2006

Pipelining Of Double Precision Floating Point Division And Square Root Operations On Field-programmable Gate Arrays

Anuja Thakkar
University of Central Florida

PIPELINING OF DOUBLE PRECISION FLOATING POINT DIVISION AND SQUARE ROOT OPERATIONS ON FIELD-PROGRAMMABLE GATE ARRAYS

by

ANUJA JAYRAJ THAKKAR
B.S. Walchand Institute of Technology, India, 2002

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2006

ABSTRACT

Many space applications, such as vision-based systems, synthetic aperture radar, and radar altimetry rely increasingly on high data rate DSP algorithms. These algorithms use double precision floating point arithmetic operations. While most DSP applications can be executed on DSP processors, the DSP numerical requirements of these new space applications surpass by far the numerical capabilities of many current DSP processors. Since the tradition in DSP processing has been to use fixed point number representation, only recently have DSP processors begun to incorporate floating point arithmetic units, even though most of these units handle only single precision floating point addition/subtraction, multiplication, and occasionally division.

While DSP processors are slowly evolving to meet the numerical requirements of newer space applications, FPGA densities have rapidly increased to parallel and surpass even the gate densities of many DSP processors and commodity CPUs. This makes them attractive platforms to implement compute-intensive DSP computations. Even in the presence of this clear advantage on the side of FPGAs, few attempts have been made to examine how wide precision floating point arithmetic, particularly division and square root operations, can perform on FPGAs to support these compute-intensive DSP applications.

In this context, this thesis presents the sequential and pipelined designs of IEEE-754 compliant double floating point division and square root operations based on low radix digit recurrence algorithms. FPGA implementations of these algorithms have the advantage of being easily testable. In particular, the pipelined designs are synthesized based on careful partial and full unrolling of the iterations in the digit recurrence algorithms. In the overall, the implementations of the sequential and pipelined designs are *common-denominator*

implementations which do not use any performance-enhancing embedded components such as multipliers and block memory. As these implementations exploit exclusively the fine-grain reconfigurable resources of Virtex FPGAs, they are easily portable to other FPGAs with similar reconfigurable fabrics without any major modifications. The pipelined designs of these two operations are evaluated in terms of area, throughput, and dynamic power consumption as a function of pipeline depth. Pipelining experiments reveal that the area overhead tends to remain constant regardless of the degree of pipelining to which the design is submitted, while the throughput increases with pipeline depth. In addition, these experiments reveal that pipelining reduces power considerably in shallow pipelines. Pipelining further these designs does not necessarily lead to significant power reduction. By partitioning these designs into deeper pipelines, these designs can reach throughputs close to the 100 MFLOPS mark by consuming a modest 1% to 8% of the reconfigurable fabric within a Virtex-II XC2VX000 (e.g., XC2V1000 or XC2V6000) FPGA.

I dedicate this thesis to my parents, Sunita Shah and Late Rajendra Shah. They have always been an inspiration in my life. Mom and Dad, I dedicate all the success I have achieved or will achieve in future to both of you.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my advisor, Dr. Abdel Ejnioui, for his tireless assistance and endless contribution.

Thanks to the committee members Drs Ronald Demara and Brian Petrasko.

The realization of this work would not be possible without the support of my husband, Jayraj. Thanks for supporting me all along the way, for sacrifices above and beyond and for your unconditional love. I would also love to thank my baby to come, for being so patient and supportive when I was putting in long hours of work.

Thanks to GOD for guiding me all the way. I would not have achieved success without His guidance and blessings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

FPGA	Field Programmable Gate Array
DSP	Digital Signal Processing
SAR	Synthetic Aperture Radar
CCD	Charge Coupled Device
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
IEEE	Institute for Electrical and Electronics Engineers
CLB	Configurable Logic Block
SOP	Sum of Products
LUT	Look Up Table
IOB	Input Output Block
MFLOPS	Mega Floating Point Operations per Second
KFLOPS	Kilo Floating Point Operations per Second

CHAPTER ONE: INTRODUCTION

In space applications, many systems require advanced digital signal processing (DSP) algorithms to address their mission needs. In particular, an entire class of satellite sub-systems, such as payload processing, data-handling, communications, guidance, navigation, and control, rely on applications of DSP techniques. As these systems evolve, the amount of data which needs to be processed increases significantly.

1.1 DSP Space Applications

Data-intensive DSP algorithms make up the foundation of many space applications. Among these applications are vision-based systems, synthetic aperture radar (SAR), and radar altimetry [1].

1.1.1 Vision-Based Systems

These vision applications interface with the ambient space through a high-resolution charge-coupled device (CCD) camera. This camera is responsible of taking images at high speed and feed them to an image DSP processor. The latter performs numerous basic low-level image processing algorithms on the received image data. This low-level processing is followed by knowledge extraction from the images. Among these vision applications are visual telemetry (i.e., collection and compression of images of deployment of various instruments such as solar arrays and antennas), vision landing (i.e., guiding a planetary module to a safe landing without assistance from a human operator), rover vision (i.e., collection and transmission of images from a planetary body's surface, planning of navigation routes on the planetary surface by computer vision), rendezvous (i.e., detecting and navigating closer to a comet or asteroid), docking (i.e.,

approaching another spacecraft for docking maneuvers), and star tracking (i.e., tracking the position of one or more starts to determine the altitude and orientation of a spacecraft).

1.1.2 Space SAR

SAR processing transforms the raw radar data into a high resolution image of the terrain scanned by radar instruments. Many space systems rely on high continuous SAR data rates for the acquisition of high resolution images. These rates, combined with compute-intensive full-resolution image processing, make space SAR an application with unusually demanding requirements. Because of these high data rates, two approaches were adopted in the past. The first consists of avoiding processing data on-board of the space spacecraft by downloading it to a ground station for additional processing while the second approach consists of processing the assembled images at low resolution. Recently, new space applications have emerged with a high level of autonomy requirements. For instance, NASA, in collaboration with the Air Force Research Labs, is exploring space vehicles that can fly in formation by using an advanced space borne differential global positioning system [2]. It is expected that this technology will result in swarms of spacecrafts flying as a virtual platform to gather significantly more and better scientific data in a totally autonomous fashion. Such autonomy requires minimal or no support from ground stations. This autonomy can be realized only if sufficient computing power is available to process on-board the received raw SAR data at full resolution.

1.2 DSP Processors vs. FPGA Devices

As customary in many DSP applications, fixed point data representation is used in order to ease the computing requirements of these applications. Subsequently, various numerical analyses have to be completed to show that the conversion to fixed point representations does not

worsen cumulative errors associated with long runtime of DSP algorithms. However, newer DSP algorithms used to support many space applications require that the data format be represented in wide bit precision to accommodate various ranges of values and eliminate problems of numerical fidelity. To support these new applications, high performance, low power, computing devices which can produce high throughput floating point computations are needed. To meet these numerical requirements, many of these applications rely on the IEEE 754-1985 binary floating point standard [3]. In fact, many DSP algorithms, which support the space applications described in the previous section, are centered on calculations involving wide-ranging floating point numbers. These numbers are used in double precision bit widths in order to accommodate the range and precisions required by these DSP algorithms. While most DSP applications can be implemented on DSP processors, only recently have DSP processors been equipped with floating point ALUs. The majority of these floating point ALUs can handle at most 32-bit wide floating point numbers particularly, ALUs introduced in DSP processors intended for audio applications [4, 5]. Even when multiple DSP processors are cascaded for increased performance, there are computing scenarios in which these processor configurations, initially thought to be necessary to provide the required throughput, fails to deliver such throughput. In fact, duplication of processors does not necessarily lead to a speedup in computation as is known in parallel processing.

Lately, FPGAs have begun to capture the attention of the DSP community as an alternative implementation technology capable of delivering significant speedups for compute-intensive DSP algorithms. Despite dire predictions at each step in technology evolution, FPGA densities continue to double approximately every 18 months as predicted by Moore's Law. With today's nanometer CMOS technology, it is now possible to deliver multi-million gate FPGAs,

which can implement complete systems integrated entirely on a single device [6, 7]. Although FPGAs are still lagging behind many ASICs in terms of raw performance, they have nevertheless crossed a gate density threshold that is usually seen in DSP processors and commodity CPUs as shown in Figure 1, Figure 2 and Figure 3 [8, 9].

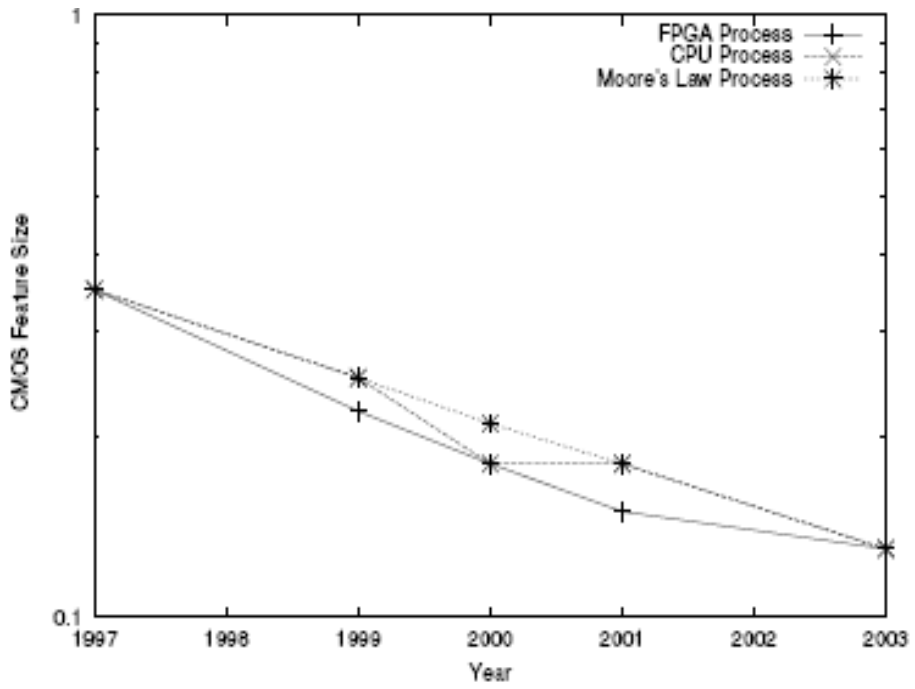

Figure 1: Expected trend in FPGA CMOS feature size.

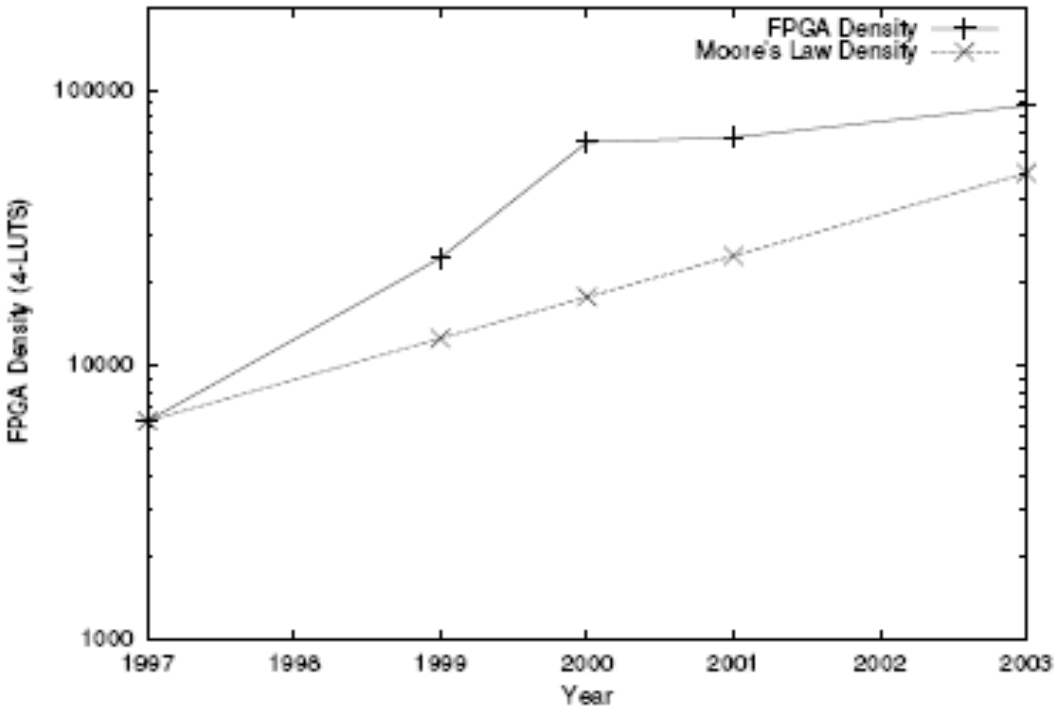

Figure 2: Expected FPGA trend in 4-LUT density.

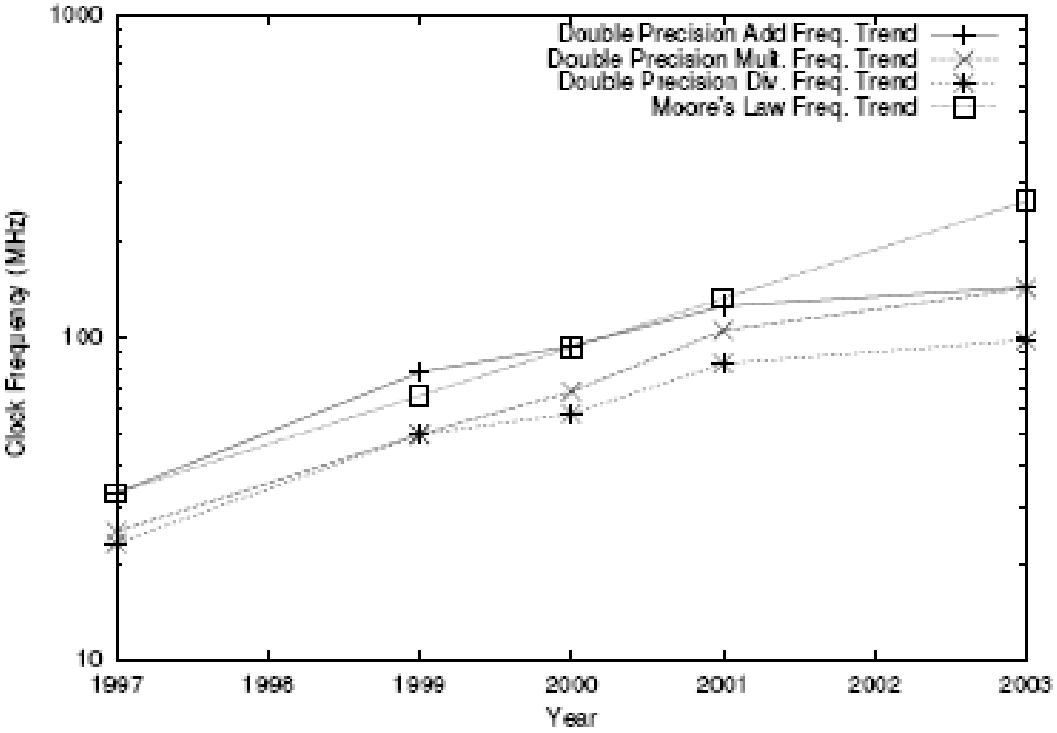

Figure 3: Expected trend in FPGA clock rates.

Starting with earlier FPGA chips, many designers have realized that it is beneficial to implement double precision floating point computations on these chips due to their mapping versatility and reconfigurability. Since then, various mapping efforts of floating point computations on FPGAs have realized performances that are steadily surpassing the performance of those computations on commodity CPUs. Figure 4 and Figure 5 show the trends of double precision floating point multiplication and division in FPGAs and commodity CPUs [9].

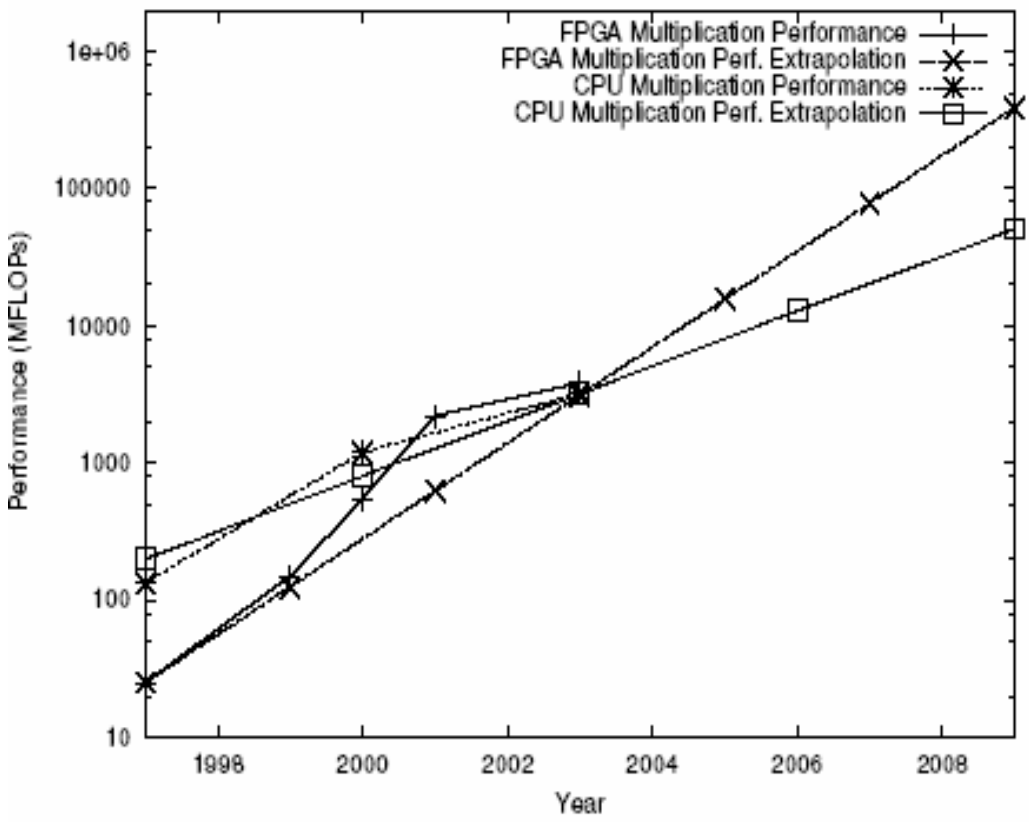

Figure 4: Expected trend in double precision floating point multiplication on FPGAs.

What is attractive about FPGAs is their diverse catalog of embedded architectural features specifically optimized for arithmetic operations. For example, Xilinx FPGAs embed carry-chains along their CLB columns designed to speedup addition with narrow operands.

These features can be exploited to support efficient floating point operations. However, exploiting these features require arithmetic algorithms, such as low-radix digit recurrence algorithms, that can be easily mapped on these structures. In fact, careful pipelining and mapping of these algorithms on specific FPGA architectures can yield easily testable implementations which can produce throughputs that are comparable to the throughputs seen in high-radix algorithms [10].

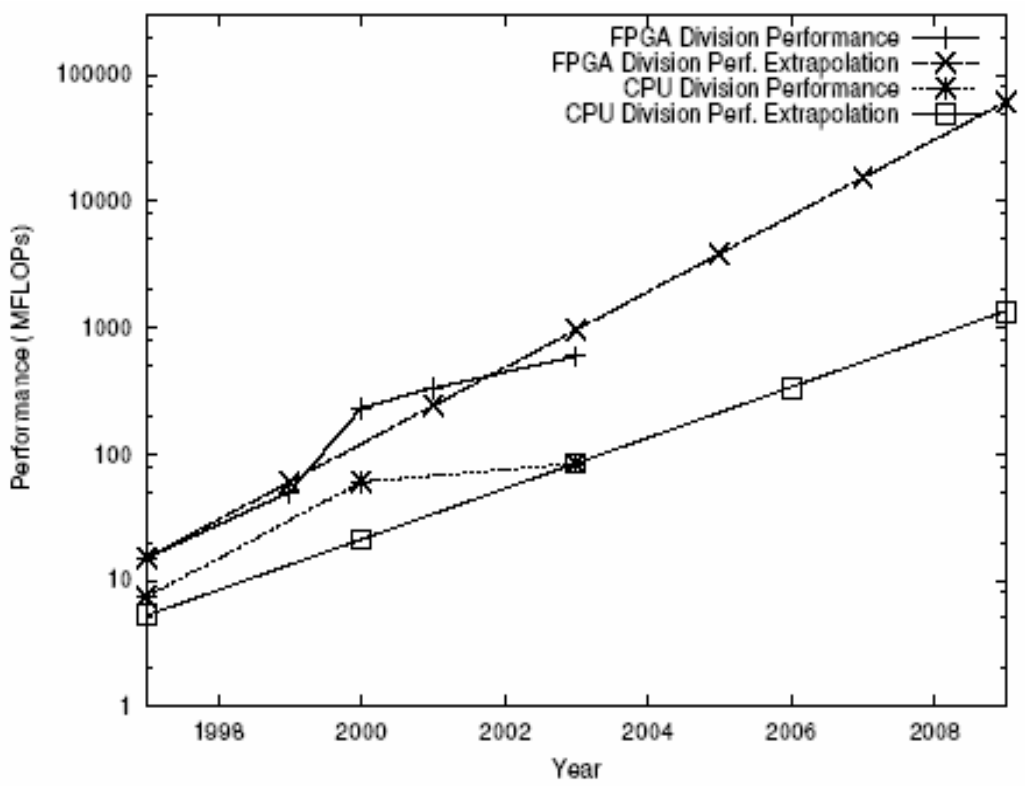

Figure 5: Expected trend in double precision floating point division on FPGAs.

Although their area overhead is marginally higher than high radix implementations in many cases, low-radix digit recurring algorithms can easily reach the 100 MFLOPS mark on some FPGA chips.

1.3 IEEE 754 Floating Point Representations

The IEEE 754 format represents floating point numbers in scientific notation. In this notation, a number consists of a base and an exponent. For example, 123.456 can be represented as 1.23456×10^2 where 1.23456 is the base part and 10^2 is the exponent part. Contrary to floating-point representations, fixed-point relies on a fixed window of representation, which limits its range and precision. Floating point representations employs a "sliding window" to accommodate the range and precision of the represented number. This allows it to represent widely varying numbers such as 1,000,000,000,000 and 0.0000000000000001 without any error accumulation.

To accommodate various ranges and precision, the IEEE 754 standard recommends four presentations, namely single precision, double precision, extended single precision and extended double precision formats as shown in Table 1[11]. The most commonly used are the single precision and double precision formats.

Table 1: IEEE 754 floating point number representations.

Format / **Parameter**	**Single**	**Single Extended**	**Double**	**Double Extended**
Significand bits	24	≥32	53	≥64
e_{max}	+127	≥+1023	+1023	≥+16383
e_{min}	-126	≤-1022	-1022	≤-16382
E_{bias}	+127	unspecified	+1023	unspecified
Exponent bits	8	≥11	11	≥15
Format bits	32	≥43	64	≥79

In general, a real number N in radix β can be represented in terms of a sign s, an exponent e and a significand S as $N = (-1)^s\cdot\beta^e\cdot S$ where $s \in \{0, 1\}$. For example, for a radix $\beta=10$, the

values $s = 1$, $e = 2$ and $S = 6.45678$ represent the number $N = (-1)^1 \cdot 10^2 \cdot 6.45678$. Similarly, for $\beta = 2$, the values $s = 0$, $e = 5$ (101_2) and $S = 1.40625$ (1.01101) represent the number $N = (-1)^0 \cdot 2^5 \cdot 1.40625 = +45$.

As shown in Figure 6, double precision numbers comprise of a sign bit, 11 bits of exponent and 52 bits of fraction. The 11 bit exponent E is an unsigned number containing a bias. The true exponent e of a floating point number is obtained by subtracting the E_{bias} from E, i.e. $e = E - E_{bias}$. The fraction f represents a 52 bit fraction in the range [0, 1) and the significand S is obtained by adding '1' (hidden bit) to the MSB of the fraction. Significand S is given by $S = 1.f$.

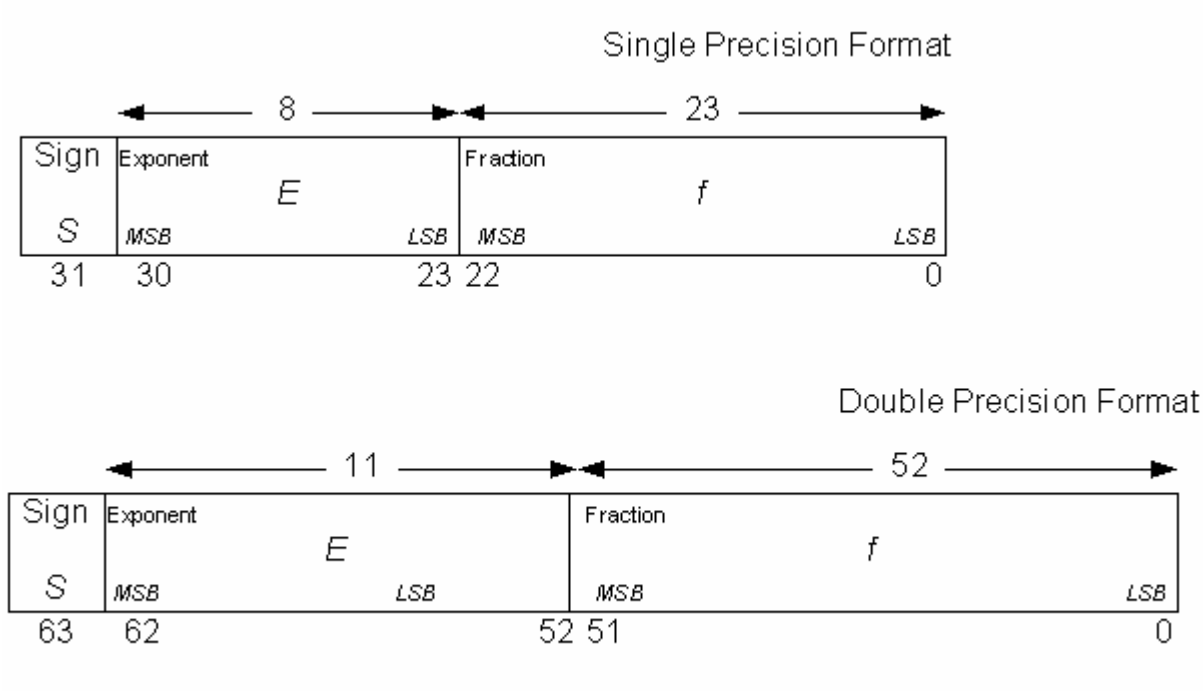

Figure 6: IEEE 754-1985 single and double precision formats.

