Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

6-2017

Pipelined Implementation of a Fixed-Point Square Root Core Using Non-Restoring and Restoring Algorithm

Vyoma Sharma
sg5232@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Sharma, Vyoma, "Pipelined Implementation of a Fixed-Point Square Root Core Using Non-Restoring and Restoring Algorithm" (2017). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Pipelined Implementation of a Fixed-Point Square Root Core Using Non-Restoring and Restoring Algorithm

by

Vyoma Sharma

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

____

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

____

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
JUNE 2017

I dedicate this work to my family and friends for their love, support and inspiration during my thesis.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Vyoma Sharma

June, 2017

Acknowledgements

I want to thank my advisor Mark A. Indovina for his support, guidance, feedback and encouragement which helped in the successful completion of my graduate research.

Abstract

Arithmetic Square Root is one of the most complex but nevertheless widely used operations in modern computing. A primary reason for the complexity is the irrational nature of the square root for non-perfect numbers and the iterative behavior required for square root computation. A typical RISC implementation of Square Root Computation can take anywhere from 200 - 300 cycles. If significant usage is encountered, this could result in an impact in run-time cost which would justify a direct hardware implementation that achieves the same result in as little as 20 clock cycles. Additionally, the implementation is pipelined to achieve even greater throughput compared to an instruction based implementation. The paper thus presents an efficient, pipelined implementation of a square root calculation core which implements a non-restoring algorithm of determining the square-root. The iteration count of the algorithm depends on the maximum size of the input and the desired resolution. A specific case of a 16-bit integer square root calculator with output resolution 0.001 is considered which requires a total of 18 iterations of the algorithm. In the implementation, each iteration is pipelined as a stage thereby resulting in an 18-stage pipelined square root computation core. The proposed algorithm utilizes standard arithmetic operations like addition, subtraction, shift and basic control statements to determine the output of each stage. The core is verified using SystemVerilog test-bench. The test-bench generates unconstrained random inputs stimulus and determines the expected value from the core device under test (DUT) by evaluating a Simulink generated model for the same stimulus. Functional coverage, implemented in the test-bench, determines reliability of the system and consequently

the duration of the test execution.

Contents

Contents **vi**

List of Figures **ix**

1 Introduction **1**

1.1 Concept of Operation 2

1.1.1 Stage I 3

1.1.2 Stage II 3

1.1.3 Stage III 4

1.2 Research Goals 5

1.3 Organization 5

2 Bibliographical Research **6**

3 System Architecture **13**

4 Detail Design **15**

4.1 Non-Restoring Algorithm 15

4.1.1 Implementation 16

4.2 Restoring Algorithm 17

4.2.1 Implementation 18

5 Verification Methodology and Results 20

5.1 Verification Methodology 20

5.2 Test Bench Design and its Components 21

5.2.1 Layered Test Bench Architecture 21

5.2.1.1 Interface 23

5.2.1.2 Stimulus 24

5.2.1.3 Scoreboard 24

5.2.1.4 Driver 25

5.2.1.5 Model 25

5.2.1.6 Monitor / Checker 25

5.2.1.7 Environment 26

6 Results 27

6.1 Non-Restoring 28

6.2 Restoring 29

7 Conclusion 32

References 34

I Source Code 38

I.1 Non-restoring Algorithm 38

I.1.1 Non-Restoring Algorithm Operation 38

I.1.2 Non-Restoring Algorithm with Pipeline architecture 39

I.1.3 Environment 56

I.1.4 Driver 57
I.1.5 Model 59
I.1.6 Monitor 64
I.1.7 Test Bench 66
I.2 Restoring Algorithm 67
I.2.1 Restoring Algorithm Operation 67
I.2.2 Non-Restoring Algorithm with Pipeline architecture 68
I.2.3 Environment 85
I.2.4 Driver 86
I.2.5 Model 89
I.2.6 Monitor 93
I.2.7 Test Bench 97

List of Figures

4.1 Non-Restoring Square Root Calculation for a single stage 16

4.2 Restoring Operation 18

4.3 Restoring Algorithm for a single stage iteration 18

5.1 Lower Layers of the Verification Test Bench 22

5.2 Test Bench Schematic for Verification 23

5.3 Interface Block Diagram 23

6.1 Coverage Report for Restoring/Non-Restoring Algorithm 27

6.2 Pipeline Data flow for Non-Restoring Algorithm 28

6.3 Data transition from Input to Output 29

6.4 Pipeline Data flow for Restoring Algorithm 30

6.5 Data transition from Input to Output 30

Chapter 1

Introduction

Square root is one of the fundamental arithmetic operations used by various signal and image processing algorithms apart from addition, subtraction, multiplication and division. Typical square root algorithms compute bits successively starting with the most significant bit and predicting the lower significant bit. The correctness of the predicted value is determined by comparing the square of the predicted number and the remainder determined for the particular stage. In this research paper, two similar algorithms to compute the square root of an integer number have been taken into consideration and extended to determine the fixed point fractional square root of the number. The paper characterizes the implementation of the two square root algorithm namely restoring algorithm and non-restoring algorithm. Non-restoring algorithm is the simpler and more efficient technique as it uses for square root implementation as it uses adders/subtractors and shift operations to determine the square root of the number. The characteristic difference between the algorithms, is their approach towards the selection of the square root digit and the partial remainder determination. The implemented system would utilize fixed-point nomenclature to represent the numbers as they require fewer pre-processing compared to floating-point numbers. Floating-point arithmetic is typically more complex, requiring more area, power and

consuming larger number of cycles for computation compared to their equivalent fixed-point arithmetic cores. However fixed-point arithmetic truncates bits lower than the significant range of the number. This becomes especially significant in case of smaller numbers whose relative value is drastically impacted by the truncated bits.

SystemVerilog is an extension of the traditional Verilog Hardware Description Language (HDL) which supports object oriented programming. The implemented test bench is constructed hierarchically with multiple modular components. A SystemVerilog test bench which performs unconstrained random verification of the functional core is utilized to verify the operation of the implemented system. The expected value of the system is determined by a model which in real-time determines the expected output of the ideal system for the same input stimulus vectors. The model was developed using Matlab and is exported using the standard Simulink HDL code library.

1.1 Concept of Operation

Considering the square root computation of a 16-bit integer number I[15:0], the result would be an 8-bit integer number B[7:0] and infinite fractional bits B[-1:$-\infty$]. The $b_i$refers to the ithe bit of the number B. The $B_n$refers to the sub-set of the B, $B[n{:}-\infty]$.

$$I = B^2$$

$$I = (b_7.2^7 + B_6)^2$$

1.1.1 Stage I

$$I = (b_7^2.2^{14} + 2.b_7.2^7.B_6. + B_6^2)$$

$$I = (b_7^2.2^{14} + 4.b_7.2^6.B_6 + B_6^2)$$

$$I - \llbracket b_7^2.2^{14} \rrbracket = (2.b_7.2^7.B_6) + B_6^2$$

The Left hand side term is the remainder of the operation of stage I, $RO1 = I - \llbracket b_7^2.2^{14} \rrbracket$ and the coefficient of 2.2^7B_6, is the partial quotient estimated at Stage 1, $QO1 = b_7$.

1.1.2 Stage II

$$RI2 = RO1 = \left(I - \llbracket b_7^2.2^{14} \rrbracket\right) = (4.b_7.2^6.(b_{6.}2^6 + B_5)). + (b_{6.}2^6 + B_5)^2)$$

$$RI2 = (4.b_7.b_6.2^{12}) + (4.b_7.B_5.2^6) + b_6^2.2^{12} + 2b_6B_5.2^6 + B_5^2$$

$$RI2 = \llbracket ((4.b_7 + b_6).b_6)\,.2^{12} \rrbracket + 2.\left|b_7b_6\right|.2^6.B_5 + B_5^2$$

$$RI2 - \llbracket ((4.QO1 + b_6).b_6)\,.2^{12} \rrbracket = 2.\left|b_7b_6\right|.2^6.B_5 + B_5^2$$

The Left hand side term is the remainder of the operation of stage II, $RO2 = RI2 - \llbracket ((4.QO1 + b_6).b_6)\,.2^{12} \rrbracket$ and the coefficient of 2.2^6B_5, is the partial quotient estimated at Stage II, $QO2 = |b_7b_6|$.

1.1.3 Stage III

$$RI3 = RO2 = RI2 - \left[\!\left[\left((4.b_7 + b_6).b_6 \right).2^{12} \right]\!\right] = 2.\left| b_7 b_6 \right|.2^6.\left(b_5.2^5 + B_4 \right) + \left(b_5.2^5 + B_4 \right)^2$$

$$RI3 = 2.\left| b_7 b_6 \right|.b_5.2^{11} + 2.\left| b_7 b_6 \right|.2^6.B_4 + b_5^2.2^{10} + 2^6.b_5.B_4 + B_4^2$$

$$RI3 = \left[\!\left[4.\left| b_7 b_6 \right|.b_5.2^{10} + b_5^2.2^{10} \right]\!\right] + (2.\left| b_7 b_6 \right| + b_5).2^6.B_4 + B_4^2$$

$$RI3 = \left[\!\left[4.\left| b_7 b_6 \right|.b_5.2^{10} + b_5^2.2^{10} \right]\!\right] + \left| b_7 b_6 b_5 \right|.2^6.B_4 + B_4^2$$

$$RI3 - \left[\!\left[(4.QO2 + b_5).b_5.2^{10} \right]\!\right] = \left| b_7 b_6 b_5 \right|.2.2^5.B_4 + B_4^2$$

The Left hand side term is the remainder of the operation of stage III, $RO3 = RI3 - \left[\!\left[(4.QO2 + b_5).b_5.2^{10} \right]\!\right]$ and the coefficient of $2.2^5 B_4$, is the partial quotient estimated at Stage II, $QO3 = \left| b_7 b_6 b_5 \right|$.

Thus, for each stage operation an additional bit of the quotient is determined, thereby, improving the resolution of the calculated square root result. Hence, the number of stage iterations required depends on the input size and is relate-able to the required output square root resolution.

Consolidating the stage operations and assuming RI1= Input Data we get,

$$RO_n = (RI_n > (QI_n \ll 2 + 1)?(RI_n - (QI_n \ll 2 + 1) : RI_n$$

$$QO_n = (QI_n \ll 1) + [(RI_n > (QI_n \ll 2 + 1)?1 : 0]$$

The quotient at the output of the last stage is the resultant square root of the configured resolution.

1.2 Research Goals

The objective of this paper is to identify and implement a simple and efficient algorithm to evaluate the square root operation on large data sets. Two algorithms have been chosen for this implementation, one is a non-restoring algorithm and the other is restoring algorithm. The paper compares the two algorithm in terms computational complexity, basic number of arithmetic operations required, number of resources needed and hardware cost.

1.3 Organization

The structure of the research paper is as follows:

• CHAPTER 2 Bibliographical Research: The chapter summarizes the existing research and algorithms on efficient computation methods for evaluating the square root of a number.

• CHAPTER 3 System Architecture: The chapter describes the requirements and design of the square root computational system.

• CHAPTER 4 Implementation: The chapter describes pipeline design implementation of the Non-Restoring and Restoring algorithms.

• CHAPTER 5 Verification Methodology and Results: The approach taken for verification of the design is discussed in this chapter.

• CHAPTER 6 Results: The results of the restoring and non-restoring algorithms are discussed in this chapter.

• CHAPTER 7 Conclusion: The conclusions is briefly discussed in this chapter.

Chapter 2

Bibliographical Research

The most important step in implementing a square root core is to identify a simple, efficient, iterative method of evaluating the square root of a given number. There are different methods of evaluating the square root like recursive approximation method and other methods. Substantial research is required to identify the method which would allow the implemented system to be power and area efficient while still providing the required boost throughput which would justify an implementation. Some of the research and methods of determining the square root of a fixed-point number are documented in the chapter.

Select research papers consulted for the work are briefly discussed in this chapter.

Wang et al. in [1] presents a new algorithm based on the conventional Non-restoring algorithm is implemented using Verilog HDL. This algorithm algorithm can be used on general purpose FPGA chips and can be used to calculate square root for a 2n bit integer. The algorithm developed is compared with the general algorithms for square root calculation based on the number of clock cycles required for the desired output, number of resources required to implement the algorithm and the number of pipeline stages required to calculate the square root of 16-bit and 32-bit Integer number. The advantage of this proposed new algorithm is that it can be im-

plemented on a general-use FPGA since it does not need a special hardware structure. Another advantage is that n number of clock cycles are needed for a 2n bit integer and the speed for processing the 2n bit integer is more as compared to other algorithm i.e. pipeline stages needed to compute the output are less.

Yamin et al. in [2] illustrates two new implementations of non-restoring algorithm which unlike the traditional algorithm does not focus on each bit of the square root rather depends upon the partial remainder. Algorithm reduces the need for additional resources, like multiplexors, multipliers, or seed generators and improves the resolution of the result for the last bit position. The iterations are simpler as either addition/subtraction is carried out based on the last iteration resultant bit.

The two algorithms differ in the implementation approach, one of the algorithms is a high-performance pipeline implementation and second other compromises on speed but is a low-cost implementation that uses less hardware. The algorithms implemented by Rachmad are proved to be time and area efficient than most of the existing algorithms. Paper clearly shows the contrast between the two implementations based on the number of cycles required for 16, 32 and 64-bit data and the number of gates required for 16, 32 and 64-bit data.

Rachmad et al. [3]in writes about a new approach to calculate square root of a given integer value. The author discusses first about the past researches on the square root calculation approach and explains the implementation of the algorithms which used the binary input decomposition approach. The algorithms discussed under previous works are Restoring algorithm and Non-restoring algorithm which compute square root on the same concept of binary input decomposition.

