SCALABLE FLOATING POINT FPGA UNITS FOR RAPID SYSTEMS PROTOTYPING

By

Irvin Ortiz Flores

A project submitted in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF PUERTO RICO

MAYAGÜEZ CAMPUS

December 2003

Approved by:

Manuel Jimenez, Ph.D. Chairperson, Graduate Comitee	Date
Rogelio Palomera, Ph.D. Member, Graduate Comitee	Date
Domingo Rodriguez, Ph.D Member, Graduate Comitee	Date
Dorothy Bollman, Ph.D Representative, Graduate School	Date
Jose Luis Cruz, Ph.D. Department Chairman	Date

Abstract

Most modern arithmetic processors rely on pipelining techniques to obtain a high throughput. Floating-point (FP) operations are often time-consuming and depend on pipelining to accelerate their processes. This project reports the development of scalable, FP arithmetic operators with a variable number of pipeline stages. An algorithm for pipeline insertion was developed and used for FP Multiplication, Addition/Subtraction, Division, and Square Root. The use of this algorithm enables operating frequencies up to 175MHz when implemented on a Xilinx Virtex II FPGA. The developed units offer scalability in terms of precision, range, and pipeline granularity. Also new topologies and improvements for supporting units were achieved. Future work includes automation of the pipeline insertion process.

Resumen

La mayoría de los microprocesadores modernos hacen uso de técnicas de “pipeline” para obtener un alto rendimiento. Las operaciones de punto flotante son generalmente complejas, por lo que estas técnicas se usan para acelerar su ejecución. Este proyecto presenta el desarrollo de unidades aritméticas escalables de punto flotante con un número variable de etapas de “pipeline”. Se desarrolló un algoritmo para la inserción de etapas de “pipeline”, el cual fue usado para las unidades de suma/resta, multiplicación, división y raíz cuadrada de punto flotante. El uso de este algorítmo en estas unidades permitió alcanzar frecuencias de operación de hasta 175MHz al ser implementadas en un FPGA Virtex II de Xilinx. Las unidades desarrolladas ofrecen escalabilidad en términos de precisión, rango dinámico y granularidad de “pipeline”. Adicionalmente, nuevas topologías y mejoras para sub-componentes fueron desarrolladas. Finalmente, la automatización del proceso de inserción de etapas de “pipeline” queda como alternativa de trabajo futuro.

Dedication

The project was made with much enthusiasm, positivism, and arduous work. I dedicate this project to God, for giving me so many opportunities to make my dreams come truth and for placing in my way the right people to help me achieve them. Also, I dedicate it to my family, specially to my parents Genaro Ortiz Rivera and Nitza M. Flores Saldaña, whose love, dedication, enthusiasm, amiability, and firmness have helped me to reach my goals. Finally I dedicate this work to my girlfriend Yolanda Crespo Méndez who helped and encouraged me during the development of this document.

Acknowledgments

I am pleased to thank Dr. Manuel Jiménez for his patience, commitment, and for allowing me accomplish this work under his supervision; Dr. Rogelio Palomera and Dr. Domingo Rodrguez for being part of my graduate committee and their mentoring.

Additionally, I acknowledge the economical and material support received from the Xilinx University Program, the University of Puerto Rico, and the PRECISE Project during the development of this project.

Also I give special thanks to my friends Oscar Acevedo and José Joaquín who where always present to support me in anything related to my project; and specially Alberto Quinchanegua, who was a co-worker in some of its stages.

Contents

List of Figures

List of Tables

Terms Definitions

1. **ASIC**: Application Specific Integrated Circuit. Build for a limited or fixed application. Typically produced in high volumes to justify their development costs. They are efficient for the designed task but very expensive in low volumes.

2. **DSP:** Digital Signal Processing.

3. **Floating Point**: This number representation format includes sign (s), exponent (e), and mantissa (m) fields. The floating point value (x)is represented by $x = -1^s \times m \times 2^e$. FP offers a dynamic wide range for large and small number representation. The format chosen in our application resembles in most aspects the IEEE 754 standard for floating point representation [22].

4. **IEEE 754 standard**: Specifies the parameters for single and double precision floating point numbers. Single precision has 32-bit representation: 23-bit mantissa, 8-bit exponent, and 1-bit sign. Double precision has 64-bit representation: 52-bit mantissa, 11-bit exponent, and 1-bit sign. Both formats have an implicit hidden one as the most significant bit of the mantissa. Mantissa is normalized in the range [1,2), which helps to make the representation for each number unique. Denormalized numbers are an optional feature of the standard, which helps to alleviate the underflows (called also gradual underflow). Special representations are used for zero, infinity, and not-a-number (NaN). The exponent value ranges from 0 to +255 and 0 to +1045 for single and double precision, respectively [22].

5. **FPGA**: Field Programmable Gate Array. FPGAs are one of the most evolved type of field programmable logic. FPGAs have increasingly become a favorite development platform for

many DSP algorithms. One of the major reasons for this trend is that FPGAs offer the functional efficiency of hardware and the programmability of software [27].

6. **Pipeline:** Technique that allows operating a circuit at high clock rates. It divides a large task into smaller size sub-tasks and overlaps their execution. Tasks are divided by the insertion of synchronizing latches. Careful selection of the latch insertion points is an important factor for obtaining optimal throughput. A pipeline consisting on k stages produces the first result after k cycles, and successive results at a rate of one per cycle. The pipeline insertion process takes advantage of those operations that can proceed concurrently, even if there is some sequential dependency. This allows for parallel processing without the need of extra computing units.

7. **VHDL**: VHSIC Hardware Description Language (VHSIC= Very High Speed Integrated Circuit). VHDL can be used to model a digital system from top-down and bottom-up design methodologies. Today, VHDL is an IEEE standard as well and ANSI standard for describing digital designs. First generated in 1981 under the VHSIC program of the US Department of Defense. First released in 1985 and standardized in 1987. Also it can be used to develop complex digital systems, when synthesized and implemented in an ASIC or a programmable platform.

Chapter 1

Introduction

This project addresses the creation of a set of scalable floating point (FP) operators based on reconfigurable hardware. These operators are intended for the rapid systems prototyping. FPGAs are the reconfigurable hardware implementation choice for these scalable operators. VHDL was the description language used to generate this set of operators. Scalability in terms of data size and pipeline granularity is controlled by VHDL generic parameters provided at compile time. Pipeline granularity adjustments allows controlling operating frequency of the developed operators.

Signal processing, scientific, and engineering algorithms often require a substantial amount of arithmetic computations to be performed at real-time or near real-time speeds [30]. The main advantage of FP arithmetic over the fixed-point scheme is the dynamic range for accommodating extremely large numbers and high precision for very small numbers. This helps to alleviate the underflow and overflow problems often seen in fixed-point formats [24]. Also it offers a robust scheme against quantization errors, improves system's precision for signal acquisition and reduces errors during internal computations. Quantization effects can be reduced to a negligible level by choosing longer mantissa and exponent registers lengths [25].

Scalable FP arithmetic cores allow for manipulating range and precision of computations to the exact user's needs. In order to achieve the so-called scalability, Hardware Description Languages (HDL's) are used. Precision and range of FP numbers can be adjusted by controlling the sizes of mantissa and exponent fields, respectively, by means of parameters passed to the HDL at compile time. The implementation of these cores on FPGAs allows for significant reduction in the turnover time of many applications, enabling their rapid prototyping.

VHDL contains useful elements to describe the behavior or structure of a digital system. This language provides support for hierarchically modeling a system in a top-down or bottom-up design methodology. Systems and subsystems can be described at any level of abstraction ranging from architecture to gate level [3]. A VHDL model can be synthesized and targeted to an Application Specific Integrated Circuit (ASIC) or to a programmable device like an FPGA. Rapid prototyping of FP units has become possible thanks to the use of VHDL and FPGA technology [24].

1.1 FPGAs Description

An FPGA is basically composed of an array of configurable logic blocks (CLBs or slices[1] in Xilinx's FPGAs), input-output ports, and programmable routing resources, as seen in Figure 1.1. The structure of a Xilinx's CLB contains multiplexers, function generators, fast carry logic and flip-flops as seen in Figure 1.2 [28]. FPGAs were originally intended to be used for "glue logic" among ASICs. Small FPGAs in the Xilinx families have from 100 to 400 CLBs, while the larger ones have over 60,000 slices. It has become possible to build entire computing systems from only a handful of FPGAs thanks to FPGAs increase in logic density and operating speed [28]. Early FPGAs were just reaching the point where they were dense enough to support a single FP unit [14]. Nowadays, FPGAs are capable of supporting several FP units.

The programmability of FPGAs allows implementations to be customized to specific system needs. An ASIC implementation is often more generic to justify its high development costs, so it may be less efficient than a specialized one [29]. Unfortunately, the high set-up costs of ASICs make them unattractive in low volumes. However, the rapid advancement and lowered costs in FPGA technology offer a viable alternative for implementing high performance DSP and highly

[1]4 slices equals 1 CLB

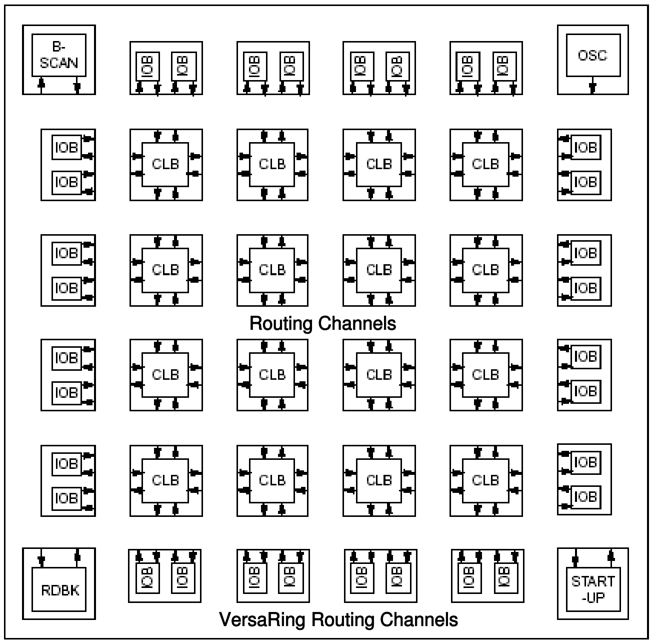

Figure 1.1: General internal structure of a Xilinx FPGA

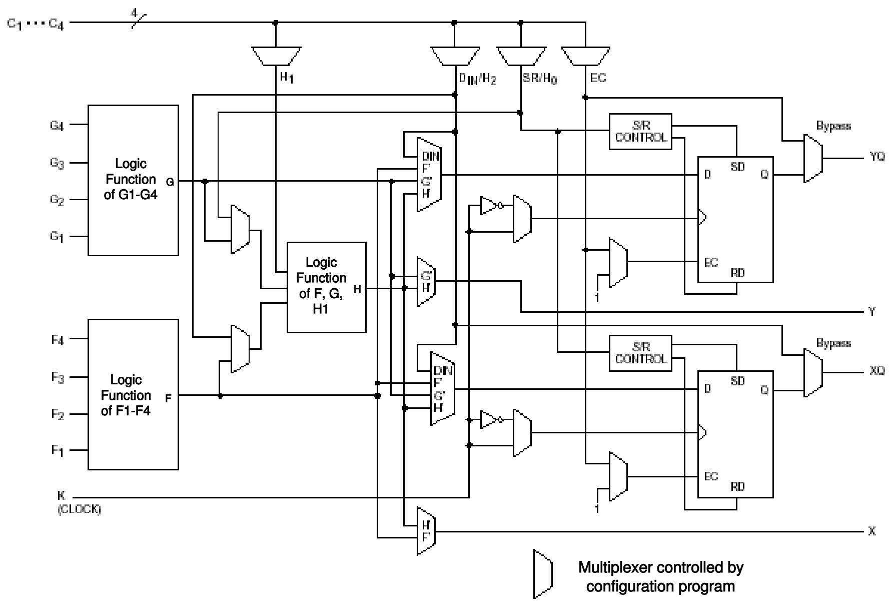

Figure 1.2: XC4000 Xilinx FPGA simplified internal structure

re-programmable solutions [29]. FPGAs offer reduced development time and costs compared to ASICs. Furthermore, many FPGAs use SRAM to store their circuit configuration. This allows for fast reprogramming, and offers the flexibility of conventional processors [29].

FPGAs are often used as powerful custom hardware for applications that require high-speed computation [9]. One of the major reasons for this trend is that FPGAs offer the functional efficiency of hardware and the programmability of software [27]. Their fast reprogrammability enables field upgrade and adaptation of hardware to run-time conditions [1]. Dawwod et Al. state the importance of FPGAs by affirming that the deployment of FPGA for filter design and implementation represents the most promising solution as compared to other solutions, including the use of dedicated DSP processors [6].

1.2 HDL Design Flow

Several typical steps are performed in the process of synthesizing an algorithm or application from a hardware description languages. The Xilinx's VHDL design flow is presented in Figure 1.3. It starts with an HDL source and ends with a netlist downloadable to a programmable device like an FPGA. Data related to timing, functionality, and consumed resources is available through synthesis reports after the circuit is synthesized. FPGA's configurable logic blocks (CLBs) are interconnected and configured during the download of the synthesized code. Finally the application is implemented in an FPGA, just like the architecture specified in the VHDL code.

1.3 Fixed-Point Arithmetic Operators

Most VHDL synthesis tools offer support for fixed-point addition/subtraction, and multiplication. There is no embedded support for fixed-point division, square root, and squaring. Implementation of these non-supported units is more complicated because their dependence among iterations and the complexity of their result generation function. Most of the used division and

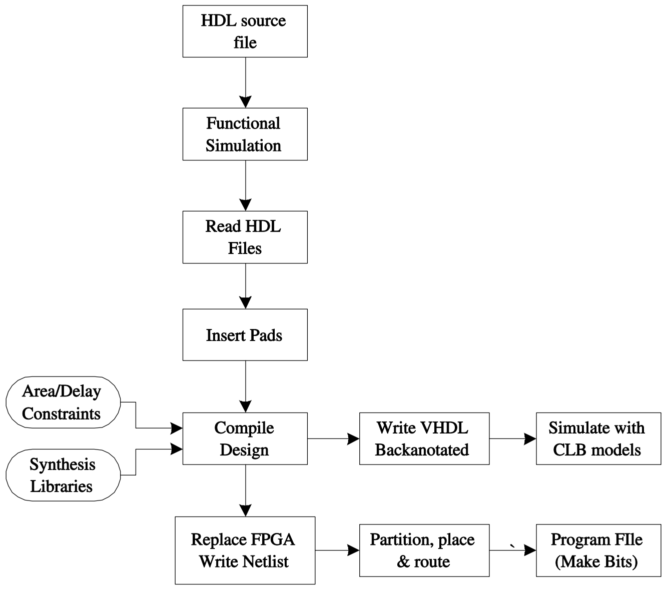

Figure 1.3: Xilinx VHDL Design Flow

square root algorithms are based on recurrences, producing one digit of the result per iteration [26].

Typical computing applications use addition, subtraction and/or multiplication. Other operations receive less attention due to their hardware complexity, long processing times, and relatively less usage. The following arithmetic components are used in FP arithmetic.

- **Comparators:** Comparators are frequently used in floating point arithmetic. They are used for operand swapping and decision-making.

- **Adders:** Addition is a basic arithmetic operation in any computational system. In special purpose computing (some signal and image processing applications), dedicated adders are required to have high throughput while latency constraints are not severe. In such cases, pipelined architectures are widely used. Traditional pipelined adders for parallel addition of two operands are based on carry save addition [5].

- **Multipliers:** Pipelined multipliers are useful in systems where arithmetic throughput is more important than latency [2]. They are also desirable for high-performance arithmetic applications such as digital signal processing. The most common type of multiplier used for pipelined applications is the array multiplier. This preference is due to its regular structure and modular design.

- **Dividers:** Division operation is a sequential operation because partial remainders are generated sequentially. Its quotient is produced only after the remainder sign is detected. Division operation is much slower than multiplication operation [4].