As an example, consider the double precision floating point number

s	E	f
0	10000000011	1001000000000000000000000000000000000000000000000000

The biased exponent E is 1027 and the unbiased exponent $e=E-E_{bias}$ =1027-1023=4. The fraction f is .5625, making the significand $S = 1.f = 1.5625$. The sign bit is set to 0. Hence, the number is $N = (-1)^0 \cdot 2^4 \cdot 1.5625 = 25$.

The IEEE standard requires that a numeric environment support addition, subtraction, multiplication, division, square root, remainder, and round-to-integer as the basic floating-point arithmetic operations. A floating point calculation often involves some approximation or rounding because the result of an operation may not be exactly representable [3].

1.4 FPGA Technology

A field-programmable gate array or FPGA is a semiconductor device used to process digital information, similar to a microprocessor. Whereas an application specific integrated circuit (ASIC) performs a particular function defined at the time of manufacture, the functionality of the FPGA is defined by a program written by someone other than the device manufacturer. Depending on the particular device, the program is either burned in permanently or semi-permanently as part of a board assembly process. In addition, it can be loaded from an external memory each time the device is powered up. This programmability gives the user access to complex integrated designs without the high engineering costs associated with ASICs.

1.4.1 Logic Resources

FPGAs come in a wide variety of sizes and with many different combinations of internal and external features. Most FPGAs are composed of relatively small blocks of programmable logic called Configurable Logic Blocks (CLB). These blocks, each of which typically contains a few registers and a few dozen low-level, configurable logic elements, are arranged in a grid and tied together using programmable interconnections as shown in Figure 7. Figure 8 shows an

architectural overview of a Virtex II FPGA. Each CLB typically consist of two to four slices as shown in Figure 9 . In a typical FPGA, the slices that make up the bulk of the device are based on lookup tables, of four or five inputs, combined with one or two single-bit registers and additional logic elements such as clock enables and multiplexers as shown in Figure 10.

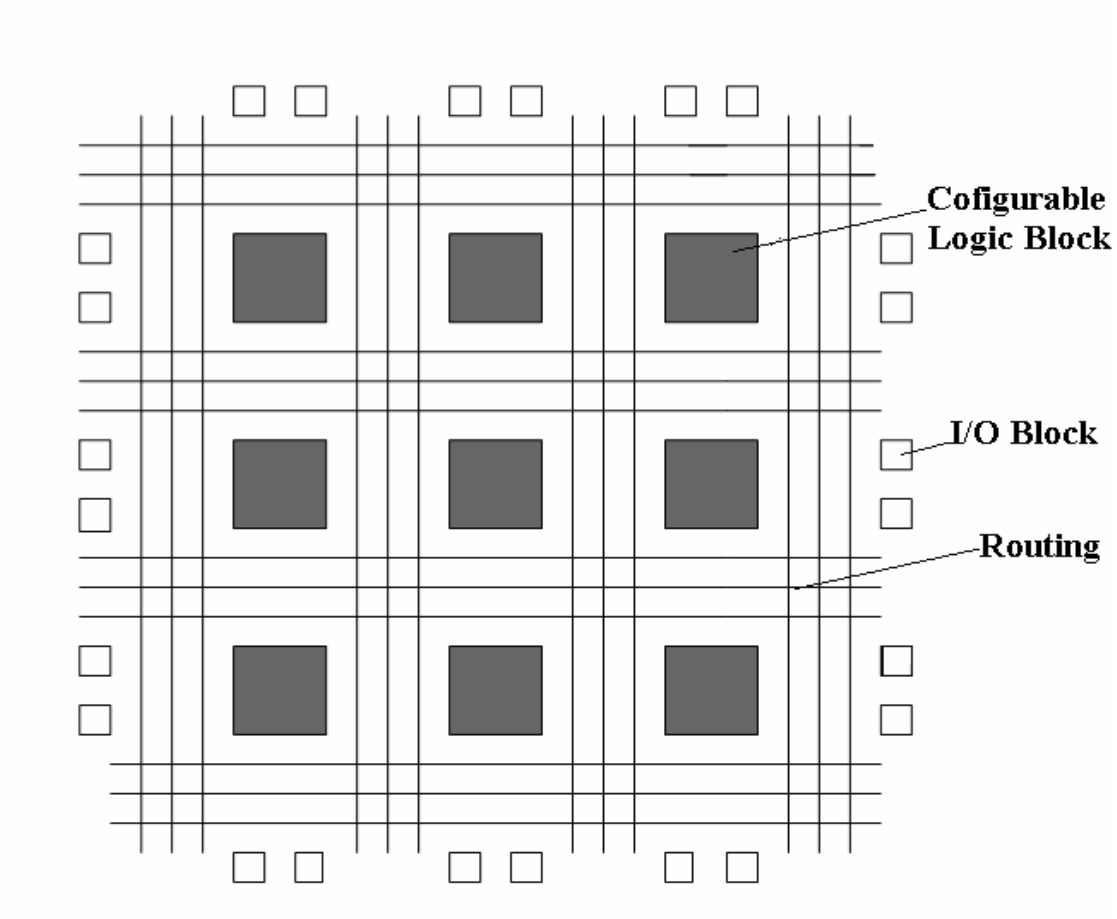

Figure 7: FPGA architecture.

In more complex FPGAs these general-purpose logic blocks are combined with higher-level arithmetic and control structures, such as multipliers and counters, in support of common types of applications such as signal processing.

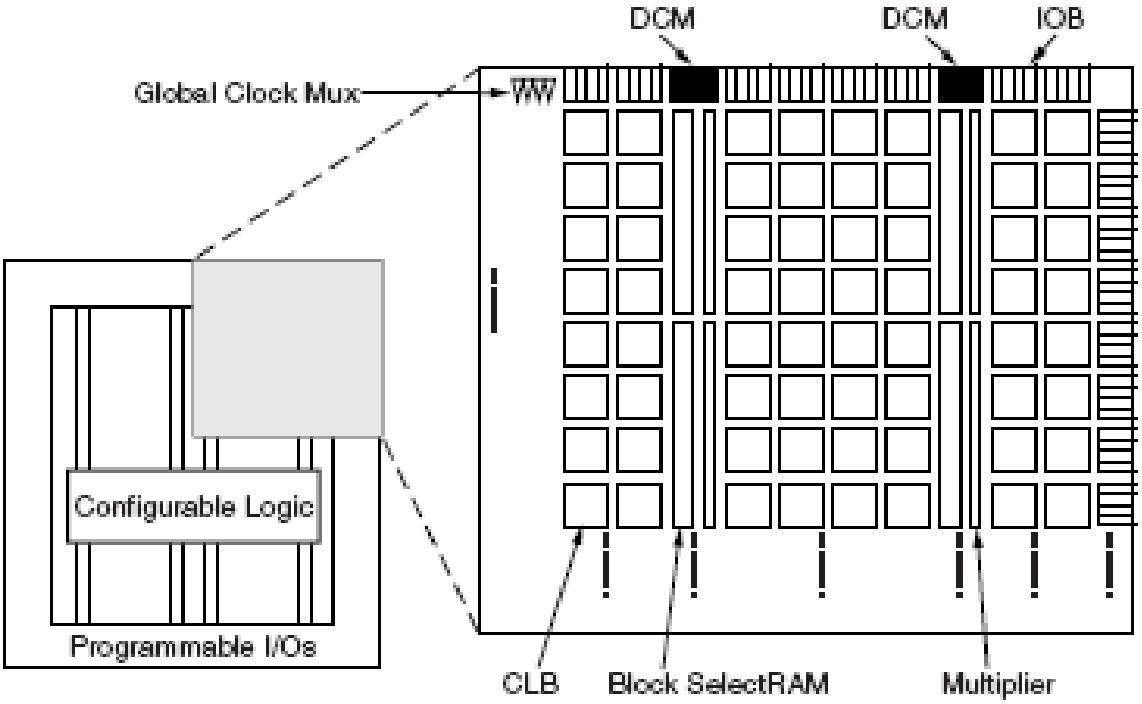

Figure 8: Virtex II architecture overview.

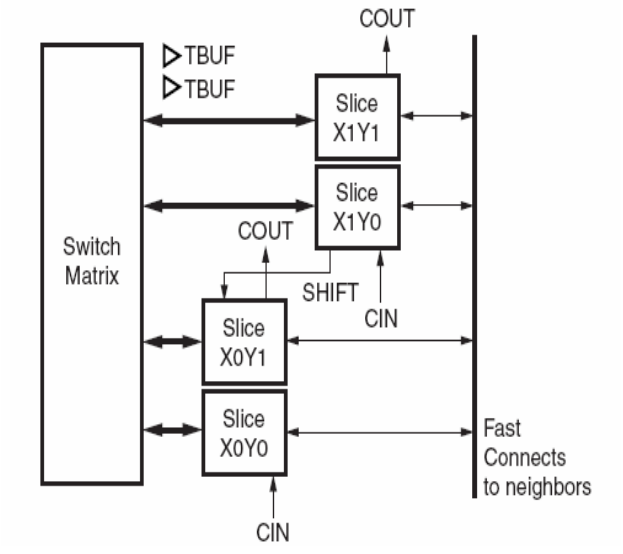

Figure 9: CLB elements.

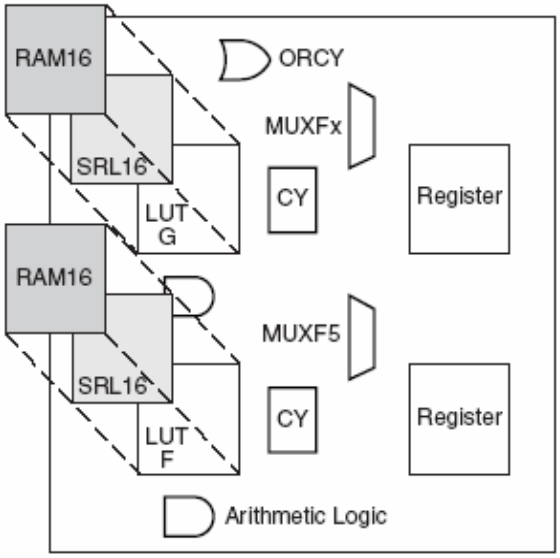

Figure 10: Slice configurations.

1.4.2 Routing Resources

Virtex-II logic resources are all connected to an identical switch matrix for access to global routing resources as shown in Figure 11.

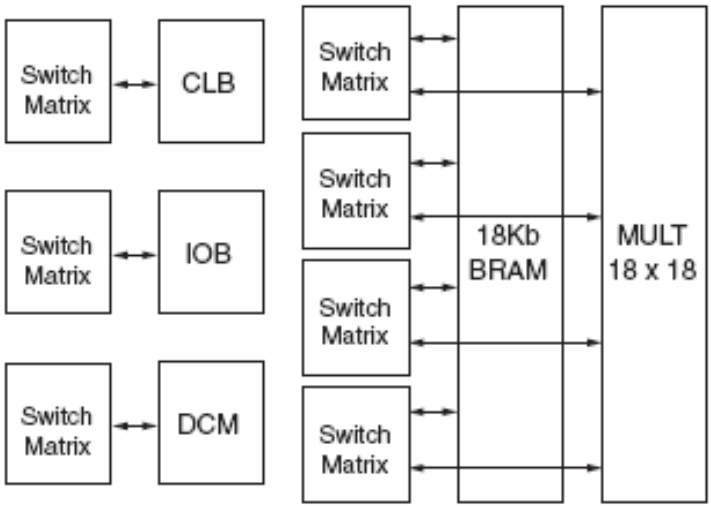

Figure 11: Active interconnect technology in Virtex-II.

Each Virtex-II device can be represented as an array of switch matrices with logic blocks attached as shown in Figure 12.

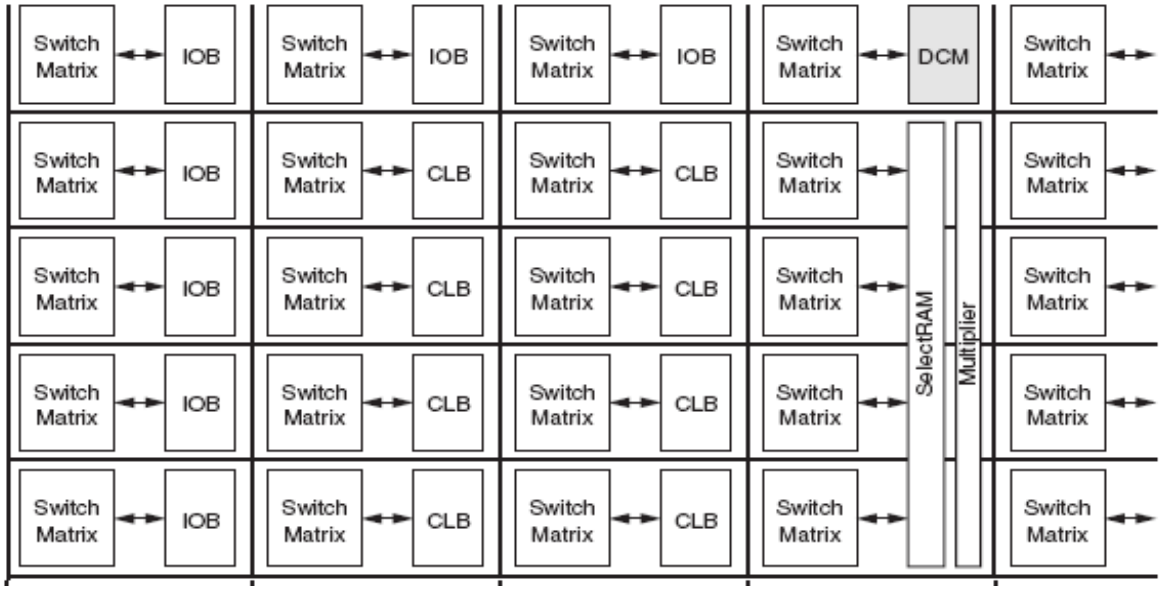

Figure 12: Routing resources in Virtex-II.

Optimum system performance and fast compile times are possible due to this regular array structure. Most Virtex-II signals are routed using the global routing resources, which are located in horizontal and vertical routing channels between each switch matrix. The hierarchical routing resources consist of long lines, hex lines, double lines, direct connect lines and fast connect lines as shown in Figure 13.

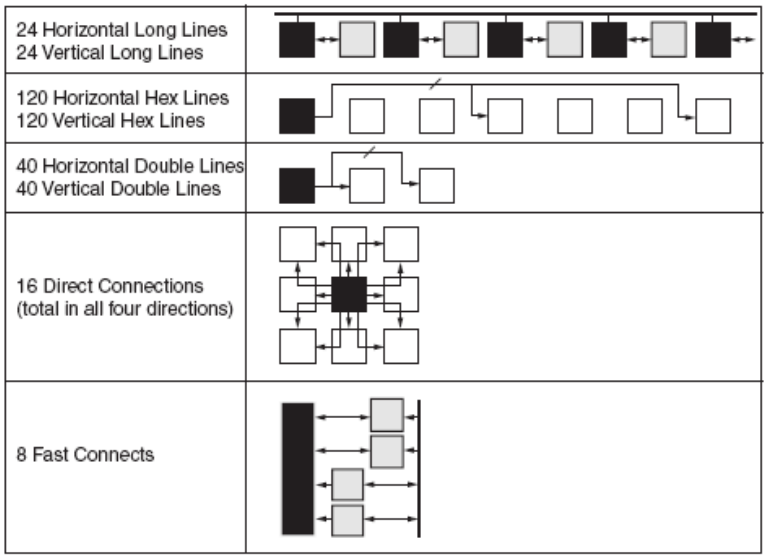

Figure 13: Hierarchical routing resources in Virtex-II.

In addition to the local and global routing, dedicated signals are also available. The dedicated signals consist of eight global clock nets per quadrant, two dedicated carry chain per slice column, one dedicated Sum-of-Products (SOP) chain per slice row, one dedicated shift chain per CLB, and three-state busses.

1.4.3 IO Resources

Virtex II I/O blocks (IOBs) are provided in groups of two or four on the perimeter of each device. Each IOB can be used as input and/or output for single-ended I/Os. Two IOBs can

be used as a differential pair. Figure 14 shows how a differential pair is connected to the same switch matrix.

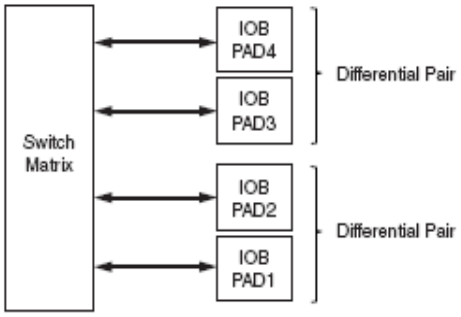

Figure 14: Virtex II input/output tile.

Virtex II IOBs are designed for high performance I/Os, supporting 19 single ended standards, as well as differential signaling with LVDS, LDT, bus LVDS and LVPECL. Figure 15 shows the supported single ended I/O standards by Virtex II.

IOSTANDARD Attribute	Output V_{CCO}	Input V_{CCO}	Input V_{REF}	Board Termination Voltage (V_{TT})
LVTTL	3.3	3.3	N/R(3)	N/R
LVCMOS33	3.3	3.3	N/R	N/R
LVCMOS25	2.5	2.5	N/R	N/R
LVCMOS18	1.8	1.8	N/R	N/R
LVCMOS15	1.5	1.5	N/R	N/R
PCI33_3	3.3	3.3	N/R	N/R
PCI66_3	3.3	3.3	N/R	N/R
PCI-X	3.3	3.3	N/R	N/R
GTL	Note (1)	Note (1)	0.8	1.2
GTLP	Note (1)	Note (1)	1.0	1.5
HSTL_I	1.5	N/R	0.75	0.75
HSTL_II	1.5	N/R	0.75	0.75
HSTL_III	1.5	N/R	0.9	1.5
HSTL_IV	1.5	N/R	0.9	1.5
HSTL_I_18	1.8	N/R	0.9	0.9
HSTL_II_18	1.8	N/R	0.9	0.9
HSTL_III _18	1.8	N/R	1.1	1.8
HSTL_IV_18	1.8	N/R	1.1	1.8
SSTL18_I(2)	1.8	N/R	0.9	0.9
SSTL18_II	1.8	N/R	0.9	0.9
SSTL2_I	2.5	N/R	1.25	1.25
SSTL2_II	2.5	N/R	1.25	1.25
SSTL3_I	3.3	N/R	1.5	1.5
SSTL3_II	3.3	N/R	1.5	1.5
AGP-2X/AGP	3.3	N/R	1.32	N/R

Figure 15: Virtex II supported single ended I/O standards.

1.4.4 Device Reconfiguration

Virtex II devices are configured by loading data into their internal configuration registers. The device can be set in a particular mode by setting the Mode bits in the configuration register as shown in Table 2.

Table 2: Configuration modes in Virtex II.

Configuration Mode	**M2**	**M1**	**M0**	**Pull-ups**
Master Serial	0	0	0	No
Slave Serial	1	1	1	No
SelectMAP	1	1	0	No
Boundary Scan	1	0	1	No
Master Serial (w/ pull-ups)	1	0	0	Yes
Slave Serial (w/ pull-ups)	0	1	1	Yes
SelectMAP (w/ pull-ups)	0	1	0	Yes
Boundary Scan (w/ pull-ups)	0	0	1	Yes

The external configuration process is a simple matter of loading the configuration bitstream into the FPGA using the selected configuration mode as illustrated in Figure 16.

Xilinx proposes two standard flows for partial reconfiguration process: Difference based and Module based flows [12]. With the Difference Based flow, the designer must manually edit a design with low-level changes. Using a low-level editing tool, such as the FPGA Editor, small changes can be made to different components, such as lookup tables, flip-flops, and I/O pins. After the changes are completed, the partial bit stream, which contains information only regarding the modifications, is generated and stored in a file.

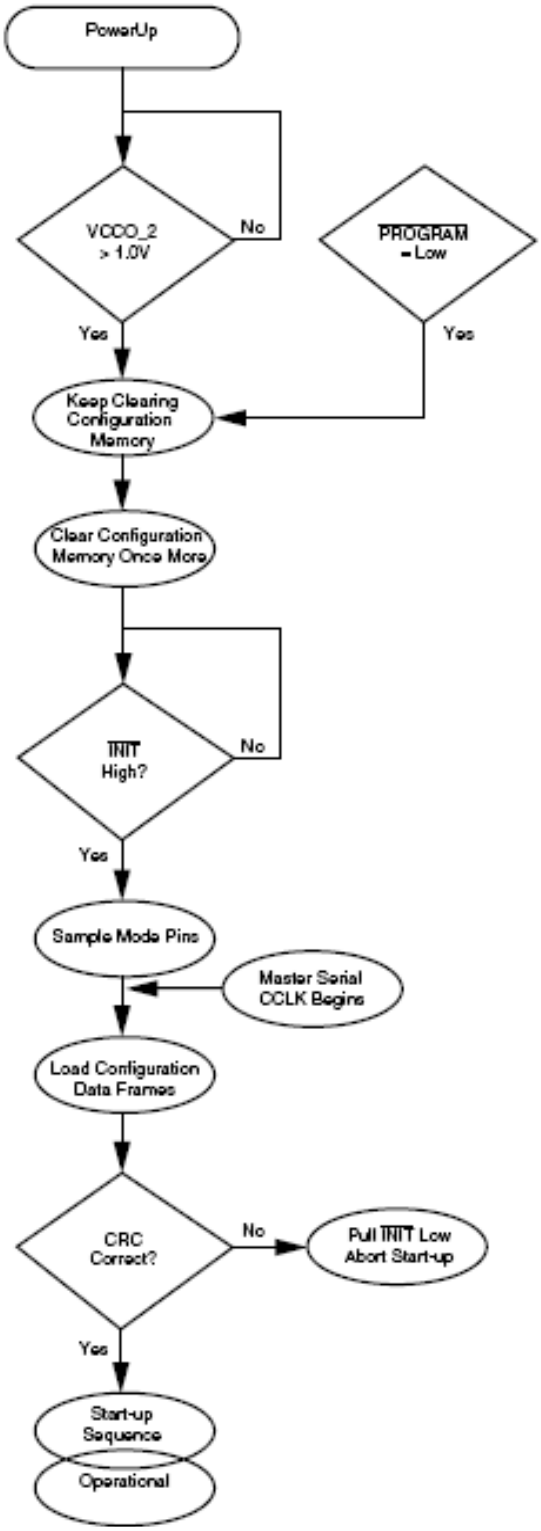

Figure 16: Configuration flow diagram.

For the Module based flow, the full design is partitioned into modules, some of which can be fixed while others can be reconfigurable. The reconfigurable fabric of the FPGA is partitioned into column-based rectangular regions in which the fixed and reconfigurable modules can be arranged based on specified area constrains. A bus macro can be used to maintain correct connections between the modules placed in these areas by sitting across the boundaries of these rectangular regions. Figure 17 shows the basic concept of this flow [12].

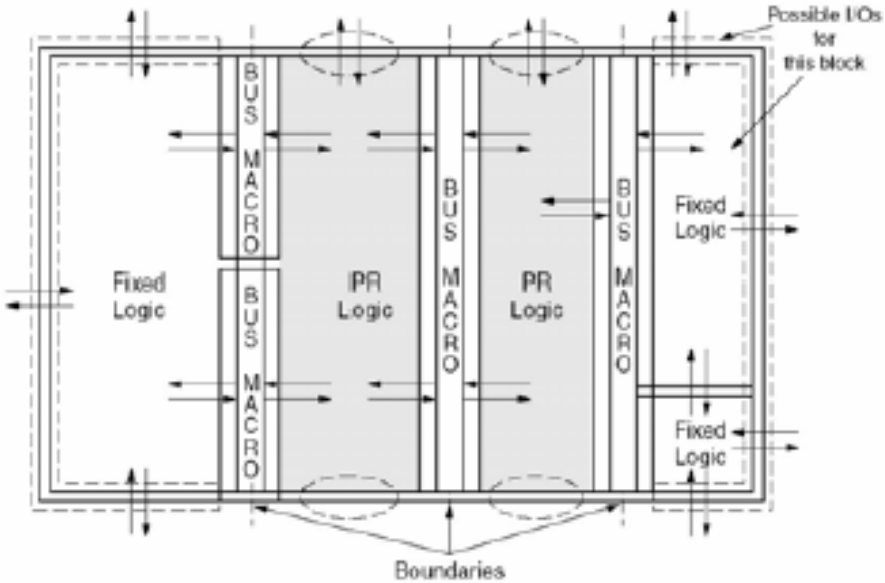

Figure 17: Layout with two reconfigurable modules.