The author discusses the digital hardware implementation of the proposed algorithm and the advantages of the new algorithm over the past researched approaches. The hardware design uses a logical gate in place of a multiplication module and uses a comparator and addition/subtraction

module. The architecture is iterative in nature. It is a simple architecture that greatly reduces the need for additional resources in the design. Thus, the number of clock cycles needed to obtain an output depends upon the n-bit input. Therefore, (n/2) + 1 clock cycles are needed by the design to complete the square root calculation. The author summarizes the simulation and synthesis results for 32-bit and 64-bit data, considers number of clock cycles, clock speed, logic elements, logic registers and combinational functions needed for the design.

Richard et al. [4].in presents implementation of two high-speed square root calculation techniques which utilizes minimum number of computations. The author concentrates more on the square root approximation algorithms which can be efficiently implemented in fixed-point arithmetic. The constraints set on the algorithms by the author are: the division operation is not being considered for calculation, the number of iteration should be less and a lookup table could be used if required.

The two iterative methods discussed in the paper to determine the square root of a single integer value are Newton-Raphson Inverse (NRI) method and Non-Linear IIR Filter (NIIRF) method. The Newton-Raphson method is not recommended for fixed-point format rather it is most suitable for floating-point systems. In this method chances of error increases if bits are used to represent internal results. Second algorithm, Non-Linear IIR Filter (NIIRF) is another iterative technique for a fixed-point implementation. Thus, the algorithms are compared for accuracy versus computational cost/workload and the author concludes it by choosing NRIIF for fixed-point math implementations.

The other high-speed square root methods discussed are Binary-Shift Magnitude estimation and equiripple error magnitude estimation for complex number magnitude estimation.

Anuja et al. presents in [5] an algorithm that calculates square root of an 8-bit fixed and floating-point number in a Field programmable Gate Array (FPGA) using the previously implemented non-restoring algorithms. The paper compares the two algorithms Restoring and Non-

Restoring algorithms and implementation of a new module Controlled subtract multiplex in the modified non-restoring algorithm.

This new algorithm proposed eliminates the components without affecting the resolution, remainder and precision of the output. The author shows that the proposed algorithm is resource efficient than the existing non-restoring algorithms. The difference highlighted between the two algorithms is that the proposed algorithm does only subtraction operation and appends 01. It also introduces a new module, controlled subtract multiplex for algorithm to compute square root for any number of input bits. The comparison is tabulated for implementation with and without optimization (CSM). Concludes by proving that the non-restoring algorithm consumes less area on the chip and using a pipeline architecture improves the speed of the system.

Yamin et al. describes in [6] two single precision floating point square root design implementations based on the existing non-restoring algorithm. The two algorithms are implemented on a general-use FPGA. The first algorithm is iterative implemented that uses a subtractor/adder component and the design implementation is low on cost. The second algorithm is implemented in a pipeline fashion and on every clock cycle it accepts a square root instruction.

The paper briefly discusses other algorithms like New-Raphson algorithm and Sweeney-Robertson-Tocher (SRT) algorithms for square root extraction. The author describes the architecture and the implementation methodology for Non-Restoring square root algorithm, Parallel-array implementation and single precision floating point square root algorithm. The algorithms are compared on the latency, errors encountered and cost of design implementation. The two iterative square root algorithms require less area on chip and the pipelined system has better performance.

Atul et al. proposes in [7] a new algorithm to calculate the square root of an N-bit unsigned number. Two architecture design implementation have been discussed. First design has a pipeline architecture and the second design has asynchronous pipeline architecture for calculating the

square root of a N-bit fixed point number. The results of the existing and new algorithms have been compared and the author shows that the new square root algorithm is far more efficient, requires less clock cycles and less hardware.

First algorithm proposed has a pipeline architecture which constitutes of a subtractor and compartor module. The second algorithm proposed with asynchronous architecture proves to use less number of resources and has increased maximum operating frequency. Paper briefly discusses the existing non-restoring square root algorithm, new technique to design pipeline architecture for the non-restoring algorithm and asynchronous design for square root calculation of a 32-bit number using modified non-restoring algorithm. The total time needed for square root calculation significantly decreases in cases of a pipeline architecture.

Puneet et al. in [8] writes about a square root algorithm based on vedic mathematics formula called Dwandwa Yoga. The inputs to the algorithm are 24-bit floating point number and 16-bit floating point output. The author describes the existing algorithms for square root calculation and compares the existing algorithms with the new implementation. The new algorithm is implemented on SPARTAN-3E FPGA, this algorithm greatly reduces the complexity of the design. Simulation and synthesis results tabulated in the paper clearly proves that the new implementation consumes less area, power and is operable at high frequencies.

Sajid et al. in [9] presents a new algorithm for square root calculation which reduces the hazards of pipelining. The new proposed algorithm is based on the non-restoring algorithm and architecture is designed such that it can be modified to adapt as per the requirements of an application.

Read before write (RBW) hazard is avoided by the improved architecture of the algorithm by replacing an if-condition with a NOT gate. Synthesis produced a multiplexer when an if-condition was encountered, a multiplexer is a costly piece of hardware. Thus, the author improves the performance of the design by introducing a NOT gate. The system is tested for varying input

bits, accuracy of the result and for perfect combination of time, area and power.

Majerski et al. in [10] describes two algorithms for square root computation of addition of two numbers. The algorithm presented in the paper is designed for high-speed digital circuits. The algorithms are based on the non-restoring technique. Algorithm implemented differs with the previously researched algorithms in a way that the addition and subtraction by taking the carry as the third bit and considering the third bit while performing addition of three summands. Two summands then form a partial remainder. The most significant bits determine the conventional notation bits.

Montuschi et al. in [11] performs a survey of the square rooting algorithm. The algorithms are discussed in detail by taking into consideration their properties. Author gives his final comments on the effective and ideal implementation. The algorithms discussed are of two types: iterative technique and approximation by real functions. The algorithms reviewed under iterative technique is based on three classes: direct method, Newton-Raphson formula based algorithms and normalization method. Pure hardware implementations can be done using direct methods and the other iterative method applications can be done in hardware and software. Some of the commonly used iterative method are Newton-Raphson algorithm, CODIC, DeLugish's and Chen's algorithm. Among all these iterative algorithms, Newton -Raphson results are accurate. Another approach for square root calculation is approximation by real functions. Taylor and McLaurin series and Chebyshev polynomials are some of the real function approximation techniques. The paper discusses in detail about the advantages and disadvantages and the application of all the algorithms. The author focuses more on the algorithms that can be implemented on hardware.

Bannur et al. in [12] discusses implementation of two non-restoring square root algorithms, one using Barrel shifter implementation and second without the usage of Barrel shifter and instead storing and manipulating the operands. The algorithms are compared on the area required

on chip, number of cycles needed for square root calculation and resources needed for implementation. The algorithms presented are simple and easy to implement and achieves better precision compared to existing algorithms.

Using additional logic at the input of the Barrel shifter reduces the number of gates needed by half. Three bits are modified in one iteration and the value depends upon the sign in the previous iteration. Second algorithm implementation avoids the usage of Barrel shifter, instead the same operation is achieved by adding registers of size 2n-bits and shifting it every cycle. Second implementation proves to be better in terms of area needed on chip and time needed to compute the result. Both implementations also discusses the floating-point implementation and rounding of the last bits.

Various supporting papers [13–20] were also studied during this research.

Chapter 3

System Architecture

The goals of the system requirement is to implement a core that calculates the square root of a given integer operand. A 16-bit unsigned integer input is considered, whose square root has to be calculated within a resolution of $\pm 1 \times 10^{-3}$. The value of the output depends upon the number of bits at the input, input resolution and the number of fractional bits generated at the output. Since, the input is 16-bit unsigned integer, thus having unity resolution, the calculated square root has 8 bits as the integer square root of the number. The rest of the bits would constitute the fractional part of the solution. Square root calculator is implemented as a recursive operation, where number of times the stage algorithm is performed determines the number of bits at the output. The resolution of the output depends on the input size, resolution of the input and number of stages. A generic formula for determining the number of stages is:

$$resolution \geq 2^{ceiling(K/2)-n}$$

Where, K is the most significant bit position of the input, n is the number of stages.

Width of the solution depends upon the number of stages. The significance of the solution bits is derived from the significant length of the input $floor((log_2(Input_{max}+1)+1)/2)$.

Therefore for the give system requirements, the value of K is 15 (16 bit number with unity resolution), thus the significant position of the MSB of the solution would be $floor((log_2(65535+1)+1)/2) = 8$. The required length of the fractional bits depends on the required resolution. Each additional bit, after the integer increases the resolution by a factor of 1/2. Therefore the resolution of the output is given by 2^{-n}, where n is the number of fractional bits. Since the system specifies the resolution to be ± 0.001, the number of stages can be derived as:

$$0.001 \geq 2^{-n}$$

This gives the possible values for n, which should be greater than 9 in order to achieve the necessary resolution. In order to minimize hardware costs, while still attaining the requirements, the minimum possible stage number of 10 is chosen to generate the fractional part.

Therefore the total number of stages, and consequently bits in the solution is 8 + 10 = 18 stages, with the MSB of the output having a significance of 2^8. The actual resolution of the output is 2^{-10}= 0.00098, which is less than the resolution requirement of the system.

Since the stage operations depend only on the previous stage output data the system can be pipelined at the stage boundaries by registering the current output and allowing the stage to process new inputs in the next clock cycle. As a result the 18 stage implementation can thus be split into 18 stage pipeline resulting in higher throughput while reducing the operational speed and size of the system.

Chapter 4

Detail Design

The non-restoring and restoring algorithm of the iterative stage of the square root operation are discussed below.

4.1 Non-Restoring Algorithm

Non-restoring algorithm is an iterative square root determination which generates one bit of a result in each iteration using two most significant bits of the input data. The partial dividend used for each iteration is determined by considering the remainder of the previous iteration and appending the two most significant bit to the least significant position of the remainder. The stage quotient is is computed as the previous iteration quotient shifted by one (to denote the position significance) and appended with predicted new bit value ($Qn = Qn - 1 \ll 1 + 1$), the subtrahend is determined as the previous iteration quotient shifted by two and appended with the new predicted quotient bit value. If the value the new dividend is greater the subtrahend then, then predicted value is accepted and the difference is forwarded as the remainder for the next iteration; else the new quotient bit is assumed to be zero and the constructed dividend is directly

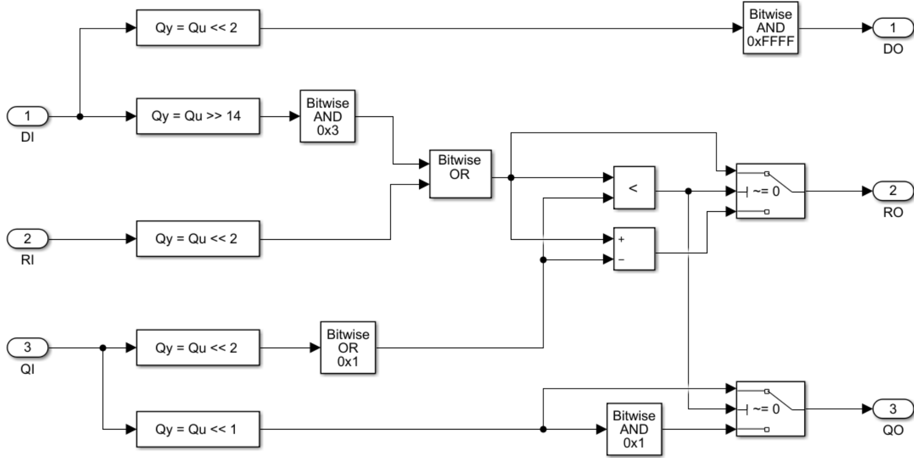

Figure 4.1: Non-Restoring Square Root Calculation for a single stage

forwarded as the remainder of the current iteration.

4.1.1 Implementation

A single iteration of the algorithm generates one bit of the square root output. The implementation is constructing by cascading multiple stages of the iteration with data storage elements isolating the stage for pipelined implementation. The Data Input to the stage is provided by the Data Output of the previous stage, which in turn is the shifted version of the original input data. The Remainder In for the current stage is provided by the Remainder Out of the previous stage. The Remainder In and the two most significant bits of the Data Input are used to construct the minuend for the stage operation. The subtrahend is determined from the Quotient In, which is provided by the Quotient Out of the previous stage and a assumed new bit value (1'b1). If the subtrahend is greater than or equal to the minuend (condition = 1), then the predicted quotient is accepted and the remainder out is updated with the non-negative difference; else the predicted quotient is discarded (changed to 0) and the minuend is forwarded as the remainder out for the current iteration.

The inputs for all stages is cascaded from the previous stage. The Quotient Input and Remainder Input of the first stage is assumed to be 0 and the input integer data is fed to the Data Input for the first stage.

```
input DI, QI, RI;
output DO, QO, RO;
wire rtemp, condition ;
assign rtemp = { RI , DI[15:14] };
assign condition = (rtemp >= {QI, 2'b01});
assign DO = {DI[13:0], 2'b00};
assign RO = condition ? rtemp - {QI,2'b01} : rtemp ;
assign QO = condition ? {QI,1'b1} : {QI,1'b0} ;
```

4.2 Restoring Algorithm

The restoring algorithm is an alternate method of computing the square root of a number, which differs from the Non-Restoring algorithm, primarily in dealing with the decision making step to determine the new quotient bit. The algorithm assumes a new quotient bit to be 1, and proceeds to evaluate the subtraction operation by deducting the newly determined subtrahend $((Q_{n-1} << 2) + 2'b01)$ from the minuend to determine the Remainder Out. An incorrect guess results in a negative remainder, which is identified by checking the sign bit (MSB) of the remainder. In the next iteration, if a negative remainder is detected, the system restores the previous non-negative remainder, by adding the previous predicted subtrahend to the remainder. The **restored** remainder is then used to calculate the output of the current stage.

