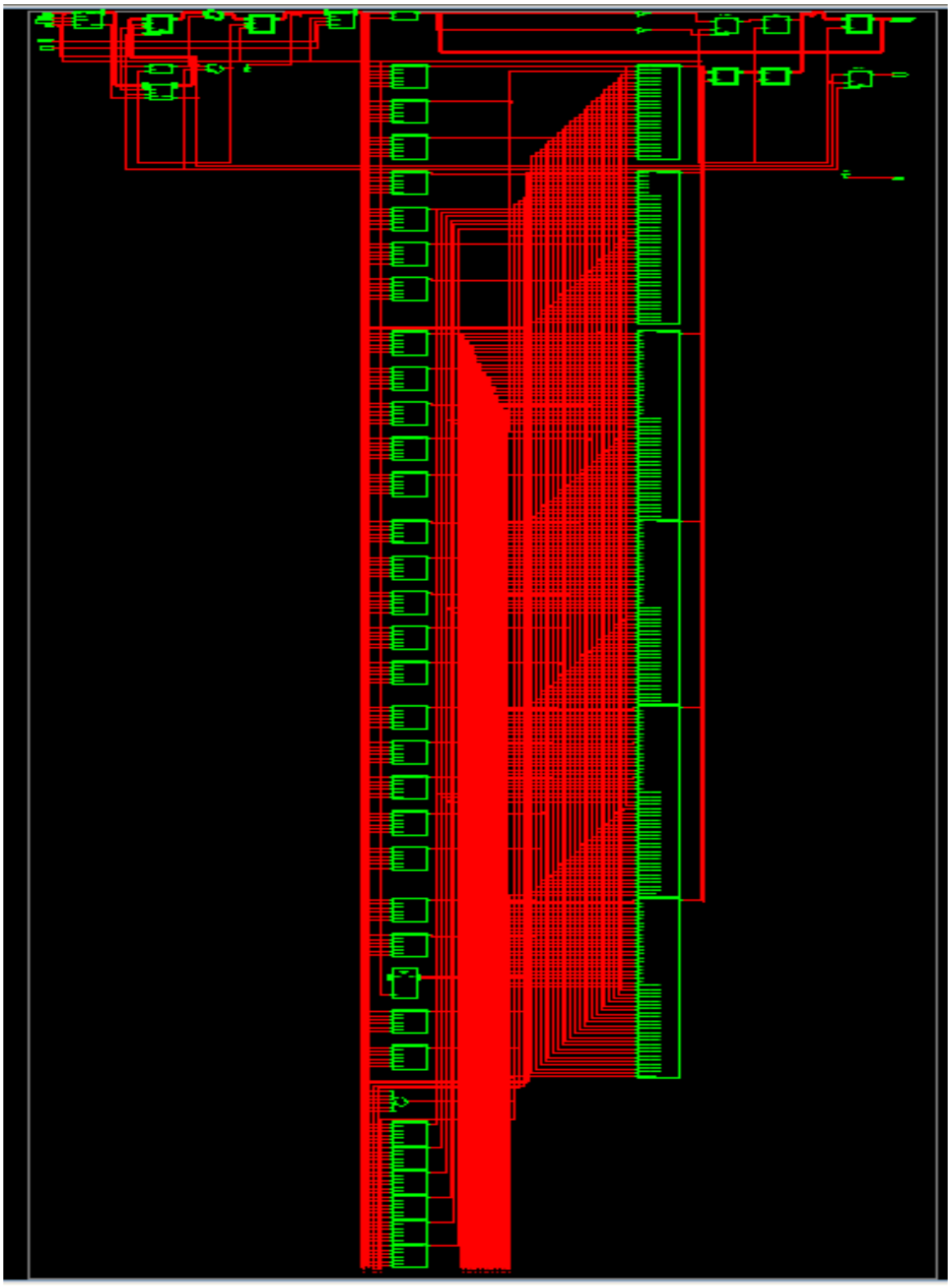
APPENDIX



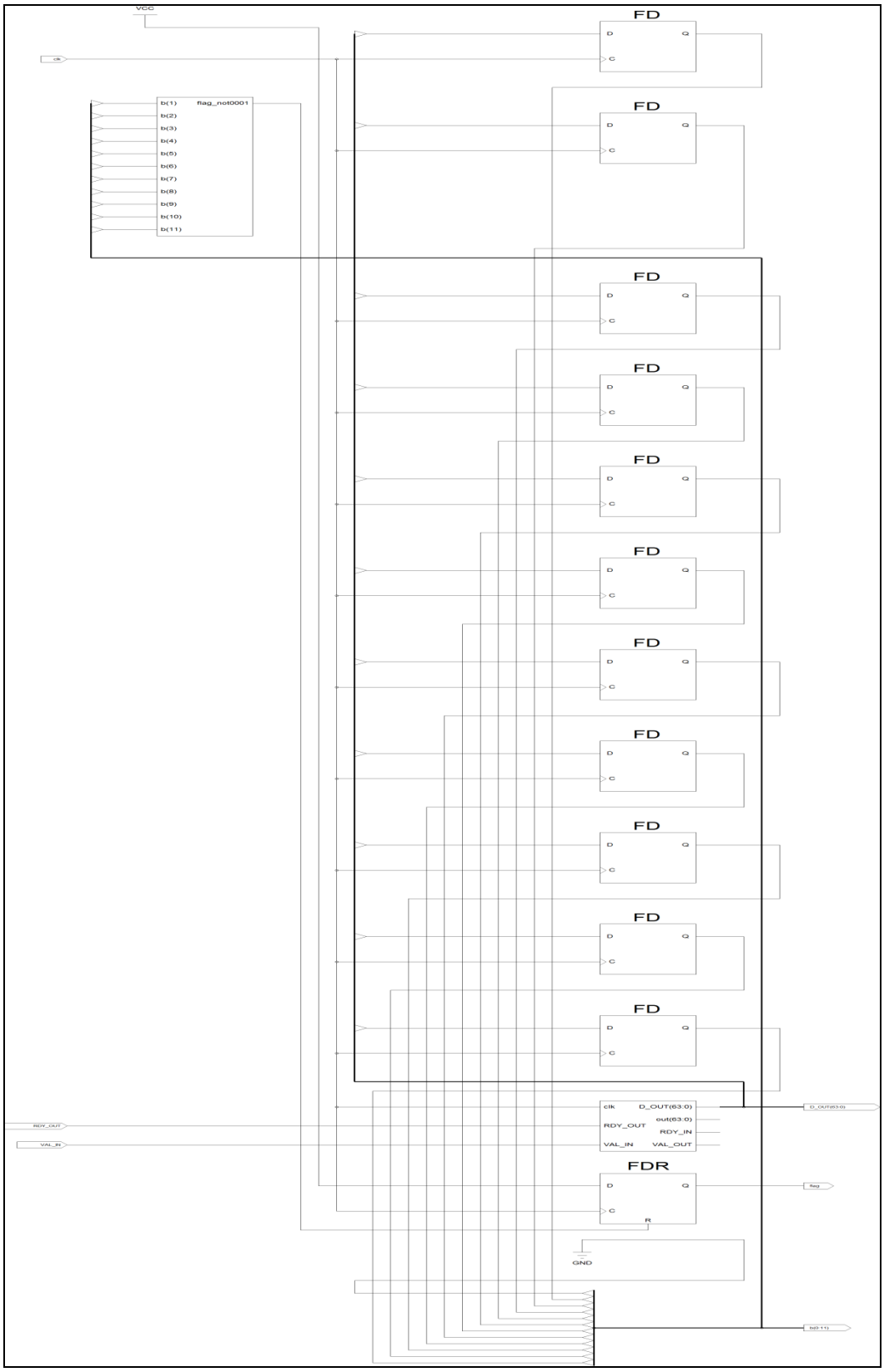
Schematic View of Double precision floating point Square Root unit



Schematic View of LCD Interfacing of Double Precision Floating Point SQRT



Schematic View of KEYBOARD Interfacing of Double Precision Floating Point SQRT



Schematic View of Double Precision floating point Square Root using BIST