1.5 Thesis Contribution

While very few attempts have been made to study wide precision floating point arithmetic on FPGAs, this thesis presents a study of IEEE 754-compliant double precision floating point operations by focusing on division and square root operations. Performance and design tradeoffs related to these two operations in particular are not well understood in terms of FPGA implementations. Based on this rationale, this thesis makes the following contributions:

(i) Contrary to established wisdom, this thesis focuses on the simplest algorithms to perform division and square root operations. The division operation is implemented based on a method similar to the pencil-and-paper method known as the sequential non-performing algorithm. On the other hand, the square root operation is implemented based on a basic non-restoring algorithm. Both algorithms are radix-2 digit recurrence algorithms.

(ii) For comparison purposes, this thesis presents the implementations of low area sequential designs and high performance pipelined designs of division and square root operations.

(iii) In order to explore the tradeoffs between area, throughput, and power consumption, this thesis partitions the pipelined designs of both operations into different pipeline depths. These different depths are used to characterize the area overhead, maximum throughput and dynamic power consumption of each operation.

(iv) While most previous implementations rely on highly optimized cores and occasionally manual layouts, all the implementation of both operations in this thesis can be qualified as *common-denominator* implementations. These implementations do not take advantage of any advanced architectural features available in the Virtex FPGAs such as Block RAMs or embedded multipliers. In addition, these implementations do not use optimized cores or any custom floor planning at all. The rationale behind this design philosophy is to quantify how much performance can be obtained by exploiting exclusively the fine-grain reconfigurable resources available in FPGAs. Such implementations have the advantage of being easily portable to other FPGA architectures with minimum modifications.

(v) Careful attention to the pipelining approach of this thesis has led to implementations whose performances are comparable to those of high-radix implementations, and in some case even superior. This approach is based on a precise unrolling of the iterations in the digit recurring algorithms.

(vi) Whereas high-radix implementations are difficult to verify, the radix-2 implementations presented in this thesis are easy to test and verify.

1.6 Thesis Outline

Chapter 2 discusses and summarizes previously proposed work related to division and square root operations. Chapter 3 and 4 present the design, implementation, and performance evaluation of the divider and square root unit respectively. Finally, chapter 5 summarizes the findings in this thesis and provides future direction for research.

CHAPTER TWO: RELATED WORK

In this chapter, a brief overview of the different designs proposed for floating-point division and square root operations are presented. In section 2.1, four different designs proposed for floating point division are briefly described while in section 2.2 five designs proposed for square root operations are briefly reviewed. Section 2.3 concludes this chapter by summarizing the reviewed designs and comparing them to the designs of the divider and square root unit proposed in this thesis.

2.1 Floating Point Division

Division algorithms can be broadly classified into five classes: digit recurrence, functional iteration, very high radix, table lookup, and variable latency [13]. Among these classes, digit recurrence algorithms are widely used since they are easy to implement. In digit recurrence, algorithms, such as restoring, non-restoring, and SRT division, rely on addition/subtraction and shift operations to complete division.

In [14], the author presents the sequential and pipelined designs of a floating point divider for three different precisions based on SRT division. The recurrence equation for this division is given by $w_{j+1} = r \cdot w_j - D \cdot q_{j+1}$, where w_x is the remainder after the xth iteration, r is the radix of the algorithm, D is the divisor, and q_x is the xth quotient digit from the most-significant bit of the quotient Q_x. In order to reduce the delay of a single stage of the pipelined version of the divider, a radix-4 divider was chosen for the significant division as shown in Figure 18. The structure of the fixed point divider is shown in Figure 19.

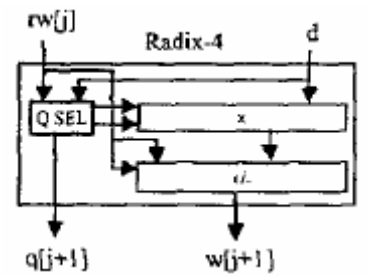

Figure 18: Radix-4 SRT single stage.

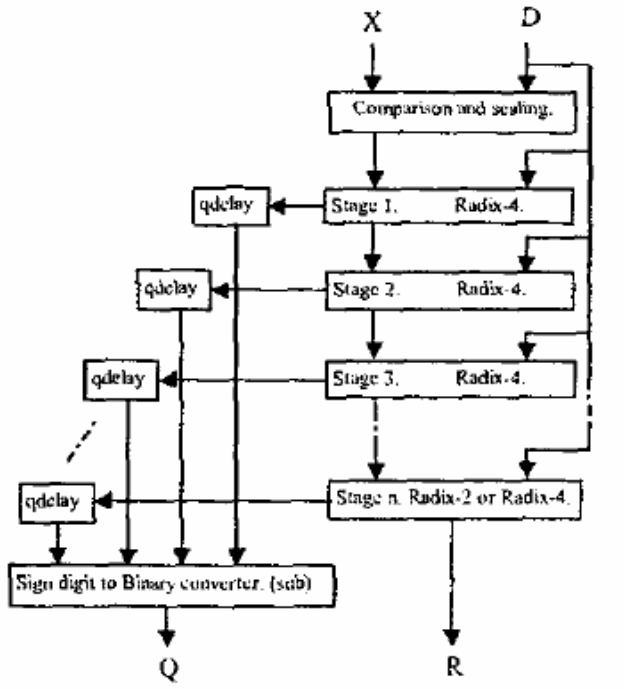

Figure 19: Fixed point divider structure.

The implemented sequential double precision floating point divider consumes 1705 slices and runs at a frequency of 3.3 MHz with a throughput of 3.17 MFLOPS on a Virtex II FPGA. This design is further pipelined to increase the throughput to 78 MFLOPS with 1.5× area overhead when implemented on a Virtex II XC2V6000 FPGA. However, the authors do not explicitly describe the pipelining approach applied to their sequential divider.

In [15], the authors present three designs of a floating point divider with three different bit precisions the largest of which is 32-bit precision. These three designs are based on iterative, array, and pipeline approaches. In particular, the pipelined design is based on the insertion of registers between division steps of an array divider. In this divider, the authors unroll the hardware for each step by rebuilding this hardware and cascading the iterative steps in the array.

Figure 20 shows the sub-pipelining of a single stage in the pipelined radix-4 divider while Figure 21 shows the structure of the pipelined divider.

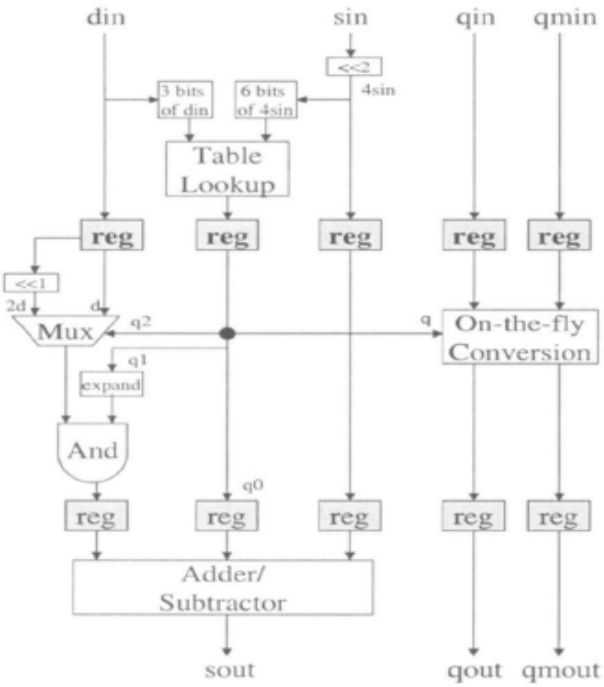

Figure 20: Two-layer pipelining of a radix-4 single stage.

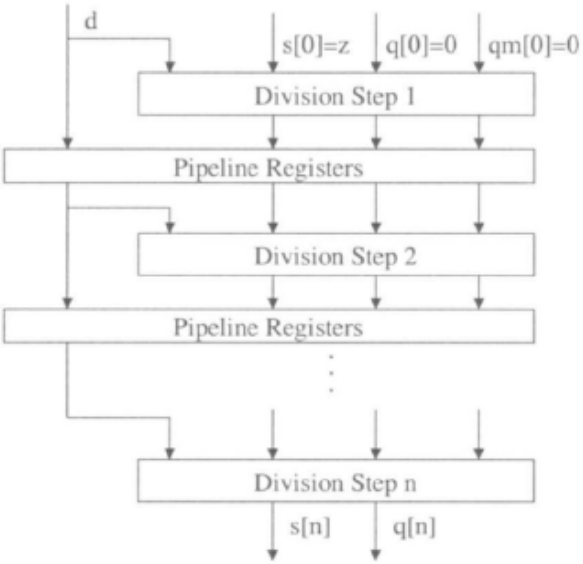

Figure 21: Pipelined implementation of SRT division.

The three designs of the divider are implemented on a Virtex II XC2V1000 FPGA. Compared to the array version of the divider, 30× improvement in throughputs are observed over 10-20× increase in the area of the pipelined version. The authors conclude that the radix-4 implementations are preferable from a performance standpoint while radix-2 implementations are preferable when area × latency or area × clock period is considered.

In [10], the authors implement IEEE-754 compliant pipelined dividers based on non-restoring and SRT divisions on a Virtex II XC2VP7 FPGA. Figure 22 shows the block architecture of the proposed divider.

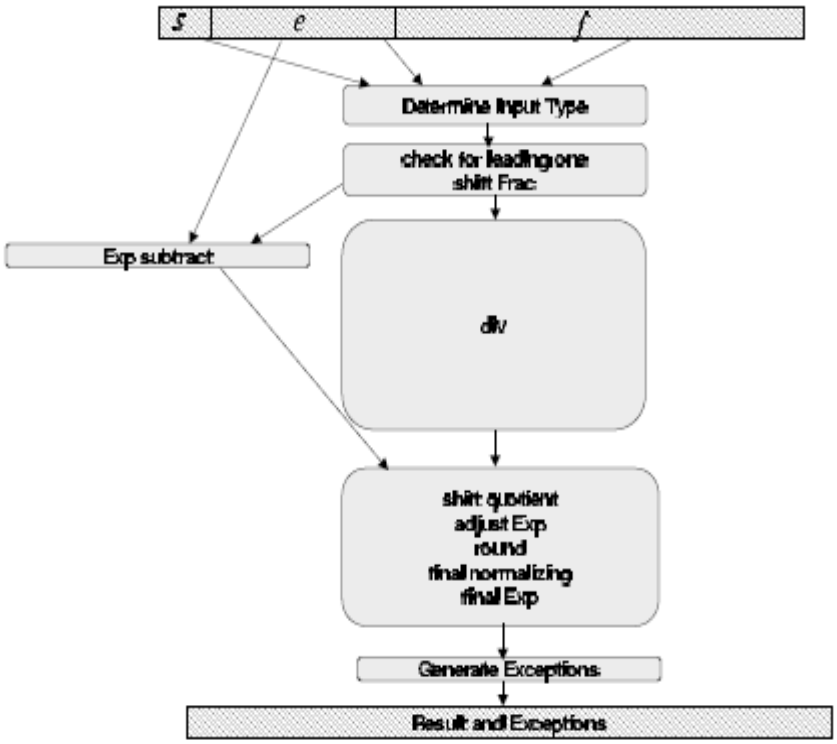

Figure 22: Block diagram of the proposed divider.

These dividers are pipelined from 32 to 68 stages depending on the selected division algorithm. The most compliant divider is a non-restoring divider pipelined into 68 stages while a low area overhead SRT divider is pipelined into 32 stages for comparison purposes. The former divider runs at 140 MHz with an area utilization of 4234 slices while the latter runs at 90 MHz with an area utilization of 3713 slices. The authors remark that the non-restoring divider can achieve a superior performance only when it is pipelined into a large number of stages. On the other hand, the SRT divider displays a slightly lower performance at the expense of a larger area.

In [16], the authors propose a scalable 32-bit pipelined design of a divider implemented on a Virtex XC2V1000 FPGA. This design is based on a radix-2 non-restoring division

algorithm. Figure 23 shows the structure of the pipelined divider while Figure 24 shows the pipelined non-restoring array divider.

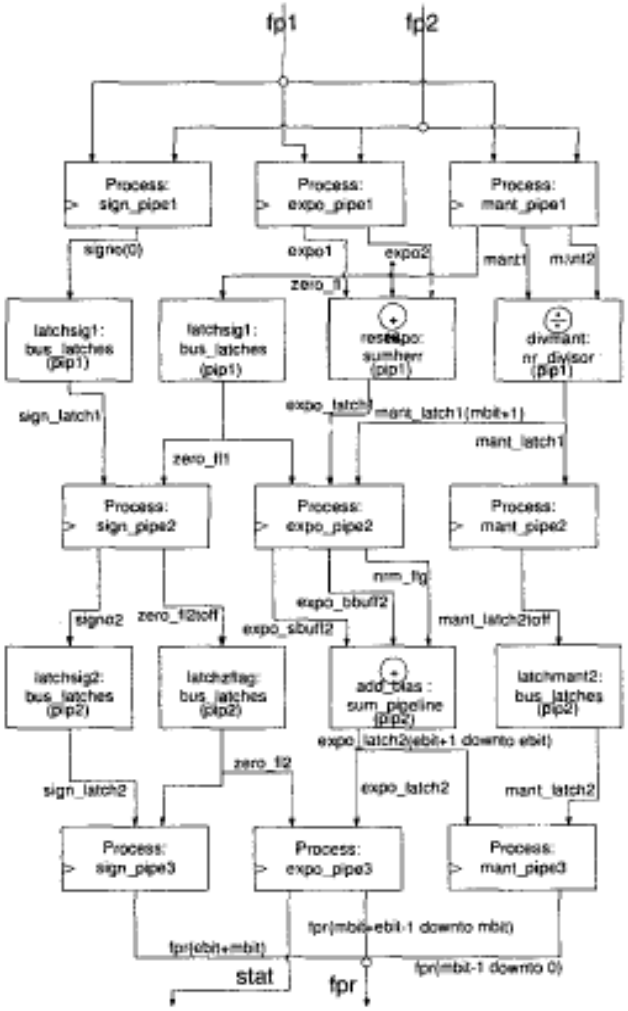

Figure 23: Structure of the pipelined divider.

A 24-stage version of the pipelined design can run at 160 MHz with an area utilization of 870 slices thus producing a throughput of 158 MFLOPS. The authors claim that by pipelining further the divider, the number of slices needed to support additional stages increases in a linear fashion. This increase in slices is caused by the need for additional latches to implement newly inserted registers.

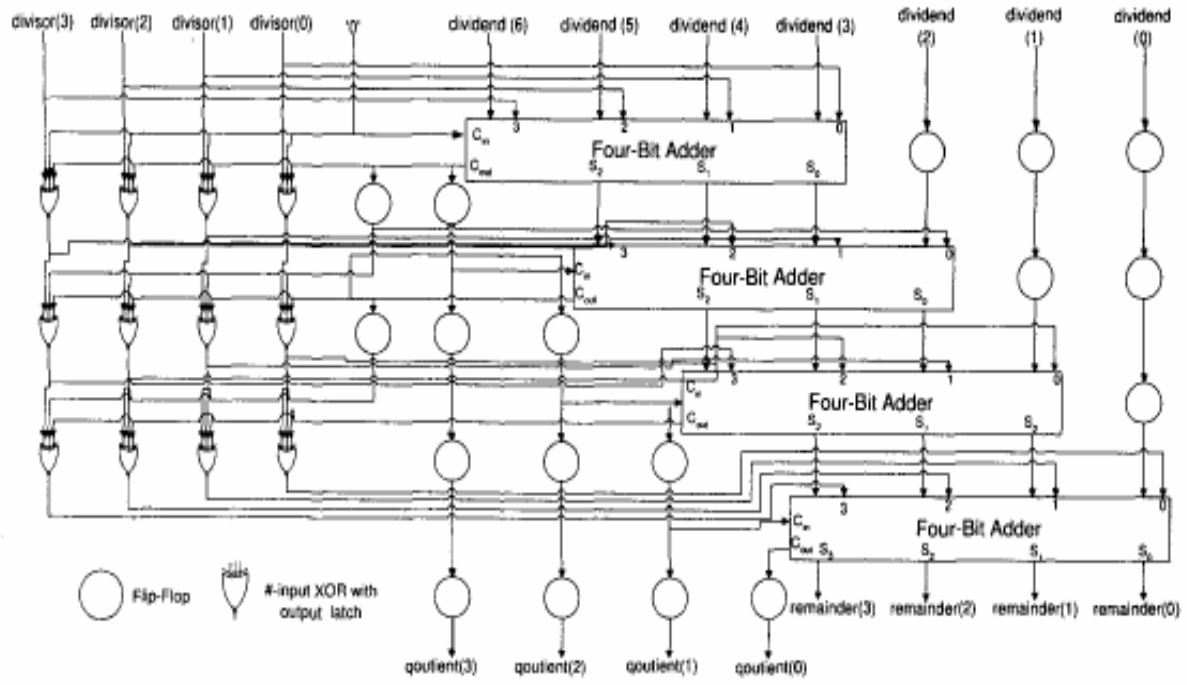

Figure 24: Pipelined non-restoring array divider.

This thesis presents a sequential divider based on a radix-2 digit-recurring non-performing sequential algorithm [17]. The iterations in the digit recurring algorithm are unrolled to various degrees to generate pipelined dividers with different pipeline depths. These depths are used to characterize the impact of pipeline depth on area cost, throughput, and dynamic power within the divider.

2.2 Floating Point Square Root Operation

Square root algorithms share numerous features with division algorithms. Among the widely known square algorithms are the traditional pencil-and-paper method, shift/subtract based restoring algorithms, non-restoring algorithms, high radix square rooting algorithms, and square rooting by convergence algorithms [18].

In [14], the authors present a non-restoring square root algorithm based on the square root recurrence equation $R_{i+1} = r \cdot R - 2 \cdot Q \cdot q_{i+1} - (q^2_{i+1} / r^{i+1})$. The non- restoring algorithm uses the digit set {1.-1} and therefore the least significant bit of the current partial root Q_i is always 1. This helps in generating the next value to add to or subtract from the shifted partial remainder $2 \cdot R_i$. The new generated value of $2 \cdot Q_i$ can be added or subtracted from the shifted partial remainder according to the sign of the quotient digit q_{i+1}. Figure 25 shows the block diagram of a square root unit while Figure 26 shows the fixed point square root structure. The authors implement a non pipelined double precision floating point square root unit which occupies 869 slices and produces a throughput of 3.99 MFLOPS at a frequency of 4.18 MHz on Virtex II XC2V6000 chip. This square root unit is further pipelined to run at 72.46 MHz with an area overhead of 1.64×.

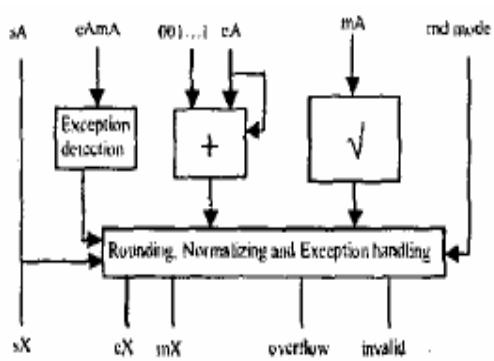

Figure 25: Floating point square root diagram.

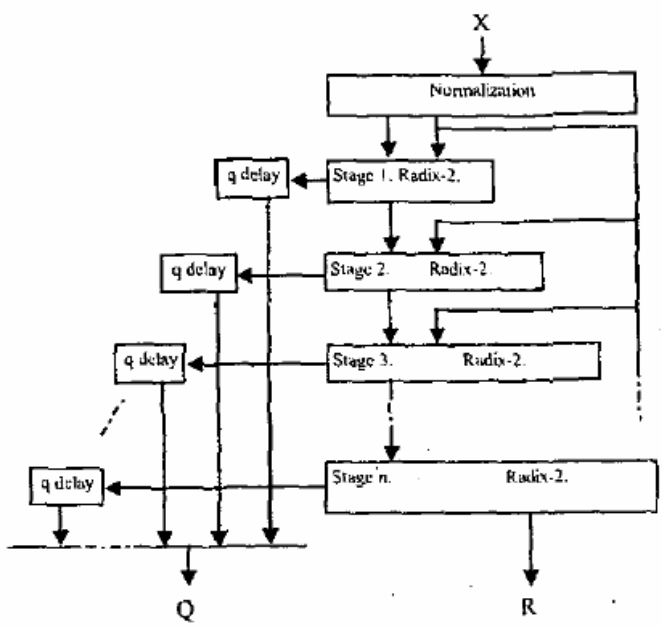

Figure 26: Fixed point square root structure.

In [19], the authors propose a single precision sequential and pipelined square root units based on a non-restoring algorithm. The pipelined version of the unit is shown in Figure 27. Although the authors did not specify the clock frequency or throughput, they stated that the single precision square root unit displays a latency of 25 clock cycles by utilizing 82 CLB function generators and 138 CLB flip-flops while its pipelined counterpart has a latency of 15 clock cycles by utilizing 408 CLB function generators and 675 CLB flip-flops on a Xilinx XC4000 FPGA.

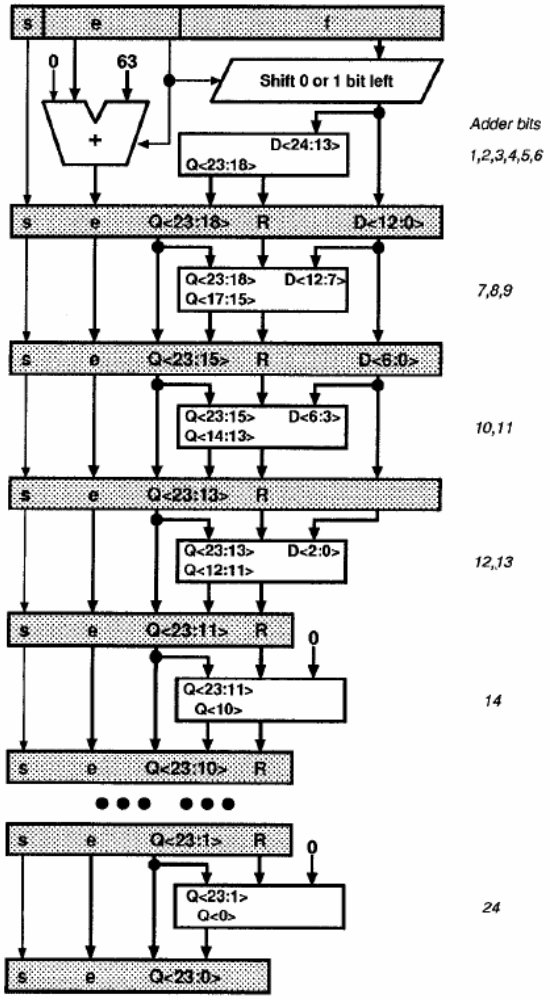

Figure 27: Pipelined implementation of a single precision square root unit.

In [15], the authors propose the implementation of a square root unit for three bit precisions based on a restoring digit recurrence algorithm. This algorithm consists primarily of a sequence of subtract and shift operations. Figure 28 shows the design of one restoring square root stage. The resulting sequential implementation runs at 153 MHz by occupying 234 slices while the pipelined implementation runs at 169 MHz by occupying 1313 slices of an XC2V1000 FPGA.

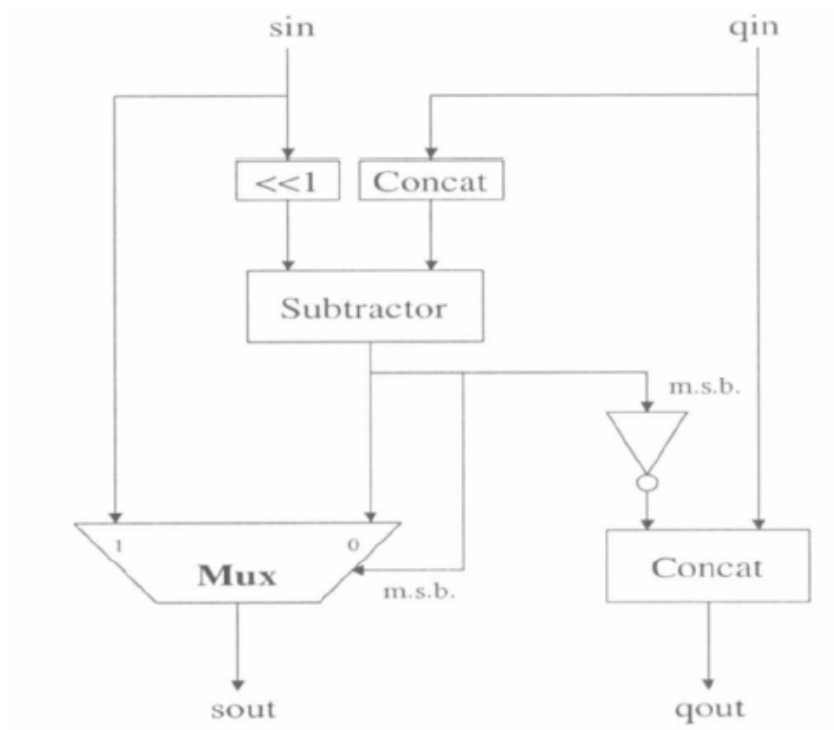

Figure 28: Block diagram of a single stage in the restoring square root algorithm.

In [10], the authors present the design of a pipelined square root unit based on the non-restoring algorithm proposed in [20]. This algorithm computes the square root by a series of additions or subtractions based on the successive values of the bits generated for the quotient. Figure 29 shows a block diagram of the steps required to perform the square root algorithm.