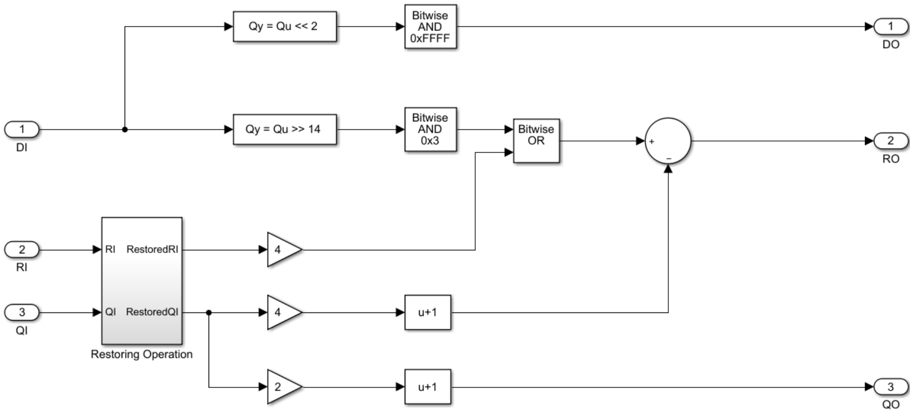

Figure 4.2: Restoring Operation

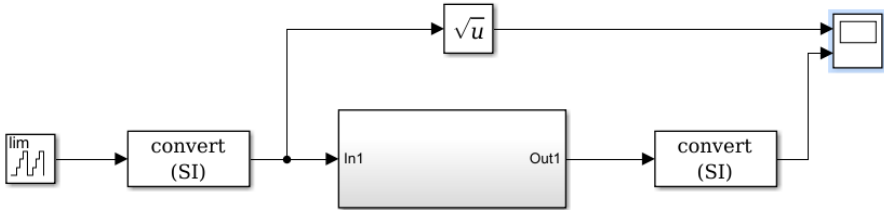

Figure 4.3: Restoring Algorithm for a single stage iteration

4.2.1 Implementation

A single iteration of the algorithm generates one bit of the square root output. The implementation is constructed by cascading multiple stages of the iteration with data storage elements isolating the stage for pipelined implementation. The Data Input to the stage is provided by the Data Output of the previous stage, which in turn is the shifted version of the original input data. The Remainder In for the current stage is provided by the Remainder Out of the previous stage. The Remainder In and the two most significant bits of the Data Input are used to construct the minuend for the stage operation. The subtrahend is determined from the Quotient In, which is provided by the Quotient Out of the previous stage and a assumed new bit value (1'b1) and the

assumed data is provided as the stage output. The minuend, based on the new predicted quotient bit 1 is subtracted from the subtrahend and the resultant remainder (+/−) is provided as the Remainder Out of the current stage. At the start of the iteration, if the partial remainder input to the block is negative the last assumed quotient bit is cleared and the subtraction performed in the previous stage is reverted and the remainder is restored to the previous value.

The inputs for all stages is cascaded from the previous stage. The Quotient Input and Remainder Input of the first stage is assumed to be 0 and the input integer data is fed to the Data Input for the first stage.

```
input DI, QI, RI;
output DO, QO, RO;
wire condition, rRestd, qRestd ;
assign condition = RI >> (2*StageNum-2) ; // Extraction of the sign-bit
assign qRestd = condition ? {QI[StageNum:1], 1'b0} : QI ; // Fixes the incorrect guess of the
quotient
assign rRestd = condition ? RI + {qRestd,1'b1} : RI ; // Fixes the incorrect guess of the
remainder
//Rest of algorithm for unconditional subtract for qn = 1'b1
assign DO = { DI[13:0], 2'b00 } ; // Left shifting the DataIn by 2
assign RO = { rRestd , DI[15:14] } - { qRestd , 2'b01 } ; // Subtra
assign QO = { qRestd , 1'b1 } ;
```

Chapter 5

Verification Methodology and Results

5.1 Verification Methodology

A test bench is implemented using SystemVerilog [21–26] to verify the operation of the restoring and non-restoring algorithm for square root computation. A stream of unconstrained 16-bit random integers is used as a input to verify system operation. The test bench contains an accurate model of the square root computation with delays to simulate pipeline operation. Therefore, when the same input is fed into the device under test (DUT) and the model provides the expected response which is compared with the DUT output to verify the accuracy of the calculation.

The steps to verify the system are generation of inputs, initial reset of DUT and reference model, capture the output of DUT and model, verification of correctness of the output and the errors or issues are reported for a solution. Since random vectors are used for verification functional coverage of the input vectors is utilized as the control parameter to decide the duration the test. The test is considered complete when all the possible inputs have been exercised.

5.2 Test Bench Design and its Components

The test bench should generate input vectors, provides the input to the DUT, determines the expected output, monitors the output of the DUT, compare the expected and actual output of the system, collect data and coverage statistics pertaining to the executed test cases, regulate test duration and monitor the overall performance of the system. A modular test bench architecture is adapted which consists of a network of objects, that perform the various operations required to test the system. A modular or layered architecture is preferred to test complex systems as it supports reuse of the developed components and the encapsulation of data and member functions; allows different blocks to be easily modified for changes in the system like change in input size, or data transmission protocols.

5.2.1 Layered Test Bench Architecture

A layered architecture is preferred as a common verification environment as it results in blocks that are reusable and extendable to be taken advantage of during test automation. There are three layers in a test bench known as Signal layer, Command layer and Scenario layer. The signal layer being the lowest layer the architecture, it connects the test bench with the RTL design. This layer comprises of DUT and has interface, modports and clocking modules. The command layer gives transaction-level interface to the next layer and drives the pins through signal layer. The command consists of a driver. The third layer is the functional layer, that consists of driver-monitor components and has a self-checking construction. This layer determines of the test cases fail or pass. This type of layered test bench is typically used for a constrained-random stimulus generation and allows the manual directed tests.

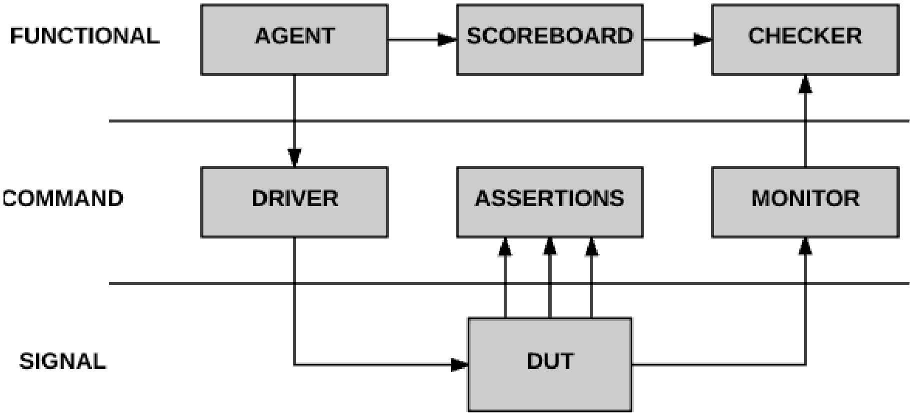

Figure 5.1: Lower Layers of the Verification Test Bench

The components of the test bench are instantiated hierarchically and called in phases to initialize, execute and complete every test [27]. The test bench components are shown in the block diagram in the figure below.

A verification methodology which has random test cases have been adapted to check the square root operation of the DUT. The inputs to the DUT are unconstrained random vectors to which the output is monitored and compared to the simulated the response of the system.The test bench is methodically is split into components to allow versatility and flexibility of use in the system.

A typical test bench consists of the following components:

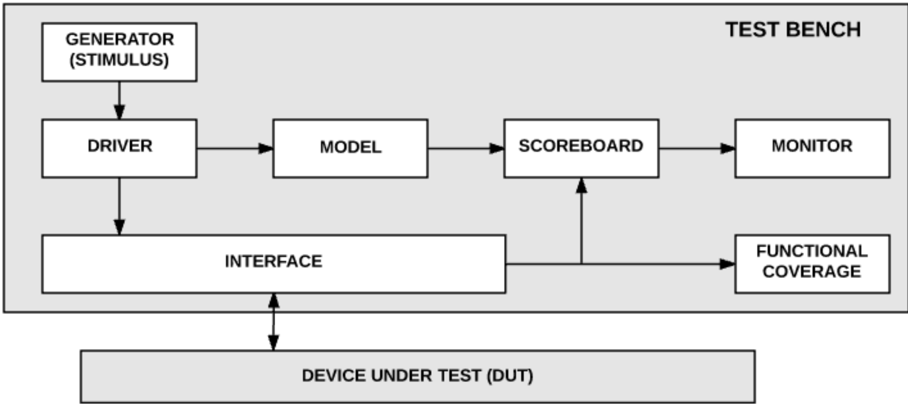

Figure 5.2: Test Bench Schematic for Verification

5.2.1.1 Interface

An interface encapsulates the required connections between a hardware block and the external system. It can also serve as a connection between the DUT, a hardware block and the test bench, which is a software component . It captures the communication between the blocks including the connectivity, directional information, clocking blocks. It ensures there is no duplication of connections, the port list for connections are compact and it easy to change to integrate to higher levels.

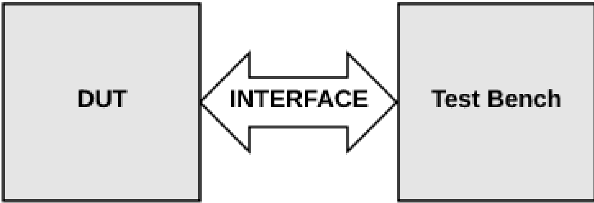

Figure 5.3: Interface Block Diagram

An external clock is is provided to the interface, which forwards it to the test bench for synchronization and device under test for synchronous operation . An virtual instance is created

for the interface class and is mapped to the outermost interface inside the different components of the test bench..

5.2.1.2 Stimulus

The stimulus module consists of a randomizable 16-bit vector which is used for the generation of inputs for the device under test and the model. In this test bench since the system should respond uniformly for all possible inputs, unconstrained random number generator is used to generate the inputs for verification.

```
/* *
* Author: Vyoma Sharma
* Rochester, NY, USA
* Description: Stimulus generates the random numbers.
* */
class stimulus;
rand bit [15:0] data;
endclass
```

In SystemVerilog, a class, containing a rand variable, is inherently defined with a randomize() function which generates a new random number based on the previously defined seed value. Therefore calling the randomize() function generates the new input which is then fed into the test bench.

5.2.1.3 Scoreboard

The scoreboard is the generic whiteboard for the test bench, it can store data which can be horizontally shared among the various components of the test bench, Typically the scoreboard may contain the actual output of the DUT, expected output provided by the model and the coverage

statistics like number of tests and other test related information. The use of scoreboard is optional when utilized with a well defined checker and functional coverage components.

5.2.1.4 Driver

The Driver operates as the conduit between the transaction layer and signal layer of the testbench. The inputs generated are at high level of abstraction and thus, these inputs are converted to actual design inputs. It accepts input data sets from the Stimulus, synchronizes and provides the input dataset to the actual input ports of the DUT. It also operates the necessary write protocols and signals required for the DUT to recognize the valid data provided by the driver.

5.2.1.5 Model

The model is high level embodiment of the requirements of the device under test. It is typically implemented with a high level of abstraction, where the requirements can be easily met, but is not suitable for synthesis and implementation. The validity of the device relies of the accuracy of the model. In the current test bench, the model required to capture the intent of the algorithm is generated from Simulink. The code is generated for the square root operation which has a 16-bit unsigned integer input and a 18-bit (8,10) fixed point output. A cascaded delay block is used to simulate the operation of the pipeline in the system.

5.2.1.6 Monitor / Checker

The monitor block monitors the output interface ports and receives and assembles the actual data from the device under test. It checks for any communication protocol violation and determines the value of the complete transactions [28]. Monitor also consists of coverage and assertions features to ensure the system behavior is met and tested. The output of the monitor is transaction level data, which in current application is the square root of the input integer.

The checker operates on the transaction data received from the monitor and the reference data from the model. The checker compares the monitor output and the model output to ensure functional correctness of the device operation. In this system, since the output of the device under test is parallel 18-bit output data, the role of the monitor is greatly reduced and therefore the monitor and checker are combined to a single test bench component [28].

5.2.1.7 Environment

The environment serves as the container in which the components of the test bench are instantiated. The various components classes are instantiated and connected to each other thereby constituting the complete test bench. The environment can also contain a program that executes the various initializes and executes the various components used, while implementing the sequence of operation and the worst-case test execution test timeout condition.

Chapter 6

Results

The implementation of the restoring and non-restoring algorithm for square root computation was successfully verified. Since the input was 16-bits the output bit with required in order to achieve 0.001 resolution was confirmed to be 18-bits with 10-bits representing the fractional portion of the calculated square root value. This gives the solution an effective resolution of 2^{-10} approximately 0.000977, which meets the system specification of at least 0.001 resolution. The resolution of the result can be improved by adding additional computation stages to the system.

Functional coverage was used to ensure that the system has been tested for all possible input values. The implemented functional coverage checks if all the bins corresponding to each possible input is set at least once before the test is terminated. This results in some of the inputs being tested more often than others but all the inputs are tested at least once and system operation is verified for all the inputs. Below are the coverage report for non-restoring and restoring algorithm.

```
                  COVERAGE REPORT
DUT input coverage:             100.000000
Detected Error Count:             0.000000
---------------------------------
```

Figure 6.1: Coverage Report for Restoring/Non-Restoring Algorithm

Issues were faced while implementing the test bench for non-restoring algorithm and restoring algorithm. In the test bench, a malfunction was detected in it's operation which substantially increased run-time and processing power required for the test bench evaluation. The root cause of the problem was identified as incorrect interfacing between the hardware signals and the software test bench. The looping statement introduced in the monitor to check at every edge of the clock if the DUT output is identical to the model output was implemented without considering synchronization with the clock. This resulted in the system evaluating an infinite loop between clock edge events causing the system to stall indefinitely at the loop. This was resolved by including an instruction to execute each iteration of the loop for each rising edge of the clock.