- **Square Rooters:** Square root is an essential and important operation in science and engineering. It may be rated, in importance, next to the four basic operations: addition, subtraction, multiplication, and division. Some typical applications are complex variables, trigonometry, error computation, and statistics. More complex applications include adaptive filtering, gradient computation for edge detection, and many others [8] [18].

1.4 Fixed-point addition

This operation has been studied extensively. Numerous approaches have been generated to optimize it. Some of them reduce the carry propagation delay while others reduce hardware consumption. The following paragraphs present some of the most common adder types.

Carry-Ripple Adder: Is the simplest and slowest adder type. Uses full adders as its basic building block. These are interconnected to create a full chain of adders.

Carry Look Ahead Adder: The fastest adder type, but it is also the most hardware consuming. It uses additional combinatorial logic for carry prediction. Carries are computed before the addition process is performed.

Carry-Save Adder: Uses full adders interconnected in a two dimensional array. Reduces the carry propagation delay by minimizing the carry propagation chain length. It is slower than the carry-look-ahead adder, but faster than the carry-ripple adder.

1.5 Fixed-point Multiplication

Multiplication is a computationally intensive operation. Normally this operator calculates several partial products and then adds them up. Speed up techniques improve the partial product stage or uses faster addition schemes. The multipliers described in the following paragraphs are among the most common ones.

Bit-serial multiplication: It uses shift and addition operations. It is similar to the hand process for multiplication.

Array multiplication: It uses full adders as it basic building block. Full adders form a 2-dimensional array which calculates the partial products and perform some part of the addition using carry-save techniques. An adder is needed for the final stage, which normally is a carry-ripple adder. This multiplier is the most hardware consuming one, but provides good delay characteristics. It is typically used in ASIC designs

Digit-serial multiplier: Similar to the bit-serial multiplier, but resolves n-bits of multiplication per cycle, where n is the digit size.

1.6 Fixed-point division and square root

Division and square root are computationally expensive arithmetic operations. Their result becomes known sequentially, beginning with the most significant bits [22]. This situation has created interest in the development of faster and more efficient division and square root algorithms. Pipelining techniques are also useful to increase their performance. The following subsections describe some of the most important division and square root algorithms.

1.6.1 Division Algorithms

Division algorithms can be divided into five classes: digit recurrence, very high radix, functional iteration, table look-up, and variable latency [20]:

Digit recurrence: The oldest class of high speed division. It retires a fixed number of quotient bits per iteration. Typically has low complexity and relatively high latency. Speed-up can be accomplished by retiring more bit per iteration, which is called higher radix division. The most common version of this algorithm is the SRT division which uses subtraction as the fundamental operator to retire a fixed number of quotient bits in every iteration.

Functional iteration: Utilizes multiplication as its fundamental operation. It has a quadratic convergence (doubles the correct quotient bits in every iteration), instead of the linear convergence of the subtractive methods.

Lookup tables: Require the use of ROMs or PLAs. They are fast, but their size grows exponentially with each bit of added accuracy.

Variable latency algorithm: Takes advantages of previously computed results by reusing them whenever is possible. Also the computations for each stage can be completed sooner than others.

1.6.2 Square Root Algorithms

Square root and division algorithms are similar in some architectural features. There are various classes of square root algorithms like the traditional pencil-and-paper method, restoring shift/subtract, binary non restoring, high radix, and by convergence [22]:

Pencil and paper method: Is the basis for the shift/subtract algorithms. Resolves two-input bits per iteration.

Restoring shift/subtract: Uses a sequence of shift and subtractions. Restores the remainder to the correct value if the trial subtraction indicates that the current quotient digit was not the right choice for a quotient digit.

Binary nonrestoring: Almost equal as the nonrestoring division. Does not restores the remainder to the correct value if the trial subtraction indicates that 1 was not the right choice for a quotient digit. In that case it performs addition in the next step instead of subtraction.

High radix square rooting: Uses the same techniques as for high radix division. Retires multiple bits per iteration.

Square Root by Convergence: Uses Newton-Raphson iterations, which involves division, addition and a single bit shift.

1.7 Pipelining in Arithmetic Units

Pipeline implementations may improve the performance of multistep arithmetic computations while also reducing their hardware cost [22]. The key figure of merit for a pipelined implementation is its computational throughput, which is defined as the number of operations that can be performed per unit of time. Most modern microprocessors use pipeline techniques to achieve the throughput requirements. Pipelining techniques also result especially useful for implementing floating-point hardware, due to the complexity and number of steps required in such operations [19] [12]. The performance achieved in pipelined FP operators is highly dependent on the approach used for introducing pipeline stages into their constituent units such as integer adders, multipliers, shifters, comparators, and others. Pipelined adders and multipliers have been widely studied and used for special purpose computing where high throughput is required and latency constraints are not severe [2]. Pipelined multipliers are desirable for high-performance arithmetic applications. The most common type of multiplier used for pipelined applications is the array multiplier. This is due to its regular and modular design.

1.8 FP Arithmetic in FPGAs

FP operations are useful for computations involving large dynamic range, but they require significantly more resources than fixed-point operations [1]. The wordlength requirements for a fixed-point algorithm would be significantly higher to support a large dynamic range (which in FP is achieved via the exponent). For example, a fixed-point CORDIC (Coordinate Rotation by Digital Computer) implementation requires almost twice the wordlength to achieve the same numerical performance of an FP implementation [29].

The use of reconfigurable hardware to perform high precision operations such as IEEE-compliant

FP has been limited in the past by FPGA resources. Most scientific algorithms require some form of fractional representation. The only available option in most programming languages is to declare an FP number. The introduction of high-speed sub-micron technology FPGAs, which offers the equivalent of over a million programmable gates and increased routing facility, allows system rates in excess of 150MHz.

1.9 Problem Statement

Many of today's FP implementations are provided as optimized netlist with prescribed parameters or as a non-customizable HDL source code. They offer high operating frequencies and are typically IEEE 754 compliant. Despite all these good characteristics, there are many applications where speed and precision constraints can be satisfied with slower and less precise units. For these cases, the usage of over-specified units has a cost in the consumption of hardware resources.

The problem addressed by this project is that of providing customized FP units for specific requirements of speed, range, and precision, without the inherent waste of hardware resources created by non-customizable pre-synthesized solutions. These units should be obtained with a high level of reusability that do not require structural modifications or re-coding for satisfying a wide range of specifications. Also they should be easily portable to multiple hardware platforms.

1.10 Objectives

The main objective of this project was the generation of a set of highly reusable FP customizable operators, ready for rapid systems prototyping. The target set of FP operators include addition/subtraction, multiplication, division, and square root; along with all the underlying fixed-point arithmetic units and supporting datapath elements.

All units should have adjustable precision and range, including those to satisfy the IEEE-754 standard for FP arithmetic. Moreover, the target speed of the units should be adjustable such that only the required hardware for the target speed is included in the synthesis. The operating frequencies should be competitive with current technologies.

The units should be modular enough to allow easy maintenance and upgrade and yet be portable to a wide range of synthesis targets.

Chapter 2

Previous work

The implementation of applications requiring dedicated arithmetic hardware are commonplace nowadays. They can be found in full custom ASICs, general-purpose processors, and particularly in digital signal processing systems [7].

Arithmetic operations are performed using either fixed-point or FP arithmetic. However, in those applications where a wide dynamic range is required, fixed point implementations are ruled-out, leaving FP as the only choice.

FPGAs have emerged as an alternative implementation platform for arithmetic circuits. Although early FPGAs were not a good alternative to implement FP arithmetic due to resources limitations, contemporary FPGAs have overcome this difficulty. This progress is evidenced in this literature research. The following sections report relevant previous work on arithmetic hardware targeting FPGAs. Section 2.1 presents work done on fixed-point arithmetic designs while section 2.2 does the same with FP arithmetic.

2.1 Fixed Point Implementations

The development of high throughput fixed point operators is very important for high performance FP units. An FP operator is basically a group of simpler units working together. A number of approaches have been found to deal with this issue by using pipeline techniques.

Traditional pipelined adders for parallel addition of two operands are typically based on carry save addition (CSA) or ripple carry adders. CSA operates at higher frequencies than ripple-carry units at the expense of a greater circuit complexity. Dadda and Piuri proposed a novel approach, which replaces the ripple-adders by fast adders, obtaining higher throughput than with CSA [5].

Asato et Al. developed a compiler to produce customized, pipelined array multipliers optimized to operate at a specified clock rate [2]. Their pipeline insertion method introduces rows of latches through the multiplier structure. It divides the array into rows of cells that operate independently from each other. The maximum operating frequency is always limited by the ripple-adder (last row). The results of this approach for a 4-stage, 32×32 array multiplier were a 33% area increase and three times the clock rate compared to a non pipelined array multiplier.

Louie and Ercegovac explored the mapping of digit-recurrence type division algorithms on a Xilinx XC4010 FPGA [17]. They studied the FPGA structure looking for suitable implementation techniques for SRT radix-2 division. They also developed a division algorithm which uses carry-propagate adders (CPA) [1] instead of CSA. They proved its usefulness when fast CPA are available, obtaining a better performance and less resources consumption than with CSA.

2.2 Floating Point Implementations

Early FP implementations on FPGA adopted custom data formats to enable a single FPGA solution, or involved multiple FPGAs for implementing IEEE 754-compliant, single precision FP

[1]Carry propagate adder is the same as carry-ripple adder

arithmetic. The use of serial arithmetic or the avoidance of some IEEE standard features was another way to deal with the early FPGAs limitations [1]. Nowadays, multiple FP units can easily fit in contemporary FPGAs. Some of the most relevant works on FP units for FPGA implementation are presented. Also, some data about these implementations like target FPGA, space complexity, operating frequency and case study architectures are offered.

Narasimhan et Al. implemented an FP adder, which used 82% of the CLBs of a Xilinx XC4005 FPGA and operated at 100 MHz [19]. They used a variation of the IEEE standard (13-bit): 9-bit mantissa , 4 -bit exponent, and 1-bit sign. Implemented as an 8-stage pipelined design.

Shirazi et Al. developed an FIR Filter (16-bit format: 1-bit sign, 6-bit exponent, and 9-bit mantissa) and a 2D FFT (18-bit format: 1-bit sign, 7-bit exponent, and 10-bit mantissa) [24]. These deviations from the FP standard were introduced due to limitations on the Splash-2 data path. They implemented FP addition, subtraction, multiplication, and division. They used division through multiplication of the inverse. The calculation of the divisor's reciprocal (mantissa) was made with the aid of an external memory. Three integer multipliers were tested: the multiplier from the Synopsys3.0a VHDL compiler, an array multiplier, and various pipelined schemes which create stages of partial products and a final addition stage. These FP operators have 3 pipeline stages except for the divider, which has 5 pipeline stages. The FP adder/subtracter, multiplier and divider operated at 8.6 MHZ, 4.9MHz, and 4.7 MHz, respectively.

Louca et Al. implemented a single precision FP adder and multiplier, using 72% of an Altera FLEX 8000 series FPGA [16]. This implementation used fixed-point bit parallel adder and digit-serial multiplier due to space limitations. Digit serial operations have some disadvantages. They require parallel-to-serial and serial-to-parallel registers and extra clock cycles to complete an operation. The pipelined version of the FP multiplier produced one result every 6-clock cycle at 15.9MHz.

Walters et Al. presented a pipelined scalable structure adaptable for filtering, convolution, and correlation tasks [30]. Used 32-bit FP arithmetic and a systolic array configuration. The FP adder was an 8-stage pipeline, and the FP Multiplier was a 13-stage pipeline. The implemented filter could be expanded over multiple boards to implement higher order filters and/or real-time processing. It was verified at 20MHz in a WILDFORCE board which has four XC4036EX-3 FPGAs with 32k x 32 SRAMs. It consumed 98% of the FPGA resources.

Ligon et Al. created a structure for matrix multiplication. It was implemented on a Xilinx 4029E, 4062XL, and 40250XV FPGAs [14]. Deep pipelining (15-stage) was used in order to reduce processing time, at the expense of latency. The FP single precision adder and multiplier used by this structure operated at 33MHz and 40 MHz, respectively. They considered four multiplication schemes for the FP multiplier: array multiplier, bit-serial multiplier, digit-serial multiplier, and both recoding multiplier.

Souani et Al. implemented FP addition with the 5-bit exponent and 16-bit mantissa [25]. The implementation target was a Xilinx XC4000 FPGA. Used 108 CLBs and operated at 7.5 MHz.

Richard et Al. presented a scalable, parallel implementation of a weight calculation architecture [29]. They employed custom FP arithmetic optimized for the target application; also created parameterized fixed-point and FP operators and more complex DSP operators, such as FIR filters, simple digital down conversion, radar receivers, beamformers, and vector multiply. They used 16-bit mantissa and the 6-bit exponent (two's complement representation) on a Xilinx XCV3200E-8 FPGA. Table 2.1 compares their weight calculation architecture with others. ASIC 0.35 μm refers to the PowerPC 7400(Altivec Processor). ASIC 0.18 μm is an estimated of ASIC 0.35 μm at this feature size.

Allan et Al. developed a method on Handel C language to produce technology independent, and variable pipelined designs [1]. This allows for parameterization of design's precision and range, and optional inclusion of IEEE-754 FP features at compile time. Single precision FP Adder and Multiplier were implemented in a Xilinx XCV1000 device, achieving 28MFLOPs. FP units were used for Two-Dimensional Fast Hartley Transform (an optimization of the FFT when only real

Architecture	Clock (MHz)	Number of processors	MFLOPS
PDSP TMS320C6701	167	1	250
ASIC 0.35 μm	100	21	32,900
ASIC 0.18 μm	190	74	225,000
FPGA XCV3200 E-8(0.18μm)	150	9	20,850

Table 2.1: Maximum throughput in weight calculation for different architectures.

numbers are required). The FHT was implemented for 1024 elements data sets; using around 59% of a Xilinx XCV1000 for single precision FP format. The design could be clocked up to 22 MHz, producing the 2D FHT of a 1024 elements data set in around 254 μs. Table 2.2 compares the time for 1K-point transform with other implementations.

Processor Clock speed(MHz)	Clock Speed (MHz)	Time for 1K-point transform (μ s)
Double BW power FFT	128	10
Texas Mem Sys TM-66	50	65
FHT Processor (Allan et Al.)	22	254
Sharc ADSP-21061	40	460
Pentium-III	800	469

Table 2.2: Comparison between the FHT processor and other systems. The first two are dedicated FFT devices while the rest are programmable DSP or supercomputers.

Three companies which offer commercial FP Cores were found. They provide the designs upon customer request at several fixed operand sizes. These cores work for the Xilinx Virtex series FPGAs. Their performance and resource usage is presented in the following paragraphs.

Designers at Dillon Engineering Inc. used ParaCore Architect IP Core to generate IEEE-754 compliant FP addition, subtraction, multiplication, division, square root, reciprocal, and conversion units (single and double precision). These units were designed for any exponent and mantissa width

[11]. Pipeline stages are configurable via ParaCore parameters. Used Goldschmidt's algorithm for division, square root and inverse. Maximum operating frequency is not specified for each operator, although operating frequencies over 100MHz are claimed on Xilinx Virtex II FPGAs. Table 2.3 presents the slice consumption of the units for different operand sizes.

Floating Point Unit	Parameters		Size
	mantissa width	exponent width	Slices
FP Adder	24	8	385
	53	11	1124
FP Subtraction	24	8	385
	53	11	1124
FP Divider	24	8	686
	53	11	3858
FP Multiplier	24	8	215
	53	11	783
FP Square Root	24	8	700
	53	11	3500

Table 2.3: Floating Point Cores area usage in a Virtex II FPGA by Dillon Engineering Inc.

Nallatech Limited Inc. offers IEEE 754-compliant Floating Point Cores which includes single precision addition and subtraction, multiplication, division, square root, float to integer conversion and integer to float conversion [15]. Cores are provided as optimized netlist. Operating frequency, latency, and resource consumption are provided in Table 2.4.

QuinetiQ Ltd. offers variable wordlength IEEE-754 compliant FP units[23], implemented as fully pipelined FP addition/subtraction, multiplication, and division, and square root. Used a fixed-point core for division and square root. Table 2.5 presents the main characteristics of these cores.