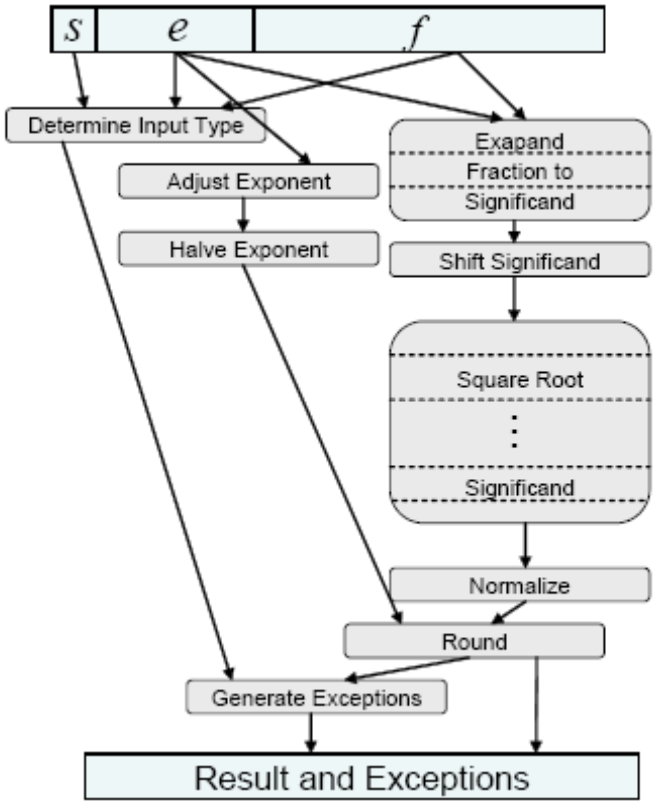

Figure 29: Steps to perform the floating point square root algorithm.

The most compliant pipelined implementation consists of 60 stages and runs at 164 MHz by occupying 2332 slices while the least compliant (i.e., lowest overhead) implementation consists of 55 stages and runs at 169 MHz by occupying 1666 slices of a Virtex II XC2VP7 FPGA.

In [16], the authors propose a scalable single-precision square root unit based on a non-restoring digit recurring algorithm. The pipelining of this unit is based on array architecture of the non-restoring algorithm. Figure 30 shows the structure of pipelined square root unit while Figure 31 shows an eight-bit non-restoring array used in a smaller square root unit.

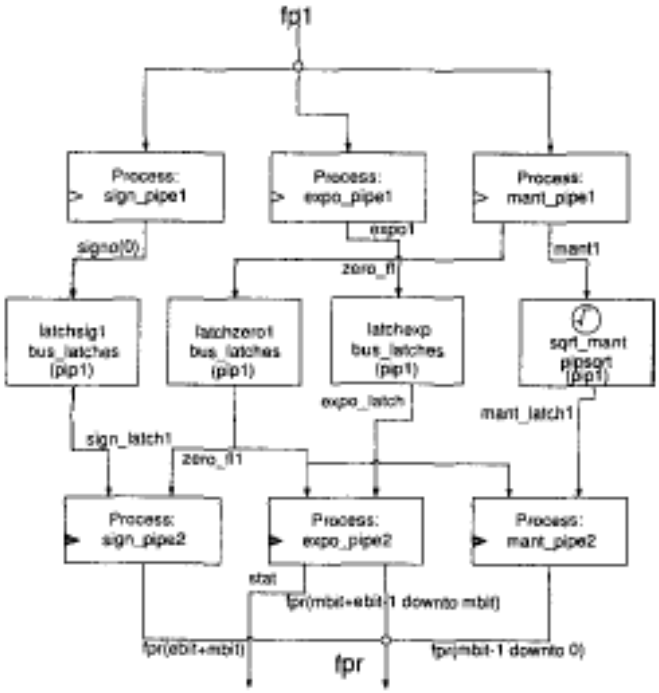

Figure 30: Structure of the pipelined square root unit.

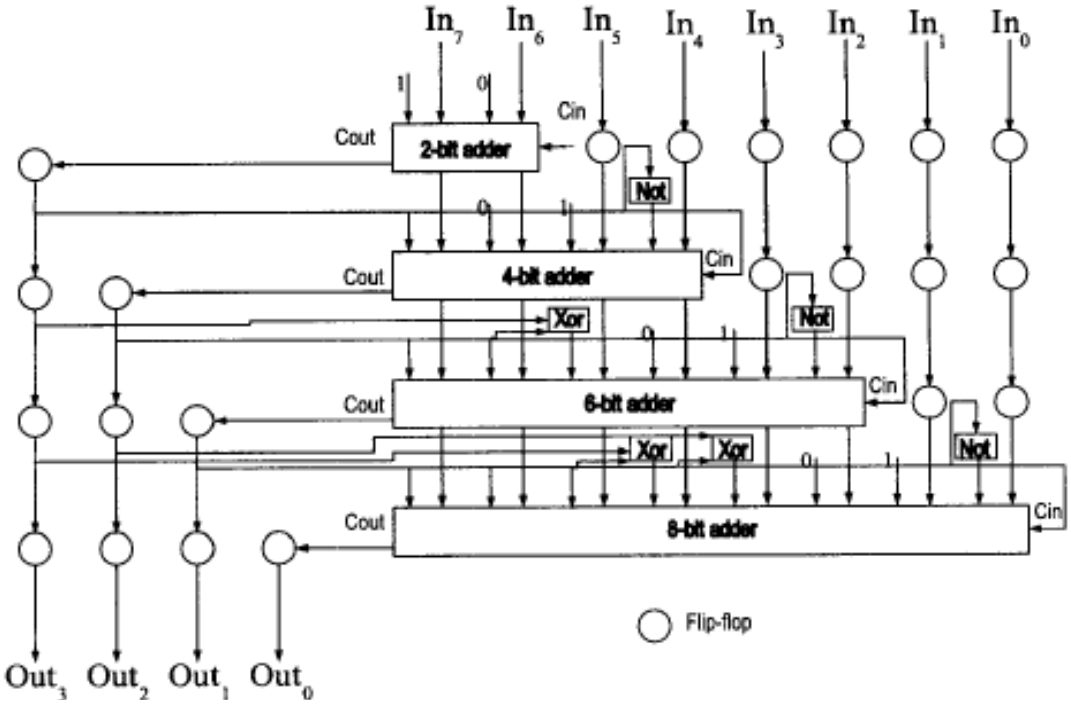

Figure 31: Eight bit non-restoring square root array.

A 12-stage single-precision version of the pipelined square root unit can run at 211 MHz with an area utilization of 302 slices thus producing a throughput of 204 MFLOPS. The same claims made by the authors regarding the dividers apply also the square root unit.

This thesis presents a sequential square root unit based on a radix-2 digit-recurring non-performing sequential algorithm [17]. The iterations in the digit recurring algorithm are unrolled to various degrees to generate pipelined square root units with different pipeline depths. These depths are used to characterize the impact of pipeline depth on area cost, throughput, and dynamic power within the unit.

2.3 Summary

Table 3 shows a summary of the reviewed dividers. This table shows the 32-bit designs highlighted in gray color. Given the diversity of FPGA chips used in these designs and the wide ranges observed in their throughputs and area overhead, it can be quite difficult to construct a meaningful comparison between these designs. As a result, a generic metric, that is independent of implementation technology, is needed to quantify the efficiency of a given design. In fact, such a metric can be easily constructed by considering the level of throughput measured in FLOPS, produced by unit of area, measured in slices. This metric, shown in the rightmost column of each table, can be used as the basis for comparison of the various designs shown in Table 3 and Table 4

In order to obtain a fair comparison of the 32-bit designs to the 64-bit designs, the throughput and area utilization of the 32-bit designs have to be halved and doubled respectively. Based on these new numbers, a scaled throughput-area ratio can be recomputed for comparison purposes.

Table 3: Summary of the reviewed dividers.

Reference	Algorithm	Radix	Precision (bits)	Pipelining (stages)	Device	Throughput (MFLOPS)	Area (Slices)	Throughput/Area (KFLOPS/Slice)
[14]	SRT	4	64	29	XC2V6000	97.81	2595	30.12
[15]	SRT	2	32	47	XC2V1000	166.66	3245	51.36
[10]	SRT, Non-restoring	2	64	68	XC2VP7	140.05	4243	33
[16]	Non-restoring	2	32	24	XC2V1000	158	870	181.6
This thesis	Pencil-and-paper	2	64	60	XC2V6000	97.81	2920	33.49

As Table 3 shows, the divider proposed in this thesis presents a throughput-area ratio that is higher than the ratio of the other 64-bit dividers. With the exception of the divider by [16], the other 32-bit divider presents a lower throughput-area ratio after it is scaled to a 64-bit precision. It is worth noting that the design proposed in [16]consumes a significant area in terms of slices in comparison to the divider proposed in this thesis as more stages are added to the pipeline. Instead, the divider in this thesis consumes an almost constant area regardless of the depth of the pipeline. This constant consumption of resources can be leveraged by maximally pipelining the design in order to reach the highest throughput possible.

Table 4 shows a summary of the reviewed square root units where the 32-bit designs are highlighted in gray color. With the exception of the design proposed by [10], this table shows that the square root unit proposed in this thesis has a throughput-area ratio that is comparable to the other 64-bit designs. It is worth noting that the authors in [10] implemented their design on an XC2VP7. This device is a high-end reconfigurable system that is fabricated in a 0.13 μm, 1.5 V CMOS process with fast switching devices. This device can reach clock frequencies that surpass by far the frequencies of the other Virtex-II devices. The divider and square root units

proposed in this thesis are based on simple radix-2 algorithms that lead to implementations that are easy to test and parameterize.

Table 4: Summary of the reviewed square root units.

Reference	Algorithm	Radix	Precision (bits)	Pipelining (stages)	Device	Throughput (MFLOPS)	Area (Slices)	Throughput/Area (KFLOPS/Slice)
[14]	Non-restoring	2	64	28	XC2V6000	69.1	1433	48.22
[19]	Non-restoring	2	32	25	XC4000	Not stated	408+675 (LUTs+FFs)	Not stated
[15]	Non-restoring	2	32	28	XC2V1000	166	1313	126.42
[10]	Non-restoring	2	64	66	XC2VP7	164.2	2332	70.41
[16]	Non-restoring	2	32	15	XC2V1000	204	302	675.9
This thesis	Non-restoring	2	64	59	XC2V6000	126.82	2700	46.97

CHAPTER THREE: DOUBLE PRECISION FLOATING POINT DIVISION

In this chapter, section 3.1 presents the architecture of the double precision floating point divider while section 3.2 presents the approach used to pipeline the divider. Section 3.3 presents the verification of the divider while 3.4 presents the experimental results of the divider and a related discussion. Finally, section 3.5 presents the conclusion of the chapter.

3.1 Architecture of the Divider

As Figure 32 shows, the divider takes as inputs two 64-bit numbers *A* and *B*, and outputs a quotient *Q* as a 64-bit number.

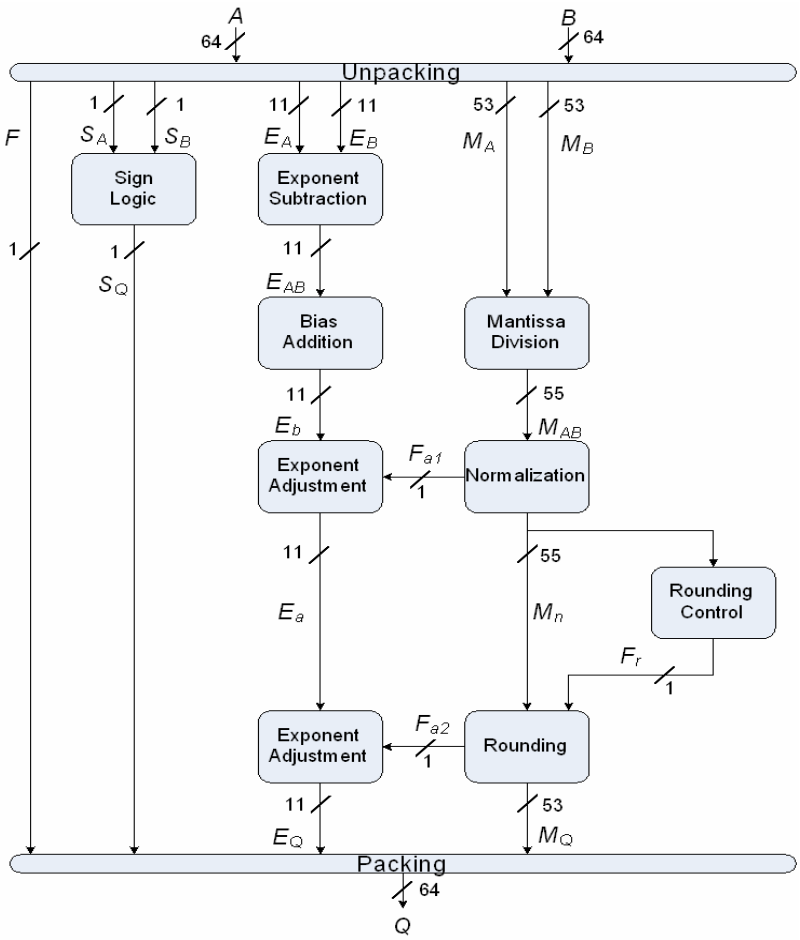

Figure 32: Double precision floating point divider.

3.1.1 *Unpacking*

The unpacking block separates the 64 bits of each number, A and B, into the sign bit S which is the most significant bit, the exponent E which is the next significant 11 bits, and the mantissa M which is the least significant 52 bits. The biased exponent and the input operand are used to determine whether the input operand is NaN (i.e., not a number), infinity, zero, or neither of these. If any of the three first conditions is true, the flag F is set and computation is halted. Otherwise, S, E and M are fed to the next appropriate blocks.

3.1.2 *Sign Logic*

This block determines the sign S_Q of the final quotient Q. The sign of the quotient is found by XORing the signs of A and B as $S_Q = S_A$ XOR S_B. Table 5 shows how the sign of the final quotient is calculated from the sign of A and B.

Table 5: Determination of sign S_Q of the final quotient Q.

S_A	S_B	S_Q
0	0	0
0	1	1
1	0	1
1	1	1

3.1.3 *Exponent Subtraction*

This block computes the exponent of the quotient by subtracting the exponent of B from that of A. The subtraction is an 11-bit operation: $E_{AB} = E_A - E_B$.

3.1.4 *Bias Addition*

This block adds the bias, which is 1023, to the output E_{AB} of the exponent subtraction block as follows: $E_b = E_{AB} + 1023$.

3.1.5 Mantissa Division

This block computes the quotient and remainder of the mantissa using a 55-bit remainder register R as follows:

```
Input: M_A, M_B  Both are 53-bit mantissas
Output: M_AB     55-bit mantissa

1: R = M_A;
2: for (i = 0; i < 55; i = i + 1)
3:    if (R – M_B) ≥ 0
4:       M_AB[54-i] = 1;
5:       R = R – M_B;
6:    else
7:       M_AB[54-i] = 0;
8:    endif
9:    R = R << 1;
10: end for
```

In line 1, the remainder is initialized with A's mantissa. In each iteration, if the difference between the contents of register R and B's mantissa is greater then or equal to 0, a 1 is inserted in the current bit position of the quotient register as shown in line 3 and 4. Next, the difference is stored in the remainder as shown in line 5. Otherwise, a 0 is inserted in the current bit position of the quotient register as shown in line 7. At the end of the iteration, the contents of the remainder register are shifted to the left by one bit as shown in line 9. Note that the insertion of bits in M_{AB} starts from the most significant bit in the first iteration and proceeds towards the least significant bit in M_{AB}. The mantissa division takes 55 clock cycles to complete.

3.1.6 Normalization

If the quotient M_{AB} is not normalized, this block normalizes it based on the quotient obtained form the mantissa division. If the most significant bit of the mantissa M_{AB} is 0, it is shifted to the left by one bit. Otherwise, the mantissa is copied to M_n as it is.

```
Input: M_AB  55-bit mantissa
Output: M_n  55-bit mantissa
        F_a1  adjust flag

1: F_a1 = 0;
2: if M_AB[54] = 0
3:    M_n[54..1] = M_AB << 1;
4:    M_n[0] = 0;
5:    F_a1 = 1;
6: else
7:    M_n = M_AB;
8: endif
```

If the most significant bit of the mantissa is 0, it is shifted to the left by one bit and copied to M_n as shown in lines 2 and 3. Next the least significant bit of M_n is updated to 0 in line 4 and the adjustment flag is set in line 5. Otherwise, the mantissa is copied to M_n as shown in line 7.

3.1.7 Rounding Control

In this block, the sticky bit is computed first. Next, the sticky, guard, and the round bits are used to determine whether rounding is necessary or not. The latter two bits are the least significant bits of the 55-bit quotient, namely M_n.

```
Input: Mn  55-bit mantissa
Output: F_r  round flag

1: F_r = 0;
2: s = (M_n[0] ∨ M_n[1] ∨ … M_n[55]);
3: if ((M_n[0] = 0) and (s = 0))
4:    if ((M_n[1] = 1) and (M_n[2] = 1))
5:       F_r = 1;
6: else
7:    if (M_n[1] = 1)
8:       F_r = 1;
9: endif
```

This rounding approach implements the round-to-nearest-even mode of rounding documented in the IEEE standard. Among the four rounding modes specified by the IEEE

standard, this mode is considered the default mode. When rounding occurs, the rounding flag F_r is set.

3.1.8 Rounding

The rounding block rounds the quotient's mantissa based on the decision taken in the rounding control block. If a rounding decision has been made, meaning when $F_r = 1$, then a 1 is added to the least significant bit of the input M_n, and the flag F_{a2} is set for exponent adjustment. Otherwise, no action is taken. The output of this block, M_Q, is the mantissa of Q.

```
Input: Mn  55-bit mantissa
       Fr   round flag
Output: MQ  53-bit mantissa
        Fa2  adjust flag

1: Fa2 = 0;
2: if (Fr = 1)
3:    MQ = Mn[52..0] + 1;
4:    Fa2 = 1;
5: else
6:    MQ = Mn[52..0];
7: endif
```

3.1.9 Exponent Adjustment

The exponent adjustment block adjusts the exponent based on the decision taken in the rounding block. If $F_{a2} = 1$, then E_a is incremented and the result is stored in E_Q. Otherwise, no action is taken. The output E_Q is the exponent of Q.

```
Input: Ea   11-bit exponent
       Fa2   adjust flag
Output: EQ  11-bit exponent

1: if (Fa2 = 1)
2:    EQ = Ea + 1;
3: else
4:    EQ = Ea;
5: endif
```

3.1.10 Packing

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

3.2 Pipelining of the Divider

In the divider, the slowest component is the block which computes the division of the mantissas. Since this block executes this division sequentially, any incoming operands have to be stopped until all iterations are complete. As a result, the throughput of the divider is significantly low. This throughput is $\tau_{seq} = \frac{1}{nd}$ where n is the number of iterations in the sequential divider while d is the execution delay of a single iteration. Pipelining this divider will definitely increase their throughputs. A straightforward way to pipeline iterative algorithms is to unroll the iterations of the loops embedded within the algorithm. In this case, the 55 iterations of the loop, which computes the division of the two mantissas, can be unrolled 55 times. As such, the throughput is $\tau_{pipe} = \frac{1}{md}$ where m is the number of unrolled iterations per stage and d is as defined above. Note that $1 \leq m \leq n$ where $m = 1$ represents a fully unrolled sequential design while $m = n$ represents the un-pipelined sequential design. However, full unrolling can theoretically increase the area cost. In fact, the area of a pipeline design can be expressed as $A_{pipe} = nc + \frac{n}{m} r$ where c is the combinational area of a single iteration, r is the number of bit registers required for a single pipeline stages, and m and n are as defined above. Note that in A_{pipe}, m varies while nc is constant regardless of how many iterations are packed into a single pipeline stage. Furthermore, A_{pipe} is at its maximum when $m = 1$. Faced with this difficulty, it

would make sense to consider (i) partially unrolling the loops, or (ii) optimizing the operations of a single iteration, in order to decrease this complexity. Considering the above factors, the divider is pipelined to various degrees in order to assess the impact of the pipeline depth on area overhead.

3.3 Verification of the Divider

This section presents the modeling of the sequential and the pipelined dividers, and shows the simulation results obtained for these dividers.

3.3.1 Modeling of the Sequential and Pipelined Dividers

To verify the divider, 5 VHDL models were developed where the first model implements the sequential divider while the remaining four models implement the 7, 14, 28, and 55-stage pipelined dividers. Table 6 shows the entities modeled in VHDL for each divider where the VHDL models of the sequential divider and its four pipelined versions total 5117 lines of code.

Table 6: Breakdown of VHDL lines of codes based on divider entities.

Module	**Entity**	**VHDL lines of code**
Sequential Divider	NON_PPL_DIVIDER (top level)	172
	UNPACK_DIVIDER	86
	SIGN_LOGIC_DIVIDER	25
	EXPONENT_SUBTRACTION_DIVIDER	23
	BIAS_ADDITION_DIVIDER	23
	DIVIDER_NON_PPL	72
	NORMALISE_DIVIDER	30
	ROUNDING_CONTROL_DIVIDER	30
	ROUND_DIVIDER	31
	EXPONENT_ADJUST_DIVIDER	27
	PACK_DIVIDER	29
	Subtotal	548
55-stage Divider	PPL_DIVIDER (top level)	172

	UNPACK_DIVIDER	86
	SIGN_LOGIC_DIVIDER	25
	EXPONENT_SUBTRACTION_DIVIDER	23
	BIAS_ADDITION_DIVIDER	23
	OVERFLOW_DIVIDE	741
	NORMALISE_DIVIDER	30
	ROUNDING_CONTROL_DIVIDER	30
	ROUND_DIVIDER	31
	EXPONENT_ADJUST_DIVIDER	27
	PACK_DIVIDER	29
	Subtotal	1,217
28-stage Divider	DIVIDER_PPL_28 (top level)	172
	UNPACK_DIVIDER	86
	SIGN_LOGIC_DIVIDER	25
	EXPONENT_SUBTRACTION_DIVIDER	23
	BIAS_ADDITION_DIVIDER	23
	DIVIDER_28	702
	NORMALISE_DIVIDER	30
	ROUNDING_CONTROL_DIVIDER	30
	ROUND_DIVIDER	31
	EXPONENT_ADJUST_DIVIDER	27
	PACK_DIVIDER	29
	Subtotal	1,178
14-stage Divider	DIVIDER_PPL_14 (top level)	172
	UNPACK_DIVIDER	86
	SIGN_LOGIC_DIVIDER	25
	EXPONENT_SUBTRACTION_DIVIDER	23
	BIAS_ADDITION_DIVIDER	23
	DIVIDER_14	630
	NORMALISE_DIVIDER	30
	ROUNDING_CONTROL_DIVIDER	30
	ROUND_DIVIDER	31
	EXPONENT_ADJUST_DIVIDER	27
	PACK_DIVIDER	29
	Subtotal	1,106
7-stage Divider	PPL_DIVIDER_7 (top level)	172
	UNPACK_DIVIDER	86
	SIGN_LOGIC_DIVIDER	25

	EXPONENT_SUBTRACTION_DIVIDER	23
	BIAS_ADDITION_DIVIDER	23
	DIVIDER_PPL_8	592
	NORMALISE_DIVIDER	30
	ROUNDING_CONTROL_DIVIDER	30
	ROUND_DIVIDER	31
	EXPONENT_ADJUST_DIVIDER	27
	PACK_DIVIDER	29
	Subtotal	1,068
Total		5,117

3.3.2 Simulation of the Divider

The VHDL model of each five different versions of the divider has been verified through extensive simulation using ModelSim 5.8. In this process, separate simulations were performed on each individual entity to insure its functional correctness.

3.3.3 Simulation of the Sequential Divider

Figure 33 shows the simulation snapshot of the top level module of the sequential divider. In this figure, two 64-bit operands, *A* and *B* being 3.75 and 1.5 respectively, are input to the divider. The highlighted a_divide output in the leftmost pane of the simulation snapshot represents the 64-bit quotient produced by the sequential divider. This sequential divider takes 58 clock cycles to produce an output as it calculates one output bit per clock cycle. While the computation progresses through the iterations of the divider, the latter cannot accept any new operands until the division operation is complete. As a result, this divider can take a new pair of operands only after every 58 cycles. The sign of the quotient is calculated as the XOR of the sign

bits of the A and B operands as described in section 3.1.2. In this case, both inputs are positive, and hence the sign of the quotient is also positive.