6.1 Non-Restoring

Cursor at TimeA and TimeB indicates that the clock period is 20 ns. The time taken to complete the entire square root computation for a single input is 360 ns (TimeC - TimeA), thus the system evaluates the 18 bit square root result in 18 clock cycles.

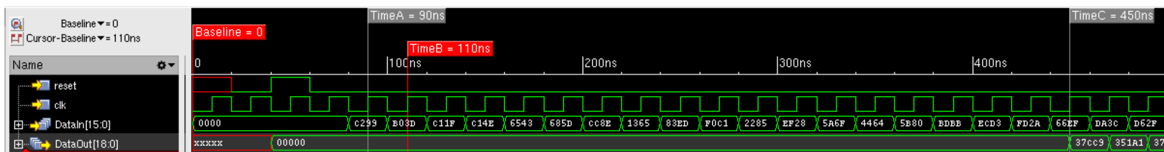

Figure 6.2: Pipeline Data flow for Non-Restoring Algorithm

The square root of the first input is provided after 18 clock cycles. It is observable that the next input to the square root computational block is evaluated before the completion of the previous input evaluation by the system, thus indicating the operation of the 18 stage pipelined system design. Compared to the non-pipelined implementation, which would consume 18 clock cycles per input evaluation , it is can be observed that, once the pipeline is filled, the system is

Table 6.1: Logic Synthesis Results for Non_Restoring Algorithm

Parameter	Pre-Scan	Post-Scan
Total Cell Area	91293.04875	101455.2031
Number of Cells	3017	2905
Worst Case Timing	19.75 ns	19.75 ns
Total Dynamic Power	3.1607 mW	3.4971 mW
DFT Test Coverage	-	100%

able to process a new data every clock cycle ., the system consumes on an average 1 clock cycle per input. The pipelining thus increases the throughput by over 17 times of the non-pipelined implementation

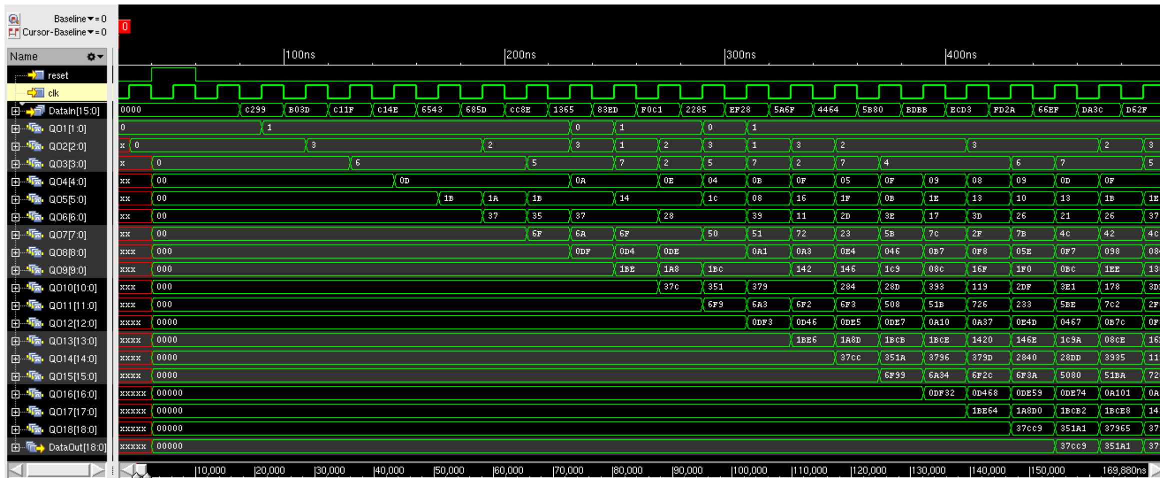

Figure 6.3: Data transition from Input to Output

The Non-Restoring algorithm implementation was synthesized using a 180 nm process design kit (PDK), results are shown in table 6.1.

6.2 Restoring

Cursor at TimeA and TimeB indicates that the clock period is 20 ns. The time taken to complete the entire square root computation for a single input is 360ns (TimeC - TimeA), thus the system

evaluates the 18 bit square root result in 18 clock cycles.

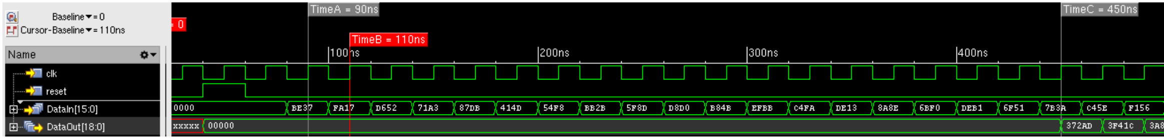

Figure 6.4: Pipeline Data flow for Restoring Algorithm

The square root of the first input is provided after 18 clock cycles. It is observable that the next input to the square root computational block is evaluated before the completion of the previous input evaluation by the system, thus indicating the operation of the 18 stage pipelined system design. Compared to the non-pipelined implementation, which would consume 18 clock cycles per input evaluation , it is can be observed that, once the pipeline is filled, the system is able to process a new data every clock cycle ., the system consumes on an average 1 clock cycle per input. The pipelining thus increases the throughput by over 17 times of the non-pipelined implementation. The termination stage of the algorithm detects the polarity of the last partial remainder and if required flip the last asserted quotient bit to 0.

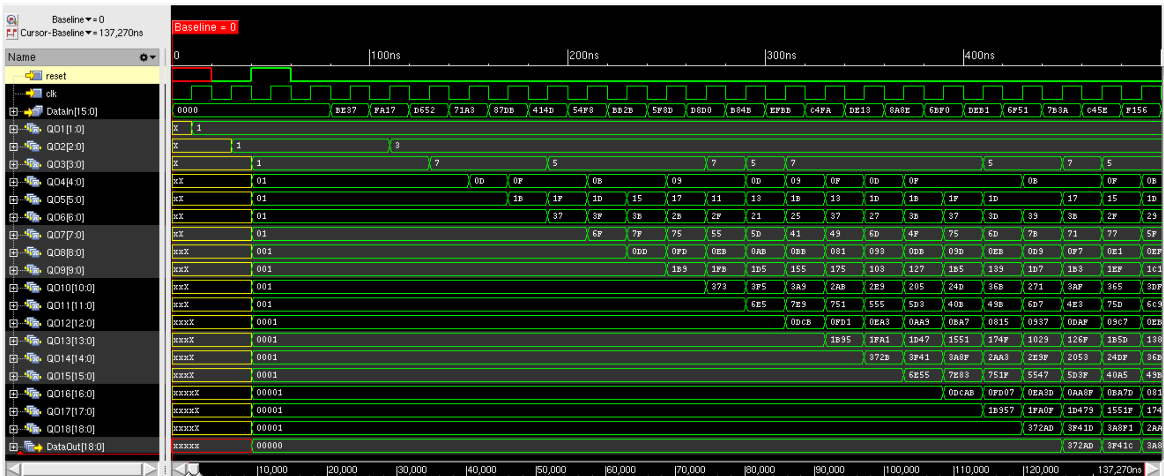

Figure 6.5: Data transition from Input to Output

Table 6.2: Logic Synthesis Results for Restoring Algorithm

Parameter	**Pre-Scan**	**Post-Scan**
Total Cell Area	93584.9385	103757.072
Number of Cells	2340	2284
Worst Case Timing	19.75 ns	19.75 ns
Total Dynamic Power	3.2308 mW	3.8427 mW
DFT Test Coverage	-	100%

The Restoring algorithm implementation was synthesized using a 180 nm process design kit (PDK), results are shown in table 6.2.

Chapter 7

Conclusion

The objective of the paper was to design a high-throughput implementation of a square root computational core. A restoring and non-restoring approaches were considered for the design of the core. The square root computation was implemented as a 18-stage pipeline which increases the throughput of the system by 17 times compared to the non-pipelined implementation. It was observed that the non-restoring approach required fewer hardware components compared to the restoring algorithm but required more control and branching functions compared to the later. Hence, the choice between restoring and non-restoring is a trade-off between hardware cost and system complexity. The restoring algorithm can be further simplified by combining the remainder restoration operation with the new remainder computation. This would significantly reduce the power consumption of the system by eliminating unnecessary signal transitions associated with decision making based on data evaluated in the current execution cycle. The non-restoring algorithm provides an accurate bit of the computed square root value per iterative stage. Thus, the number of bits required would always correspond to the number of stages in the system. The restoring algorithm returns a bit of the quotient which is checked only in the next stage of the iteration. Hence, to determine n accurate bits we require n+1 stages with the last stage being a

termination stage whose function is to identify if the determined quotient bit is correct and is in compliance with the partial remainder obtained from the previous stage. Thus, an n-bit square root computation using the restoring algorithm would require n+1 clock cycles for evaluation.

The described implementations have been verified using a SystemVerilog test bench which generates random unconstrained vectors to test the operation of the restoring and non-restoring algorithm. The square root reference model in the test bench which is used to determine the expected output value of the device under test and is generated from the standard Simulink library. Using a reference model source reduced the possibility of wrong error detection and increased the reliability of validation of the DUT. The implemented functional coverage ensured that all possible inputs to the square root core have been evaluated before test is completed, ensuring that the system operation is tested for all possible values in the input range. The implementation have been synthesized in 180nm technology standard and a comparison of the two implementations have been performed.

References

[1] Y. Z. X. Wang, Q. Ye, and S. Yang, "A new algorithm for designing square root calculators based on fpga with pipeline technology," in *2009 Ninth International Conference on Hybrid Intelligent Systems*, vol. 1, Aug 2009, pp. 99–102.

[2] Y. Li and W. Chu, "A new non-restoring square root algorithm and its vlsi implementations," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, Oct 1996, pp. 538–544.

[3] R. V. W. Putra, "A novel fixed-point square root algorithm and its digital hardware design," in *International Conference on ICT for Smart Society*, June 2013, pp. 1–4.

[4] R. G. Lyons, *High-Speed Square Root Algorithms*. Wiley-IEEE Press, 2012, pp. 243–250. [Online]. Available: http://ieeexplore.ieee.org.ezproxy.rit.edu/xpl/articleDetails.jsp?arnumber=6241243

[5] A. Nanhe, G. Gawali, S. Ahire, and K. Sivasankaran, "Implementation of fixed and floating point square root using nonrestoring algorithm on fpga," in *International Journal of Computer and Electrical Engineering, Vol. 5, No. 5, October 2013*, vol. 3, Feb 2013, pp. 226–230.

[6] Y. Li and W. Chu, "Implementation of single precision floating point square root on fp-

gas," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, Apr 1997, pp. 226–232.

[7] A. Rahman and Abdullah-Al-Kafi, "New efficient hardware design methodology for modified non-restoring square root algorithm," in *2014 International Conference on Informatics, Electronics Vision (ICIEV)*, May 2014, pp. 1–6.

[8] P. Kachhwal and B. C. Rout, "Novel square root algorithm and its fpga implementation," in *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*, July 2014, pp. 158–162.

[9] I. Sajid, M. M. Ahmed, and S. G. Ziavras, "Pipelined implementation of fixed point square root in fpga using modified non-restoring algorithm," in *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 3, Feb 2010, pp. 226–230.

[10] S. Majerski, "Square-root algorithms for high-speed digital circuits," in *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*, June 1983, pp. 99–102.

[11] P. Montuschi and P. M. Mezzalama, "Survey of square rooting algorithms," *IEE Proceedings E - Computers and Digital Techniques*, vol. 137, no. 1, pp. 31–40, Jan 1990.

[12] J. Bannur and A. Varma, "The vlsi implementation of a square root algorithm," in *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*, June 1985, pp. 159–165.

[13] M. D. Ercegovac, "On digit-by-digit methods for computing certain functions," in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, Nov 2007, pp. 338–342.

[14] C. K. Piromsopa and P. Chongsatitvatana, "An fpga implementation of a fixed-point square root operation," *Chulalongkorn University*, 09 2002.

[15] Y. Li and W. Chu, "Parallel-array implementations of a non-restoring square root algorithm," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, Oct 1997, pp. 690–695.

[16] D. R. Llamocca-Obregon, "A core design to obtain square root based on a non-restoring algorithm," in *Proceedings XI Workshop IBERCHIP*, 03 2005.

[17] E. L. Oberstar, *Fixed-Point Representation & Fractional Math*. Oberstar Consulting, 2007.

[18] K. N. Vijeyakumar, V. Sumathy, P. Vasakipriya, and A. D. Babu, "Fpga implementation of low power high speed square root circuits," in *2012 IEEE International Conference on Computational Intelligence and Computing Research*, Dec 2012, pp. 1–5.

[19] B. Yang, D. Wang, and L. Liu, "Complex division and square-root using cordic," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, April 2012, pp. 2464–2468.

[20] W. Chu and Y. Li, "Cost/performance tradeoff of n-select square root implementations," in *Proceedings 5th Australasian Computer Architecture Conference. ACAC 2000 (Cat. No.PR00512)*, 2000, pp. 9–16.

[21] IEEE, *IEEEStd 1364 - 2005 IEEE Standard for Verilog Hardware Description Language*, IEEE Std. "IEEEStd 1364 - 2005", 04 2006.

[22] C. Spear and G. Tumbush, *SystemVerilog for Verification, Third Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2012.

[23] J. Bergeron, I. Books24x7, and Books24x7.com, *Writing testbenches using SystemVerilog*, 1st ed. New York: Springer Science & Business Media, 2006;2007;.

[24] C. E. Cummings and T. Fitzpatrick, "Ovm & uvm techniques for terminating tests," 2011.

[25] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design*. Springer US, 2006.

[26] S. A. Wadekar, “A rt level verification method for soc designs,” in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, Sept 2003, pp. 29–32.

[27] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti, “Synthesis of system verilog assertions,” in *Proceedings of the Design Automation Test in Europe Conference*, vol. 2, March 2006, pp. 1–6.

[28] A. Shetty and D. H. Mahmoodi, *System Verilog Testbench Tutorial Using Synopsys EDA Tools*. Nano-Electronics & Computing Research Center School of Engineering San Francisco State University San Francisco, CA, 2011.