Floating Point Core	Operating Frequency (MHz)			Size	Latency
	SG -6	SG -7	SG -8	(Slices)	(Clock cycles)
Adder	115	129	147	376	15
Subtraction	115	129	147	376	15
Multiplier	116	130	152	593	23
Square Root	121	135	154	543	34
Division	120	133	155	980	35
FP to integer	143	162	171	134	4
Integer to FP	155	164	178	106	3

Table 2.4: Floating Point Cores area usage and performance in a Virtex II FPGA by Nallatech Limited Inc., tested at different speed grades (SG)

2.3 Previous work overview

The previous sections reported the most relevant work related to this project. Some of the main points are summarized:

- A time effective design should include pipeline insertion to achieve high throughput units. Almost every consulted work uses pipeline techniques and demonstrated their usefulness.
- The internal FPGA structure should be carefully studied to obtain optimal designs when targeting this implementation platform.
- Addition is a very common operation. FPGAs support very well this operation. CSA can be replaced by fast CPA offered by the FPGAs.
- Resource limitations should be observed carefully when dealing with speed-area trade off. Some implementations use custom data sizes different from that of the IEEE standard, when resource consumption is an important constraint.

Floating Point	Parameters		Frequency (MHz)		Size	Latency
Unit	mantissa width	exponent width	SG -6	SG -8	Slices	Clock cycles
FP Adder	8	6	153	199	121	10
	12	6	143	193	158	10
	16	6	153	188	208	11
	20	6	147	182	247	11
	14	8	137	169	306	11
FP Divider	8	6	202	259	124	11
	12	6	182	238	220	15
	16	6	166	216	348	19
	20	6	154	203	512	23
	24	8	142	189	711	27
FP Multiplier	8	6	130	164	67	5
	12	6	129	165	119	6
	16	6	125	157	229	6
	24	8	122	165	171	6
	20	6	122	152	326	6
FP Square Root	16	6	173	191	620	19

Table 2.5: Floating Point Cores area usage and performance in a Virtex II FPGA by QuinetiQ ltd, tested at different speed grades (SG)

- Serial arithmetic leads to hardware savings, but requires parallel-serial interface and longer processing time.

- Designs can be expanded over multiple FPGAs when resources limitations occur. The major drawback is that partitioning and routing gets more complicated. Also interconnection delays between FPGAs are generally higher, which limits the maximum operating frequency.

- Nallatech Limited Inc., QuinetiQ ltd, and Dillon Engineering Inc. appear to be the best FP units implementations. They have commercial designs with operating speeds exceeding 100MHz and provide a set of FP units and DSP solutions. Their designs are not provided as VHDL source code. Each core is sold with fixed features and does not provides so much flexibility. Also, these cores are less portable because they use dedicated resources from specific FPGA families.

Chapter 3

Design Methodology

The prototypes of the integer and FP units were designed as structural descriptions composed of several design hierarchies. A bottom-up design methodology has been followed using VHDL. Each design is accessed as a structural COMPONENT which accepts the sizes of the operand fields as GENERIC parameters. These parameters are passed along the hierarchical structure specifying the widths of the mantissa, exponent fields, and pipeline parameters; and accordingly, generating the appropriate sized units. This allowed for a modular and highly reusable set of units. The developed FP prototypes are adjustable to meet the IEEE 754 standard. Units were designed for parallel input/output interface. Xilinx FPGAs were used as synthesis target.

FPGAs unit's building blocks were developed and carefully inspected looking for improvement opportunities. Section 3.1 presents some of the improvement procedures, focusing on those of major impact in the FP units performance.

Array-type architectures were extensively used. They allow for an efficient implementation of pipeline insertion algorithms, but they consume more hardware resources than other architectures. Pipelining in these structures is accomplished by latch insertion between array's rows's and synchronizing the input/output ports. Pipeline techniques were used to increase the units throughput and to control their granularity. A detailed description of the developed pipeline insertion algorithm is presented in Section 3.2. Section 3.3 shows how these methodologies were used in the implementation of the FP units.

3.1 Operand Improvement Techniques

The general improvement procedure was based on careful inspections of the FP units VHDL code. For example, a for-loop is a typical resource consuming structure. That is because the synthesis tools unroll all loops and generate hardware for each loop iteration. Another procedure was the inspection of components, focusing on the most commonly used by the FP operators. For example, some structures can be substituted by fast adders, which have good support in current FPGAs. Improvements in square root and division operators were made by changing the rows of basic cells by fast adders. Some of the basic cell's functionality was moved outside the array in order to allow the use of fast adders. In the case of square root, the majority of the XORs receive a combination of input operands that allowed for additional simplifications. These implementational changes are explained in Sections 4.1.6 and 4.1.7.

Pipeline stages were inserted as rows of latches though the array structures, improving their timing characteristics. This task was done using the variable pipeline insertion algorithm presented in [21]. Pipeline depth and operand size were controlled by means of VHDL generic parameters, specified at compile time.

3.2 Pipeline Techniques

An algorithm for pipeline insertion in structures such as adders, multipliers, and multi-stage operators was developed. It receives two main parameters, which include the number of circuit stages (s) and the number of pipeline stages (p). The algorithm generates (x) cells of granularity (g_1) $= \lceil s/p \rceil$ where $x = mod(s/p)$ and $(p - x)$ cells of granularity (g_2) $= \lfloor s/p \rfloor$; where $\lceil . \rceil$ and $\lfloor . \rfloor$ denote the ceiling and floor of the enclosed expression. This technique has been used for the fixed-point adder, multiplier, divider, square root, barrel shifter, normalizer, and shift register. These pipelined operators form part of the developed FP units. Under this technique, the granularity of each component of an FP unit can be independently adjusted.

The selection of the pipeline depth parameters involves multiple steps. These steps, which are performed manually, are summarized in Figure 3.1 . Once the desired operating frequency is established, an initial implementation is obtained using minimal values for the pipeline parameters. An analysis on the timing reports is done identifying the design's bottlenecks. Pipeline parameters are increased until either the desired operating frequency is obtained or the finest possible granularity of the component is reached. Several refinement cycles might be required until reaching this point. The desired frequency might not be achieved when the pipelining capacity of the component is exhausted.

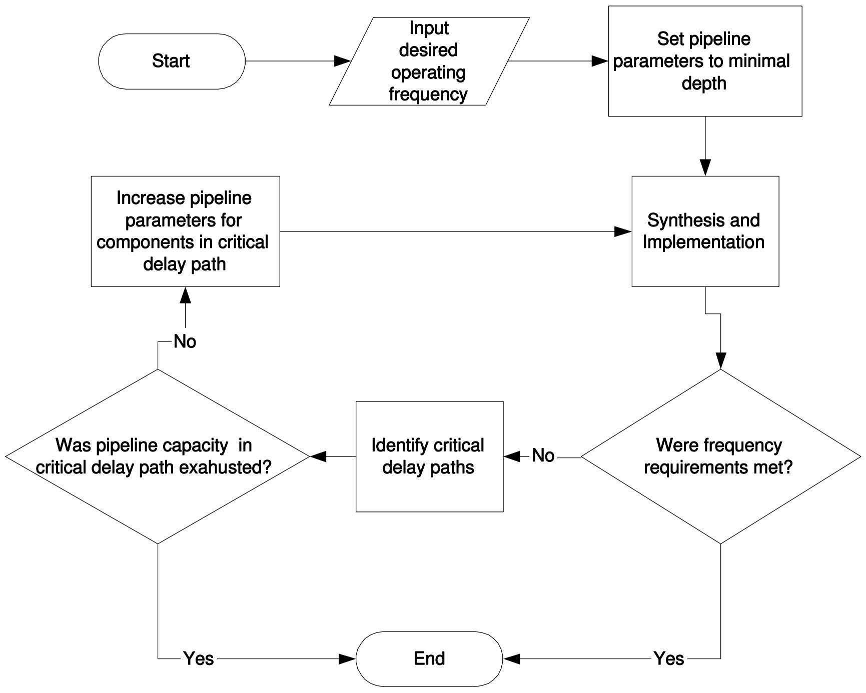
Start
Input desired operating frequency
Set pipeline parameters to minimal depth
Increase pipeline parameters for components in critical delay path
Synthesis and Implementation
No
Was pipeline capacity in critical delay path exahusted?
Identify critical delay paths
No
Were frequency requirements met?
Yes
End
Yes

Figure 3.1: Pipeline Improvement Algorithm

3.3 Architecture of FP Units

The following sections describe the main architectural features of the developed arithmetic units. Discussion is made by FP units and their subcomponents. Data related to operating frequency and FPGA slices consumption are presented.

3.3.1 Floating Point Adder

The basic structure of the FP adder has been designed to provide scalable mantissa and exponent fields as well as a variable number of pipeline stages. Figure 3.2 shows the basic adder's structure. Exponent and mantissa field widths are specified through parameters *ebit* and *mbit*, respectively. The amount of pipeline stages is specified through five parameters *(pip1, pip2a, pip2, pip3a, pip3b)*. An example of FP addition with *ebit* = 3 and *mbit* = 9 is presented in Table 3.1. Descriptions of the main operators, components and processes used by the FP Adder, along with their associated pipeline parameters are provided in the following sub-sections.

Operation	Operand 1			Operand 2		
	Sign 1	Exponent 1	Mantissa 1	Sign 2	Exponent 2	Mantissa 2
Initial inputs	0	010	011011010	1	100	010011101
Operand swapping	1	100	010011101	0	010	011011010
Result sign(larger operator's sign)	Sign is 1. Subtraction instead of addition (opposite signs)					
Appending mantissa's implicit one	1	100	**1**.010011101	*	010	**1**.011011010
Exponent subtraction ($ebit1 - ebit2$)	100-010=010 (2_d)					
Shift *mant*2 2 places to the right	1	100	1.010011101	*	***	**0.0**10110110
Mantissa Addition	1.010011101 - 0.010110110 = 0.111100111 (subtraction because of opposite signs)					
Leading zero detection (1 zero)	1	010	**0**.111100111	*	***	**********
Mantissa normalization	1	010	1.11100111**0**	*	***	**********
Exponent adjustment	leading zero amount is subtracted to resultant exponent: 010-001=001					
Implicit one removal	1	001	*.111001110	*	***	**********
Final result	1	001	111001110	*	***	**********

Table 3.1: Floating Point Addition example

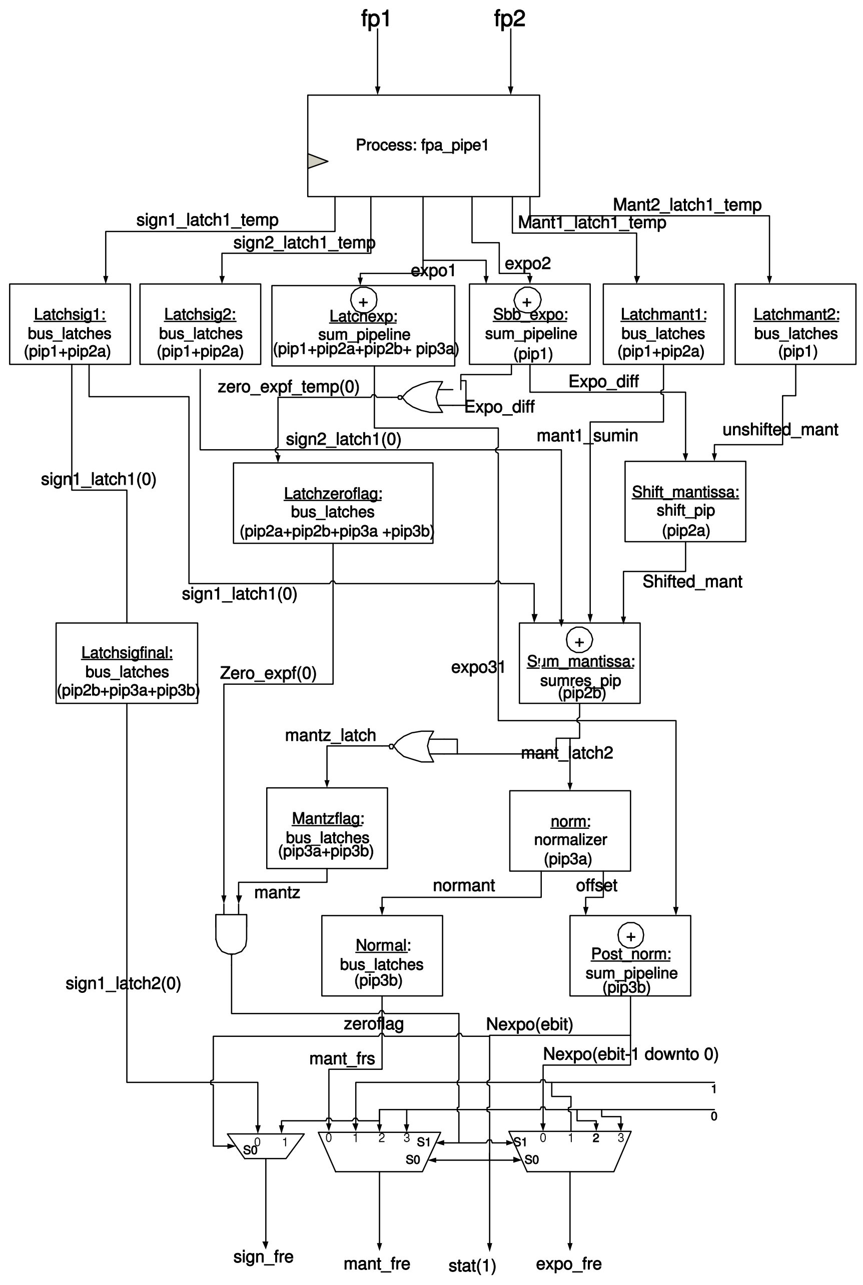
fp1
fp2
Process: fpa_pipe1
sign1_latch1_temp
sign2_latch1_temp
expo1
expo2
Mant2_latch1_temp
Mant1_latch1_temp
Latchsig1: bus_latches (pip1+pip2a)
Latchsig2: bus_latches (pip1+pip2a)
Latchexp: sum_pipeline (pip1+pip2a+pip2b+ pip3a)
Sbb_expo: sum_pipeline (pip1)
Latchmant1: bus_latches (pip1+pip2a)
Latchmant2: bus_latches (pip1)
zero_expf_temp(0)
Expo_diff
Expo_diff
unshifted_mant
sign2_latch1(0)
mant1_sumin
sign1_latch1(0)
Latchzeroflag: bus_latches (pip2a+pip2b+pip3a +pip3b)
Shift_mantissa: shift_pip (pip2a)
Shifted_mant
sign1_latch1(0)
Latchsigfinal: bus_latches (pip2b+pip3a+pip3b)
Zero_expf(0)
expo31
Sum_mantissa: sumres_pip (pip2b)
mantz_latch
mant_latch2
Mantzflag: bus_latches (pip3a+pip3b)
norm: normalizer (pip3a)
mantz
normant
offset
Normal: bus_latches (pip3b)
Post_norm: sum_pipeline (pip3b)
sign1_latch2(0)
zeroflag
Nexpo(ebit)
mant_frs
Nexpo(ebit-1 downto 0)
1
0
S0
S1
sign_fre
mant_fre
stat(1)
expo_fre

Figure 3.2: Base structure of pipelined FP Adder.

Operators:

Comparator: This operator is used in various stages of an FP unit. There are multiple comparator architectures. Four schemes were tested, which include: comparison by subtraction, using the synthesis tool's comparator, two modular approaches based on 2-bit number comparator, and a behavioral scheme which is further described.

The subtraction scheme subtracts both numbers and evaluates the result's sign to determine the comparison result. The modular comparator is based on 2-bit two-number comparator, arranged as a binary tree, as seen in Figure 3.3. The internaloperationof each module is explained in Table 3.2. Each stage reduces the size of the numbers to be compared through, while maintaining their relative magnitude. The process is completed when both numbers are reduced to 1-bit. A variant of this scheme uses the synthesis tool to implement the basic cell, while maintaining the same structure of the modular comparator. The synthesis tool seems to use a scheme similar to the subtraction scheme because their performance is similar as the operand sizes are increased. The behavioral scheme used a VHDL function that converts incoming vectors types to integers types. The result for this comparator is set by using VHDL properties for integer types.