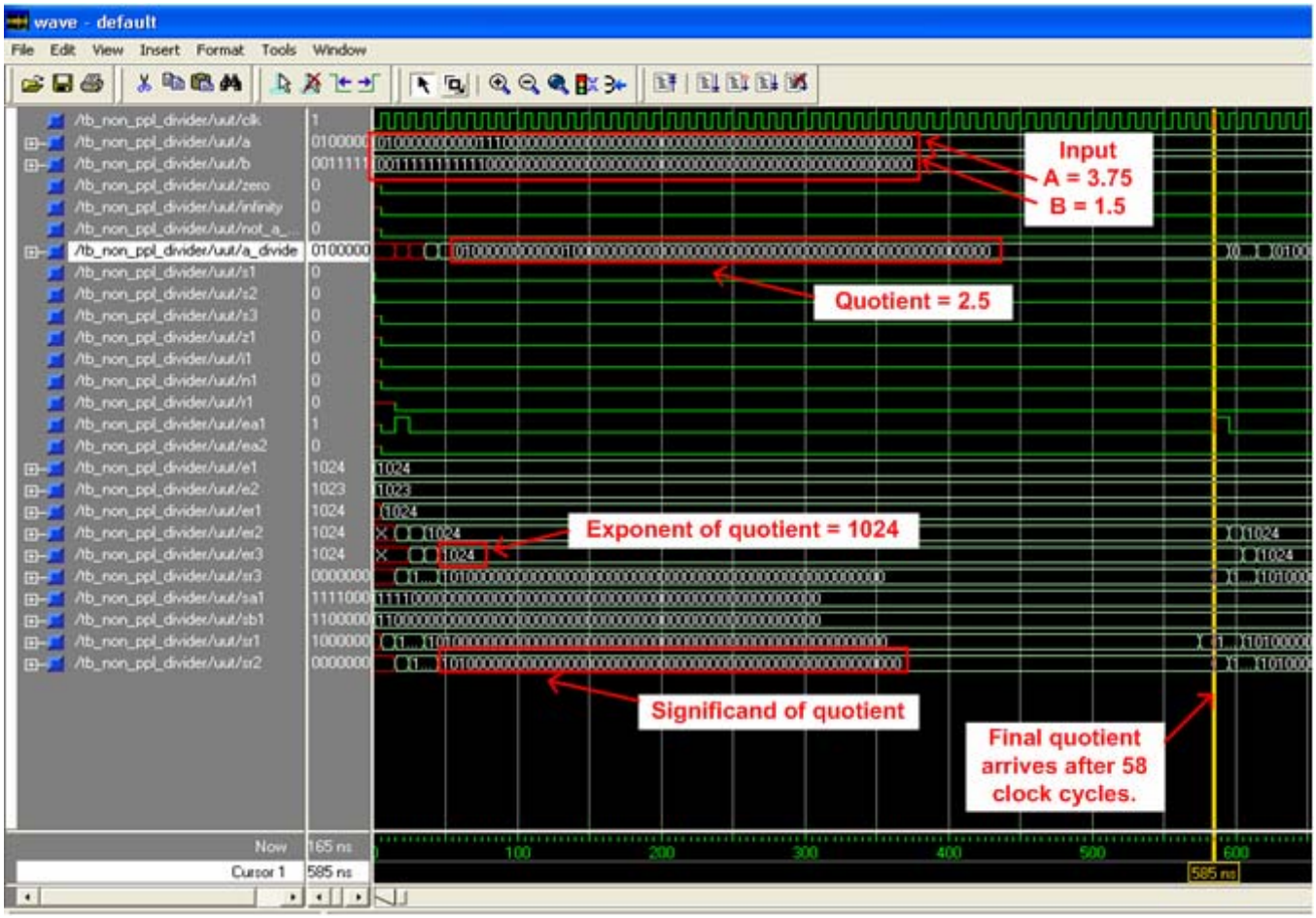

Figure 33: Simulation snapshot of the sequential divider.

On the other hand, the 11-bit exponents of A and B are made biased as described in section 3.1.4. In this case, $A = 3.75_{10} = 11.11_2 = 1.111 \times 10^1$. Since A's exponent is 1, then $E_A = 1 + 1023 = 1024_{10} = 10000000000_2$. Also, since $B = 1.5_{10} = 1.1_2 = 1.1 \times 10^0$, its exponent is $E_B = 0 + 1023 = 1023_{10} = 01111111111_2$. Based on these two exponents, E_{AB} and E_b can be calculated as $E_{AB} = E_A - E_B = 1024 - 1023 = 1$ and $E_b = 1023 + E_{AB} = 1023 + 1 = 1021_{10} = 10000000000_2$ respectively. Finally, the quotient mantissa

can be computed as $M_Q = 2^{E_b} \times (1 + fraction) = 2^1 \times (1 + 2^{-2}) = 2 \times 1.25 = 2.5$. In the simulation snapshot, the quotient a_divide represents the expected output M_Q.

3.3.4 Simulation of the 55-stage Pipelined Divider

In this divider, pipeline registers are inserted after each iteration of the mantissa division (i.e., m = 1) as shown in Figure 34. Since 55 iterations are required to calculate the quotient of the division, 55 registers are inserted in the mantissa divider. This maximum pipeline depth produces maximum throughput. After an initial latency of 55 clock cycles, a new output is produced every clock cycle.

Figure 35 shows a simulation snapshot of the 55-stage pipelined divider. The simulation illustrated in this figure consists of feeding the first pair of operands to the divider, namely A = 3.74 and B = 1.5, in the first clock cycle, followed by a second pair of operands, namely A = 10.5 and B = 2.5.

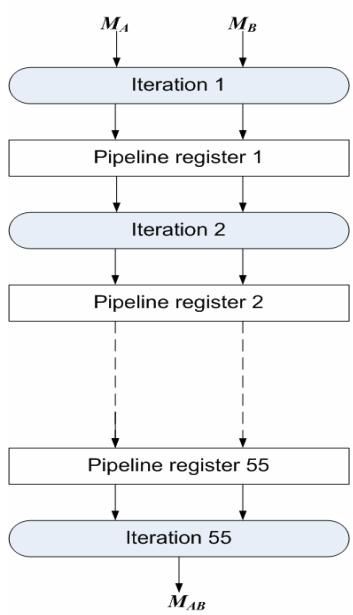

Figure 34: Register placement in the 55-stage pipelined divider.

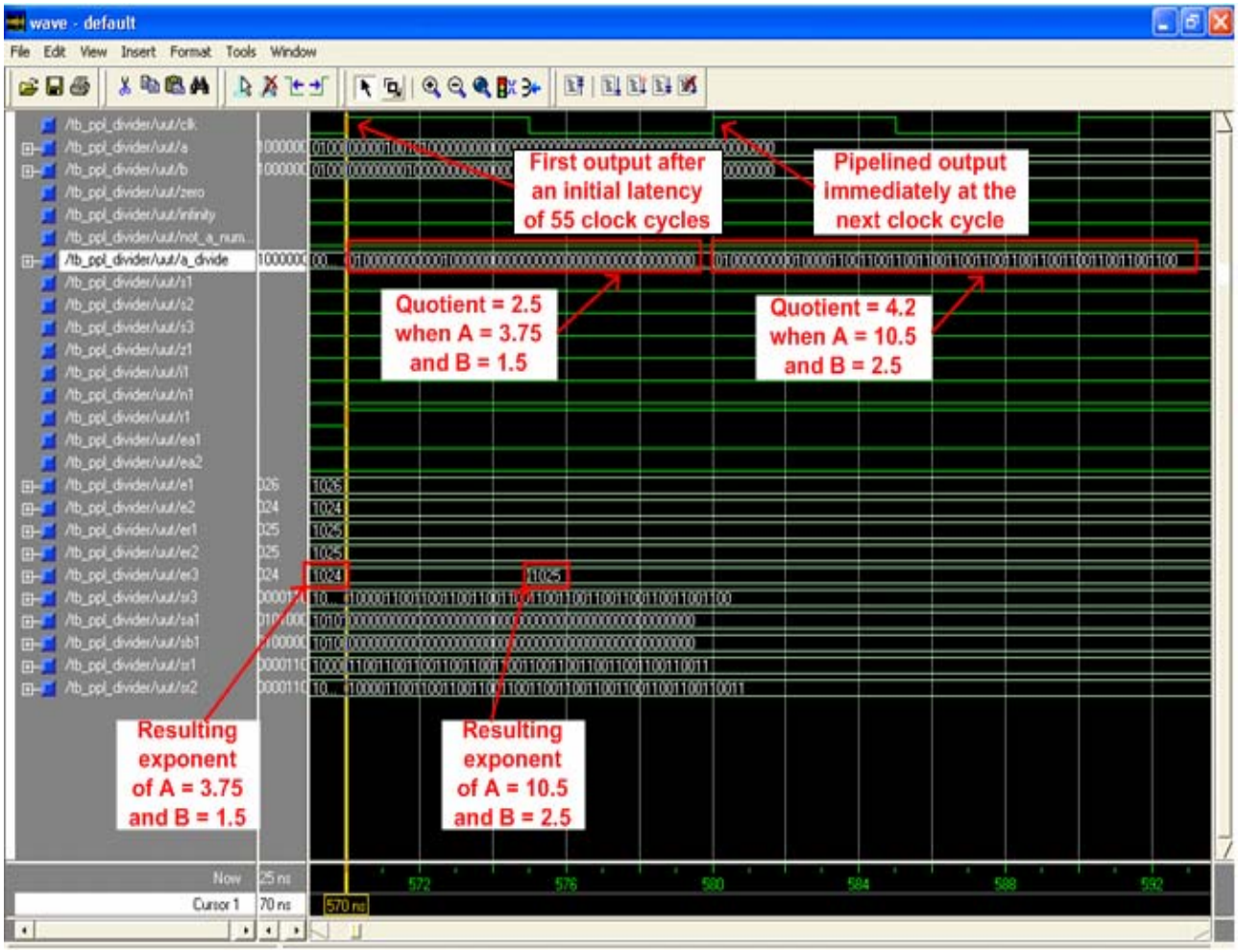

Figure 35: Simulation snapshot of the 55-stage pipeline divider.

The simulation run goes through 55 cycles before the first quotient 2.5 appears at the output of the divider. This is due to the 55-cycle initial latency of this divider. Immediately after the first quotient, the second quotient 4.2 appears at the output of the divider in the following cycle.

3.3.5 Simulation of the 28-stage Pipelined Divider

In this divider, pipeline registers are inserted after every two iterations of the mantissa division (i.e., $m = 2$) as shown in Figure 36. This results in the insertion of 28 pipeline registers.

The achieved throughput is roughly half of the throughput achieved by the 55-stage divider with a 28-cycle initial latency.

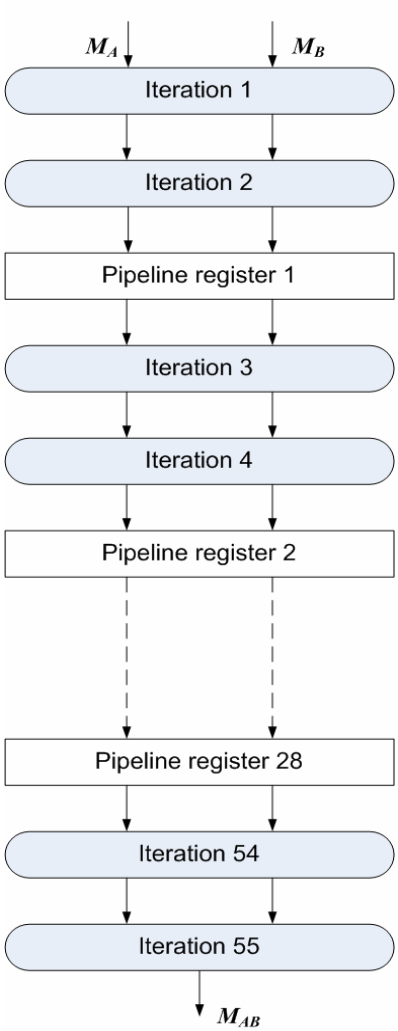

Figure 36: Register placement in the 28-stage pipelined divider.

Figure 37 shows a simulation snapshot of the 28-stage pipelined divider. Similarly to the simulation of the 55-stage divider, this simulation consists of feeding the same two pairs of operands in consecutive cycles to the divider. This simulation run show how both quotients appear in consecutive cycles after the 28-cycle initial latency of the pipelined divider.

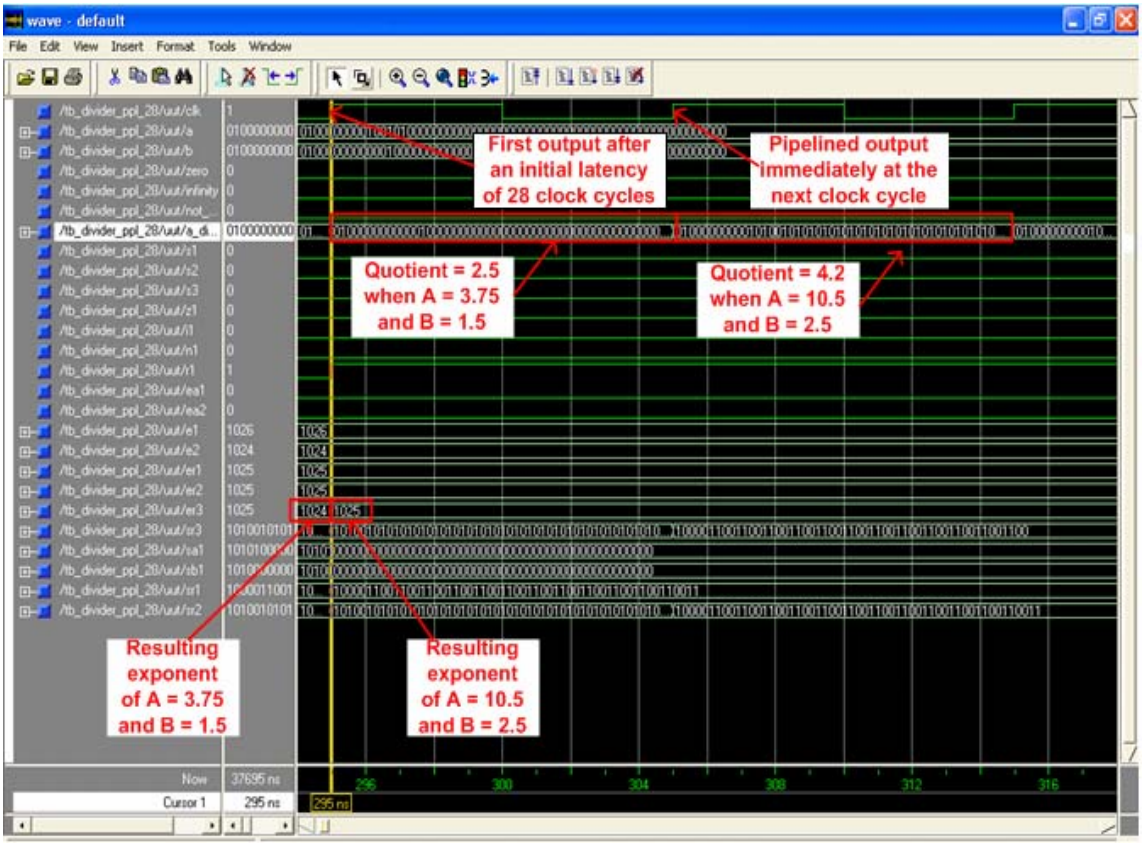

Figure 37: Simulation snapshot of the 28-stage pipelined divider.

3.3.6 Simulation of the 14-stage Pipelined Divider

In this divider, pipeline registers are inserted after every four iterations of the mantissa division (i.e., $m = 4$) as shown in Figure 38. In total, 14 pipeline registers are inserted in the pipeline. The achieved throughput of this divider is roughly half of the throughput of the 28-stage divider.

Figure 39 shows a simulation snapshot of the 14-stage pipelined divider. Similarly to the simulation of the 55-stage divider, this simulation consists of feeding the same two pairs of operands in consecutive cycles to the divider. This simulation run show how both quotients appear in consecutive cycles after the 14-cycle initial latency of the pipelined divider.

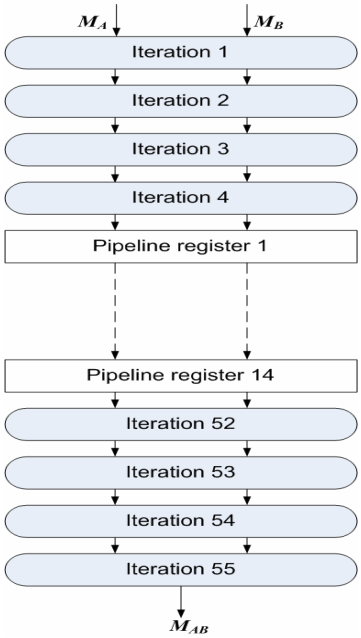

Figure 38: Register placement in the 14-stage pipelined divider.

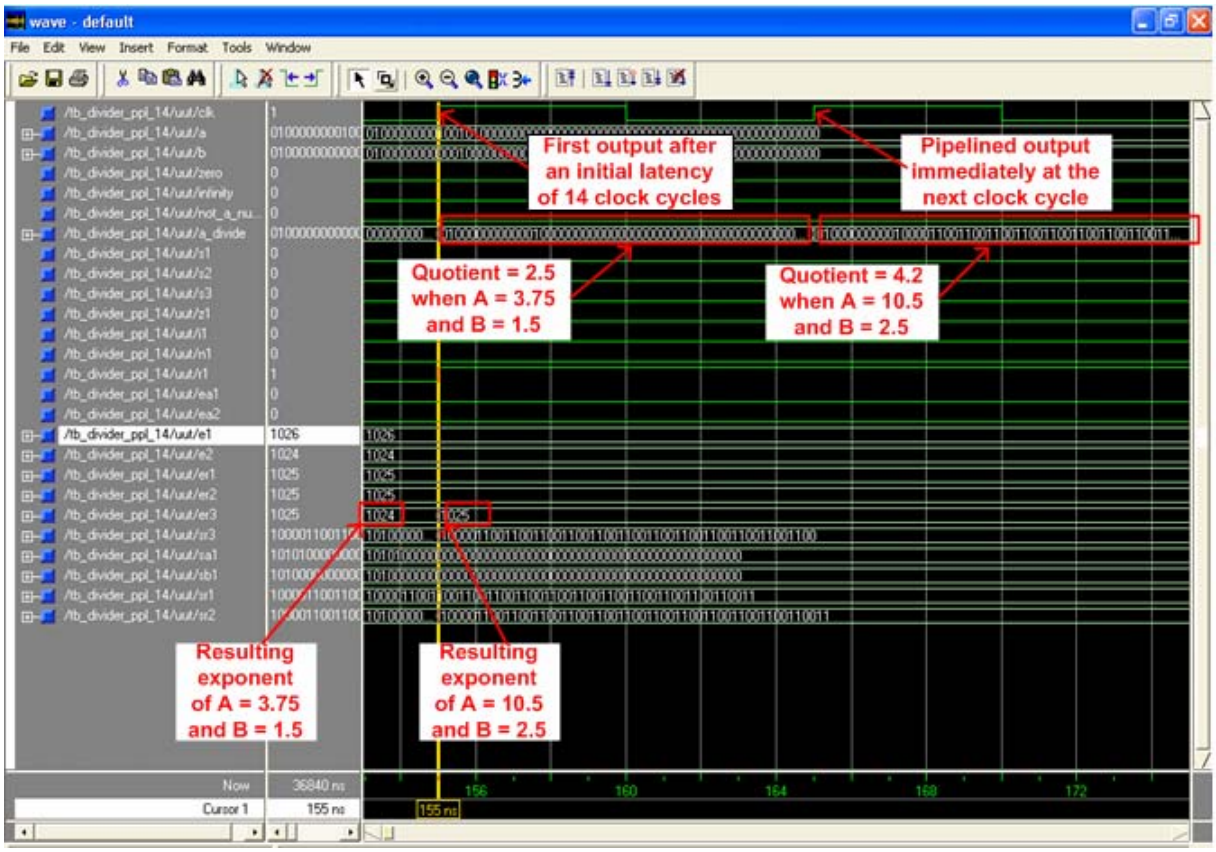

Figure 39: Simulation snapshot of the 14-stage pipelined divider.

3.3.7 Simulation of the Seven-stage Pipelined Divider

In this divider, pipeline registers are inserted after every eight iterations of the mantissa division (i.e., $m = 8$) as shown in Figure 40. In total, 7 pipeline registers are inserted in the entire divider. The throughout achieved by this divider is roughly half of the throughput achieved by the 14-stage divider.

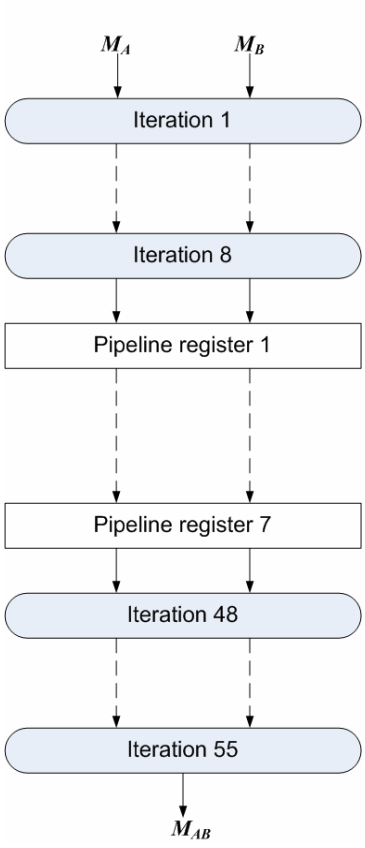

Figure 40: Register placement in the seven-stage pipelined divider.

Figure 41 shows a simulation snapshot of the 7-stage pipelined divider. Similar to the simulation of the 55-stage divider, this simulation consists of feeding the same two pairs of operands in consecutive cycles to the divider. This simulation run show how both quotients appear in consecutive cycles after the seven-cycle initial latency of the pipelined divider.

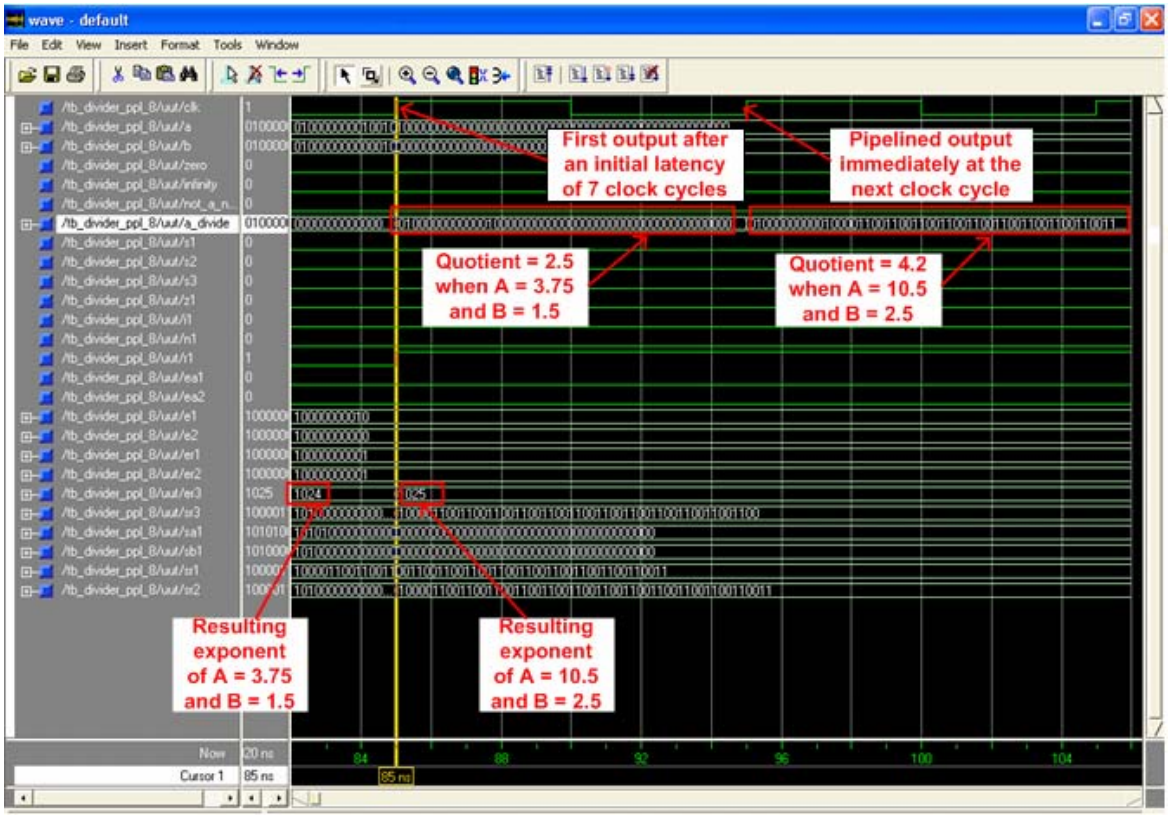

Figure 41: Simulation snapshot of the seven-stage pipelined divider.

3.4 Evaluation of the Divider

In this section, the results of the synthesized divider are compared to earlier related dividers implemented on FPGAs. This comparison is followed by an evaluation of the impact of pipeline depth on area, throughput, and dynamic power of the divider.

3.4.1 Divider Design Comparison

The divider unit was modeled in VHDL, simulated in ModelSim 5.8, synthesized using Synplify Pro 7.2, and placed using Xilinx ISE 5.2. Table 7 shows the implementation results of the divider units on the Virtex XCV1000 chip. These units were mapped on this chip in order to compare our results with the sequential results obtained in [17] where no pipelined

implementations of the units were presented. Sequential implementation of the divider has a throughput that is quite comparable to the throughputs in [17]. This implementation is significantly low in area overhead measured in terms of LUTs as shown in Table 7 although its throughput hovers around the single MFLOP mark. In contrast, the pipelined implementation displays a throughput that is significantly higher in this unit. In fact, the throughput of the pipelined divider is 62× higher than its iterative counterpart. The 62× increase in the divider throughput leads to a modest 13× increase in slices. This shows that the pipelined implementation of the divider is highly efficient since the gain in performance is offset by a relatively low cost in area overhead.

Table 7: Implementation results of the divider units on the Virtex XCV1000.

Unit	**Clock Period (ns)**	**Clock Frequency (MHz)**	**Throughput (MFLOPS)**	**Slices**	**LUTs**	**Flip-Flops**
Sequential divider	15.33	65.20	1.03	222	443 (1%)	274
Pipelined divider	14.90	67.10	64	2915	5830 (23%)	5734

With the exception of a few attempts, most previously published designs of division address only single or parameterizable precision floating point implementation [15]. For a meaningful comparison, it would make sense to map both units on the same chips used in [10, 14]. Table 8 contrasts our implementation results to the results obtained in [14].

Table 8 : Performance of the divider units on the Virtex II XC2V6000.