Appendix I

Source Code

I.1 Non-restoring Algorithm

I.1.1 Non-Restoring Algorithm Operation

```
1  /*
2  *
3  *Description: Non-Restoring Algorithm implementation for
4  *Square-root computation for a single stage
5  *
6  *Author: Vyoma Sharma
7  *
8  */
9  module SqrtStage(DI, QI, RI, DO, QO, RO);
10 parameter StageNum = 20;
11 input wire [15:0] DI;
```

```
12 input wire [StageNum:0] QI;
13 input wire [2*StageNum:0] RI;
14 output wire [15:0]  DO;
15 output wire [StageNum:0] QO;
16 output wire [2*StageNum:0] RO;
17 wire [2*StageNum:0] rtemp;
18 wire condition;
19 assign rtemp = {RI, DI[15:14]};
20 assign condition = (rtemp >= {QI,2'b01});
21 assign DO = {DI[13:0],2'b00};
22 assign RO = condition ? rtemp - {QI,2'b01} : rtemp;
23 assign QO = condition ? {QI,1'b1} : {QI,1'b0};
24 endmodule // SQRTNRST
```

I.1.2 Non-Restoring Algorithm with Pipeline architecture

```
1 /*
2 *
3 *Description: Non-Restoring Algorithm implementation
4 *for Square-root computation with pipeline architecture
5 *
6 *Author: Vyoma Sharma
7 *
8 */
9
10 `define tscale `timescale 1ns/1ns
```

```
11 module SQRTNRST (
12       clk,
13         reset,
14         DataIn,
15         DataOut,
16         test_mode,
17         scan_in0,
18       scan_en,
19         scan_out0
20        );
21 input
22   reset,                  // system reset
23   clk;                    // system clock
24 input
25   scan_in0,               // test scan mode data input
26   scan_en,                // test scan mode enable
27   test_mode;              // test mode select
28 output
29   scan_out0;    // test scan mode data output
30 input
31   wire  [15:0]    DataIn;
32 output
33   reg [18:0] DataOut;
34 // Local Interconnect Definitions for Stage 1
35     reg  [1:0]  QI1 ;
```

```
36      reg  [2:0]  RI1 ;
37      reg  [15:0] DI1 ;
38      wire [15:0] DO1 ;
39      wire [1:0]  QO1 ;
40      wire [2:0]  RO1 ;
41  // Local Interconnect Definitions for Stage 2
42      reg   [2:0] QI2 ;
43      reg   [4:0] RI2 ;
44      reg   [15:0]  DI2 ;
45      wire  [15:0]  DO2 ;
46      wire  [2:0] QO2 ;
47      wire  [4:0] RO2 ;
48  // Local Interconnect Definitions for Stage 3
49      reg   [3:0] QI3 ;
50      reg   [6:0] RI3 ;
51      reg   [15:0]  DI3 ;
52      wire  [15:0]  DO3 ;
53      wire  [3:0] QO3 ;
54      wire  [6:0] RO3 ;
55  // Local Interconnect Definitions for Stage 4
56      reg   [4:0] QI4 ;
57      reg   [8:0] RI4 ;
58      reg   [15:0]  DI4 ;
59      wire  [15:0]  DO4 ;
60      wire  [4:0] QO4 ;
```

```
61       wire   [8:0] RO4 ;
62 // Local Interconnect Definitions for Stage 5
63       reg    [5:0] QI5 ;
64       reg    [10:0]  RI5 ;
65       reg    [15:0]  DI5 ;
66       wire   [15:0]  DO5 ;
67       wire   [5:0] QO5 ;
68       wire   [10:0]  RO5 ;
69 // Local Interconnect Definitions for Stage 6
70       reg    [6:0] QI6 ;
71       reg    [12:0]  RI6 ;
72       reg    [15:0]  DI6 ;
73       wire   [15:0]  DO6 ;
74       wire   [6:0] QO6 ;
75       wire   [12:0]  RO6 ;
76 // Local Interconnect Definitions for Stage 7
77       reg    [7:0] QI7 ;
78       reg    [14:0]  RI7 ;
79       reg    [15:0]  DI7 ;
80       wire   [15:0]  DO7 ;
81       wire   [7:0] QO7 ;
82       wire   [14:0]  RO7 ;
83 // Local Interconnect Definitions for Stage 8
84       reg    [8:0] QI8 ;
85       reg    [16:0]  RI8 ;
```

```
86      reg   [15:0]  DI8 ;
87      wire  [15:0]  DO8 ;
88      wire  [8:0] QO8 ;
89      wire  [16:0]  RO8 ;
90  // Local Interconnect Definitions for Stage 9
91      reg   [9:0] QI9 ;
92      reg   [18:0]  RI9 ;
93      reg   [15:0]  DI9 ;
94      wire  [15:0]  DO9 ;
95      wire  [9:0] QO9 ;
96      wire  [18:0]  RO9 ;
97  // Local Interconnect Definitions for Stage 10
98      reg   [10:0]  QI10 ;
99      reg   [20:0]  RI10 ;
100     reg   [15:0]  DI10 ;
101     wire  [15:0]  DO10 ;
102     wire  [10:0]  QO10 ;
103     wire  [20:0]  RO10 ;
104 // Local Interconnect Definitions for Stage 11
105     reg   [11:0]  QI11 ;
106     reg   [22:0]  RI11 ;
107     reg   [15:0]  DI11 ;
108     wire  [15:0]  DO11 ;
109     wire  [11:0]  QO11 ;
110     wire  [22:0]  RO11 ;
```

```
111 // Local Interconnect Definitions for Stage 12
112     reg   [12:0]  QI12 ;
113     reg   [24:0]  RI12 ;
114     reg   [15:0]  DI12 ;
115     wire  [15:0]  DO12 ;
116     wire  [12:0]  QO12 ;
117     wire  [24:0]  RO12 ;
118 // Local Interconnect Definitions for Stage 13
119     reg   [13:0]  QI13 ;
120     reg   [26:0]  RI13 ;
121     reg   [15:0]  DI13 ;
122     wire  [15:0]  DO13 ;
123     wire  [13:0]  QO13 ;
124     wire  [26:0]  RO13 ;
125 // Local Interconnect Definitions for Stage 14
126     reg   [14:0]  QI14 ;
127     reg   [28:0]  RI14 ;
128     reg   [15:0]  DI14 ;
129     wire  [15:0]  DO14 ;
130     wire  [14:0]  QO14 ;
131     wire  [28:0]  RO14 ;
132 // Local Interconnect Definitions for Stage 15
133     reg   [15:0]  QI15 ;
134     reg   [30:0]  RI15 ;
135     reg   [15:0]  DI15 ;
```

```
136     wire  [15:0]  DO15 ;
137     wire  [15:0]  QO15 ;
138     wire  [30:0]  RO15 ;
139  // Local Interconnect Definitions for Stage 16
140     reg   [16:0]  QI16 ;
141     reg   [32:0]  RI16 ;
142     reg   [15:0]  DI16 ;
143     wire  [15:0]  DO16 ;
144     wire  [16:0]  QO16 ;
145     wire  [32:0]  RO16 ;
146  // Local Interconnect Definitions for Stage 17
147     reg   [17:0]  QI17 ;
148     reg   [34:0]  RI17 ;
149     reg   [15:0]  DI17 ;
150     wire  [15:0]  DO17 ;
151     wire  [17:0]  QO17 ;
152     wire  [34:0]  RO17 ;
153  // Local Interconnect Definitions for Stage 18
154     reg   [18:0]  QI18 ;
155     reg   [36:0]  RI18 ;
156     reg   [15:0]  DI18 ;
157     wire  [15:0]  DO18 ;
158     wire  [18:0]  QO18 ;
159     wire  [36:0]  RO18 ;
160   SqrtStage #(1) Stage1 (
```

```
161             .DI(DI1),
162             .QI(QI1),
163             .RI(RI1),
164             .DO(DO1),
165             .QO(QO1),
166             .RO(RO1)    );
167     SqrtStage #(2) Stage2 (
168             .DI(DI2),
169             .QI(QI2),
170             .RI(RI2),
171             .DO(DO2),
172             .QO(QO2),
173             .RO(RO2)    );
174     SqrtStage #(3) Stage3 (
175             .DI(DI3),
176             .QI(QI3),
177             .RI(RI3),
178             .DO(DO3),
179             .QO(QO3),
180             .RO(RO3)    );
181     SqrtStage #(4) Stage4 (
182             .DI(DI4),
183             .QI(QI4),
184             .RI(RI4),
185             .DO(DO4),
```

```
186             .QO(QO4),
187             .RO(RO4)    );
188    SqrtStage #(5) Stage5 (
189             .DI(DI5),
190             .QI(QI5),
191             .RI(RI5),
192             .DO(DO5),
193             .QO(QO5),
194             .RO(RO5)    );
195    SqrtStage #(6) Stage6 (
196             .DI(DI6),
197             .QI(QI6),
198             .RI(RI6),
199             .DO(DO6),
200             .QO(QO6),
201             .RO(RO6)    );
202    SqrtStage #(7) Stage7 (
203             .DI(DI7),
204             .QI(QI7),
205             .RI(RI7),
206             .DO(DO7),
207             .QO(QO7),
208             .RO(RO7)    );
209    SqrtStage #(8) Stage8 (
210             .DI(DI8),
```

```
211             . QI ( QI8 ) ,
212             . RI ( RI8 ) ,
213             .DO(DO8) ,
214             .QO(QO8) ,
215             .RO(RO8)     ) ;
216     SqrtStage #(9) Stage9 (
217             . DI ( DI9 ) ,
218             . QI ( QI9 ) ,
219             . RI ( RI9 ) ,
220             .DO(DO9) ,
221             .QO(QO9) ,
222             .RO(RO9)     ) ;
223     SqrtStage #(10) Stage10 (
224             . DI ( DI10 ) ,
225             . QI ( QI10 ) ,
226             . RI ( RI10 ) ,
227             .DO(DO10) ,
228             .QO(QO10) ,
229             .RO(RO10)      ) ;
230     SqrtStage #(11) Stage11 (
231             . DI ( DI11 ) ,
232             . QI ( QI11 ) ,
233             . RI ( RI11 ) ,
234             .DO(DO11) ,
235             .QO(QO11) ,
```

```
236            .RO(RO11)      );
237     SqrtStage #(12) Stage12 (
238            .DI(DI12),
239            .QI(QI12),
240            .RI(RI12),
241            .DO(DO12),
242            .QO(QO12),
243            .RO(RO12)      );
244     SqrtStage #(13) Stage13 (
245            .DI(DI13),
246            .QI(QI13),
247            .RI(RI13),
248            .DO(DO13),
249            .QO(QO13),
250            .RO(RO13)      );
251     SqrtStage #(14) Stage14 (
252            .DI(DI14),
253            .QI(QI14),
254            .RI(RI14),
255            .DO(DO14),
256            .QO(QO14),
257            .RO(RO14)      );
258     SqrtStage #(15) Stage15 (
259            .DI(DI15),
260            .QI(QI15),
```

```
261             .RI(RI15),
262             .DO(DO15),
263             .QO(QO15),
264             .RO(RO15)    );
265     SqrtStage #(16) Stage16 (
266             .DI(DI16),
267             .QI(QI16),
268             .RI(RI16),
269             .DO(DO16),
270             .QO(QO16),
271             .RO(RO16)    );
272     SqrtStage #(17) Stage17 (
273             .DI(DI17),
274             .QI(QI17),
275             .RI(RI17),
276             .DO(DO17),
277             .QO(QO17),
278             .RO(RO17)    );
279     SqrtStage #(18) Stage18 (
280             .DI(DI18),
281             .QI(QI18),
282             .RI(RI18),
283             .DO(DO18),
284             .QO(QO18),
285             .RO(RO18)    );
```

```
286    always @( posedge clk , posedge reset ) begin
287      if(reset == 1'b1) begin
288        DataOut <= 0;
289        QI1 <= 'b0;
290        RI1 <= 'b0;
291        DI1 <= 'b0;
292        DI2 <= 'b0;
293        QI2 <= 'b0; //Quotient Input
294        RI2 <= 'b0; //Remainder Input
295        QI3 <= 'b0;
296        RI3 <= 'b0;
297        DI3 <= 'b0;
298        QI4 <= 'b0;
299        RI4 <= 'b0;
300        DI4 <= 'b0;
301        QI5 <= 'b0;
302        RI5 <= 'b0;
303        DI5 <= 'b0;
304        QI6 <= 'b0;
305        RI6 <= 'b0;
306        DI6 <= 'b0;
307        QI7 <= 'b0;
308        RI7 <= 'b0;
309        DI7 <= 'b0;
310        QI8 <= 'b0;
```

```
311           RI8  <=  'b0;
312           DI8  <=  'b0;
313           QI9  <=  'b0;
314           RI9  <=  'b0;
315           DI9  <=  'b0;
316           QI10  <=  'b0;
317           RI10  <=  'b0;
318           DI10  <=  'b0;
319           QI11  <=  'b0;
320           RI11  <=  'b0;
321           DI11  <=  'b0;
322           QI12  <=  'b0;
323           RI12  <=  'b0;
324           DI12  <=  'b0;
325           QI13  <=  'b0;
326           RI13  <=  'b0;
327           DI13  <=  'b0;
328           QI14  <=  'b0;
329           RI14  <=  'b0;
330           DI14  <=  'b0;
331           QI15  <=  'b0;
332           RI15  <=  'b0;
333           DI15  <=  'b0;
334           QI16  <=  'b0;
335           RI16  <=  'b0;
```

```
336         DI16 <= 'b0;
337         QI17 <= 'b0;
338         RI17 <= 'b0;
339         DI17 <= 'b0;
340         QI18 <= 'b0;
341         RI18 <= 'b0;
342         DI18 <= 'b0;
343       end else begin
344         QI1 <= 1'b0    ;
345         RI1 <= 2'b00   ;
346         DI1 <= DataIn  ;
347         // Data Handoff to Stage 2
348         QI2 <= QO1   ;
349         RI2 <= RO1   ;
350         DI2 <= DO1   ;
351         // Data Handoff to Stage 3
352         QI3 <= QO2   ;
353         RI3 <= RO2   ;
354         DI3 <= DO2   ;
355         // Data Handoff to Stage 4
356         QI4 <= QO3   ;
357         RI4 <= RO3   ;
358         DI4 <= DO3   ;
359         // Data Handoff to Stage 5
360         QI5 <= QO4   ;
```

```
361        RI5  <= RO4   ;
362        DI5  <= DO4   ;
363        // Data Handoff to Stage 6
364        QI6  <= QO5   ;
365        RI6  <= RO5   ;
366        DI6  <= DO5   ;
367        // Data Handoff to Stage 7
368        QI7  <= QO6   ;
369        RI7  <= RO6   ;
370        DI7  <= DO6   ;
371        // Data Handoff to Stage 8
372        QI8  <= QO7   ;
373        RI8  <= RO7   ;
374        DI8  <= DO7   ;
375        // Data Handoff to Stage 9
376        QI9  <= QO8   ;
377        RI9  <= RO8   ;
378        DI9  <= DO8   ;
379        // Data Handoff to Stage 10
380        QI10  <= QO9  ;
381        RI10  <= RO9  ;
382        DI10  <= DO9  ;
383        // Data Handoff to Stage 11
384        QI11  <= QO10   ;
385        RI11  <= RO10   ;
```

```
386         DI11  <=  DO10   ;
387         //  Data  Handoff  to  Stage  12
388         QI12  <=  QO11   ;
389         RI12  <=  RO11   ;
390         DI12  <=  DO11   ;
391         //  Data  Handoff  to  Stage  13
392         QI13  <=  QO12   ;
393         RI13  <=  RO12   ;
394         DI13  <=  DO12   ;
395         //  Data  Handoff  to  Stage  14
396         QI14  <=  QO13   ;
397         RI14  <=  RO13   ;
398         DI14  <=  DO13   ;
399         //  Data  Handoff  to  Stage  15
400         QI15  <=  QO14   ;
401         RI15  <=  RO14   ;
402         DI15  <=  DO14   ;
403         //  Data  Handoff  to  Stage  16
404         QI16  <=  QO15   ;
405         RI16  <=  RO15   ;
406         DI16  <=  DO15   ;
407         //  Data  Handoff  to  Stage  17
408         QI17  <=  QO16   ;
409         RI17  <=  RO16   ;
410         DI17  <=  DO16   ;
```

```
411         // Data Handoff to Stage 18
412         QI18 <= QO17  ;
413         RI18 <= RO17  ;
414         DI18 <= DO17  ;
415         DataOut = QO18  ;
416       end
417     end
418 endmodule // SQRTNRST
```