Input Condition	Output B	Output A
$B_1B_0 > A_1A_0$	1	0
$B_1B_0 < A_1A_0$	0	1
$B_1B_0 = A_1A_0$	0	0

Table 3.2: Internal operation of each module of the modular comparator

Adder: Figure 3.4 contains the selected architecture for pipelined fixed-point addition, which provides scalability in terms of data size and pipeline stages. This architecture reduces the carry propagation chain length by inserting latches across it. Fast adders are used for each stage. The synthesis tool adder is used for this purpose because of its advantage over the CLA adder. Additional latches are used to propagate inner stage carry and input data to be used on subsequent

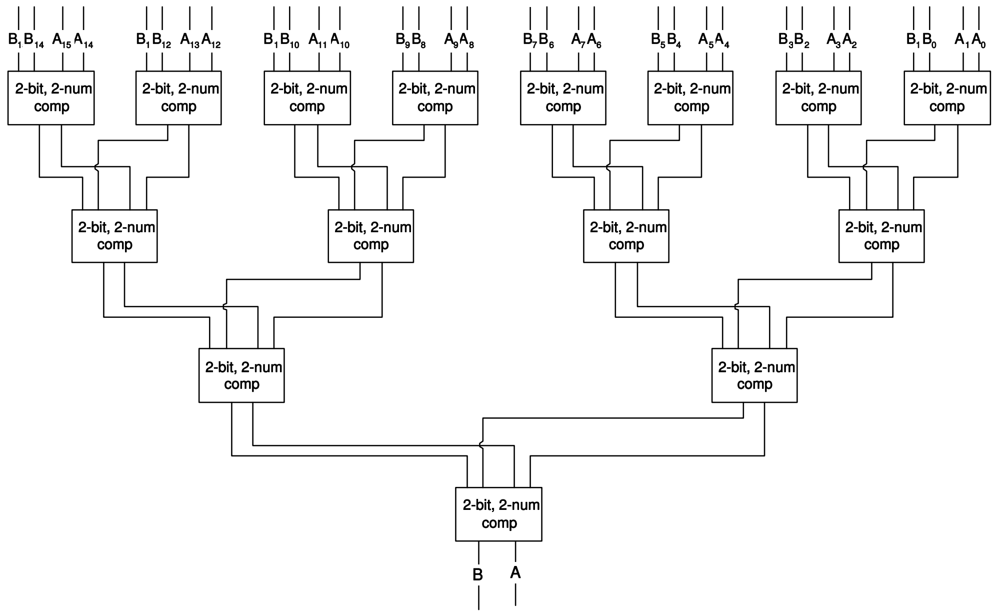

Figure 3.3: 16-bit Modular Comparator

stages. These latches also balance the output delays. Similar pipelined adder's architectures have been proposed by Dadda and Piuri [5]. One of them is similar to the developed approach except for the use of a ripple-carry adder instead of a fast adder. The other architecture is based on carry save addition (CSA). The CSA scheme has greater circuit complexity than the selected scheme when implemented in FPGAs. On the other hand, the CSA scheme has better delay characteristics when implemented in ASICs. These CSA uses half adders which are faster than full adders in ASIC implementations.

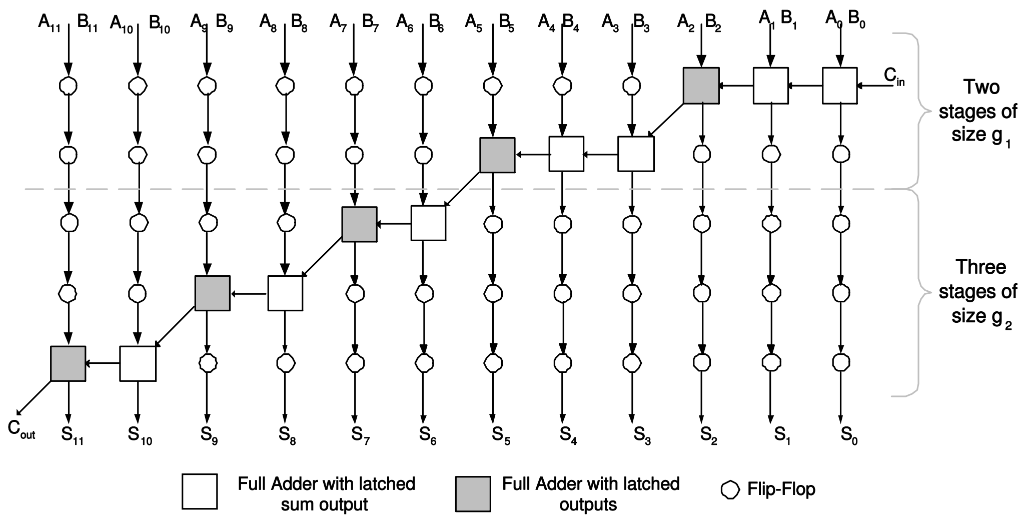
A11 B11 A10 B10 A9 B9 A8 B8 A7 B7 A6 B6 A5 B5 A4 B4 A3 B3 A2 B2 A1 B1 A0 B0
Cin
Two stages of size g1
Three stages of size g2
Cout S11 S10 S9 S8 S7 S6 S5 S4 S3 S2 S1 S0
Full Adder with latched sum output
Full Adder with latched outputs
Flip-Flop

Figure 3.4: 12-bit pipelined fixed-point adder: $s = 12, p = 5$.

Shifter: A right shifter is used to denormalize the smaller mantissa as required by the exponent equalization step. The shifter, whose structure is illustrated in Figure 3.5, uses a log-2 right shift scheme based on multiplexers. This scheme was found to be similar to that proposed by Heo [10]. The implemented structure is scalable in terms of data size and pipeline stages. It has two inputs and one output. IN is an input bit vector of size n; the one to be shifted. $SHIFT$ is an input bit vector of size m; it specifies the number of shiftings to be made. The maximum value of m is given by $m = log_2(n)$. Also m is the amount of multiplexer stages (s) needed to implement the shifter. OUT is an output bit vector of size n; it represents the IN shifted $SHIFT$ times to the right.

The idea behind this type of shifter is the fact that for vector E, each bit has its own shift weight. Bit 0 specifies 0 or 1 shifting, bit one specifies 0 or two, bit 2 specifies 0 or 4, etc. So each one of these bits control a multiplexer in which one input is a bit vector and the other is the same bit vector with inserted zeros at the left and a right displacements of bits equal to the amount of inserted zeros.

The advantages of this scheme are that for given values of n and m, shiftings are made in the same period of time, no matter the amount of shifting needed. Also it uses less hardware than a traditional barrel shifter because of it's logarithmic grow which is a function of the size of the data. An example of a 24-bit input number (101101011001010001101011) and a shift amount of 11_d (01011) is presented in Table 3.3.

Multiplexer Stage	Shift Amount (binary format)	Input operand
		101101011001010001101011
1	0 (bit #4)	101101011001010001101011
2	1 (bit #3)	**00000000**1011010110010100
3	0 (bit #2)	**00000000**1011010110010100
4	1 (bit #1)	**0000000000**10110101100101
5	1 (bit #0)	**00000000000**1011010110010

Table 3.3: Example of the right shifter. Shifting amount is 11.

Normalizer and leading zero detector: IEEE standard for FP numbers specifies that FP numbers should be normalized (mantissa value is in the range [1,2)). However, FP addition requires denormalized numbers in their internal structure. A leading zero detector is needed to detect how many shifts are necessary to normalize a number. The normalizer uses that information to eliminate all the leading zeros by left shifting. A topology, shown in Figure 3.6, has been developed, which follows a structure similar to that of the shifter. The last multiplexer's output is the normalized version of the unit's input. The result of each multi-input nor-gate is combined to form

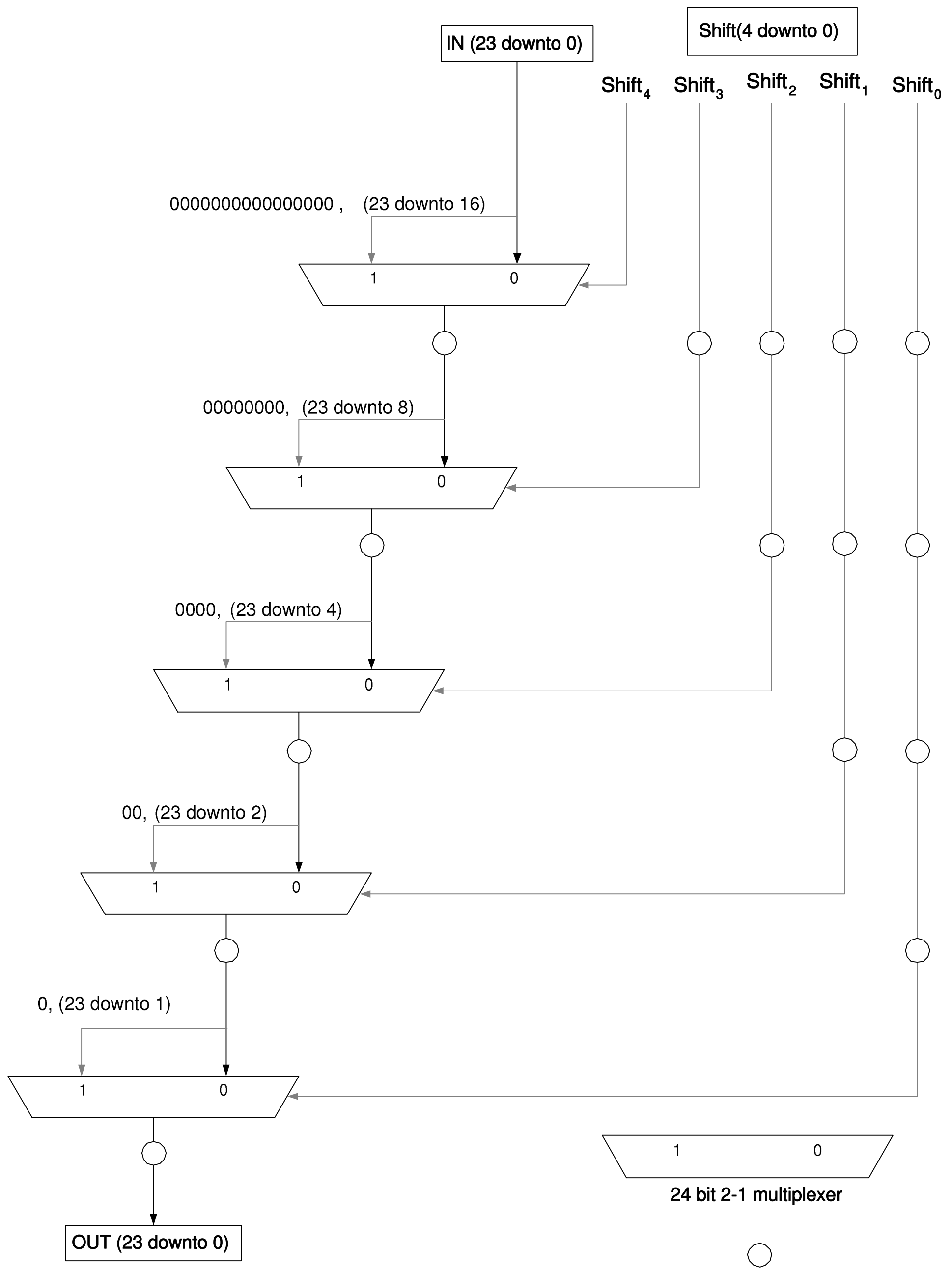

Figure 3.5: Scalable pipelined shifter.

the total leading-zero amount. This topology improves over previous approaches by performing the zero leading detection and mantissa normalization in a single step, without the requirement of independent operations. Also flip-flops can be inserted at the multiplexer's and comparator's output, allowing the operation of the circuit at higher frequencies. A numerical example of this unit with input operand (000000000000000001101011) and 17 leading zeros is presented in Table 3.4.

Multiplexer Stage	Operand transformations	Leading zero amount (5-bit)
	000000000000000001101011	*****
1	01101011**0000000000000000**	1****
2	01101011**0000000000000000**	10***
3	01101011**0000000000000000**	100**
4	01101011**0000000000000000**	1000*
5	1101011**00000000000000000**	10001

Table 3.4: Example of the normalizing and zero detection unit.

FP Adder Components:

Sbb_expo (*pip*1): Subtracts both exponents to determine the number of shifting positions when denormalizing the smaller mantissa.

Latchexp ($pip1, pip2a, pip2, pip3a$): Adds one to the greater exponent to allow shifting the binary point of the mantissa by one place to the left.

Shift_mantissa (*pip*2*a*): Right shifts the mantissa of the smaller input operand. The shifting amount is specified by the result of the exponent subtraction.

Sum_mantissa (*pip*2*b*): Mantissa addition or subtraction depending on the sign of the input operands.

Normalizer (*pip*3*a*): Detects leading zeros on the mantissa and normalizes it.

Post_norm (*pip*3*b*): Adjusts the exponent result by subtracting the number of leading zeros provided by the normalizer.

Bus_latches: Maintain the data integrity through the pipeline.

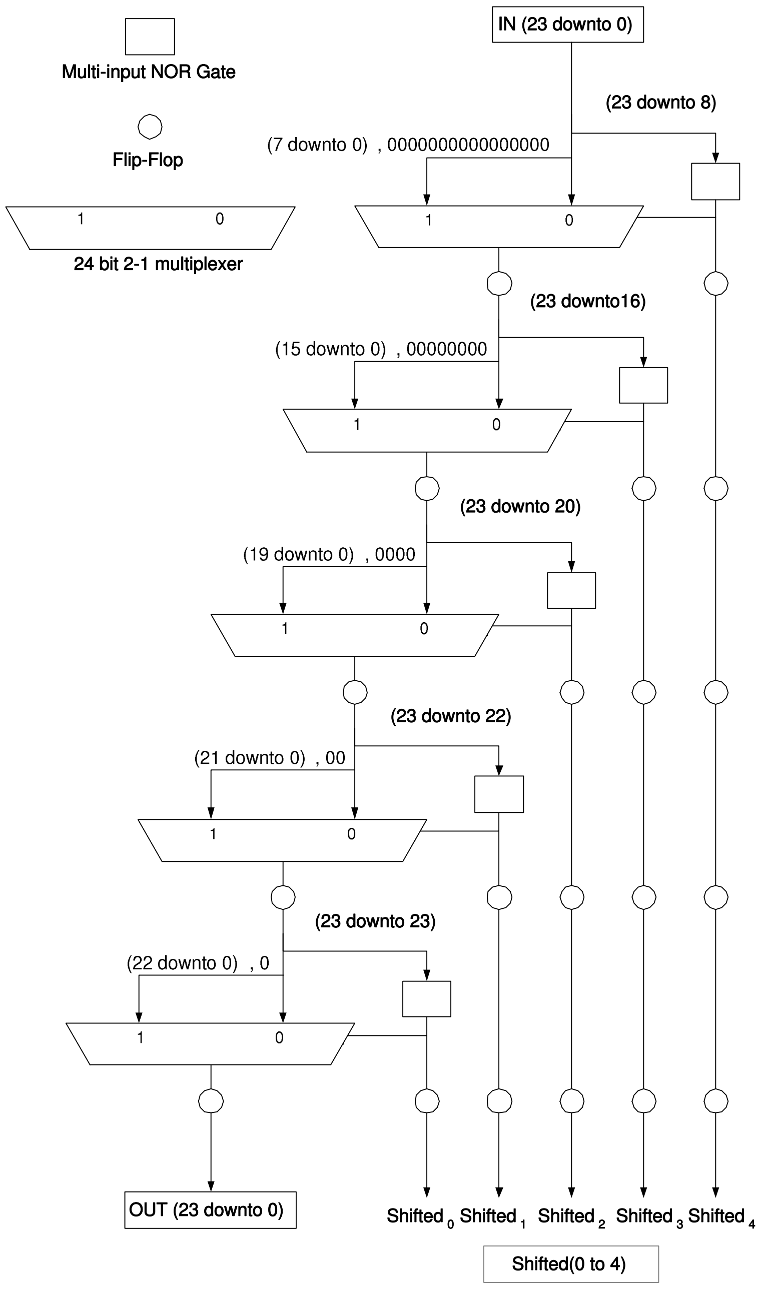

Figure 3.6: Normalizing and zero detection unit.