Unit	**Clock Period (ns)**	**Clock Frequency (MHz)**	**Clock Cycles**	**Latency (ns)**	**Throughput (MFLOPS)**	**Area (Slices)**	**Throughput/Area (KFLOPS/Slice)**
Sequential divider	9.93	100.70	60	595.8	1.60	284	5.63
Array divider [14]	300	3.33	1	300	3.17	1705	1.85
Pipelined divider (60 stages)	9.75	102.50	1	9.75	97.81	2920	33.49
Pipelined divider (29 stages) [14]	12.20	81.96	1	12.20	78.17	2595	30.12

Column 1 shows the units to be compared and the number of stages in the pipelined units while column 2 shows the clock period of the critical path of each unit. Column 3 shows the clock frequency of the implemented unit while column 4 shows the number of clock cycles required to produce a single output. Column 5 shows the latency or the time required to produce a single output while column 6 shows the throughput of each unit measured in Mega floating point operations per second (MFLOPS). Column 7 shows the number of slices needed to implement the unit of the XC2V6000 chip while column 8 shows the throughput per area of each unit. While our non-pipelined units are sequential in nature, the units in [14] are not. As a result, the areas required to implement our divider units are significantly smaller than the areas required to implement the units in [14]. They can occupy only 0.16× of the divider compared to the areas of the array units in [14]. On the other hand, the latencies in our sequential unit is 1.98× higher for the divider than the latency of the non-pipelined units in [14] although the clock periods in our unit is significantly lower than the clock periods in the non-pipelined units in [14]. However, the latencies of our pipelined unit is 0.79× smaller for the divider than the latencies of the pipelined units in [14]. This can be attributed to the high degree of pipelining introduced in our units. In fact, the iterative computations in our divider units were fully unrolled to yield a 55-stage pipeline in the mantissa divider block of the divider unit. In contrast, the pipelined divider in [14] consists of only 29 stages. Considering this difference in pipeline stages, the areas measured in slices of our implemented pipelined unit is obviously higher than the area of the pipelined unit in [14].

Although the number of pipeline stages differs over implementations, a relatively accurate metric to assess the efficiency of these various implementations can be derived by considering the ratio of throughput over area shown in column 8 of Table 8. This ratio gives a

rough idea about the level of throughput produced by a single slice regardless of the area used in the implementation. Based on this ratio, our pipelined divider is 1.11× more efficient that the pipeline divider in [14]. Although our design is sequential in nature and does not take advantage of other radices to reduce delay as done in [14], it is clear that it is quite comparable in performance to the design in [14]. Furthermore, it is worth noting that designs based on radix-2 computations, such as ours, are easier to implement and test. Note that these throughputs were achieved by merely spreading spatially the computations across the LUTs of the FPGA. The performance of our unit can be boosted further by constructing highly optimized layouts of the units which take advantage of the block RAMs and embedded multipliers within the Virtex FPGA.

3.4.2 Throughput Evaluation

In order to understand the impact of loop unrolling on area and throughput, the divider was pipelined into different depths as described in section 3.3.2. The pipeline depth, or number of pipeline stages, depends on the degree of unrolling of the iterative loops in the division unit of the mantissa. Experiments were conducted to measure area and performance parameters by partitioning the mantissa's divider into pipelines of 1, 7, 14, 28, and 55 stages. The four last depths can be obtained by embedding eight iterations, four iterations, two iterations, and one iteration per stage respectively. Figure 42 shows clock frequency and throughput across the mentioned pipeline depths for the divider units on the Xilinx XC2V6000-4 chip.

The figure shows that the frequency and throughput of a divider increases as the pipeline depth increases in a non linear fashion. The sequential divider unit is considered as one-stage

pipeline. Note that in these designs, the iterative computations of the loop are performed in the same small area leading to a short clock period, and subsequently a high clock frequency.

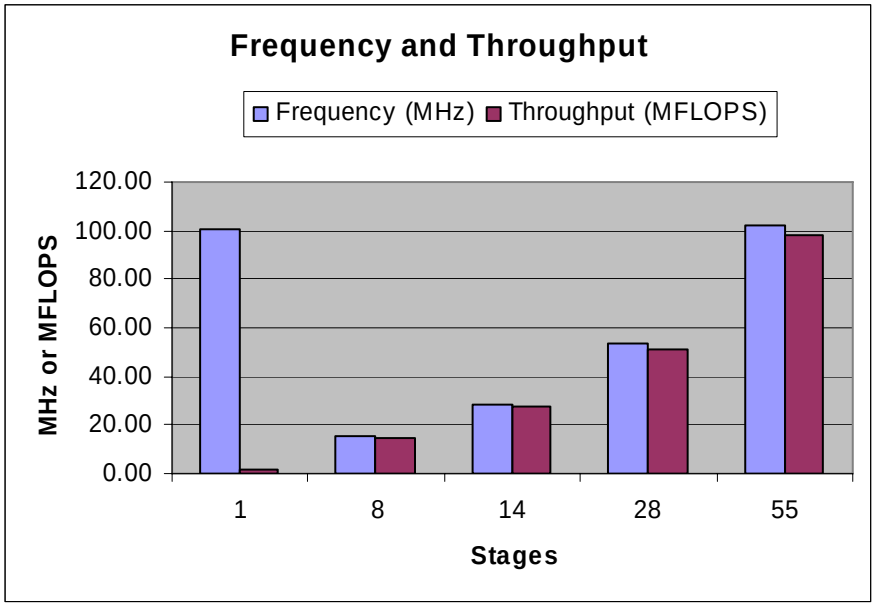

Figure 42: Clock frequencies and throughputs of the dividers.

However, the 55 iterations of the loop have to be completed for division computations before an output is produced. This leads to a significantly low throughput as shown in Figure 42. This shows that a complete unrolling of the loop in the iterative design of the divider can provide higher level of throughputs. However, this gain in throughput can be accompanied by an area penalty.

3.4.3 Area Evaluation

Figure 43 shows area cost, measured in slices, LUTs, and flip-flops for various pipeline depths of the divider units on XC2V6000-4. As this figure shows, the overall area increases as the pipeline becomes deep. Among LUTs and flip-flops, the increase seems to be more pronounced for the latter. This can be explained by the fact that, except for the sequential

design, the pipelined designs have roughly the same amount of combinational logic regardless of the degree of loop unrolling.

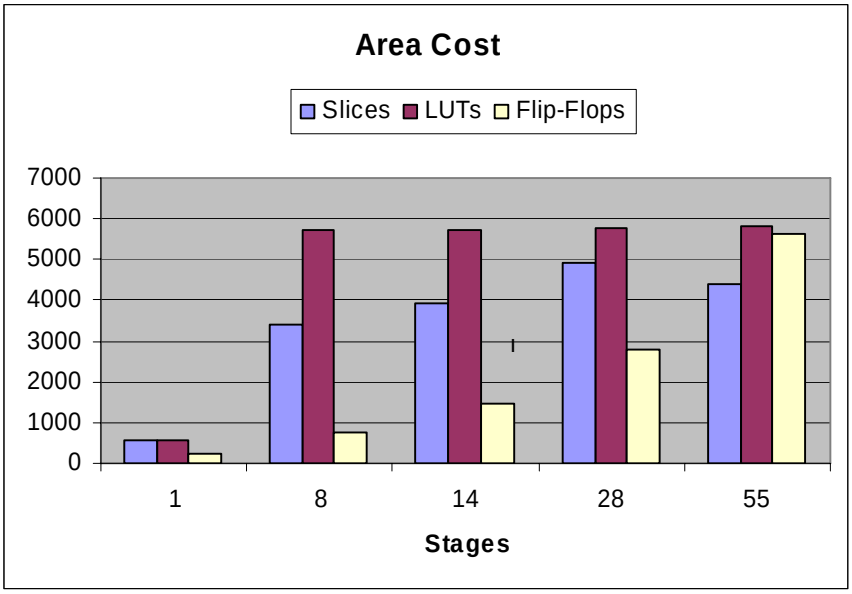

Figure 43: Area costs of the dividers.

As such, the increase in area tends to affect the number of flip-flops since flip-flops are gradually added to implement the increase in the number of inter-stage registers required to support deeper pipelines. By examining the trend lines of this increase, it is clear that it is non-linear across the implemented divider units. With regard to slices, it seems that their numbers reach their maximum in the 28-stage pipeline. The trend line of this increase is an extremely flat bell curve spanning the pipelines of 8 to 55 stages. This shows that the degree of unrolling does not impact significantly the slice area needed to implement the pipeline. One can speculate that roughly the same slice area is used to implement the same amount of combinational logic embedded in the four pipelines. As more stages are added to a pipeline, the flip-flops contained in the same slices are being used to implement the increasing numbers of pipeline registers. If

additional flip-flops are needed, additional slices are marshaled to provide the needed flip-flop thus leading to a slight increase in slices.

When considering the throughput and area cost, one can quantify the incurred area penalty associated with throughput gain as a ratio. Figure 44 shows the throughput-area ratio of the divider units on XC2V6000-4. This ratio can be usefully used in measuring the level of throughput provided by unit of area expressed as a single slice.

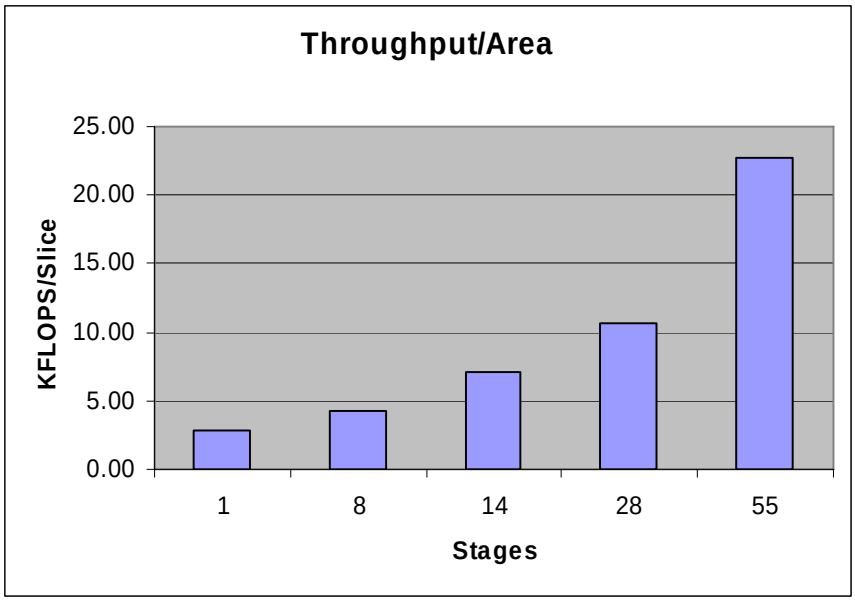

Figure 44 : Throughput-area ratios of the dividers.

As this figure shows, this ratio increases as the pipeline depth increases. In fact, by further unrolling the loop, and subsequently adding more stages to the pipeline, one is not adding combinational logic, but merely shortening the critical path in each pipeline stage. This results in a speedup of the clock of the pipeline without a significant increase in area. The net effect is a visible increase in the throughput-area ratio. These observations suggest that in iterative designs, maximum performance benefits can be obtained by totally unrolling the iterative loops of the computations without incurring a significant area penalty. This is somewhat counter-intuitive

considering that advanced pipelining in ASIC implementations always lead to a gradual increase in area penalty. This increase can reach a point where the throughput-area ratio starts to gradually decrease as more stages are added to the pipeline after which partitioning further the pipeline can only yield diminishing returns in terms of throughput-area ratio.

3.4.4 Dynamic Power Evaluation

In order to understand the impact of loop unrolling on dynamic power consumption, the divider was pipelined into different depths as described in section 3.3.2. Each pipelined divider is placed and routed using Xilinx ISE 6.2i on a Virtex XCV600 since the latter is the closest chip in architecture to the XC2V6000 chip used in the area and throughput experiments that is provided by Xilinx tools for power analysis. After setting the simulation clock period to 10 ns, a simulation run of 700 ns is performed on each divider by feeding a diverse set of 70 input vectors. The output of this simulation produces a vcd file which is fed to the XPower tool packaged with Xilinx ISE software tools. XPower reads the vcd file and generates a report showing the switching power consumed by various components such as IO blocks, logic, routing, clocking, etc…Figure 45 shows the adopted methodology to conduct power analysis on the pipelined dividers. For a focused power analysis, we disregarded the power consumed by all device-dependent components such as IO blocks. In return, only design-dependent power components were taken into account. These components are the switching power of the clock tree, design logic, and routing signals. Figure 46 shows the dynamic power consumed by the pipelined dividers.

As shown in the figure, dynamic power decreases as pipeline depth increases. However, the values of dynamic power consumed by the 14, 28, and 55-stage dividers are all below 7,000

mW in contrast to the 142,750 mW consumed by the 8-stage divider based on the log scale of the y-axis in Figure 46. The latter figure is equivalent to 21.54× more power than the power consumed by the 14-stage divider. Note that the amount of combinational logic is roughly the same in all the dividers. The only difference is the amount of logic embedded within a single stage of the divider.

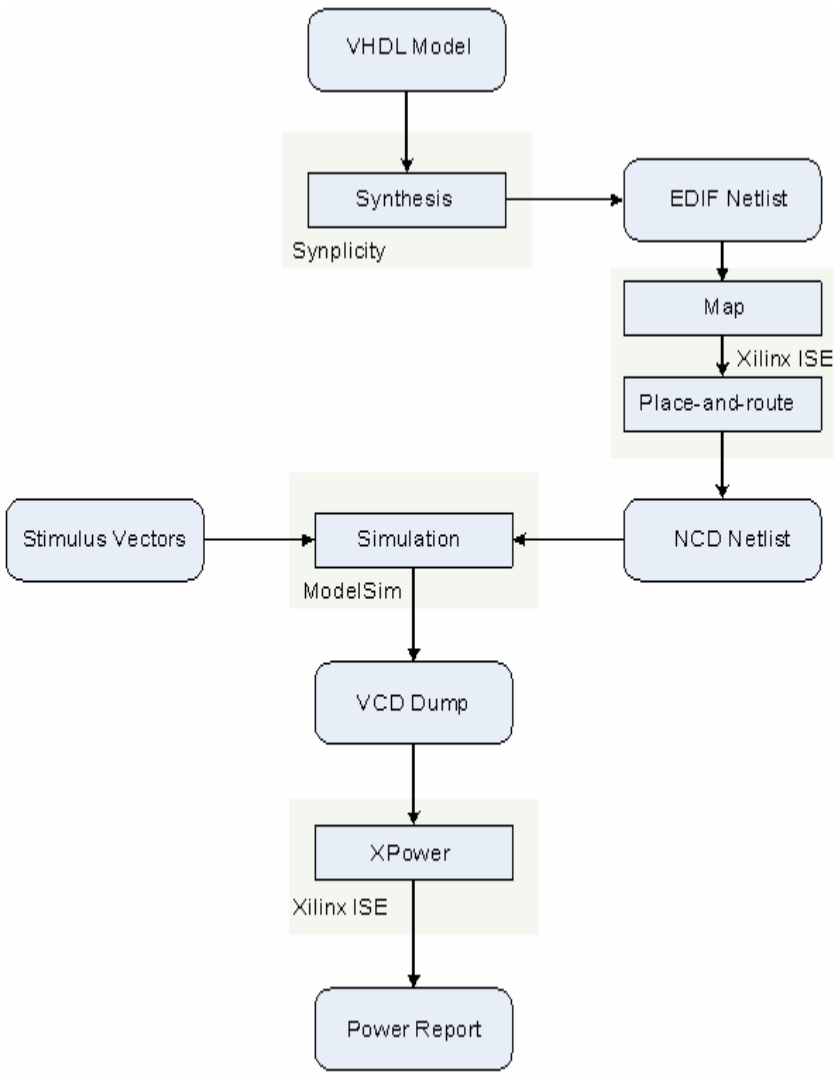

Figure 45 : Power analysis methodology.

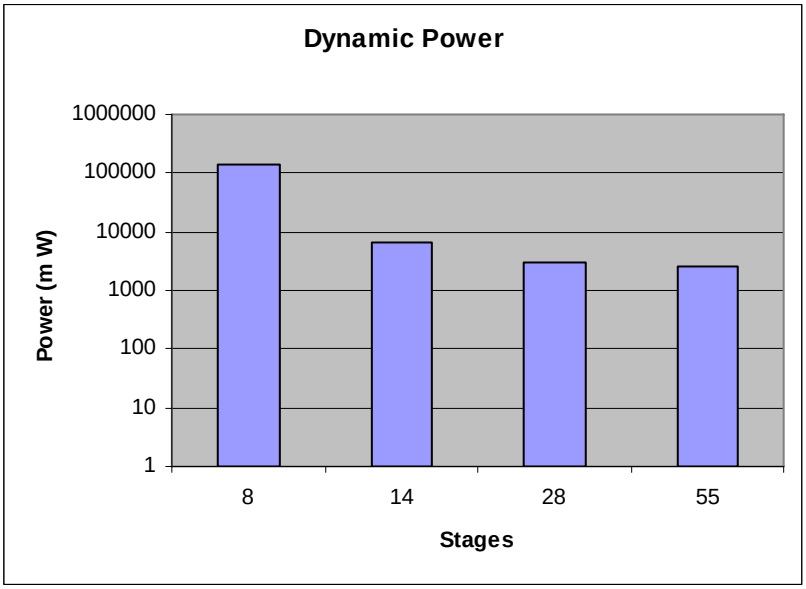

Figure 46 : Dynamic power of the dividers.

In the case of an eight-stage divider, there are eight iterations embedded within a single stage. These eight iterations present a logic network that contains numerous paths which do not have necessarily the same length. In particular, long paths are notorious for displaying glitching behavior. In fact, glitching activity increases with signal length [21]. By inserting registers at various points in the design, the amount of interconnect between registers is reduced thereby reducing the amount of glitching plaguing these signals. As a result, the dynamic power displayed by the divider is also reduced. However, Figure 46 shows that this reduction in dynamic power tends to slow down as the divider is pipelined further. Apparently, there is a large initial payoff in power reduction in the initial pipelined design (e.g., eight-stage pipeline) up to a point beyond which further pipelining does not return any substantial reduction in dynamic power (e.g., 14-, 28-, and 55-stage pipeline). This can be explained by the fact that in shallow pipelined designs, glitching can make up to 80% of the total power consumed by the divider. As the pipeline depth of the design is slightly increased, glitching activity can fall to

levels below 40% of total power displayed by the divider [22]. This sudden fall in glitching activity explains the decreasing returns observed in deeper pipelines. In essence, while pipelining can improve throughput in a non-linear fashion, its impact on reducing power is limited to shallow pipelines.

3.5 Conclusion

This chapter presents the design of IEEE-compliant double precision floating point sequential and parallel dividers. This design is based on a low-radix iterative division algorithm known as the binary version of the pencil-and-paper method. The pipelining of the divider was based on partial and full unrolling of the loops in the iterative mantissa division. The implementation of this divider did not take advantages of any advanced architectural features available in high end FPGA chips or use any pre-designed architecture-specific arithmetic cores. The experiments reveal that this divider can produce maximum throughput when the iteration of the computational loops are totally unrolled without incurring a significant area penalty. While the sequential divider occupies less than 3% of an XC2V6000 FPGA chip, its pipelined counterparts can produce throughputs that exceed the 100 MFLOPS mark by consuming at most a modest 8% of the chip area. These throughputs surpass by far the throughputs of division on many processors [9]. Such performances are indeed perfectly suited to accelerate numeric applications.

CHAPTER FOUR: DOUBLE PRECISION FLOATING POINT SQUARE ROOT UNIT

In this chapter, section 4.1 presents the architecture of the double precision floating point square root unit while section 4.2 describes the approach used to pipeline this unit. Section 4.3 presents the verification of the unit while section 4.4 explains the experimental results conducted on the unit. Finally, section 4.5 concludes the chapter.

4.1 Architecture of the Square Root Unit

As Figure 47 shows, the square root units takes as input a 64-bit number A and outputs its square root R as a 64-bit number.

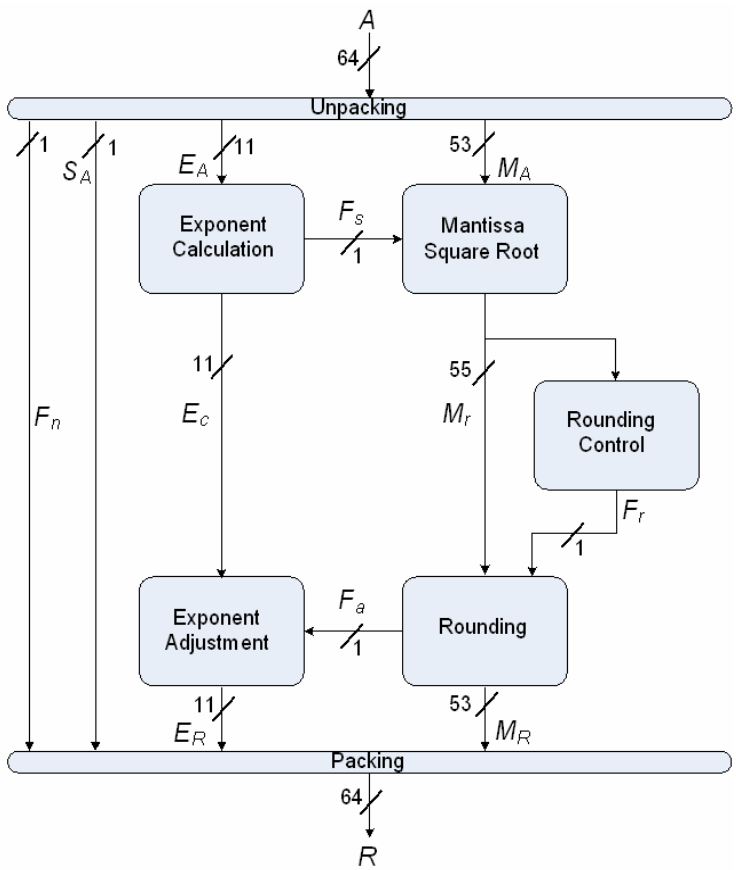

Figure 47: Double precision floating point square root unit.

4.1.1 Unpacking

The unpacking block separates the 64 bits of A in a manner similar to the one performed by the unpacking block of the divider.

4.1.2 Exponent Calculation

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

```
Input: E_A  11-bit exponent
Output: E_c  11-bit exponent
        F_s  shift flag

1: F_s = 0;
1: if (E_A is even)
2:    E_c = (E_A + 1022) >> 1;
3:    F_s = 1;
4: else if (E_A is odd)
5:    E_c = (E_A + 1023) >> 1;
6: end if
```

If the biased exponent is even, it is added to 1022 and divided by two. In addition, the shift flag F_s is set to indicate that the mantissa should be shifted to the left by 1 bit before computing its square root. Note that before shifting, the mantissa bits are stored in the 53-bit register M_A as 1.xxxx… After shifting to the left, M_A contents become 1x.xxx… If the biased exponent is odd, it is added to 1023 and divided by two.

4.1.3 Mantissa Square Root

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

```
Input: M_A  53-bit mantissa
       F_s  1-bit shift flag
Output: M_r 55-bit mantissa
```

```
1: Mr = 000…0;
2: T = 01000…0;
3: for (i = 0; i < 55; i = i + 1)
4:    if (i = 0)
5:       if (Fs = 1)
6:          MA = MA[53].MA[51..0].0;
7:       end if
8:       X = MA >> 1;
9:    else
10:      if (X ≥ T)
11:         X = (X – T) << 1;
12:         T = T[(56-i)..0] >> 1;
13:         T[56-i] = 1;
14:      else
15:         X = X << 1;
16:         T = T[56-i)..0] >> 1;
17:         T[56-i] = 0;
18:      end if
19:   end if
20: end for
21: Mr = T[56..2];
```

In the first iteration, the contents of M_A are shifted to the left and stored in X if the shift flag F_s is on as shown in lines 4, 5, and 6. Otherwise, these contents are shifted to the right by one bit and stored in X as shown in line 8. In other iterations, X is compared to T as shown in line 10. If it is greater or equal to T, the contents of X are subtracted from those of T, shifted to the left, and stored in X as shown in line 11. The contents of T are shifted to the left by one bit starting from the current pointer position as shown in line 12. Then, a 1 is inserted in the current bit position in T as shown in line 13. Note that in each iteration, a pointer points to the current bit that will be replaced in T. If X is less than T, its contents are shifted to the left and stored in X as shown in line 15. The contents of T are shifted to the right by one bit starting from the current pointer position as shown in line 16. Then, a 0 is inserted in the current bit position in T as shown in line 17. After the last iteration, the contents of T, with the exception of the two least

significant bits, are copied to M_r as shown in line 21. The computation of the mantissa's square root takes 55 clock cycles to complete.

4.1.4 Rounding Control

In this block, the rounding decision is executed in the same way as the rounding control block used in division. The only difference is that the mantissa M_r in the square root unit is a 53-bit number instead of the 55-bit mantissa M_n used in the rounding control of the divider. Of course, this rounding control implements the same rounding mode used in the divider.

4.1.5 Rounding

The rounding block rounds the mantissa based on the same approach used in the divider. M_R is the mantissa of R.

```
Input: Mr   53-bit mantissa
       Fr   round flag
Output: MR  53-bit mantissa
        Fa  adjust flag

      1: Fa = 0;

2: if (Fr = 1)
3:    MR = Mr + 1;
4:    Fa = 1;
5: else
6:    MR = Mr;
7: end if
```

4.1.6 Exponent Adjustment

The exponent adjustment block adjusts the exponent based on the same approach used in the exponent adjustment block of the divider.