I.1.3 Environment

```
1  /*
2  * Description: Parent class which contains the
3  *              instances of the test bench component.
4  *
5  *Author: Vyoma Sharma
6  *
7  *
8  */
9      class environment;
10             driver drvr;
11             scoreboard sb;
12             monitor mntr;
13     function new(virtual intf intfvalDUT, virtual intf
         intfvalModel);
14           sb = new();
```

```
15              drvr = new(intfvalDUT, intfvalModel, sb);
16            mntr = new(intfvalDUT, intfvalModel, sb);
17              fork
18               mntr.check();
19           mntr.cov_check();
20              join_none
21              endfunction
22
23         endclass
```

I.1.4 Driver

```
1  /*
2  *Description: Driver reads the random variable
3  *from the stimulus and gives it to the model.
4  *It reads the value from stimulus and writes
5  *the data in DUT based on clock.
6  *
7  *Author: Vyoma Sharma
8  *
9  */
10
11    class driver;
12
13          stimulus sti;
14          scoreboard sb;
```

```
15         virtual intf intf_DUT;
16
17     virtual intf intf_model;
18
19         function new;
20       input virtual intf intf_DUT;
21       input virtual intf intf_model;
22       input scoreboard sb;
23       begin
24          this.intf_DUT = intf_DUT;
25          this.intf_model = intf_model;
26              this.sb = sb;
27          this.sti = new();
28         end
29
30       endfunction
31
32         task reset();
33
34        intf_DUT.DataIn = 0;
35        intf_model.DataIn = 0;
36             @ (negedge intf_DUT.clk);
37             intf_DUT.reset = 0;
38        intf_model.reset = 0;
39             @ (negedge intf_DUT.clk);
```

```
40                      intf_DUT.reset = 1;
41            intf_model.reset = 1;
42                      @ (negedge intf_DUT.clk);
43                      intf_DUT.reset = 0;
44            intf_model.reset = 0;
45
46              endtask
47
48              task drive(input integer iteration);
49          int loc;
50          repeat (iteration) begin
51          loc = sti.randomize();
52          @(negedge intf_DUT.clk);
53          intf_DUT.DataIn = sti.data;
54          intf_model.DataIn = sti.data;
55
56          end
57
58        endtask
59      endclass
```

I.1.5 Model

```
1 /*
2 *
3 *File Name: hdlsrc\model\model.v
```

```
4   *
5   * Generated by MATLAB 9.2 and HDL Coder 3.10
6   *
7   *
8   * -- ------------------------------------------
9   * -- Rate and Clocking Details
10  * -- ------------------------------------------
11  * Model base rate: 0.2
12  * Target subsystem base rate: 0.2
13  *
14  * ---------------------------------------------
15  * ---------------------------------------------
16  *
17  * Module: model
18  * Source Path: model
19  * Hierarchy Level: 0
20  *
21  */
22  module model
23            (clk,
24             reset,
25             DataIn,
26             DataOut);
27    input   clk;
28    input   reset;
```

```
29    input   [15:0] DataIn;  // uint16
30    output  [18:0] DataOut;  // ufix19_En10
31    reg [15:0] Delay_reg [0:18];  // ufix16 [18]
32    wire [15:0] Delay_reg_next [0:18];  // ufix16 [18]
33    wire [15:0] Delay_out1;  // uint16
34    wire [18:0] Sqrt_out1;  // ufix19_En10
35    always @(posedge clk or posedge reset)
36      begin : Delay_process
37        if (reset == 1'b1) begin
38          Delay_reg[0] <= 16'b0000000000000000;
39          Delay_reg[1] <= 16'b0000000000000000;
40          Delay_reg[2] <= 16'b0000000000000000;
41          Delay_reg[3] <= 16'b0000000000000000;
42          Delay_reg[4] <= 16'b0000000000000000;
43          Delay_reg[5] <= 16'b0000000000000000;
44          Delay_reg[6] <= 16'b0000000000000000;
45          Delay_reg[7] <= 16'b0000000000000000;
46          Delay_reg[8] <= 16'b0000000000000000;
47          Delay_reg[9] <= 16'b0000000000000000;
48          Delay_reg[10] <= 16'b0000000000000000;
49          Delay_reg[11] <= 16'b0000000000000000;
50          Delay_reg[12] <= 16'b0000000000000000;
51          Delay_reg[13] <= 16'b0000000000000000;
52          Delay_reg[14] <= 16'b0000000000000000;
53          Delay_reg[15] <= 16'b0000000000000000;
```

```
54            Delay_reg[16] <= 16'b0000000000000000;
55            Delay_reg[17] <= 16'b0000000000000000;
56            Delay_reg[18] <= 16'b0000000000000000;
57          end
58          else begin
59            Delay_reg[0] <= Delay_reg_next[0];
60            Delay_reg[1] <= Delay_reg_next[1];
61            Delay_reg[2] <= Delay_reg_next[2];
62            Delay_reg[3] <= Delay_reg_next[3];
63            Delay_reg[4] <= Delay_reg_next[4];
64            Delay_reg[5] <= Delay_reg_next[5];
65            Delay_reg[6] <= Delay_reg_next[6];
66            Delay_reg[7] <= Delay_reg_next[7];
67            Delay_reg[8] <= Delay_reg_next[8];
68            Delay_reg[9] <= Delay_reg_next[9];
69            Delay_reg[10] <= Delay_reg_next[10];
70            Delay_reg[11] <= Delay_reg_next[11];
71            Delay_reg[12] <= Delay_reg_next[12];
72            Delay_reg[13] <= Delay_reg_next[13];
73            Delay_reg[14] <= Delay_reg_next[14];
74            Delay_reg[15] <= Delay_reg_next[15];
75            Delay_reg[16] <= Delay_reg_next[16];
76            Delay_reg[17] <= Delay_reg_next[17];
77        Delay_reg[18] <= Delay_reg_next[18];
78          end
```

```
79        end
80      assign Delay_out1 = Delay_reg[18];
81      assign Delay_reg_next[0] = DataIn;
82      assign Delay_reg_next[1] = Delay_reg[0];
83      assign Delay_reg_next[2] = Delay_reg[1];
84      assign Delay_reg_next[3] = Delay_reg[2];
85      assign Delay_reg_next[4] = Delay_reg[3];
86      assign Delay_reg_next[5] = Delay_reg[4];
87      assign Delay_reg_next[6] = Delay_reg[5];
88      assign Delay_reg_next[7] = Delay_reg[6];
89      assign Delay_reg_next[8] = Delay_reg[7];
90      assign Delay_reg_next[9] = Delay_reg[8];
91      assign Delay_reg_next[10] = Delay_reg[9];
92      assign Delay_reg_next[11] = Delay_reg[10];
93      assign Delay_reg_next[12] = Delay_reg[11];
94      assign Delay_reg_next[13] = Delay_reg[12];
95      assign Delay_reg_next[14] = Delay_reg[13];
96      assign Delay_reg_next[15] = Delay_reg[14];
97      assign Delay_reg_next[16] = Delay_reg[15];
98      assign Delay_reg_next[17] = Delay_reg[16];
99      assign Delay_reg_next[18] = Delay_reg[17];
100     Sqrt u_Sqrt (.din(Delay_out1),  // uint16
101                  .dout(Sqrt_out1)  // ufix19_En10
102                  );
103     assign DataOut = Sqrt_out1;
```

```
104 endmodule  // model
```

I.1.6 Monitor

```
1 /*
2 * Description: Monitor checks the result from
3 *        DUT with the values in scoreboard.
4 *
5 * Author:  Vyoma Sharma
6 *
7 */
8 class monitor;
9 int errcount;
10 int count;
11 scoreboard sb;
12 virtual intf intf_DUT;
13 virtual intf intf_model;
14
15 covergroup inputcov@(negedge intf_DUT.clk);
16  Inputdata:  coverpoint intf_DUT.DataIn{option.auto_bin_max =
      65536;}
17 endgroup
18  function new (virtual intf intf_DUT, virtual intf intf_model,
      scoreboard sb);
19   begin
20        this.intf_DUT = intf_DUT;
```

```
21          this.intf_model = intf_model;
22        this.inputcov= new();
23          this.sb = sb;
24        errcount = 0;
25        count = 1;
26    end
27  endfunction
28    task check();
29        forever @ (posedge intf_DUT.clk) begin
30      if(intf_model.DataOut === intf_DUT.DataOut) begin
31    end else begin
32    $display(" * ERROR * %t : Expected value : %h Actual value :
33         %h",$time, intf_model.DataOut, intf_DUT.DataOut);
34    errcount = errcount + 1;
35    end
36    end
37    endtask
38
39    task cov_check(); begin
40      while(1) begin
41          @ (posedge intf_DUT.clk) begin
42        if (((count++)%32767) == 0) begin
43        if(inputcov.get_coverage()==100)begin
44        showcoverage();
45        $stop;
```

```
46                    end
47                  end
48                end
49              end
50            end
51        endtask
52
53    task showcoverage();
54    $display("COVERAGE REPORT");
55    $display("DUT input coverage:%f",inputcov.get_coverage());
56    $display("Detected Error Count:%f",errcount);
57    $display("——————————————————");
58    endtask
59 endclass
```

I.1.7 Test Bench

```
1 /*
2 *Description: Interface controls the
3 *             execution of the system
4 *Author: Vyoma Sharma
5 *
6 */
7 program testbench(intf intf_DUT, intf intf_model);
8
9       environment env;
```

```
10
11        initial begin
12
13        env = new(intf_DUT, intf_model);
14
15        env.drvr.reset();
16
17        env.drvr.drive(1000000000);
18      end
19 endprogram
```

I.2 Restoring Algorithm

I.2.1 Restoring Algorithm Operation

```
1 /*
2 *
3 *Description: Restoring Algorithm implementation
4 *for Square-root computation for a single stage
5 *
6 *Author: Vyoma Sharma
7 *
8 */
9
10 module SqrtStage(DI,QI,RI,DO,QO,RO);
11
```

```
12 parameter StageNum = 20;
13
14 input wire [15:0] DI;
15 input wire [StageNum:0] QI;
16 input wire [2*StageNum:0] RI;
17 output wire [15:0] DO;
18 output wire [StageNum:0] QO;
19 output wire [2*StageNum:0] RO;
20
21 wire condition;
22 wire [2*StageNum:0] rRestd;
23 wire [StageNum:0] qRestd;
24
25 // Restoring Operation
26   assign condition = RI >> (2*StageNum-2) ;
27   assign qRestd = condition ? {QI[StageNum:1], 1'b0}: QI;
28   assign rRestd = condition ? RI + {qRestd,1'b1}: RI;
29
30 //Rest of algorithm for unconditional subtract for qn = 1'b1
31 assign DO = {DI[13:0], 2'b00};
32 assign RO = {rRestd, DI[15:14]} - {qRestd, 2'b01};
33 assign QO = {qRestd, 1'b1};
34 endmodule // SQRTRST
```