FP Adder Processes:

Fpa_pipe1: Compares the input operands and swaps them if necessary. Also adds the implicit hidden one to the mantissas.

3.3.2 Floating Point Multiplier

The basic structure of the FP multiplier has been designed to provide scalable mantissa and exponent fields as well as a variable number of pipeline stages. Figure 3.7 shows the general organization of the FP multiplier. Exponent and mantissa widths are specified through parameters *ebit* and *mbit*, respectively. An example of FP Multiplication with $ebit = 3$ and $mbit = 9$ is presented in Table 3.5. The number of pipeline stages is specified through three parameters $(pip1, pip1b, pip2)$. Descriptions of the main operators, components and processes used by the FP Multiplier, indicating their associated pipeline parameters are provided in the following sub-sections.

Operation	Operand 1			Operand 2		
	Sign 1	Exponent 1	Mantissa 1	Sign 2	Exponent 2	Mantissa 2
Initial inputs	0	010	011011010	1	100	110011101
Appending mantissa's implicit one	0	010	1.011011010	1	100	1.110011101
Result sign(XOR of the signs)	**1**	010	1.011011010	*	100	1.110011101
Exponent addition	1	**110**	1.011011010	*	***	1.110011101
Mantissa multiplication (upper half)	1.011011010 × 1.110011101 = **10.10010011**0110110010					
Mantissa normalization	10.10010011 ⇒ 1.010010011					
Post-normalization exponent adjustment	Add one in the next step due to mantissa normalization					
Exponent bias subtraction	110-011+1=100 Exponent bias is 011(3_d)					
Implicit one removal	1	100	*.010010011	*	***	**********
Final result	1	100	010010011	*	***	**********

Table 3.5: Floating Point Multiplication example

Operators:

Array multiplier: Figure 3.8 shows the method used to insert pipeline stages to the array multiplier. Its architecture is scalable in terms of data size and pipeline stages. Its structure resembles

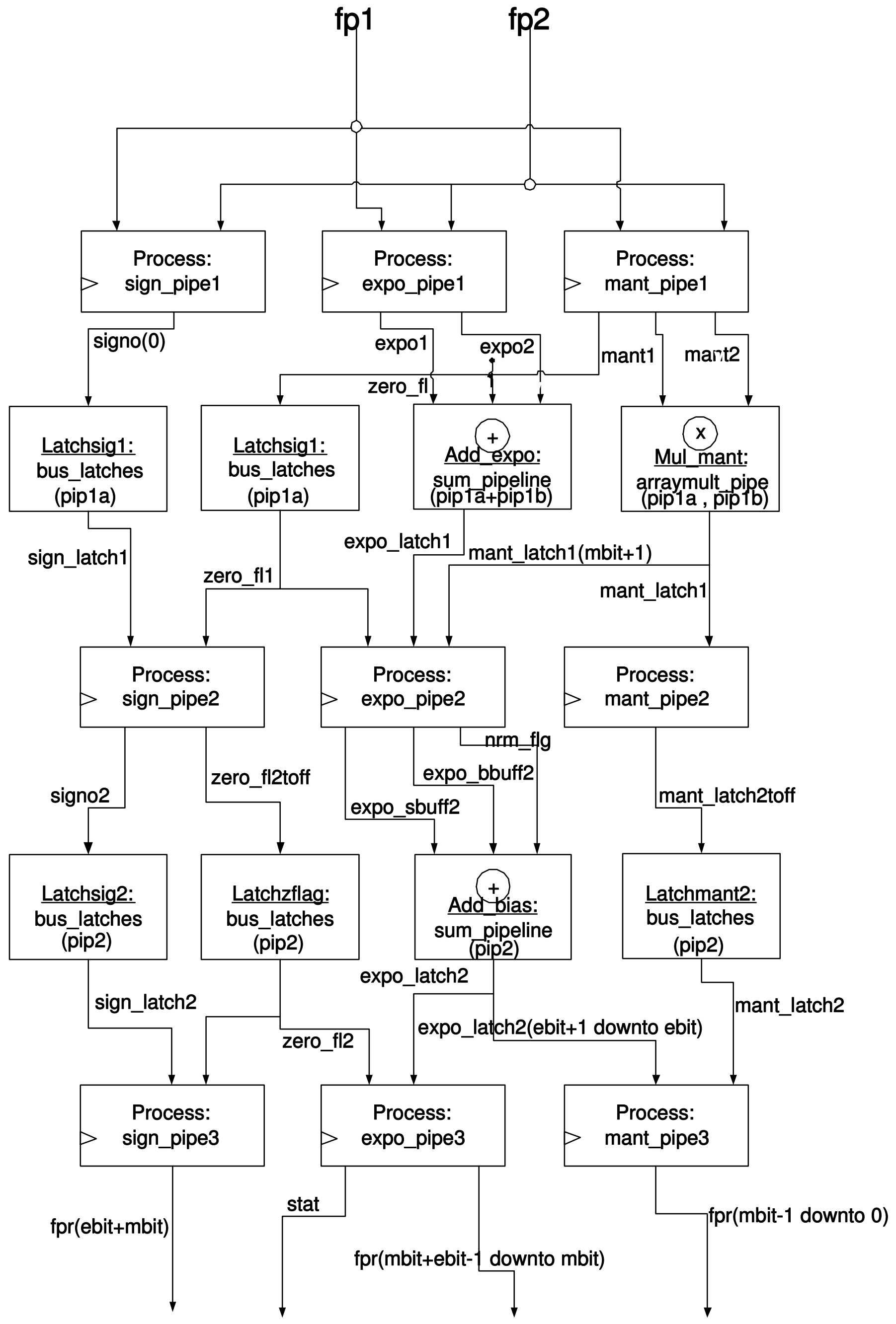

Figure 3.7: Base structure of pipelined FP Multiplier.

the manual process for tree multipliers, which appears in Figure 3.9. Some features are based on the scheme proposed by Asato [2]. Asato et Al. created a compiler to determine the pipeline insertion point for an array multiplier, given its data size and desired operating frequency. Pipeline stages were inserted as rows of latches between the multiplier's rows.

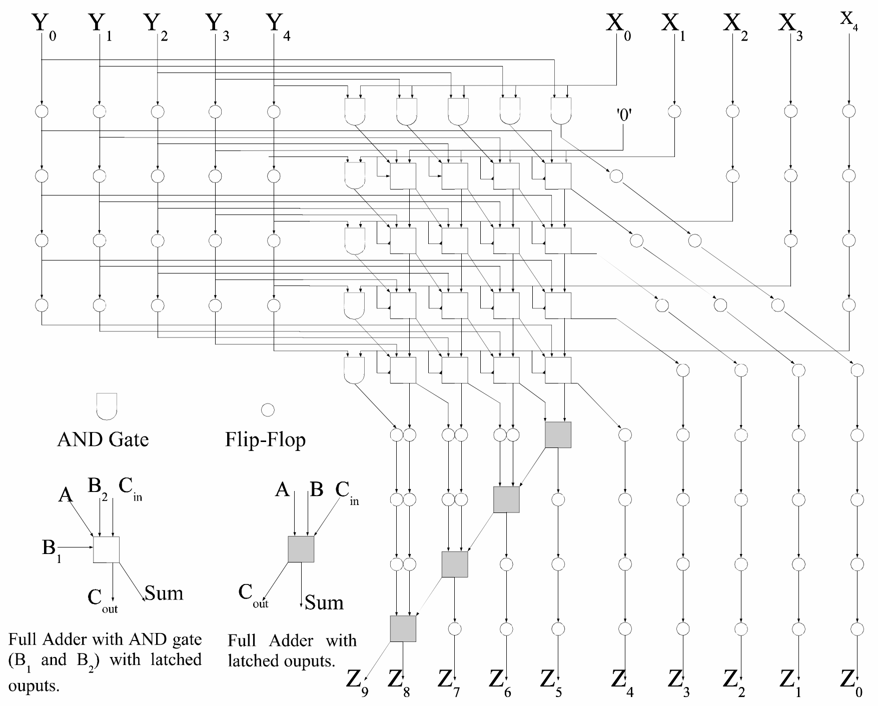
Y_0 Y_1 Y_2 Y_3 Y_4
X_0 X_1 X_2 X_3 X_4
'0'
AND Gate
Flip-Flop
A B_2 C_{in}
B_1
C_{out} Sum
Full Adder with AND gate (B_1 and B_2) with latched ouputs.
A B C_{in}
C_{out} Sum
Full Adder with latched ouputs.
Z_9 Z_8 Z_7 Z_6 Z_5 Z_4 Z_3 Z_2 Z_1 Z_0

Figure 3.8: 5x5 full-pipelined array multiplier.

Adder: Uses the same structure as those in the FP Adder.

$$\begin{array}{cccccccccc}
 & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
 & & & & & a_4x_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & a_4x_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & a_4x_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & a_4x_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
+ & a_4x_4 & a_3x_4 & a_2x_4 & a_1x_4 & a_0x_4 & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

Figure 3.9: 5-bit $\times$ 5-bit multiplication example

FP Multiplier Components:

Mult_mant ($pip1a, pip1b$): This is an ($mbit$+1)-wide pipelined array multiplier. The top-portion of the array has $pip1a$ stages. The bottom-portion has $pip1b$ stages. Performs mantissa multiplication.

Add_expo ($pip1a, pip1b$): Performs exponent addition using *ebit* as the operand width.

Add_bias ($pip2$): This is an ($ebit + 2$) bit adder. Performs exponent adjustments due to mantissa normalization and the subtraction of the exponent bias.

FP Multiplier Processes:

Sign_pipe1: Performs XOR of the signs.

Expo_pipe1: Pass the exponents.

Mant_pipe1: Zero detection and adds the implicit hidden one to the mantissa.

Sign_pipe2: Pass the zero flag and the sign.

Expo_pipe2: Prepares the operands for exponent bias subtraction by selectively performing two's complement.

Mant_pipe2: Normalizes the mantissa.

Sign_pipe3: Modifies the sign in case of zero result.

Expo_pipe3: Modifies the exponent in case of overflow or underflow or zero result. Also set the status overflow and zero flag.

Mant_pipe3: Set overflow and underflow conditions.

3.3.3 Floating Point Division

The basic structure of the FP divider provides scalable mantissa and exponent fields as well as a variable number of pipeline stages. The general organization of the FP divider, which resembles the structure of the FP Multiplier, is presented in Figure 3.10. Exponent and mantissa widths are specified through parameters *ebit* and *mbit*, respectively. An example of FP Division with $ebit = 3$ and $mbit = 9$ is presented in Table 3.6. The amount of pipeline stages is specified through two parameters $(pip1, pip2)$. Descriptions of the main operators, components and processes used by the FP Divider, indicating their associated pipeline parameters are provided in the following sub-sections.

Operation	Operand 1			Operand 2		
	Sign 1	Exponent 1	Mantissa 1	Sign 2	Exponent 2	Mantissa 2
Initial inputs	0	110	110011101	1	010	011011010
Appending mantissa's implicit one	0	110	1.110011101	1	010	1.011011010
Result sign(XOR of the signs)	**1**	110	1.110011101	*	010	1.011011010
Exponent subtraction	1	**100**	1.110011101	*	***	1.011011010
Mantissa division	1.110011101 ÷ 1.011011010 = 1.010001000					
Mantissa normalization (not needed here)	1	100	1.010001000	*	***	**********
Exponent bias addition	100+011=111 Exponent bias is 011(3_d)					
Implicit one removal	1	111	*.010001000	*	***	**********
Final result	1	111	010001000	*	***	**********

Table 3.6: Floating Point Division example

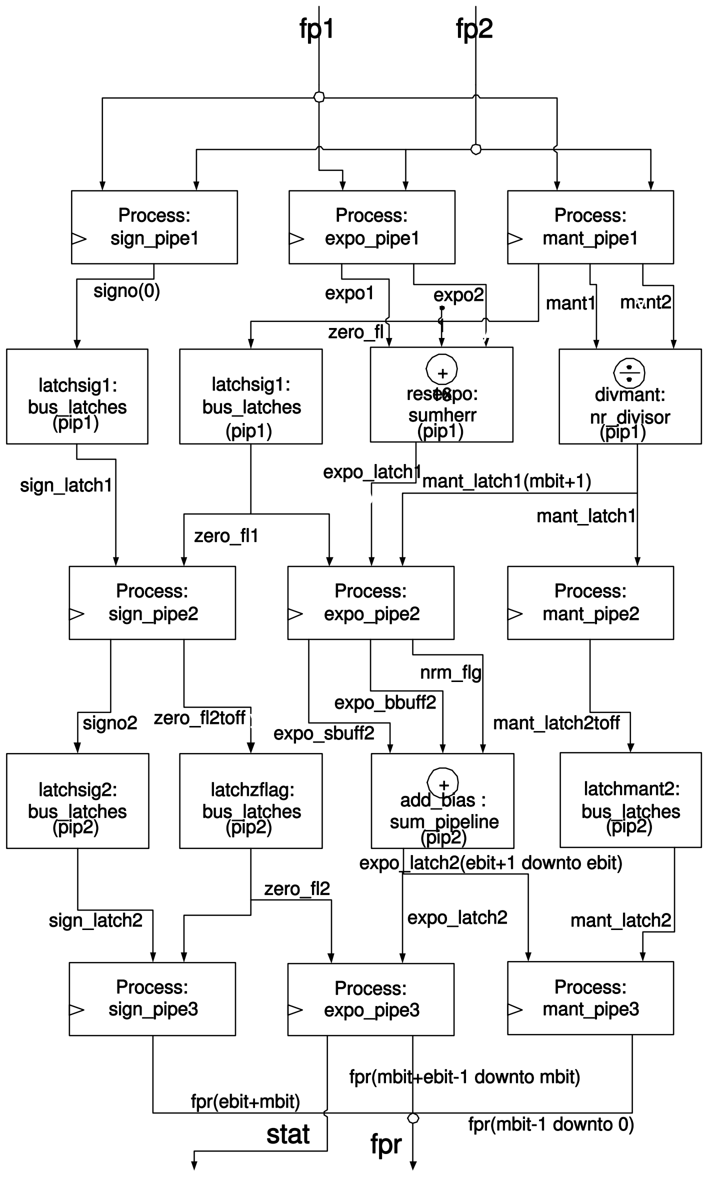

Figure 3.10: Base structure of pipelined FP Divider.

Operators:

Non Restoring Array Divider: The selected architecture for fixed-point division is a non-restoring array divider, as seen in Figure 3.11. It belongs to the digit-recurrence class. It is obtained by unfolding the iterations of a radix-2 digit-recurrence divider. The basic structure of this divider is a controlled adder/subtracter (CAS). The major drawback of this divider is that the carry signal propagates through each row of the array until it arrives to the leftmost bottom corner of the array. This scheme was selected because of its regular structure, suitability for a scalable implementation, and further pipeline insertion. Pipeline insertion between rows improves its timing characteristics.

The array divider inputs are the dividend and the divisor; its outputs are the quotient and the remainder. Note that the divisor, quotient, and remainder posses the same bit-length (n-bit) while the dividend is of ($2n - 1$-bit). The first row performs subtraction. The other rows perform subtraction or addition depending on the sign of the previous partial remainder. If the partial remainder is negative, the next row performs addition instead of subtraction. The correct remainder is restored in the next row. An example of this process is presented in Figure 3.12.

Adder: Uses the same structure as those in the FP Adder.

FP Divider Components:

Div_mant ($pip1$): This is a ($mbit + 1$)-bit pipelined array divider. Performs mantissa division.
Res_expo ($pip1$): Performs exponent subtraction using $ebit$ as the operand width.
Add_bias ($pip2$): This is an ($ebit + 2$) bit adder. Performs exponent adjustments due to mantissa normalization and exponent bias addition. The exponent bias addition compensates for the problem created by the exponents subtraction stage.