4.1.7 Packing

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

4.2 Pipelining of the Square Root Unit

In the square root unit, the slowest component is the block which computes the square root of the mantissa. Since this block executes the square root operation sequentially, any incoming operand will have to be stopped until all iterations are complete. As a result, the throughput of the square root unit is significantly low. This throughput is $\tau_{seq} = \frac{1}{nd}$ where n is the number of iterations in the sequential square root unit while d is the execution delay of a single iteration. Pipelining this square root block will definitely increase its throughput. Hence, this unit is pipelined in a way similar to the pipelining approach described in section 3.2 of Chapter 3 for the divider.

4.3 Verification of the Square Root Unit

This section presents the modeling of the sequential and the pipelined square root units as well as the simulation results obtained for these square root units.

4.3.1 Modeling of the Sequential and Pipelined Square Root Units

To verify the square root, five VHDL models were developed where the first model implements the sequential square root unit while the remaining four models implement the seven, 14, 28, and 55-stage pipelined square root units. Table 9 shows the entities modeled in

VHDL for each square root unit along with the numbers of lines of VHDL code resulting in a total of 4,943 lines.

Table 9: Breakdown of VHDL lines of codes based on the entities of the square root units.

Module	Entity	VHDL lines of code
Sequential Square Root Unit	SQUARE_ROOT (top level)	158
	UNPACK_SQRT	56
	EXPO_CALC	66
	SQRT_NON_PPL	93
	ROUND_CONTROL_SQRT	29
	ROUND_SQRT	58
	EXPO_ADJUST_SQRT	56
	PACK_SQRT	30
	Subtotal	546
55-stage Square Root Unit	SQUARE_ROOT_PPL (top level)	158
	UNPACK_SQRT	56
	EXPO_CALC	66
	STAGE2	782
	ROUND_CONTROL_SQRT	29
	ROUND_SQRT	58
	EXPO_ADJUST_SQRT	56
	PACK_SQRT	30
	Subtotal	1,235
28-stage Square Root Unit	SQUARE_ROOT_28 (top level)	158
	UNPACK_SQRT	56
	EXPO_CALC	66
	TRIAL_SQRT	636
	ROUND_CONTROL_SQRT	29
	ROUND_SQRT	58
	EXPO_ADJUST_SQRT	56
	PACK_SQRT	30
	Subtotal	1,089
14-stage Square Root Unit	SQUARE_ROOT_14 (top level)	158
	UNPACK_SQRT	56
	EXPO_CALC	66
	TRIAL_SQRT_14	594

	ROUND_CONTROL_SQRT	29
	ROUND_SQRT	58
	EXPO_ADJUST_SQRT	56
	PACK_SQRT	30
	Subtotal	1,047
7-stage Square Root Unit	SQUARE_ROOT_7 (top level)	158
	UNPACK_SQRT	56
	EXPO_CALC	66
	TRIAL_SQRT_7	573
	ROUND_CONTROL_SQRT	29
	ROUND_SQRT	58
	EXPO_ADJUST_SQRT	56
	PACK_SQRT	30
	Subtotal	1,026
Total		4,943

4.3.2 Simulation of the Square Root Units

The VHDL models of the five versions of the square root units have been verified through extensive simulation using ModelSim 5.8. In this process, separate simulations were performed on each individual unit to insure its functional correctness.

4.3.3 Simulation of the Sequential Square Root Unit

Figure 48 shows the simulation snapshot of the top level entity of the sequential square root unit. In this figure, a 64-bit operand A = 30.25 is input to the square root. The highlighted a_sqrt output in the leftmost pane of the simulation snapshot represents the 64-bit square root produced by the sequential square root unit. This sequential square root unit takes 65 clock cycles to produce an output as it calculates one output bit per clock cycle. As the computation progresses through the iterations of the square root, the unit cannot accept any new

operands until the square root operation is complete. As a result, this square root can take a new operand only after every 65 cycles.

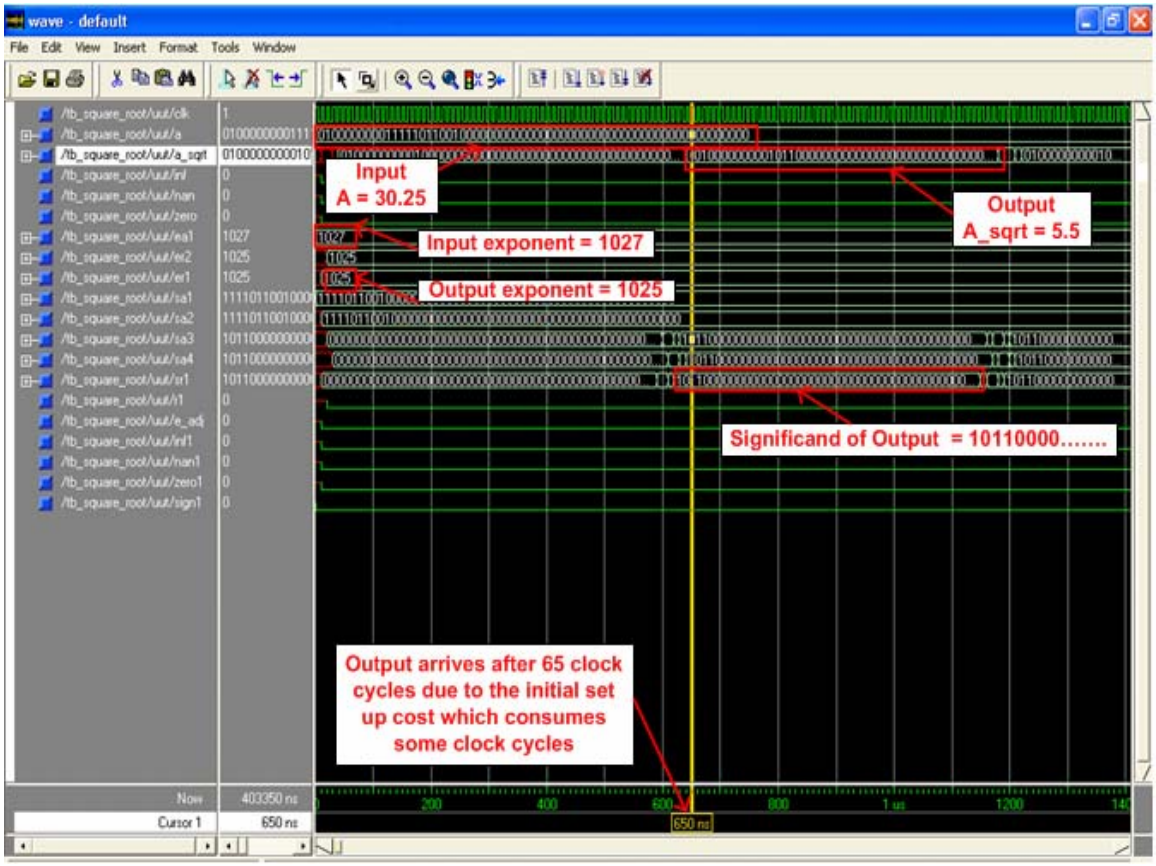

Figure 48: Simulation snapshot of the sequential square root unit.

The 11-bit exponent of A is converted to a biased exponent as described in section 3.1.4. In this case, $A = 30.25_{10} = 11110.11001_2 = 1.111011001 \times 10^4$. Since A's exponent E_A = 4+1023 = 1027 which is odd, then $E_c = (E_A + 1023) \div 2 = 2050_{10} \div 2 = 1025_{10} = 10000000011_2$. Based on this result, $E_R = 1025_{10} = 10000000011_2$ as the rounding bit = 0. Finally, the square root mantissa can be computed as $M_R = 2^{E_R} \times (1 + fraction) = 2^2 \times (1 + 2^{-2} + 2^{-3}) = 4 \times 1.375 = 5.5$. In the simulation snapshot, the quotient a_sqrt represents the expected output M_R.

4.3.4 Simulation of the 55-stage Pipelined Square Root Unit

In this square root unit, pipeline registers are inserted after each iteration of the mantissa square root (i.e., $m = 1$) as shown in Figure 34. Since 55 iterations are required to calculate the quotient of the square root, 55 registers are inserted in the mantissa square root block. This maximum pipeline depth produces maximum throughput. After an initial latency of 55 clock cycles, a new output is produced every clock cycle. Figure 49 shows a simulation snapshot of the 55-stage pipelined square root unit.

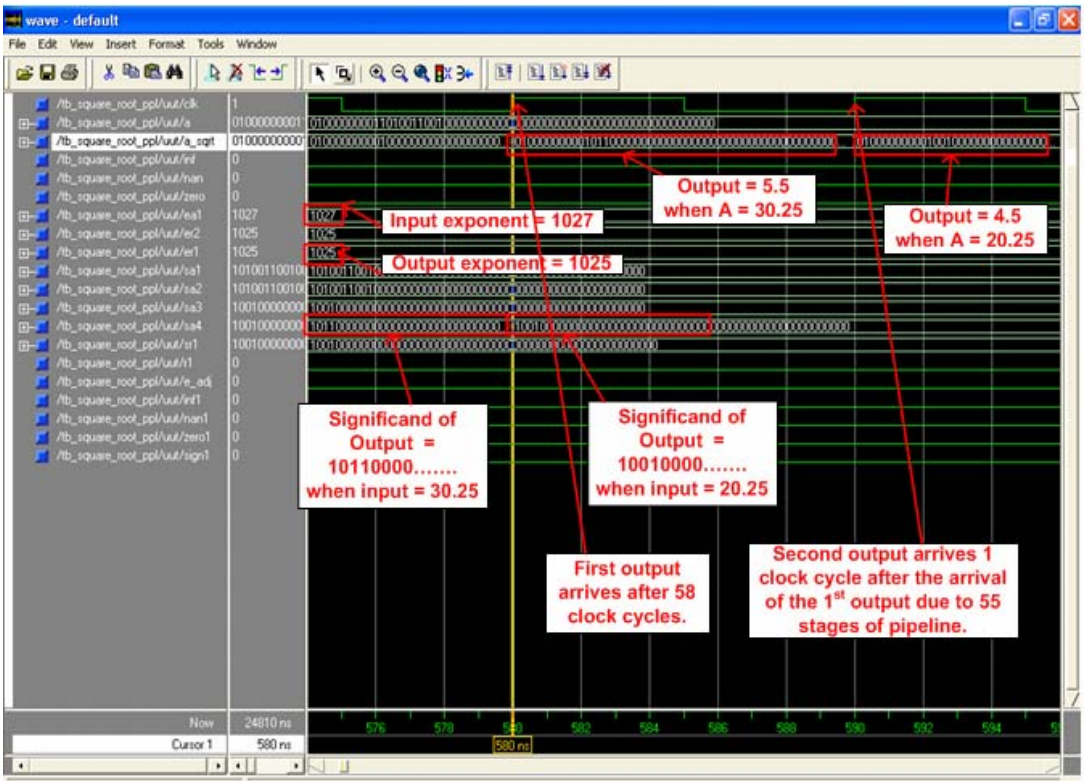

Figure 49: Simulation snapshot of the 55-stage pipelined square root unit.

The simulation illustrated in this figure consists of feeding the first operand to the square root, namely $A = 30.25$, in the first clock cycle, followed by a second operand, namely $A = 20.25$, in the second cycle. The simulation run goes through 55 cycles before the first square root 5.5 appears at the output of the square root unit. This is due to the 55-cycle initial latency of this

unit. Immediately after the first remainder, the second remainder 4.5 appears at the output of the unit in the following cycle.

4.3.5 Simulation of the 28-stage Pipelined Square Root Unit

In this square root unit, pipeline registers are inserted after every two iterations of the mantissa square root (i.e., m = 2) as shown in Figure 36. This results in the insertion of 28 pipeline registers. The achieved throughput is roughly half of the throughput achieved by the 55-stage square root with a 28-cycle initial latency.

Figure 50 shows a simulation snapshot of the 28-stage pipelined square root. Similar to the simulation of the 55-stage square root, this simulation consists of feeding the same operands in consecutive cycles to the square root. This simulation run shows how both square roots appear in consecutive cycles after the 28-cycle initial latency of the pipelined square root unit.

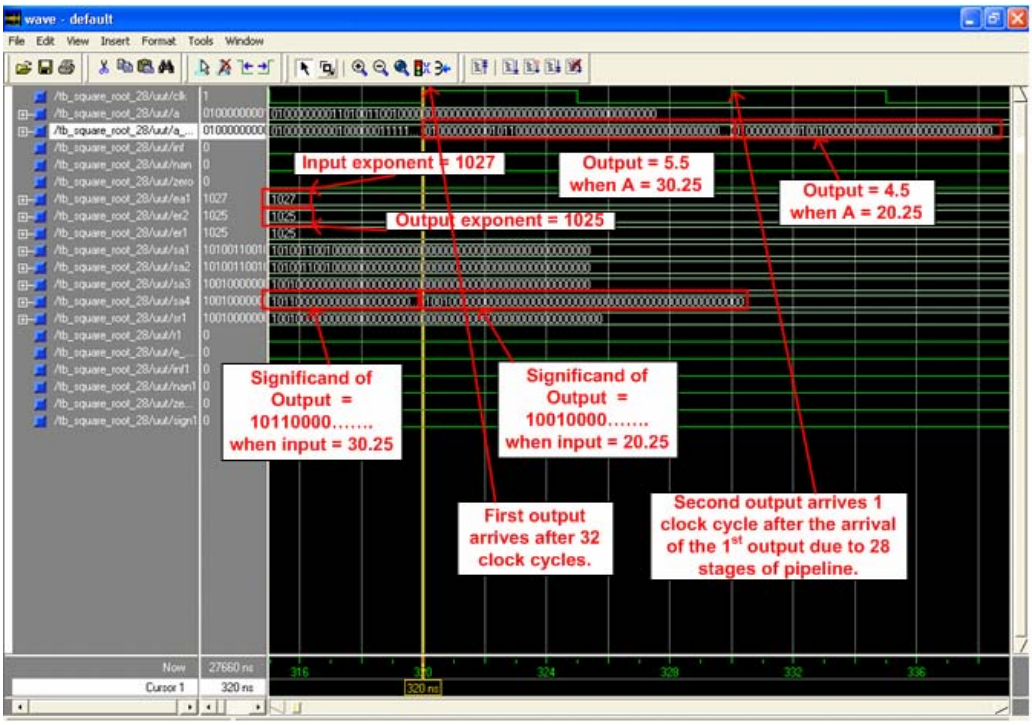

Figure 50: Simulation snapshot of the 28-stage pipelined square root unit.

4.3.6 Simulation of 14-stage Pipelined Square Root Unit

In this square root, pipeline registers are inserted after every four iterations of the mantissa square root (i.e., m = 4) as shown in Figure 38. In total, 14 pipeline registers are inserted in the pipeline. The achieved throughput of this square root unit is roughly half of the throughput of the 28-stage square root unit.

Figure 51 shows a simulation snapshot of the 14-stage square root. Similar to the simulation of the 55-stage square root unit, this simulation consists of feeding the same two operands in consecutive cycles to the square root. This simulation run shows how both remainders appear in consecutive cycles after the 14-cycle initial latency of the pipelined square root unit.

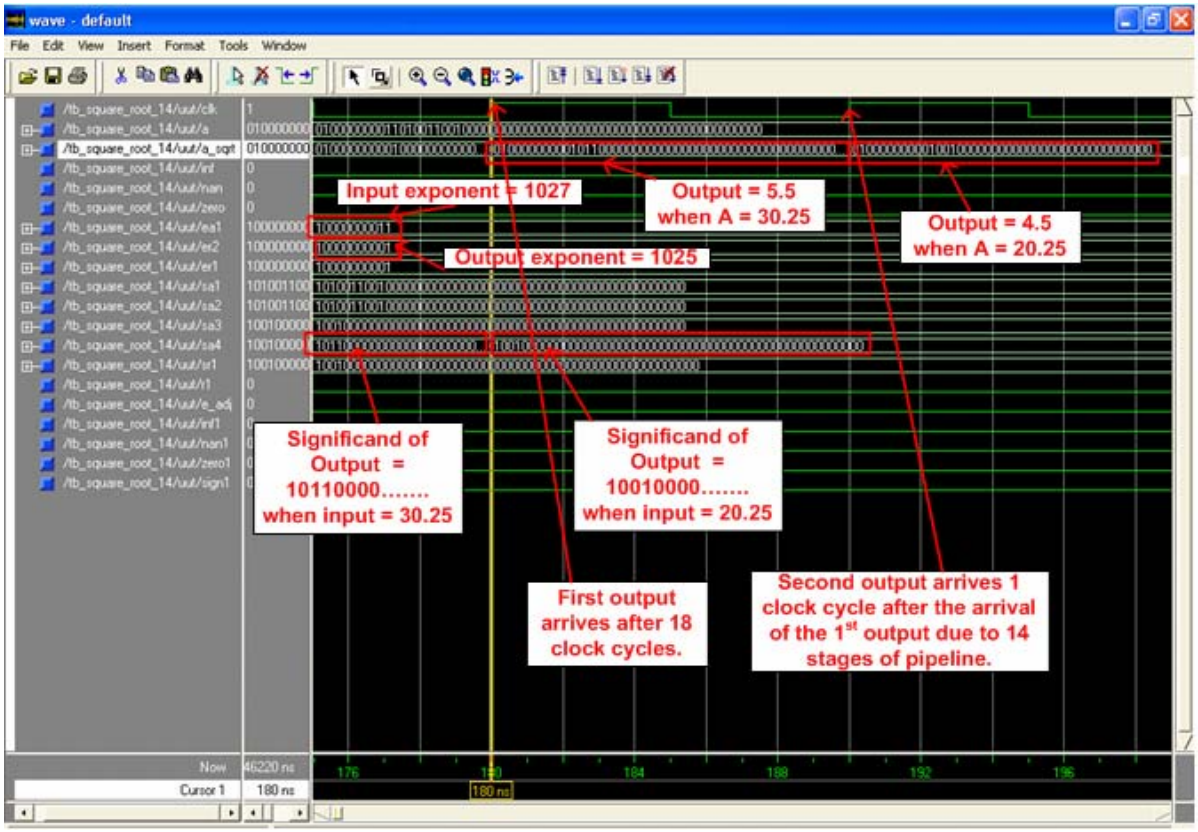

Figure 51: Simulation snapshot of the 14-stage pipelined square root unit.

4.3.7 Simulation of the Seven-stage Pipelined Square Root Unit

In this square root, pipeline registers are inserted after every eight iterations of the mantissa square root (i.e., m = 8) as shown in Figure 40. In total, seven pipeline registers are inserted in the entire square root unit. The throughout achieved by this square root is roughly half of the throughput achieved by the 14-stage square root unit.

Figure 52 shows a simulation snapshot of the 7-stage pipelined square root unit. Similarly to the simulation of the 55-stage square root unit, this simulation consists of feeding the same two operands in consecutive cycles to the square root. This simulation run shows how both remainders appear in consecutive cycles after the seven-cycle initial latency of the pipelined square root unit.

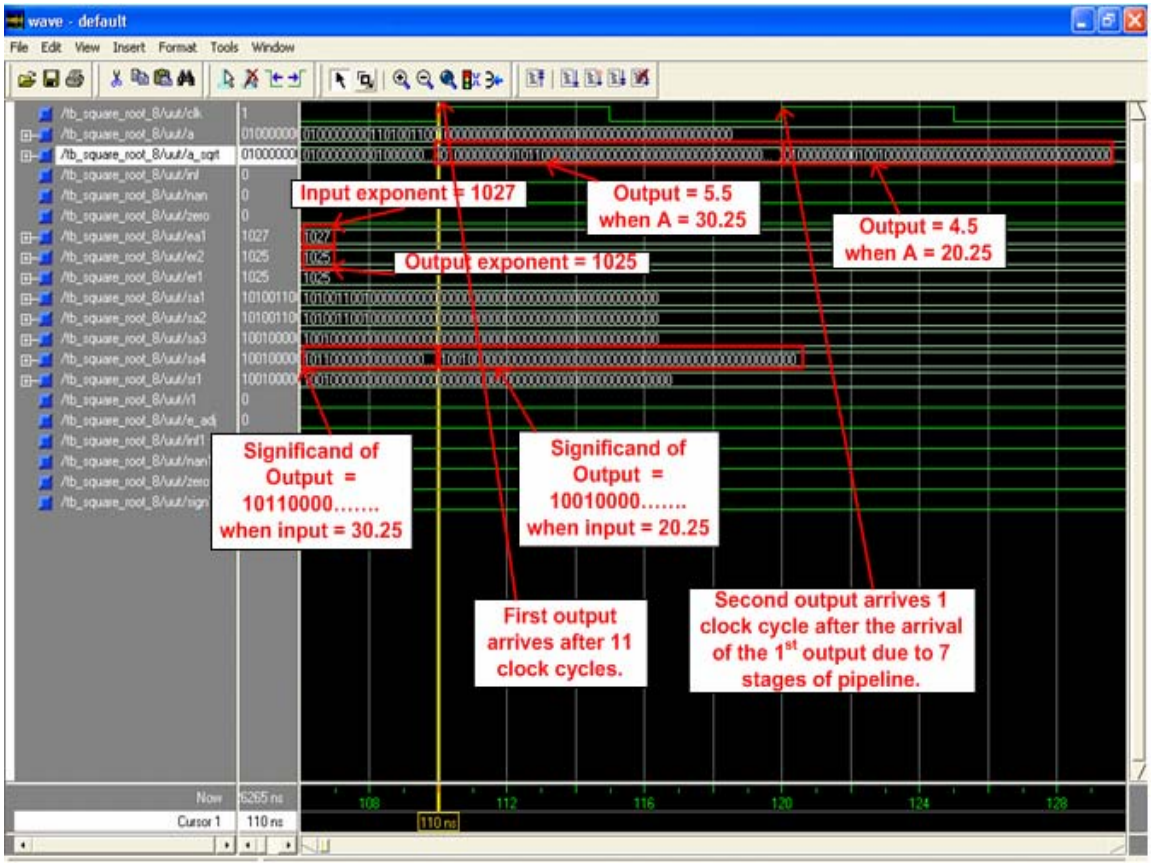

Figure 52: Simulation snapshot of the seven-stage pipelined square root unit.

4.4 Evaluation of the Square Root Unit

In this section, the results of the synthesized square root units are compared to earlier related square root units implemented on FPGAs. This comparison is followed by an evaluation of the impact of pipeline depth on area, throughput, and dynamic power of the square root.

4.4.1 Square Root Design Comparison

The square root unit was modeled in VHDL, simulated in ModelSim 5.8, synthesized using Synplify Pro 7.2, and placed using Xilinx ISE 5.2. Table 10 shows the implementation results of the units on the Virtex XCV1000 chip. These units were mapped on this chip in order to compare our results with the sequential results obtained in [17] where no pipelined implementations of the units were presented.

Table 10 : Implementation results of the square root units on the Virtex XCV1000.

Unit	Clock Period (ns)	Clock Frequency (MHz)	Throughput (MFLOPS)	Slices	LUTs	Flip-Flops
Sequential square root	16.81	59.50	0.96	400	818 (3%)	438
Pipelined square root	14.33	69.80	66.55	2699	5364 (21%)	3165

Sequential implementation of the square root has a throughput that is quite comparable to the throughputs in [17]. This implementation is significantly low in area overhead measured in terms of LUTs as shown in Table 10 although its throughput hovers around the single MFLOP mark. In contrast, the pipelined implementation displays a throughput that is significantly higher. In fact, the throughput of the pipelined square root unit is 69× higher than its iterative counterpart. The 69× increase in the square root throughput leads to a modest 6× increase in

slices. This shows that the pipelined implementation of the square root is highly efficient since the gain in performance is offset by a relatively low cost in area overhead.

With the exception of a few attempts such as the one in [14], most previously published designs of square root units address only single or parameterizable precision floating point implementations [10, 15, 19]. For a meaningful comparison, it would make sense to map both units on the same chips used in [10, 14]. Table 11 contrasts our implementation results to the results obtained in [14].

Table 11: Performance of the square root units on the Virtex II XC2V6000.

Unit	Clock Period (ns)	Clock Frequency (MHz)	Clock Cycles	Latency (ns)	Throughput (MFLOPS)	Area (Slices)	Throughput/Area (KFLOPS/Slice)
Sequential square root	9.03	110.70	59	532.77	1.79	405	4.41
Array square root [14]	239	4.18	1	239	3.99	869	4.59
Pipelined square root (59 stages)	7.52	132.98	1	7.52	126.82	2700	46.97
Pipelined square root (29 stages) [14]	13.80	72.46	1	13.80	69.10	1433	48.22

Column 1 shows the units to be compared and the number of stages in the pipelined units while column 2 shows the clock period of the critical path of each unit. Column 3 shows the clock frequency of the implemented unit while column 4 shows the number of clock cycles required to produce a single output. Column 5 shows the latency, which is the time required to produce a single output, while column 6 shows the throughput of each unit measured in Mega floating point operations per second (MFLOPS). Column 7 shows the number of slices needed to implement the unit of the XC2V6000 chip while column 8 shows the throughput per area of each unit. While our non-pipelined units are sequential in nature, the units in [14] are not. As a

result, the areas required to implement our square root units are significantly smaller than the areas required to implement the units in [14]. They can occupy only 0.46× of the square root unit area compared to the areas of the array units in [14]. On the other hand, the latencies in our sequential unit is 2.22× higher for the square root than the latency of the non-pipelined units in [14]. However, the latencies of our pipelined unit is 0.54× smaller for the square root units than the latencies of the pipelined units in [14]. This can be attributed to the high degree of pipelining introduced in our units. In fact, the iterative computations in our square root units were fully unrolled to yield a 55-stage pipeline in the mantissa square root block of the square root unit. In contrast, the pipelined square roots in [14] consist of only 28 stages. Considering this difference in pipeline stages, the areas measured in slices of our implemented pipelined unit is obviously higher than the area of the pipelined unit in [14].