I.2.2 Non-Restoring Algorithm with Pipeline architecture

```
1  /*
2  *
3  *Description: Restoring Algorithm implementation
4  *for Square-root computation with pipeline architecture
5  *
6  *Author: Vyoma Sharma
7  *
8  */
9  `define tscale `timescale 1ns/1ns
10 module SQRTRST (
11     clk,
12     reset,
13     DataIn,
14     DataOut,
15     test_mode,
16     scan_in0,
17     scan_en,
18     scan_out0
19        );
20   input
21     reset,        // system reset
22     clk;          // system clock
23   input
24     scan_in0,     // test scan mode data input
25         scan_en,     // test scan mode enable
```

```
26        test_mode;  // test mode select
27     output
28        scan_out0;  // test scan mode data output
29     input
30        wire [15:0] DataIn;
31     output
32        reg [18:0] DataOut;
33  // Local Interconnect Definitions for Stage 1
34        reg   [1:0]   QI1;
35        reg   [2:0]   RI1;
36        reg   [15:0]  DI1;
37        wire  [15:0]  DO1;
38        wire  [1:0]   QO1;
39        wire  [2:0]   RO1;
40  // Local Interconnect Definitions for Stage 2
41        reg     [2:0]  QI2;
42        reg     [4:0]  RI2;
43        reg     [15:0]   DI2;
44        wire    [15:0]   DO2;
45        wire    [2:0]  QO2;
46        wire    [4:0]  RO2;
47  // Local Interconnect Definitions for Stage 3
48        reg     [3:0]  QI3;
49        reg     [6:0]  RI3;
50        reg     [15:0]   DI3;
```

```
51      wire   [15:0]   DO3;
52      wire   [3:0]  QO3;
53      wire   [6:0]  RO3;
54  // Local Interconnect Definitions for Stage 4
55      reg    [4:0]  QI4;
56      reg    [8:0]  RI4;
57      reg    [15:0]    DI4;
58      wire   [15:0]    DO4;
59      wire   [4:0]    QO4;
60      wire   [8:0]  RO4;
61  // Local Interconnect Definitions for Stage 5
62      reg    [5:0]   QI5;
63      reg    [10:0]   RI5  ;
64      reg    [15:0]   DI5  ;
65      wire   [15:0]   DO5  ;
66      wire   [5:0]  QO5  ;
67      wire   [10:0]   RO5  ;
68  // Local Interconnect Definitions for Stage 6
69      reg    [6:0]    QI6  ;
70      reg    [12:0]   RI6  ;
71      reg    [15:0]   DI6  ;
72      wire   [15:0]   DO6  ;
73      wire     [6:0]    QO6  ;
74      wire   [12:0]   RO6  ;
75  // Local Interconnect Definitions for Stage 7
```

```
76      reg   [7:0]  QI7 ;
77      reg   [14:0]  RI7 ;
78      reg   [15:0]  DI7 ;
79      wire  [15:0]  DO7 ;
80      wire  [7:0] QO7 ;
81      wire  [14:0]  RO7 ;
82 // Local Interconnect Definitions for Stage 8
83      reg   [8:0] QI8 ;
84      reg   [16:0]  RI8 ;
85      reg   [15:0]  DI8 ;
86      wire  [15:0]  DO8 ;
87      wire  [8:0] QO8 ;
88      wire  [16:0]  RO8 ;
89 // Local Interconnect Definitions for Stage 9
90      reg   [9:0] QI9 ;
91      reg   [18:0]  RI9 ;
92      reg   [15:0]  DI9 ;
93      wire  [15:0]  DO9 ;
94      wire  [9:0] QO9 ;
95      wire  [18:0]  RO9 ;
96 // Local Interconnect Definitions for Stage 10
97      reg   [10:0]  QI10 ;
98      reg   [20:0]  RI10 ;
99      reg   [15:0]  DI10 ;
100     wire  [15:0]  DO10 ;
```

```
101      wire   [10:0]   QO10 ;
102      wire   [20:0]   RO10 ;
103 // Local Interconnect Definitions for Stage 11
104      reg    [11:0]   QI11 ;
105      reg    [22:0]   RI11 ;
106      reg    [15:0]   DI11 ;
107      wire   [15:0]   DO11 ;
108      wire   [11:0]   QO11 ;
109      wire   [22:0]   RO11 ;
110 // Local Interconnect Definitions for Stage 12
111      reg    [12:0]   QI12 ;
112      reg    [24:0]   RI12 ;
113      reg    [15:0]   DI12 ;
114      wire   [15:0]   DO12 ;
115      wire   [12:0]   QO12 ;
116      wire   [24:0]   RO12 ;
117 // Local Interconnect Definitions for Stage 13
118      reg    [13:0]   QI13 ;
119      reg    [26:0]   RI13 ;
120      reg    [15:0]   DI13 ;
121      wire   [15:0]   DO13 ;
122      wire   [13:0]   QO13 ;
123      wire   [26:0]   RO13 ;
124 // Local Interconnect Definitions for Stage 14
125      reg    [14:0]   QI14 ;
```

```
126     reg    [28:0]  RI14 ;
127     reg    [15:0]  DI14 ;
128     wire   [15:0]  DO14 ;
129     wire   [14:0]  QO14 ;
130     wire   [28:0]  RO14 ;
131 // Local Interconnect Definitions for Stage 15
132     reg    [15:0]  QI15 ;
133     reg    [30:0]  RI15 ;
134     reg    [15:0]  DI15 ;
135     wire   [15:0]  DO15 ;
136     wire   [15:0]  QO15 ;
137     wire   [30:0]  RO15 ;
138 // Local Interconnect Definitions for Stage 16
139     reg    [16:0]  QI16 ;
140     reg    [32:0]  RI16 ;
141     reg    [15:0]  DI16 ;
142     wire   [15:0]  DO16 ;
143     wire   [16:0]  QO16 ;
144     wire   [32:0]  RO16 ;
145 // Local Interconnect Definitions for Stage 17
146     reg    [17:0]  QI17 ;
147     reg    [34:0]  RI17 ;
148     reg    [15:0]  DI17 ;
149     wire   [15:0]  DO17 ;
150     wire   [17:0]  QO17 ;
```

```
151     wire  [34:0]  RO17 ;
152 // Local Interconnect Definitions for Stage 18
153     reg   [18:0]  QI18 ;
154     reg   [36:0]  RI18 ;
155     reg   [15:0]  DI18 ;
156     wire  [15:0]  DO18 ;
157     wire  [18:0]  QO18 ;
158     wire  [36:0]  RO18 ;
159   SqrtStage #(1) Stage1 (
160           .DI(DI1),
161           .QI(QI1),
162           .RI(RI1),
163           .DO(DO1),
164           .QO(QO1),
165           .RO(RO1)   );
166   SqrtStage #(2) Stage2 (
167           .DI(DI2),
168           .QI(QI2),
169           .RI(RI2),
170           .DO(DO2),
171           .QO(QO2),
172           .RO(RO2)   );
173   SqrtStage #(3) Stage3 (
174           .DI(DI3),
175           .QI(QI3),
```

```
176             .RI(RI3),
177             .DO(DO3),
178             .QO(QO3),
179             .RO(RO3)    );
180     SqrtStage #(4) Stage4 (
181             .DI(DI4),
182             .QI(QI4),
183             .RI(RI4),
184             .DO(DO4),
185             .QO(QO4),
186             .RO(RO4)    );
187     SqrtStage #(5) Stage5 (
188             .DI(DI5),
189             .QI(QI5),
190             .RI(RI5),
191             .DO(DO5),
192             .QO(QO5),
193             .RO(RO5)    );
194     SqrtStage #(6) Stage6 (
195             .DI(DI6),
196             .QI(QI6),
197             .RI(RI6),
198             .DO(DO6),
199             .QO(QO6),
200             .RO(RO6)    );
```

```
201  SqrtStage #(7) Stage7 (
202          .DI(DI7),
203          .QI(QI7),
204          .RI(RI7),
205          .DO(DO7),
206          .QO(QO7),
207          .RO(RO7)   );
208  SqrtStage #(8) Stage8 (
209          .DI(DI8),
210          .QI(QI8),
211          .RI(RI8),
212          .DO(DO8),
213          .QO(QO8),
214          .RO(RO8)   );
215  SqrtStage #(9) Stage9 (
216          .DI(DI9),
217          .QI(QI9),
218          .RI(RI9),
219          .DO(DO9),
220          .QO(QO9),
221          .RO(RO9)   );
222  SqrtStage #(10) Stage10 (
223          .DI(DI10),
224          .QI(QI10),
225          .RI(RI10),
```

```
226             .DO(DO10),
227             .QO(QO10),
228             .RO(RO10)     );
229     SqrtStage #(11) Stage11 (
230             .DI(DI11),
231             .QI(QI11),
232             .RI(RI11),
233             .DO(DO11),
234             .QO(QO11),
235             .RO(RO11)     );
236     SqrtStage #(12) Stage12 (
237             .DI(DI12),
238             .QI(QI12),
239             .RI(RI12),
240             .DO(DO12),
241             .QO(QO12),
242             .RO(RO12)     );
243     SqrtStage #(13) Stage13 (
244             .DI(DI13),
245             .QI(QI13),
246             .RI(RI13),
247             .DO(DO13),
248             .QO(QO13),
249             .RO(RO13)     );
250     SqrtStage #(14) Stage14 (
```

```
251             .DI(DI14),
252             .QI(QI14),
253             .RI(RI14),
254             .DO(DO14),
255             .QO(QO14),
256             .RO(RO14)      );
257     SqrtStage #(15) Stage15 (
258             .DI(DI15),
259             .QI(QI15),
260             .RI(RI15),
261             .DO(DO15),
262             .QO(QO15),
263             .RO(RO15)      );
264     SqrtStage #(16) Stage16 (
265             .DI(DI16),
266             .QI(QI16),
267             .RI(RI16),
268             .DO(DO16),
269             .QO(QO16),
270             .RO(RO16)      );
271     SqrtStage #(17) Stage17 (
272             .DI(DI17),
273             .QI(QI17),
274             .RI(RI17),
275             .DO(DO17),
```

```
276             .QO(QO17),
277             .RO(RO17)      );
278    SqrtStage #(18) Stage18 (
279             .DI(DI18),
280             .QI(QI18),
281             .RI(RI18),
282             .DO(DO18),
283             .QO(QO18),
284             .RO(RO18)      );
285    always @( posedge clk, posedge reset ) begin
286      if(reset == 1'b1) begin
287        DataOut <= 0;
288        QI1 <= 'b0;
289        RI1 <= 'b0;
290        DI1 <= 'b0;
291        DI2 <= 'b0;
292        QI2 <= 'b0; //Quotient Input
293        RI2 <= 'b0; //Remainder Input
294        QI3 <= 'b0;
295        RI3 <= 'b0;
296        DI3 <= 'b0;
297        QI4 <= 'b0;
298        RI4 <= 'b0;
299        DI4 <= 'b0;
300        QI5 <= 'b0;
```

```
301         RI5  <=  'b0;
302         DI5  <=  'b0;
303         QI6  <=  'b0;
304         RI6  <=  'b0;
305         DI6  <=  'b0;
306         QI7  <=  'b0;
307         RI7  <=  'b0;
308         DI7  <=  'b0;
309         QI8  <=  'b0;
310         RI8  <=  'b0;
311         DI8  <=  'b0;
312         QI9  <=  'b0;
313         RI9  <=  'b0;
314         DI9  <=  'b0;
315         QI10  <=  'b0;
316         RI10  <=  'b0;
317         DI10  <=  'b0;
318         QI11  <=  'b0;
319         RI11  <=  'b0;
320         DI11  <=  'b0;
321         QI12  <=  'b0;
322         RI12  <=  'b0;
323         DI12  <=  'b0;
324         QI13  <=  'b0;
325         RI13  <=  'b0;
```

```
326         DI13 <= 'b0;
327         QI14 <= 'b0;
328         RI14 <= 'b0;
329         DI14 <= 'b0;
330         QI15 <= 'b0;
331         RI15 <= 'b0;
332         DI15 <= 'b0;
333         QI16 <= 'b0;
334         RI16 <= 'b0;
335         DI16 <= 'b0;
336         QI17 <= 'b0;
337         RI17 <= 'b0;
338         DI17 <= 'b0;
339         QI18 <= 'b0;
340         RI18 <= 'b0;
341         DI18 <= 'b0;
342       end else begin
343         QI1 <= 1'b0    ;
344         RI1 <= 2'b00   ;
345         DI1 <= DataIn  ;
346         // Data Handoff to Stage 2
347         QI2 <= QO1   ;
348         RI2 <= RO1   ;
349         DI2 <= DO1   ;
350         // Data Handoff to Stage 3
```

```
351        QI3  <=  QO2    ;
352        RI3  <=  RO2    ;
353        DI3  <=  DO2    ;
354        // Data Handoff to Stage 4
355        QI4  <=  QO3    ;
356        RI4  <=  RO3    ;
357        DI4  <=  DO3    ;
358        // Data Handoff to Stage 5
359        QI5  <=  QO4    ;
360        RI5  <=  RO4    ;
361        DI5  <=  DO4    ;
362        // Data Handoff to Stage 6
363        QI6  <=  QO5    ;
364        RI6  <=  RO5    ;
365        DI6  <=  DO5    ;
366        // Data Handoff to Stage 7
367        QI7  <=  QO6    ;
368        RI7  <=  RO6    ;
369        DI7  <=  DO6    ;
370        // Data Handoff to Stage 8
371        QI8  <=  QO7    ;
372        RI8  <=  RO7    ;
373        DI8  <=  DO7    ;
374        // Data Handoff to Stage 9
375        QI9  <=  QO8    ;
```

```
376         RI9  <=  RO8    ;
377         DI9  <=  DO8    ;
378         // Data Handoff to Stage 10
379         QI10  <=  QO9  ;
380         RI10  <=  RO9  ;
381         DI10  <=  DO9  ;
382         // Data Handoff to Stage 11
383         QI11  <=  QO10    ;
384         RI11  <=  RO10    ;
385         DI11  <=  DO10    ;
386         // Data Handoff to Stage 12
387         QI12  <=  QO11    ;
388         RI12  <=  RO11    ;
389         DI12  <=  DO11    ;
390         // Data Handoff to Stage 13
391         QI13  <=  QO12    ;
392         RI13  <=  RO12    ;
393         DI13  <=  DO12    ;
394         // Data Handoff to Stage 14
395         QI14  <=  QO13    ;
396         RI14  <=  RO13    ;
397         DI14  <=  DO13    ;
398         // Data Handoff to Stage 15
399         QI15  <=  QO14    ;
400         RI15  <=  RO14    ;
```

```
401          DI15 <= DO14   ;
402          // Data Handoff to Stage 16
403          QI16 <= QO15   ;
404          RI16 <= RO15   ;
405          DI16 <= DO15   ;
406          // Data Handoff to Stage 17
407          QI17 <= QO16   ;
408          RI17 <= RO16   ;
409          DI17 <= DO16   ;
410          // Data Handoff to Stage 18
411          QI18 <= QO17   ;
412          RI18 <= RO17   ;
413          DI18 <= DO17   ;
414          DataOut = RO18[34] ? QO18 - 1'b1 : QO18 ;
415        end
416     end
417 endmodule // SQRTNRST
```