FP Divider Processes:

Sign_pipe1: Performs XOR of the signs.
Expo_pipe1: Prepares exponents for subtraction.

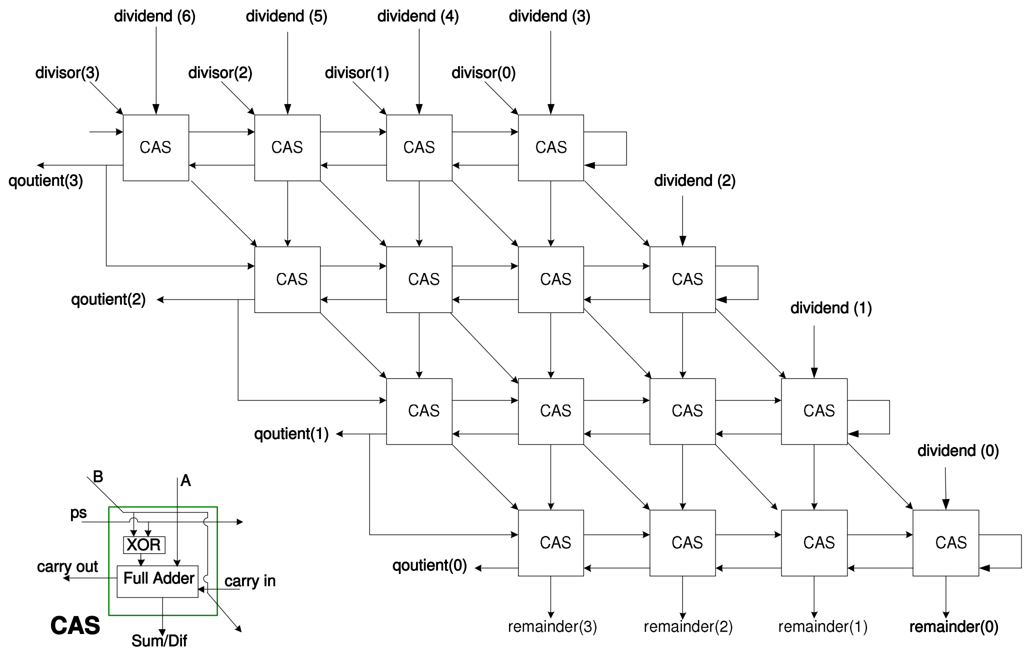

Figure 3.11: Nonrestoring Array Divider [22]

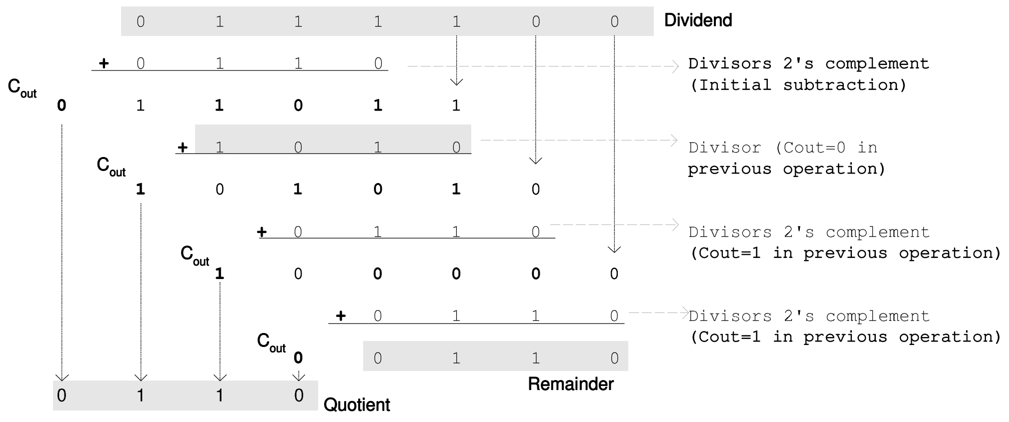
Dividend
Divisors 2's complement (Initial subtraction)
Divisor (Cout=0 in previous operation)
Divisors 2's complement (Cout=1 in previous operation)
Divisors 2's complement (Cout=1 in previous operation)
C_{out}
Remainder
Quotient

Figure 3.12: Example of a 7-bit dividend(60_d), 4-bit divisor(10_d) NR division

Mant_pipe1: Zero detection and adds the implicit hidden one to the mantissa.

Sign_pipe2: Pass the zero flag and the sign.

Expo_pipe2: Prepares the operands for exponent bias addition.

Mant_pipe2: Normalizes the mantissa.

Sign_pipe3: Modifies the sign in case of zero result.

Expo_pipe3: Modifies the exponent in case of underflow or zero result. Also set the status flags.

Mant_pipe3: Set underflow conditions.

3.3.4 Floating Point Square Root

The basic structure of the FP square root provides scalability in terms of mantissa, exponent fields, and variable number of pipeline stages. Figure 3.13 shows the general organization of the FP square root. Exponent and mantissa widths are specified through parameters *ebit* and *mbit*, respectively. An example of FP Square Root with $ebit = 3$ and $mbit = 9$ is presented in Table 3.7. The number of pipeline stages is specified through one parameter ($pip1$). Descriptions of the main operators, components, and processes used by the FP Square Root, indicating their associated pipeline parameter are provided in the following sub-sections.

Operation	Operand		
	Sign	Exponent	Mantissa
Initial input	0	111	110011101
Appending mantissa's implicit one	0	111	1.110011101
Result sign(input sign should be +)	**0**	111	1.110011101
Left shift mantissa one place (if exponent is odd)	0	111	11.10011101**0**
Subtract one to the exponent (if it is odd)	0	11**0**	11.100111010
Right shift exponent (one place if it was odd)	0	**0**11	11.100111010
Mantissa Square Root	$\sqrt{11.100111010} = 01.1110$		
Mantissa already normalized in the range (2,1]	0	011	0**1**.111000000
Implicit one removal	0	011	**.111000000
Final result	0	011	111000000

Table 3.7: Floating Point Square Root example

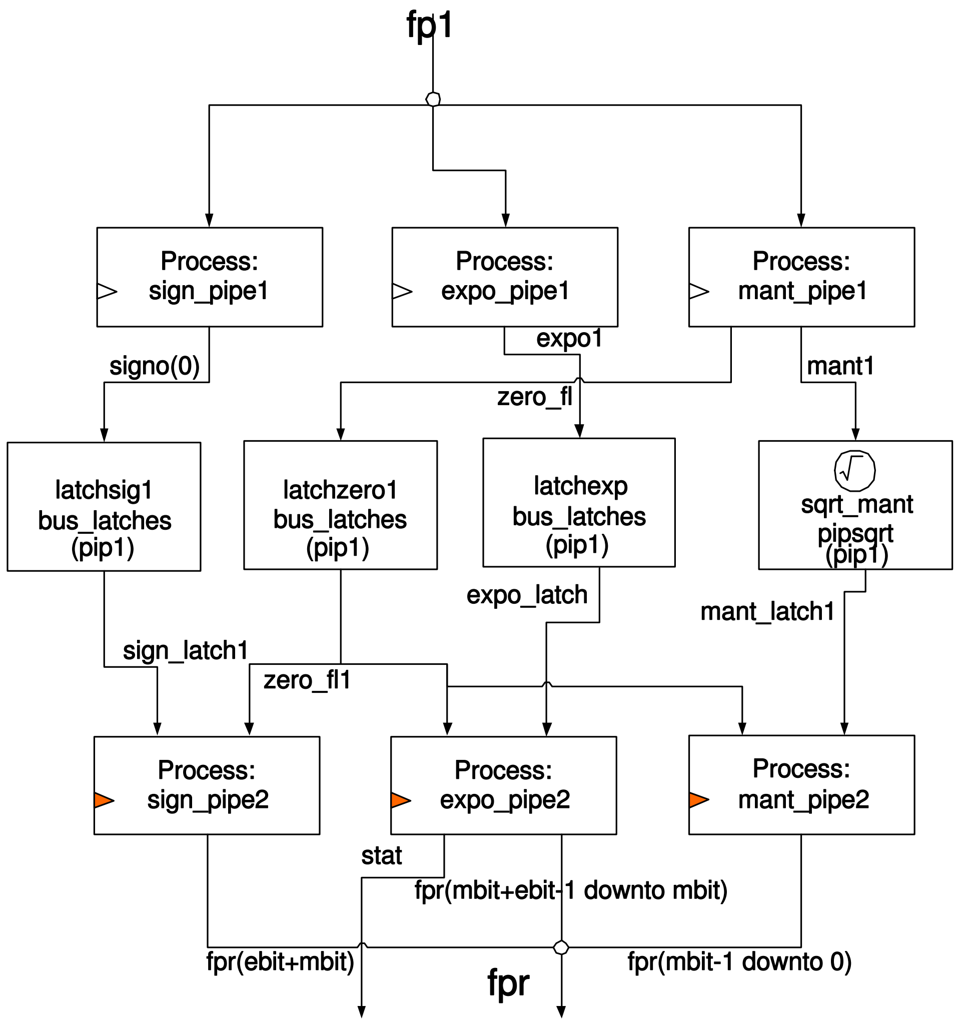

Figure 3.13: Base structure of pipelined FP Square Root

Operators:

Non Restoring Array Square Root: The structure of the non-restoring Array Square root is similar to the Nonrestoring Array Divider, as seen in Figure 3.14. It was selected for implementation for its suitability for pipeline insertion. This design uses a CAS as its basic building block. The amount of CAS is equal to $N = (n^2 + (n/2))$ where n is the size of the input operand. The value of n is restricted to positive and even integers. That is because each row of the array accepts two bits from the input operand. The input operand is *Numin* (n-bit) and the output operand *numout* has a bit size of $n/2$.

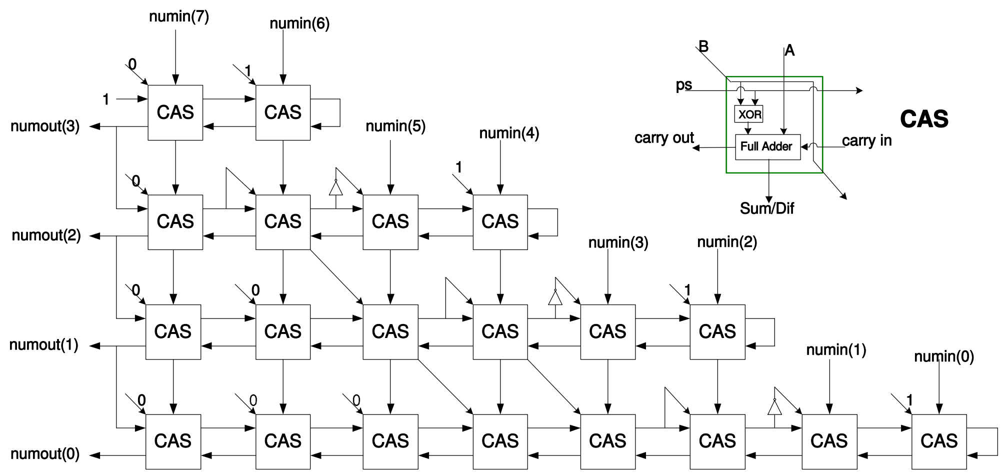

Figure 3.14: Nonrestoring Array Square Root [13]

The partial results of the integer square root are generated sequentially, just like division. The data flow in this architecture is sequential, i.e., when an input is applied, a valid output appears as the carries propagate along the entire structure of CAS. This carry propagation sets the critical delay path. Pipelining can be applied by inserting latches between rows. An example of a non-restoring square root operating is presented in Figure 3.15.

Adder: Uses the same structure as those in the FP Adder.

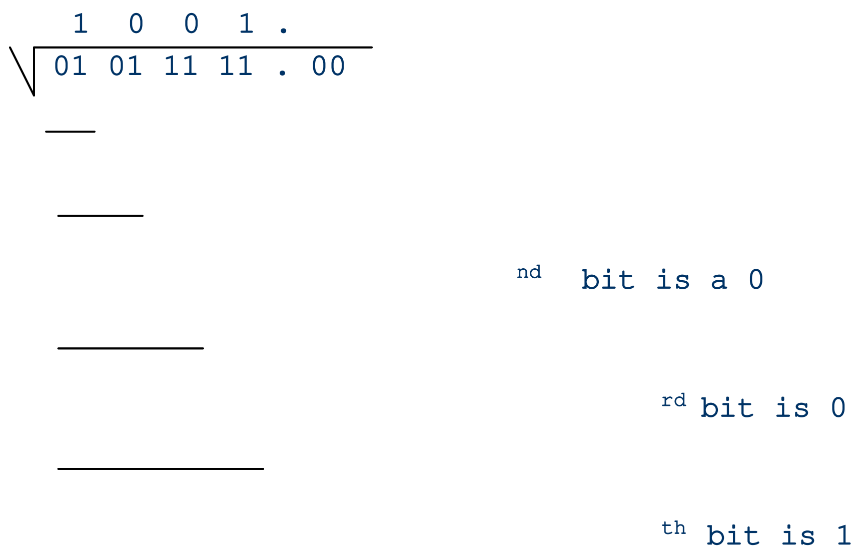

Figure 3.15: 8-bit NR Square Root Example

FP Square Root Components:

Sqrt_mant (*pip*1): This is a $[(mbit + 1) + mod((mbit + 1)/2)]$-bit pipelined array square root. Performs mantissa square root.

FP Square Root Processes:

Sign_pipe1: Pass the signs.

Expo_pipe1: Right shift the exponent by 1-bit position. This is the same as dividing the exponent by two.

Mant_pipe1: Zero detection and addition of the implicit hidden one to the mantissa. Left shift the mantissa by one place if the exponent is odd.

Sign_pipe2: Modifies the sign in case of zero result.

Expo_pipe2: Modifies the exponent in case of underflow or zero result. Also sets the status flags.

Mant_pipe2: Sets underflow conditions.

Chapter 4

Synthesis and Improvement Results

Developed units were tested using test data, looking for correct handling of exceptions and numerical results. This task was made by running testbenches using the simulation tools from ALDEC Active HDL. Test values were converted to FP format and added to the testbench. Testbench results were converted back to decimal format for numerical verification. Also architectural tests were made by comparing the architecture obtained after synthesis and the initial architectural design. Finally, the units were synthesized and implemented for various data sizes and pipeline depths. This allowed for a better analysis on the units scalability.

Some of the developed units were achieved with the aid of the VHDL synthesis tool. This tool supports operations like carry propagation and multiplication. This contributed to the creation of improved components for FP operators. Other improvements were obtained by custom designs and modifications to existing arithmetic architectures. Improved components were integrated to the FP operators, yielding better synthesis results than with the previously used structures. Area reduction was enabled through the modifications. Delay reductions were achieved in all the structures with the aid of pipeline insertion.

Operating frequency and slice consumption data are provided for each designed unit. Sections 4.1 and 4.2 offer the details for non-pipelined and pipelined structures, respectively. Section 4.3 provides these details for FP units, using pipelined components. A comparison between the developed units and commercial implementations is made is Section 4.4.

4.1 Non-Pipelined Integer Units

Non-pipelined operators offer a lower bound for the operating frequency of the designed units. This section presents each non pipelined integer operator with a brief comparison of other considered schemes. Justification is given for the selected schemes.

4.1.1 Comparators

The comparators were designed as simple combinational blocks where pipeline insertion was not considered a priority. Initially, the comparators were not identified as bottleneck in the FP units. However as the FP units performance grew up, the comparator's speed became relatively slow. The synthesis tool's comparator and the modular comparators were the best in terms of delay and slice usage as seen in Figure 4.1 and Figure 4.2, respectively. The selection between these approaches should be done considering the data size, maximum allowable area, and delay of the design. The synthesis tool's comparator was selected to be incorporated into the FP units because of overall advantages.