Although the number of pipeline stages differs over implementations, a relatively accurate metric to assess the efficiency of these various implementations can be derived by considering the ratio of throughput over area shown in column 8 of Table 11. This ratio gives a rough idea about the level of throughput produced by a single slice regardless of the area used in the implementation. Based on this ratio, our pipelined square root is 0.03× less efficient that the pipeline square root unit in [14]. While our design is sequential in nature and does not take advantage of other radices to reduce delay as is the case in [14], it is clear that it is quite comparable in performance to the design proposed in [14]. Furthermore, it is worth noting that designs based on radix-2 computations, such as ours, are easier to implement and test than those of high-radices implemented in [14]. Note that these throughputs were achieved by merely spreading spatially the computations across the LUTs of the FPGA. The performance of our unit

can be boosted further by constructing highly optimized layouts of the units which take advantage of the block RAMs and embedded multipliers within the Virtex FPGA.

4.4.2 Throughput Evaluation

In order to understand the impact of loop unrolling on area and throughput, the square root unit was pipelined into different depths as described in section 4.2. The pipeline depth, or number of pipeline stages, depends on the degree of unrolling of the iterative loops in the square root unit block of the mantissa. Experiments were conducted to measure area and performance parameters by partitioning the mantissa's square root block into pipelines of one, seven, 14, 28, and 55 stages. The four last depths can be obtained by embedding eight iterations, four iterations, two iterations, and one iteration per stage respectively. Figure 53 shows the clock frequencies and throughputs for the mentioned pipeline depths of the square root unit on the Xilinx XC2V6000-4 chip. The figure shows that the frequency and throughput of a square root unit increases as the pipeline depth increases in a non linear fashion. The sequential square root unit is considered as one-stage pipeline. Note that in these designs, the iterative computations of the loop are performed in the same small area leading to a short clock period, and subsequently a high clock frequency. However, the 55 iterations of the loop have to be completed for square root computations before an output is produced. This leads to a significantly low throughput as shown in Figure 53. This shows that a complete unrolling of the loop in the iterative design of the square root block can provide higher level of throughputs. However, this gain in throughput may be accompanied by an area penalty.

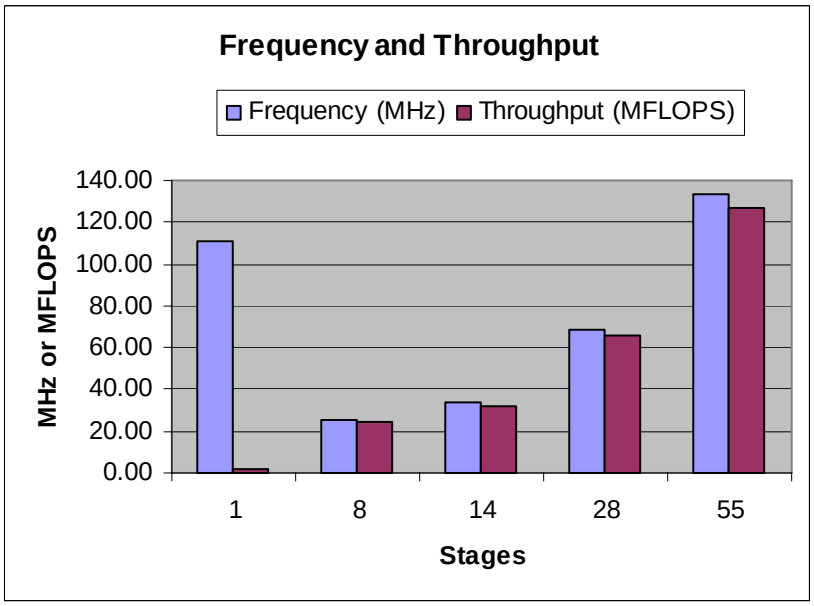

Figure 53: Clock frequencies and throughputs of the square root units.

4.4.3 Area Evaluation

Figure 54 shows the area cost, measured in slices, LUTs, and flip-flops for various pipeline depths of the square root units on XC2V6000-4. As this figure shows, the overall area increases as the pipeline becomes deep. Among LUTs and flip-flops, the increase seems to be more pronounced for the latter. This can be explained by the fact that, except for the sequential design, the pipelined designs have roughly the same amount of combinational logic regardless of the degree of loop unrolling. As such, the increase in area tends to affect the number of flip-flops since flip-flops are gradually added to implement the increase in the number of inter-stage registers required to support deeper pipelines. By examining the trend lines of this increase, it is clear that it is non-linear across the implemented square root units. With regard to slices, it seems that their numbers reach their maximum in the 28-stage pipeline. The trend line of this increase is an extremely flat bell curve spanning the pipelines of eight to 55 stages. This shows that the

degree of unrolling does not impact significantly the slice area needed to implement the pipeline. One can speculate that roughly the same slice area is used to implement the same amount of combinational logic embedded in the four pipelines. As more stages are added to a pipeline, the flip-flops contained in the same slices are being used to implement the increasing numbers of pipeline registers. If additional flip-flops are needed, additional slices are marshaled to provide the needed flip-flop thus leading to a slight increase in slices.

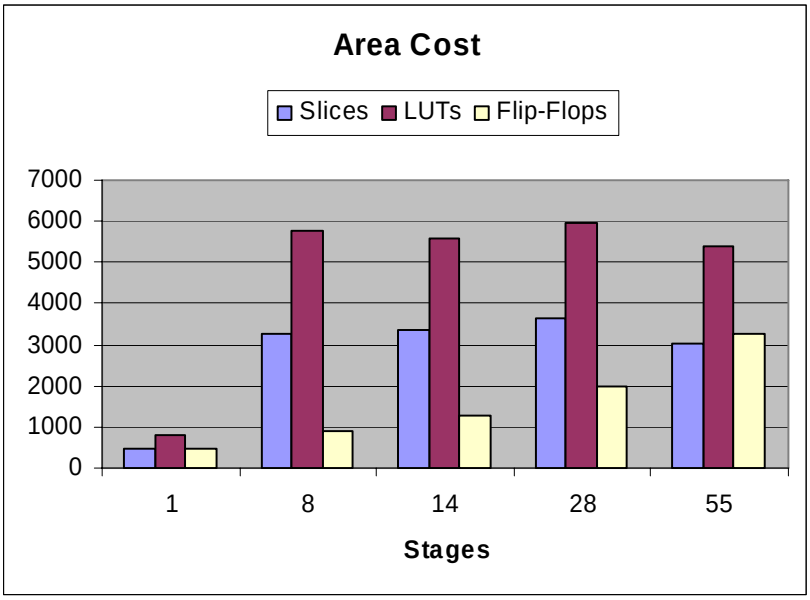

Figure 54: Area cost of the square root units.

When considering the throughput and area cost, one can quantify the incurred area penalty associated with throughput gain as a ratio. Figure 55 shows the throughput-area ratios of the square root units on XC2V6000-4. This ratio can be used in measuring the level of throughput provided by unit of area expressed as a single slice. As this figure shows, this ratio increases as the pipeline depth increases. In fact, by further unrolling the loop, and subsequently adding more stages to the pipeline, one is not adding combinational logic, but merely shortening the critical path in each pipeline stage. This results in a speedup of the clock of the pipeline

without a significant increase in area. The net effect is a visible increase in the throughput-area ratio. These observations suggest that in iterative designs, maximum performance benefits can be obtained by totally unrolling the iterative loops of the computations without incurring a significant area penalty. This is somewhat counter-intuitive considering that advanced pipelining in ASIC implementations always lead to a gradual increase in area penalty.

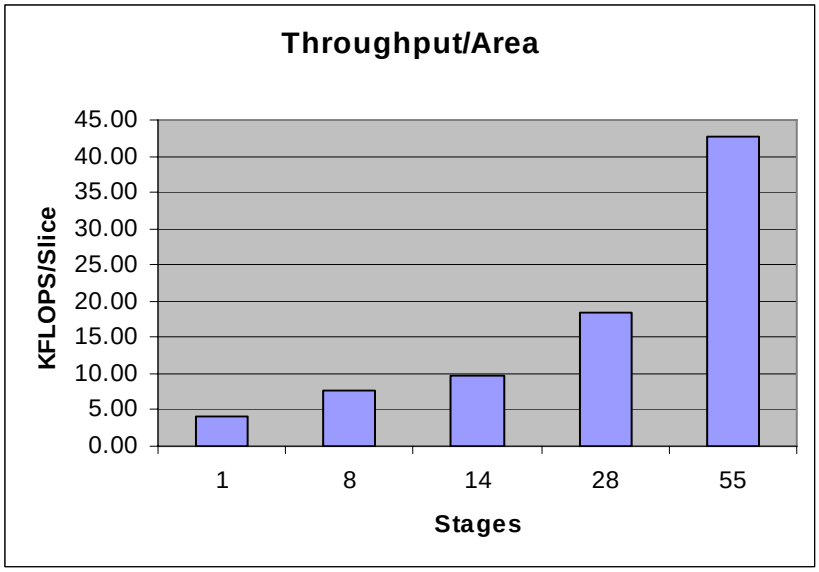

Figure 55: Throughput-area ratio of the square root units.

As this figure shows, this ratio increases as the pipeline depth increases. In fact, by further unrolling the loop, and subsequently adding more stages to the pipeline, one is not adding combinational logic, but merely shortening the critical path in each pipeline stage. This results in a speedup of the clock of the pipeline without a significant increase in area. The net effect is a visible increase in the throughput-area ratio. These observations suggest that in iterative designs, maximum performance benefits can be obtained by totally unrolling the iterative loops of the computations without incurring a significant area penalty. This is somewhat counter-intuitive considering that advanced pipelining in ASIC implementations always lead to a gradual increase

in area penalty. This increase can reach a point where the throughput-area ratio starts to gradually decrease as more stages are added to the pipeline after which partitioning further the pipeline can only yield diminishing returns in terms of throughput-area ratio.

4.4.4 Dynamic Power Evaluation

To study the effect of loop unrolling on dynamic power consumption, the square root unit was pipelined into different depths as described in section 4.2. Each pipelined square root unit is placed and routed using the same tools and simulation set up as described in section 3.4.4 for the divider. Before running the post place and route simulation, Xilinx ISE 6.2i is set to generate a vcd file after the simulation is over. This vcd file which is fed to the XPower tool packaged with Xilinx ISE software tools. XPower reads the vcd file and generates a report showing the switching power consumed by various components such as IO blocks, logic, routing, clocking, etc. The adopted methodology for power analysis is similar to the one described Figure 45. We disregarded the power consumed by all device-dependent components such as IO blocks. Only design-dependent power components were taken into account. These components are the switching power of the clock tree, design logic, and routing signals. Figure 56 shows the dynamic power consumed by the pipelined square root.

Figure 56 show that dynamic power decreases as the pipeline depth increases. Similar trend is observed in the dynamic power evaluation of the divider in section 3.4.4. However, the values in dynamic power consumed by the 28 and 55 stage pipeline square root are all below 3,000 mW in contrast to the 47,641 mW consumed by the 8 stage square root based on the log scale of the y-axis of Figure 56. The power consumed by the 8 stage square root is 17.23× and more compared to the power consumed by the 28 stage square root. The amount of the

combinational logic is roughly the same across all square root units. The only difference is the amount of logic embedded in single stage of the square root.

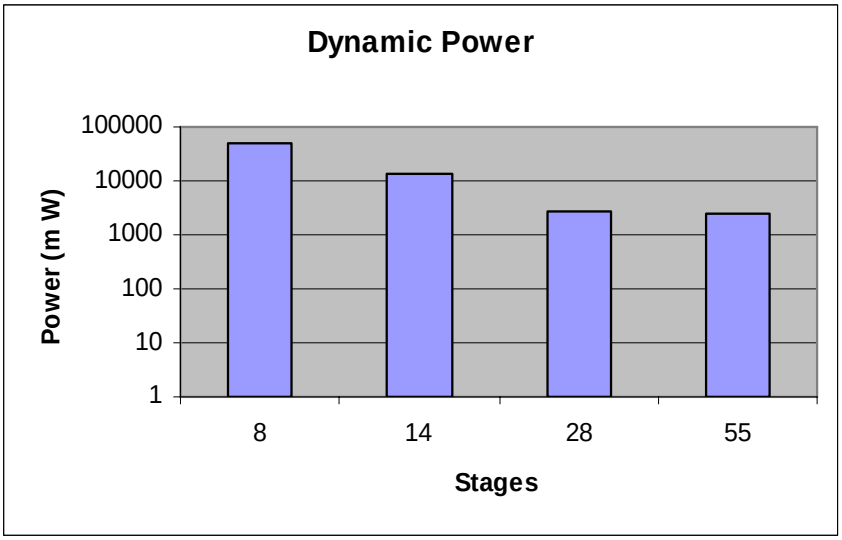

Figure 56: Dynamic power of the square root units.

In case of the eight stage square root, there are eight iterations embedded within a single stage. These eight iterations present a logic network that contains various paths of varied lengths. In particular, long paths are notorious for displaying glitching behavior. In fact, glitching activity increases with signal length [21]. As the pipeline depth increases, more registers are added at various points of the design. This reduces the amount of interconnect between registers thereby reducing the glitching plaguing these signals. This results in the reduction of dynamic power consumed by the square root as the pipeline depth increases. However, Figure 56 shows that this reduction in dynamic power tends to slow down as the square root is pipelined further. It is observed that there is a huge power benefit when the 8 stage design is pipelined to 14 stage and 28 stage square root. However, this power benefit cannot be seen much as the design is pipelined further to 55 stages. This can be explained by the fact that in shallow pipelined designs, glitching can make up to 80% of the total power consumed by the square root. As the pipeline

depth of the design is slightly increased, glitching activity can fall to levels below 40% of total power displayed by the square root [22]. This sudden fall in glitching activity explains the decreasing returns observed in deeper pipelines. In essence, while pipelining can improve throughput in a non-linear fashion, its impact on reducing power is limited to shallow pipelines.

4.5 Conclusion

This chapter presents the design of an IEEE-compliant double precision floating point sequential and parallel square root unit. This design is based on a low-radix iterative square root algorithm known as the binary version of the pencil-and-paper method. The pipelining of the square root was based on partial and full unrolling of the loops in the iterative mantissa square root block. The implementation of this square root unit did not take advantages of any advanced architectural features available in high end FPGA chips or use any pre-designed architecture-specific arithmetic cores. The experiments reveal that this square root can produce maximum throughput when the iteration of the computational loops are totally unrolled without incurring a significant area penalty. While the sequential square root unit occupies less than 1% of an XC2V6000 FPGA chip, its pipelined counterparts can produce throughputs over 100 MFLOPS by consuming 7× more area as its sequential counterpart.

CHAPTER FIVE: CONCLUSION

This thesis presents the sequential and pipelined designs of division and square root operations on FPGAs. These designs are based on radix-2 digit recurrence algorithms. While the division design is based on a sequential non-performing algorithm, the square root unit is based on an iterative non-restoring algorithm. These designs are all IEEE 754-compliant double precision floating point implementations. These implementations are built in a way to allow them to use only the most common reconfigurable resources available in most FPGAs. As a result, they should be highly portable to most popular FPGA devices.

The pipelining of these designs reveal that the area overhead tends to remain constant regardless of the degree of pipelining to which the design is submitted. In fact, the unrolling of the iterations of the digit recurrence algorithm consumes the same amount of logic resources regardless of how many iterations are packed per pipeline stage. The only minor change in area overhead is due to the usage of flip-flops to insert registers between the pipeline stages. These flip-flops are recalled from the already used slices to implement the combinational logic of the stages.

With regard to throughput, it seems to increase as the pipeline depth increases. By decreasing the delay of each stage, it is possible to increase the overall pipeline throughput. This can be realized by reducing the number of iterations embedded in a single pipeline stage. This trend reaches its maximum when a pipeline stage consists of one iteration of the digit recurring algorithm. This trend can be pushed further by introducing sub-pipelining in each stage. It remains to be seen how much throughput can be achieved using the sub-pipelining approach.

Since these computations are intended for space application where power budgets are a primary concern, this thesis examines pipelining as a means to reduce dynamic switching power. Pipelining reveals that it reduces power considerably in shallow pipelines. Pipelining further these designs does not necessarily lead to significant power reduction. In fact, the pay-off curve in terms or dynamic power reduction seems to go down as more stages are added to the pipeline. This has been attributed to the fact that pipelining reduces the amount of glitching activity on the logic paths by the insertion of pipeline registers. The latter acts as a reducing factor on the length of these glitching-prone logic paths. Glitching activity is by no means the only component that contributes to the overall dynamic power budget. Other components such as the switching of the clock lines and the order of input arrival to switching logic LUTs can increase the amount of switching within the power budget. To reduce power further, it would be worthwhile using other techniques such as clock gating, guarded evaluation, bus multiplexing, or pre-computing [23].

In summary, the pipelined implementations of double precision floating point division and square root operations, based on elementary digit recurrence algorithms, are able to reach the 100 MFLOPS mark by consuming only a minor 1% of the fine-grain resources of an XC2V6000 FPGA. Note that this chip does not even belong to the high-end full-blown reconfigurable system-on-chips such as the Virtex-II Pro FPGA series. Although they tend to consume more power, these chips have a much faster clock than the XC2VX000 chips. Table 12 summarizes the implemented designs.

While this thesis undertakes to examine the effect of pipelining radix-2 digit recurrence algorithms of division and square root operations on area, throughput, and dynamic power in

FPGA implementations, it nevertheless raises new questions which would be interesting to pursue as directions for future research:

Table 12: Summary of the implemented designs.

	Sequential Divider	Sequential Square Root Unit	Pipelined Divider	Pipelined Square Root Unit
Algorithm	Non-Performing	Non-Restoring	Non-Performing	Non-Restoring
Radix	2	2	2	2
Precision (bits)	64	64	64	64
Pipelining (stages)	1	1	60	59
Device	XC2V6000	XC2V6000	XC2V6000	XC2V6000
Clock Period (ns)	9.93	9.03	9.75	7.52
Clock Frequency (MHz)	100.70	110.70	102.50	132.98
Clock Cycles	60	59	1	1
Latency (ns)	595.8	532.77	9.75	7.52
Throughput (MFLOPS)	1.60	1.79	97.81	126.82
Area (Slices)	284	405	2920	2700
Throughput/Area (KFLOPS/Slice)	5.63	4.41	33.49	46.97

How does wide bit representation affects the area, throughput, and power performance of these designs? It is meant by wide bit representation arbitrarily large bit fields to support extremely large and small numbers with high accuracy. These numbers are commonly used in many scientific applications such weather modeling and astronomical simulations. Based on the findings of this thesis, one can speculate that wider bit representations will not affect the latency per pipeline stage. However, it will increase the number of iterations needed to produce a single output granted that digit recurring algorithms are used. If that is the case, the maximum throughput can be achieved only if the iterations are fully unrolled to the point where a single iteration is embedded per pipeline stage. With regard to area and power, it seems that the results

obtained for double precision representations may remain valid for arbitrarily wide bit representations. These speculations need to be confirmed with actual experimentation.

If high radix algorithms are used instead, how would they affect the performance of division and square root operations? High radix digit recurrence algorithms have the advantage of retiring several digits per iteration. This has the effect of reducing the number of iterations to compute the quotient or square root. From a pipelining perspective, this has the effect or reducing the maximum pipeline depth. However, this advantage comes at the cost of a noticeable increase in the area required to support computing multi-digits per iteration. By increasing the area of a single iteration, the area of a single pipeline stage will subsequently increase, which may lead to an increase in its latency. If that is the case, it is not clear how reducing the number of pipeline stages and increasing the latency of each stage will affect the overall area, throughput, and power consumption of a pipeline based on these high radix algorithms.

Can advanced architectural features available in current FPGAs be used to support floating point arithmetic operations? Many high-end FPGAs contain embedded multipliers and blocks of memory. Several studies show that these embedded structures tend to consume less power than the fine-grain reconfigurable fabric of these FPGAs [24]. It is tempting to take advantage of these structures by using arithmetic algorithms that can be easily mapped on these structures. For instance, many division and square root algorithms are based on table lookup operations. These operations can be easily mapped on BlockRAMs available in Virtex FPGAs. It would be interesting to see whether it is possible to reach throughputs in the GHz range by mapping these algorithms on these embedded structures.

How do the implementations of elementary functions fair on FPGAs? While square root operations are less common than division, elementary functions such as the logarithm, exponential, and the powering operations are even more infrequent than square root operations. However, they began recently to occur more frequently thanks to newer applications in DSP [25], 3D graphics [26], scientific computing, artificial neural networks, and multimedia applications [27]. Although these operations have always been implemented as software routines, these routines provide accurate results, but are often too slow to meet the needs of numerical-intensive applications [28, 29]. In addition, these operations have been rarely implemented in hardware due to the prohibitive costs of their table lookup requirements. Lately, improved algorithms which reduce significantly the hardware costs of these operations have been proposed [30]. These algorithms can serve as the basis for high-performance FPGA implementations targeting numerical applications. Preliminary scrutiny suggests that these implementations cannot be realized unless the advanced architectural features available within high-end FPGAs are heavily exploited.

Addressing these issues can expand the levels of performance needed to support numerically intense applications in general, and space DSP applications in particular.

LIST OF REFERENCES

1. S. M. Parkes, “DSP (Demanding Space-based Processing!): The Path Behind and the Road Ahead,” *Proc. International Workshop on Digital Signal Processing Techniques for Space Applications*, pp. 1-11, Sep. 23-25 1998.
2. F. H. Bauer, K. Hartman, J. P. How, J. Bristow, D. Weidow, F. Busse, "Enabling Spacecraft Formation Flying through Spaceborn GPS and Enhanced Automation Technology," *Institute of Navigation GPS Conference*, Sep. 1999.
3. IEEE Standard Board, "IEEE Standard for Binary Floating-point Arithmetic," The Institute for Electrical and Electronics Engineers, 1985.
4. J. Eiler, A. Ehliar, D. Liu, "Using Low Precision Floating Point Numbers to Reduce Memory Cost for MP3 Decoding," *Proc. IEEE International Workshop on Multimedia Signal Processing*, Siena, Italy, Sep. 2004.
5. J. Fridman, Z. Greenfield, "The TigerSHARC DSP Architecture," *Proc. IEEE Micro*, pp. 66-76, Jan.-Feb. 2000.
6. K. Keutzer, "Challenges in CAD for the One Million Gate FPGAs", *Proc. 5th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, pp.133-134, Feb. 1997.
7. Xilinx, Inc, "Virtex - II Platform FPGAs: Complete Data Sheet," 2005.
8. B. L. Hutchings, B. E. Nelson, "GigaOp DSP on FPGA," *VLSI Signal Processing Systems*, Hingham ,MA, vol. 36, pp. 41-55, 2004.

9. K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-Point Performance," *International Conference on Field-Programmable Gate Arrays*, Monterey, CA, pp. 171-181, Feb. 2004.

10. G. Govindu, R. Scrofano, V. K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing,".*International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2005.

11. S. Paschalakis, *Moment Methods and Hardware Architectures for High Speed Binary, Greyscale and Colour Pattern Recognition*, Ph D Dissertation, Dept. of Electronics, University of Kent at Canterbury, Aug. 2001.

12. Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Application Note, Nov. 2003.

13. S. F. Oberman, M. J. Flynn, "Division Algorithms and Implementations," *IEEE Transactions on Computers,* vol. 26, no.8, pp. 833-854, Aug. 1997.

14. B. Lee, N. Burgess, "Parameterisable Floating-Point Operations on FPGA," *Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1064-1068, 2002.

15. X. Wang, B. E. Nelson, "Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 195-203, 2003.

16. I. Ortiz, M. Jimenez, "Scalable Pipeline Insertion in Floating-Point Division and Square Root Units," *Proc. IEEE Midwest Symposium on Circuits and Systems,* vol. II, pp. 225-228, 2004.

17. S. Paschalakis, "Double Precision Floating-Point Arithmetic on FPGAs," *IEEE Conference on Field-Programmable Technology (FPT),* pp. 352-358, 2003.

18. B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, 2000.

19. Y. Li, W. Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs," *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pp. 226-232, 1997.

20. Y. Li, W.Chu, "A New Non-Restoring Square Root Algorithm and its VLSI Implementations," *Proc. International Conference on Computer Design*, Oct. 1996.

21. G. K. Yeap, *Practical Low Power Digital VLSI Design*, Kluwer Academic Publishers, Second Edition, 2001.

22. N. Rollins, M. Wirthlin, "Reducing Energy in FPGA Multipliers through Glitch Reduction," Bringham Young University, Dept. of Electrical and Computer Engineering, 2005.

23. Actel Inc., "Design for Low Power in Actel Antifuse FPGAs," Application Note, Sep. 2000.

24. S. Choi, R. Scrofano, V. K. Prasanna, J. W. Jang, "Energy-Efficient Signal Processing using FPGAs," *ACM International Symposium on FPGAs*, Feb. 2003, pp. 225-234.

25. D. M. Lewis,"114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications" *IEEE Journal of Solid-State Circuits*, vol. 30, no. 12, pp. 1547-1553.

26. D. Harris, "A Powering Unit for an OpenGL Lighting Engine," *Proc. 35th Asilomar Conference on Signals, Systems, and Computers*, pp. 1641-1645, 2001.

27. J.-A. Piňero, J. D. Bruguera, J.M. Muller, "Faithful Powering Computation Using Table Look-Up and Fusion Accumulation Tree," *Proc. IEEE 15th International Symposium on Computer Architecture*, pp. 40-47, 2001.

28. W. Cody, W. Waite, *Software Manual for Elementary Functions*, Prentice-Hall, 1980.

29. S. Gal, B. Bachelis, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26-45, Mar. 1991.

30. J.-A. Piñero, M. D. Ercegovac, J. D. Bruguera, "Algorithm and Architecture for Logarithm, Exponential, and Powering Computation," *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1085-1096, Sep. 2004.