I.2.3 Environment

```
1 /*
2 * Description: Parent class which contains the
3 *              instances of the test bench component.
4 *
5 *Author: Vyoma Sharma
6 *
```

```
7  *
8  */
9      class environment;
10             driver drvr;
11             scoreboard sb;
12             monitor mntr;
13     function new(virtual intf intfvalDUT, virtual intf
           intfvalModel);
14            sb = new();
15            drvr = new(intfvalDUT, intfvalModel, sb);
16          mntr = new(intfvalDUT, intfvalModel, sb);
17            fork
18             mntr.check();
19         mntr.cov_check();
20            join_none
21            endfunction
22
23      endclass
```

I.2.4 Driver

```
1  /*
2  *Description: Driver reads the random variable
3  *from the stimulus and gives it to the model.
4  *It reads the value from stimulus and writes
5  *the data in DUT based on clock.
```

```
6  *
7  *Author: Vyoma Sharma
8  *
9  */
10
11   class driver;
12
13         stimulus sti;
14         scoreboard sb;
15         virtual intf intf_DUT;
16
17     virtual intf intf_model;
18
19         function new;
20       input virtual intf intf_DUT;
21       input virtual intf intf_model;
22       input scoreboard sb;
23       begin
24          this.intf_DUT = intf_DUT;
25          this.intf_model = intf_model;
26              this.sb = sb;
27          this.sti = new();
28         end
29
30       endfunction
```

```
31
32          task reset();
33
34         intf_DUT.DataIn = 0;
35         intf_model.DataIn = 0;
36               @ (negedge intf_DUT.clk);
37               intf_DUT.reset = 0;
38         intf_model.reset = 0;
39               @ (negedge intf_DUT.clk);
40               intf_DUT.reset = 1;
41         intf_model.reset = 1;
42               @ (negedge intf_DUT.clk);
43               intf_DUT.reset = 0;
44         intf_model.reset = 0;
45
46          endtask
47
48          task drive(input integer iteration);
49        int loc;
50        repeat (iteration) begin
51        loc = sti.randomize();
52       @(negedge intf_DUT.clk);
53        intf_DUT.DataIn = sti.data;
54        intf_model.DataIn = sti.data;
55
```

```
56          end
57
58        endtask
59      endclass
```

I.2.5 Model

```
1  /*
2  *
3  *File Name: hdlsrc\model\model.v
4  *
5  * Generated by MATLAB 9.2 and HDL Coder 3.10
6  *
7  *
8  * -- ---------------------------------------------
9  * -- Rate and Clocking Details
10 * -- ---------------------------------------------
11 * Model base rate: 0.2
12 * Target subsystem base rate: 0.2
13 *
14 * ---------------------------------------------
15 * ---------------------------------------------
16 *
17 * Module: model
18 * Source Path: model
19 * Hierarchy Level: 0
```

```
20  *
21  */
22  module model
23              (clk,
24               reset,
25               DataIn,
26               DataOut);
27    input   clk;
28    input   reset;
29    input   [15:0] DataIn;  // uint16
30    output  [18:0] DataOut;  // ufix19_En10
31    reg [15:0] Delay_reg [0:18];  // ufix16 [18]
32    wire [15:0] Delay_reg_next [0:18];  // ufix16 [18]
33    wire [15:0] Delay_out1;  // uint16
34    wire [18:0] Sqrt_out1;  // ufix19_En10
35    always @(posedge clk or posedge reset)
36      begin : Delay_process
37        if (reset == 1'b1) begin
38          Delay_reg[0] <= 16'b0000000000000000;
39          Delay_reg[1] <= 16'b0000000000000000;
40          Delay_reg[2] <= 16'b0000000000000000;
41          Delay_reg[3] <= 16'b0000000000000000;
42          Delay_reg[4] <= 16'b0000000000000000;
43          Delay_reg[5] <= 16'b0000000000000000;
44          Delay_reg[6] <= 16'b0000000000000000;
```

```
45              Delay_reg[7] <= 16'b0000000000000000;
46              Delay_reg[8] <= 16'b0000000000000000;
47              Delay_reg[9] <= 16'b0000000000000000;
48              Delay_reg[10] <= 16'b0000000000000000;
49              Delay_reg[11] <= 16'b0000000000000000;
50              Delay_reg[12] <= 16'b0000000000000000;
51              Delay_reg[13] <= 16'b0000000000000000;
52              Delay_reg[14] <= 16'b0000000000000000;
53              Delay_reg[15] <= 16'b0000000000000000;
54              Delay_reg[16] <= 16'b0000000000000000;
55              Delay_reg[17] <= 16'b0000000000000000;
56              Delay_reg[18] <= 16'b0000000000000000;
57           end
58           else begin
59              Delay_reg[0] <= Delay_reg_next[0];
60              Delay_reg[1] <= Delay_reg_next[1];
61              Delay_reg[2] <= Delay_reg_next[2];
62              Delay_reg[3] <= Delay_reg_next[3];
63              Delay_reg[4] <= Delay_reg_next[4];
64              Delay_reg[5] <= Delay_reg_next[5];
65              Delay_reg[6] <= Delay_reg_next[6];
66              Delay_reg[7] <= Delay_reg_next[7];
67              Delay_reg[8] <= Delay_reg_next[8];
68              Delay_reg[9] <= Delay_reg_next[9];
69              Delay_reg[10] <= Delay_reg_next[10];
```

```
70            Delay_reg[11] <= Delay_reg_next[11];
71            Delay_reg[12] <= Delay_reg_next[12];
72            Delay_reg[13] <= Delay_reg_next[13];
73            Delay_reg[14] <= Delay_reg_next[14];
74            Delay_reg[15] <= Delay_reg_next[15];
75            Delay_reg[16] <= Delay_reg_next[16];
76            Delay_reg[17] <= Delay_reg_next[17];
77        Delay_reg[18] <= Delay_reg_next[18];
78          end
79        end
80      assign Delay_out1 = Delay_reg[18];
81      assign Delay_reg_next[0] = DataIn;
82      assign Delay_reg_next[1] = Delay_reg[0];
83      assign Delay_reg_next[2] = Delay_reg[1];
84      assign Delay_reg_next[3] = Delay_reg[2];
85      assign Delay_reg_next[4] = Delay_reg[3];
86      assign Delay_reg_next[5] = Delay_reg[4];
87      assign Delay_reg_next[6] = Delay_reg[5];
88      assign Delay_reg_next[7] = Delay_reg[6];
89      assign Delay_reg_next[8] = Delay_reg[7];
90      assign Delay_reg_next[9] = Delay_reg[8];
91      assign Delay_reg_next[10] = Delay_reg[9];
92      assign Delay_reg_next[11] = Delay_reg[10];
93      assign Delay_reg_next[12] = Delay_reg[11];
94      assign Delay_reg_next[13] = Delay_reg[12];
```

```
95    assign Delay_reg_next[14] = Delay_reg[13];
96    assign Delay_reg_next[15] = Delay_reg[14];
97    assign Delay_reg_next[16] = Delay_reg[15];
98    assign Delay_reg_next[17] = Delay_reg[16];
99    assign Delay_reg_next[18] = Delay_reg[17];
100   Sqrt u_Sqrt (.din(Delay_out1),  // uint16
101                .dout(Sqrt_out1)  // ufix19_En10
102                );
103   assign DataOut = Sqrt_out1;
104 endmodule  // model
```

I.2.6 Monitor

```
1  /*
2  *
3  * File Name: hdlsrc\model\model.v
4  *
5  * Generated by MATLAB 9.2 and HDL Coder 3.10
6  *
7  *
8  * -- ------------------------------------------------
9  * -- Rate and Clocking Details
10 * -- ------------------------------------------------
11 * Model base rate: 0.2
12 * Target subsystem base rate: 0.2
13 *
```

```
14 * ————————————————————————————————
15 * ————————————————————————————————
16 *
17 * Module: model
18 * Source Path: model
19 * Hierarchy Level: 0
20 *
21 */
22 module model
23           (clk,
24            reset,
25            DataIn,
26            DataOut);
27   input   clk;
28   input   reset;
29   input   [15:0] DataIn;  // uint16
30   output  [18:0] DataOut;  // ufix19_En10
31   reg [15:0] Delay_reg [0:18];  // ufix16 [18]
32   wire [15:0] Delay_reg_next [0:18];  // ufix16 [18]
33   wire [15:0] Delay_out1;  // uint16
34   wire [18:0] Sqrt_out1;  // ufix19_En10
35   always @(posedge clk or posedge reset)
36     begin : Delay_process
37       if (reset == 1'b1) begin
38         Delay_reg[0] <= 16'b0000000000000000;
```

```
39          Delay_reg[1] <= 16'b0000000000000000;
40          Delay_reg[2] <= 16'b0000000000000000;
41          Delay_reg[3] <= 16'b0000000000000000;
42          Delay_reg[4] <= 16'b0000000000000000;
43          Delay_reg[5] <= 16'b0000000000000000;
44          Delay_reg[6] <= 16'b0000000000000000;
45          Delay_reg[7] <= 16'b0000000000000000;
46          Delay_reg[8] <= 16'b0000000000000000;
47          Delay_reg[9] <= 16'b0000000000000000;
48          Delay_reg[10] <= 16'b0000000000000000;
49          Delay_reg[11] <= 16'b0000000000000000;
50          Delay_reg[12] <= 16'b0000000000000000;
51          Delay_reg[13] <= 16'b0000000000000000;
52          Delay_reg[14] <= 16'b0000000000000000;
53          Delay_reg[15] <= 16'b0000000000000000;
54          Delay_reg[16] <= 16'b0000000000000000;
55          Delay_reg[17] <= 16'b0000000000000000;
56          Delay_reg[18] <= 16'b0000000000000000;
57        end
58        else begin
59          Delay_reg[0] <= Delay_reg_next[0];
60          Delay_reg[1] <= Delay_reg_next[1];
61          Delay_reg[2] <= Delay_reg_next[2];
62          Delay_reg[3] <= Delay_reg_next[3];
63          Delay_reg[4] <= Delay_reg_next[4];
```

```
64            Delay_reg[5] <= Delay_reg_next[5];
65            Delay_reg[6] <= Delay_reg_next[6];
66            Delay_reg[7] <= Delay_reg_next[7];
67            Delay_reg[8] <= Delay_reg_next[8];
68            Delay_reg[9] <= Delay_reg_next[9];
69            Delay_reg[10] <= Delay_reg_next[10];
70            Delay_reg[11] <= Delay_reg_next[11];
71            Delay_reg[12] <= Delay_reg_next[12];
72            Delay_reg[13] <= Delay_reg_next[13];
73            Delay_reg[14] <= Delay_reg_next[14];
74            Delay_reg[15] <= Delay_reg_next[15];
75            Delay_reg[16] <= Delay_reg_next[16];
76            Delay_reg[17] <= Delay_reg_next[17];
77        Delay_reg[18] <= Delay_reg_next[18];
78          end
79        end
80      assign Delay_out1 = Delay_reg[18];
81      assign Delay_reg_next[0] = DataIn;
82      assign Delay_reg_next[1] = Delay_reg[0];
83      assign Delay_reg_next[2] = Delay_reg[1];
84      assign Delay_reg_next[3] = Delay_reg[2];
85      assign Delay_reg_next[4] = Delay_reg[3];
86      assign Delay_reg_next[5] = Delay_reg[4];
87      assign Delay_reg_next[6] = Delay_reg[5];
88      assign Delay_reg_next[7] = Delay_reg[6];
```

```
89    assign Delay_reg_next[8] = Delay_reg[7];
90    assign Delay_reg_next[9] = Delay_reg[8];
91    assign Delay_reg_next[10] = Delay_reg[9];
92    assign Delay_reg_next[11] = Delay_reg[10];
93    assign Delay_reg_next[12] = Delay_reg[11];
94    assign Delay_reg_next[13] = Delay_reg[12];
95    assign Delay_reg_next[14] = Delay_reg[13];
96    assign Delay_reg_next[15] = Delay_reg[14];
97    assign Delay_reg_next[16] = Delay_reg[15];
98    assign Delay_reg_next[17] = Delay_reg[16];
99    assign Delay_reg_next[18] = Delay_reg[17];
100   Sqrt u_Sqrt (.din(Delay_out1),  // uint16
101               .dout(Sqrt_out1)  // ufix19_En10
102               );
103   assign DataOut = Sqrt_out1;
104 endmodule  // model
```

I.2.7 Test Bench

```
1 /*
2 *Description: Interface controls the
3 *             execution of the system
4 *Author: Vyoma Sharma
5 *
6 */
7 program testbench(intf intf_DUT, intf intf_model);
```

```
8
9        environment env;
10
11       initial begin
12
13       env = new(intf_DUT, intf_model);
14
15       env.drvr.reset();
16
17       env.drvr.drive(1000000000);
18     end
19 endprogram
```