4.1.2 Adder

Two alternatives were evaluated: Ripple-connected Carry Look Ahead (RCLA) and the one provided by the synthesis tool. The RCLA adder is configured as a chain of a 4, 3, 2 or 1 bit carry-look-ahead adders. The purpose of four types of adders is to compose an adder with the desired size using the minimum amount of carry-look-ahead adders connected in carry-ripple way. In this way some advantages of the carry-look-ahead scheme were obtained while low amount of levels of logic is maintained.

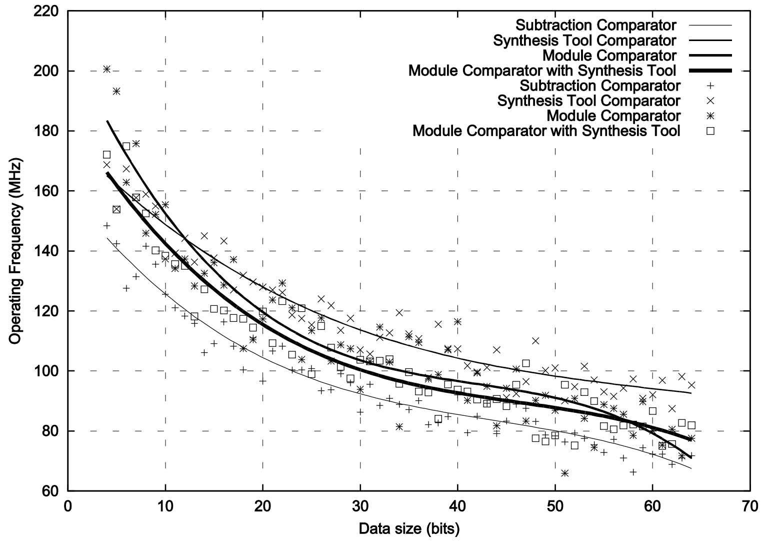

Figure 4.1: Operating frequency of comparator schemes

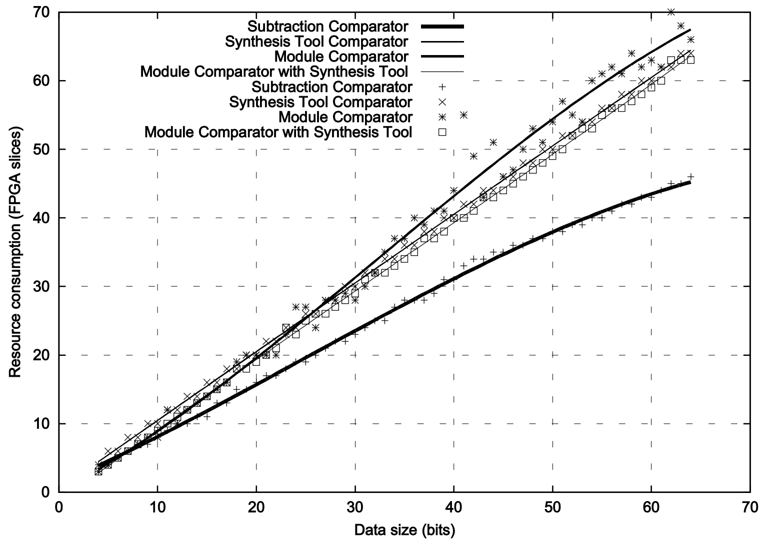

Figure 4.2: FPGA slices consumption of comparator schemes

The synthesis tool's fixed-point adder was selected instead of the RCLA. Figure 4.3 shows that this fixed-point adder has a better performance and uses less FPGA slices than the RCLA adder. This last observation can be obtained from Figure 4.4. These advantages of the synthesis tool's adder come from the support provided by the FPGA through the fast carry propagation logic. The synthesis tool's adder was a component for the pipelined adder.

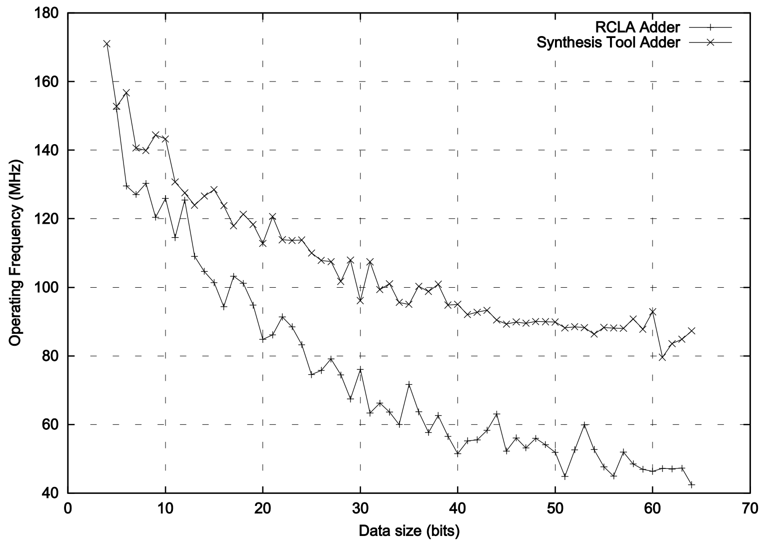

Figure 4.3: Operating frequency of two addition schemes

4.1.3 Multiplier

Two multiplier schemes were tested: array multiplier and the synthesis tool's multipliers. The synthesis tool's multiplier performance is better than the array multiplier's in terms of speed and resources, which is seen on Figure 4.5 and Figure 4.6. The synthesis tool multiplier's disadvantages are its inability to support variable pipelining and its dependence on the 18×18 hardware multipliers provided by the target FPGA. The synthesis tool's multipliers uses a variable amount of hardware multipliers depending on the operand sizes, as seen in Table 4.1. These were the reasons for choosing the array multiplier instead of the synthesis tool's multiplier.

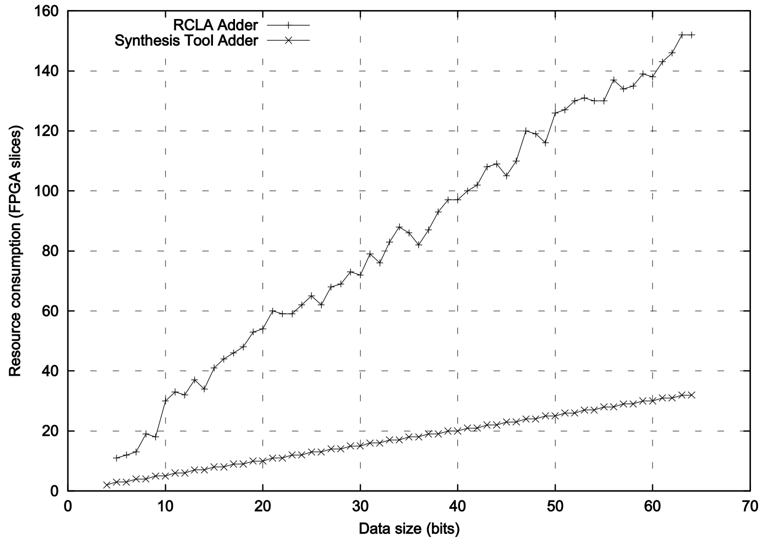

Figure 4.4: FPGA slices consumption of two addition schemes

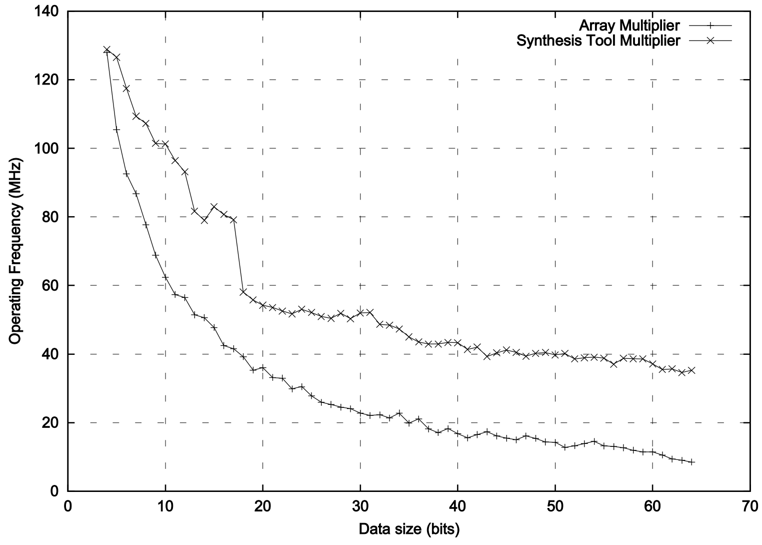

Figure 4.5: Operating frequency of two multiplication schemes

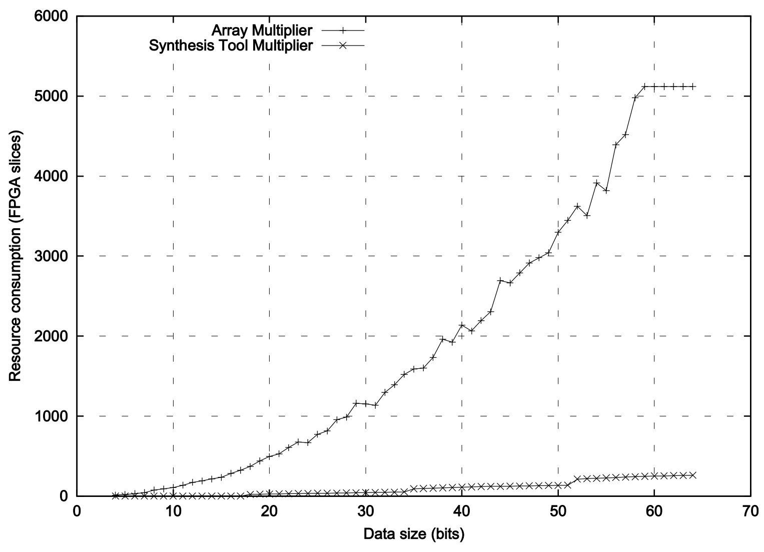

Figure 4.6: FPGA slices consumption of two multiplication schemes. Note that the synthesis tool multipliers use dedicated hardware multipliers as seen in Table 4.1

Data size range	18×18 Hardware multipliers	Slices
4 to 17 bits	1	0
18 to 34 bits	4	22 to 52
35 to 51 bits	9	92 to 137
52 to 64 bits	16	213 to 262

Table 4.1: Synthesis tool's multiplier usage of hardware multipliers. The target FPGA has 48 hardware multipliers

4.1.4 Shifter

This scheme allowed parallel shifting with operating frequency independent from the amount of shifting positions. Figure 4.7 shows its performance at different data sizes, which decreases because of the additional multiplexers and routing resources needed to increment it's data size. Its space complexity grows linearly with its data size as seen in Figure 4.8. Furthermore, this shifter is easily pipelined by inserting latches at multiplexer's outputs. This shifter replaced a previously used design which created each of the shifting possibilities, and used a multiplexer to select that corresponding to the desired number of positions. It was discarded because of its high space consumption.

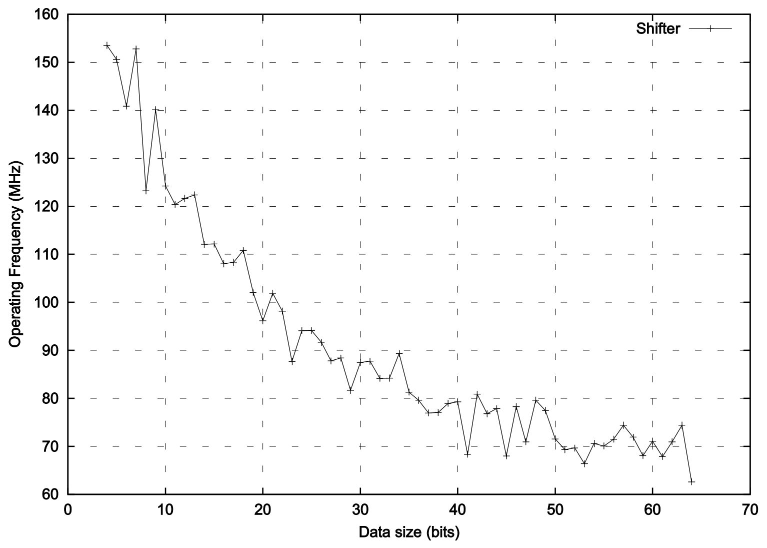

Figure 4.7: Operating frequency for the shifter

4.1.5 Normalizer and leading-zero detector

This new topology was designed as a scalable pipelined unit. This scheme was incorporated into the FP Adder, enabling operation over 150 MHz. Figures 4.9 and 4.10 present the operating frequencies and slice usage at different data sizes, which are affected by the additional multiplexers, multi-input NOR gates, and routing resources needed to increment it's data size.

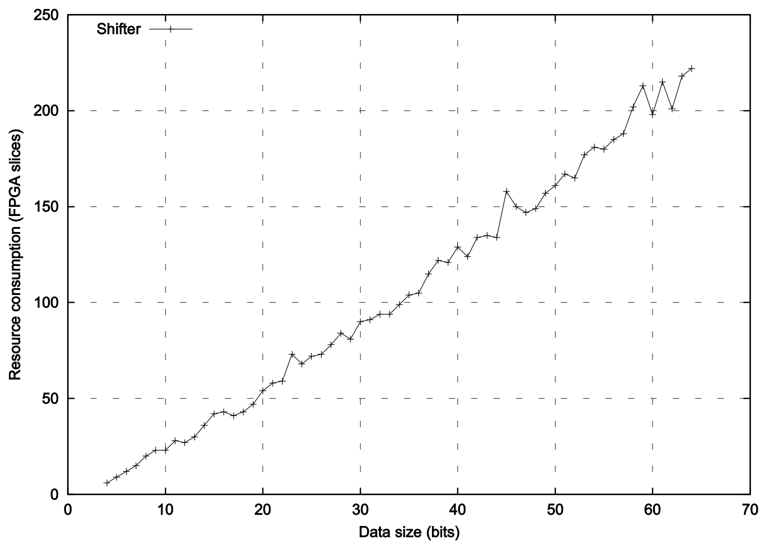

Figure 4.8: FPGA slices consumption of the shifter

The previously used scheme consisted of a for-loop, which tested each of the mantissa bits, and another for-loop for the mantissa normalization. A similar scheme was used by Athanas et Al. to locate the leading-one [30]. In this approach they created a 6-bit word to indicate in which of the six mantissa nibbles the leading-one resides, if any. Another stage built the shift value based on the 6-bit word and the identified nibble. Finally the shift value was used to adjust the FP exponent and perform mantissa normalization. This scheme caused a bottleneck in the FP Adder, limiting its operating frequency at 30-40 MHZ.

4.1.6 Divider

Improvements to the basic structure of the NR divider were made by changing the rows of basic cells by fast adders. Part of the cell functionality was moved outside the array in order to allow the use of fast adders. The architecture of the improved divider is shown in Figure 4.13. These modifications produced improved results in terms of operating frequency and slice consumption as seen in Figures 4.11 and 4.12, respectively. Additional performance improvement can be achieved

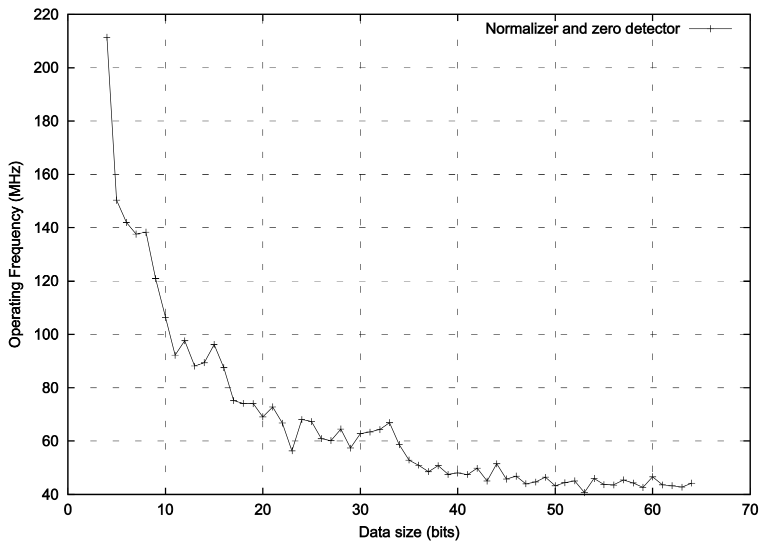

Figure 4.9: Operating frequency for the normalizing and zero detecting unit

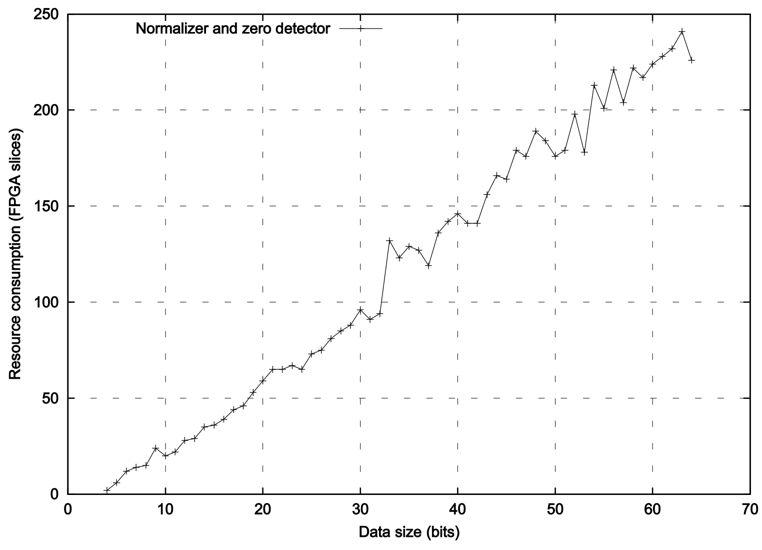

Figure 4.10: FPGA slices consumption for the normalizing and zero detecting unit

by the insertion of pipeline stages in each fast adder (just when all the rows are pipelined). Slice occupation increased almost linearly for the division operator. Slice consumption was half the consumption of the non-improved NR array divider.

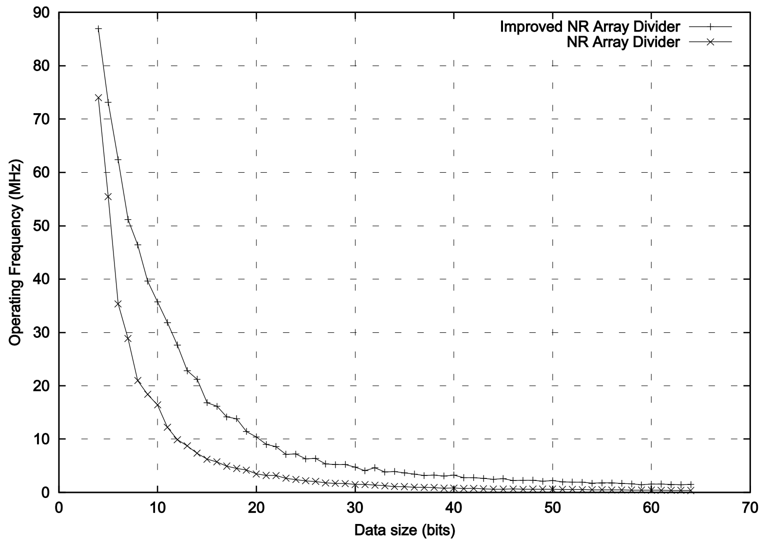

Figure 4.11: Operating frequency for the division units

4.1.7 Square root

The square root was subjected to the same optimizations made on the division unit. Additionally, the majority of its XORs receive inputs like the ones on Table 4.2 which allow for simplifications. The improved square root unit is presented in Figure 4.14. These modifications made the square root yield a better improvement percent than the divider's, as seen in Figures 4.15 and 4.16. Also slice occupation increased linearly for the modified square root operator.

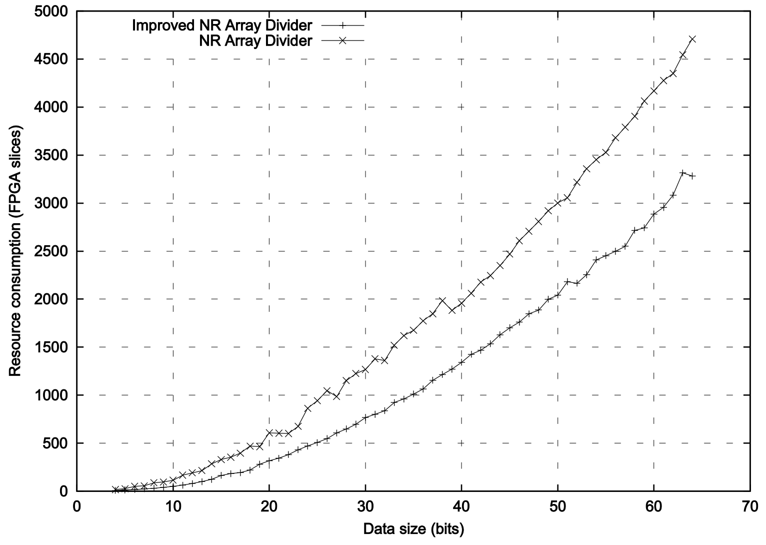

Figure 4.12: FPGA slices consumption for the division units

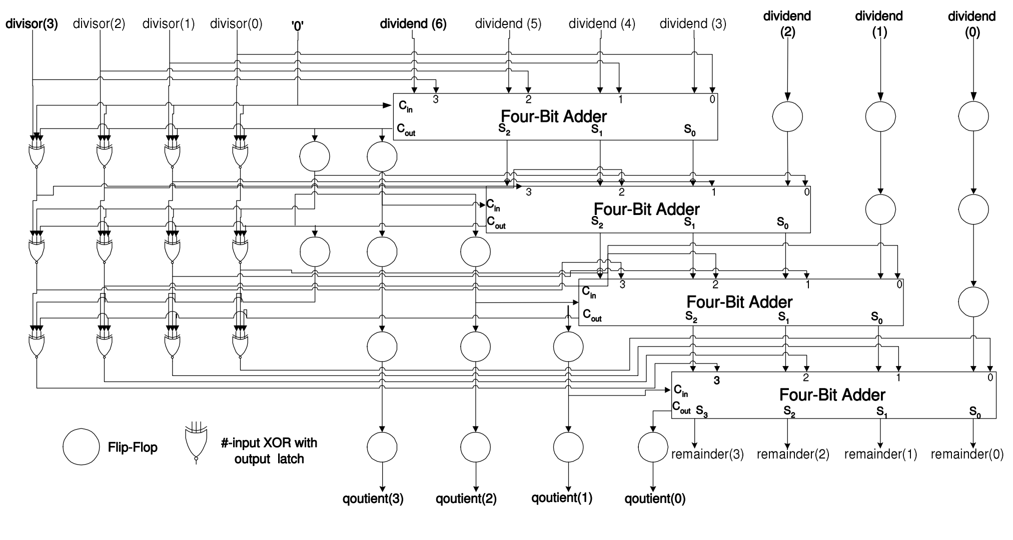

Figure 4.13: Improved Nonrestoring Array Divider (4-bit divisor, 7-bit dividend).

Input operand combination	Output
A = B	0
A $\neq$ B	1
A = 1	$\overline{B}$
A = 0	B

Table 4.2: Properties of 2-inputs (A and B) XOR gates

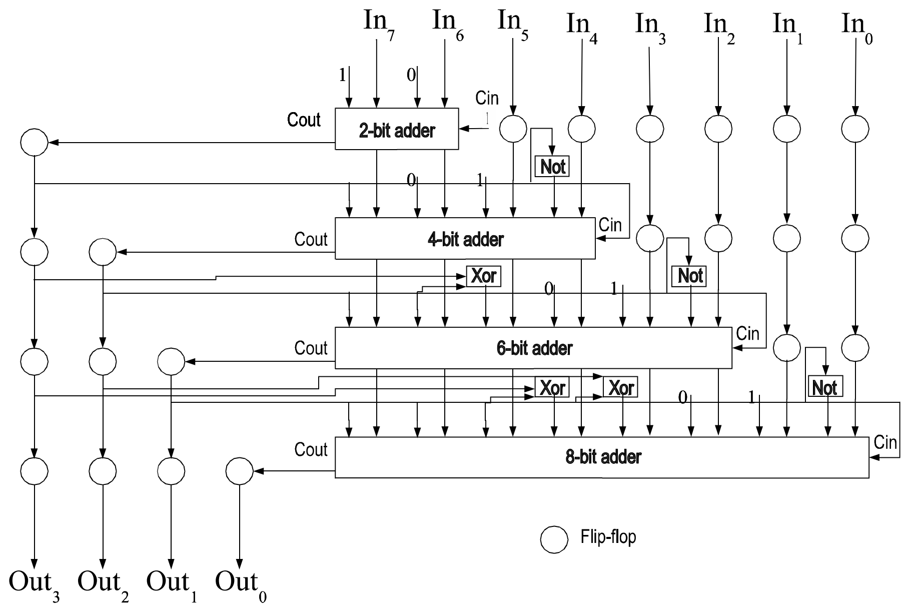

Figure 4.14: Improved Nonrestoring Array Square Root (8-bit input, 4-bit result)

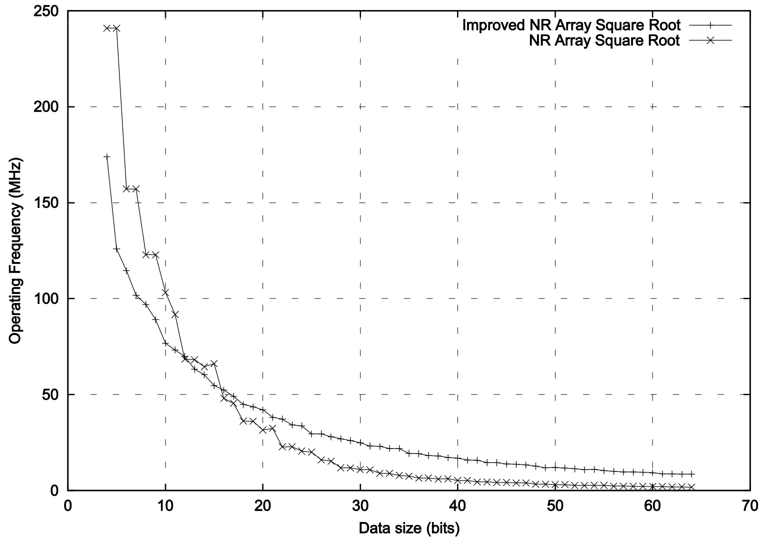

Figure 4.15: Operating frequency for the square root units

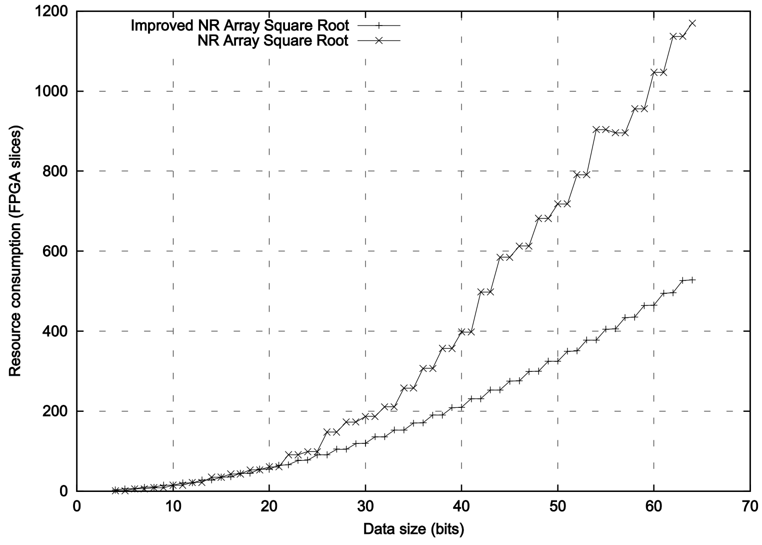

Figure 4.16: FPGA slices consumption for the square root units

4.2 Pipelined Integer Units

The behavior of the integer units provides information on how their FP counterpart will work. Information related to operating frequency and resource consumption is offered in Figure 4.17 and 4.18, respectively. These data were taken by implementing fixed-point units at data sizes required for a single precision FP unit.

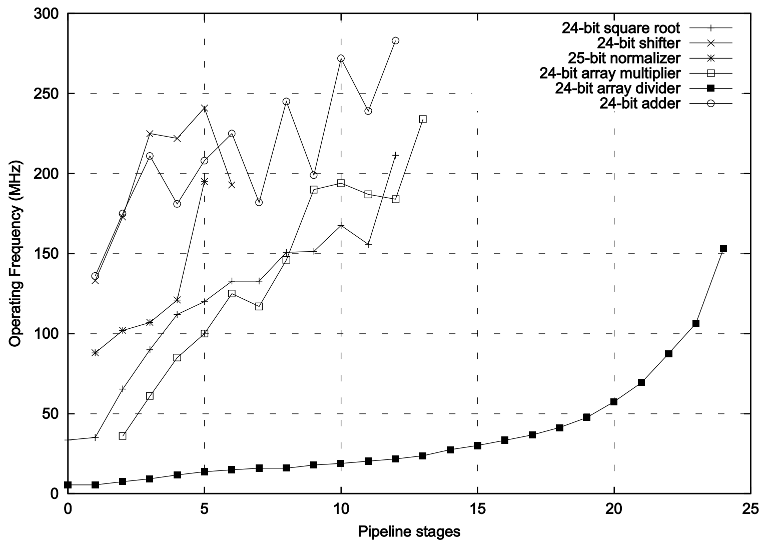

Figure 4.17: Fixed Point Units Operating Frequency Results

All the units increased their performance when the pipeline granularity was increased. Note that the square root goes up to 12 pipeline stages because every stage retires two bits. It achieves operating frequencies up to 211MHz. Operands can not exceed certain amount of pipeline stages depending on architectural limitations. The irregularities in the operating frequency plots are due to the variability in the interconnection delays in FPGAs, which sometimes counts for a 50% of the critical delay path.

Almost all the units overpass 200MHz of operating frequency. Also the array divider and array multipliers were the most resource consuming structures. Increase in slice usage is mainly due to

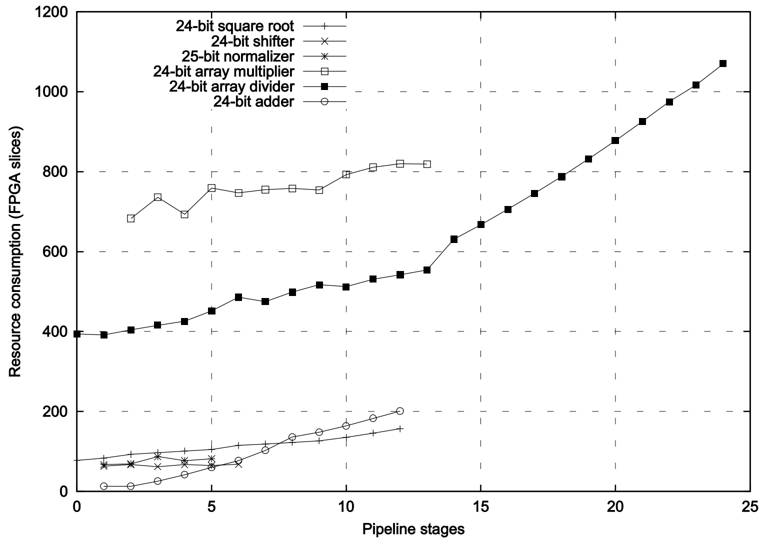

Figure 4.18: Fixed Point Units Slice Usage Results

the increased in latches usage when inserting pipeline stages.

4.3 Floating Point Units

The FP unit performance data presented in the following subsections were obtained by using the pipeline insertion method presented in Section 3.2. Pipeline techniques were successful in achieving higher operating frequencies.

4.3.1 FP Adder

The effect of varying the number of pipeline stages on the speed of the FP adder is illustrated in Figure 4.19. This graph shows that increasing the number of stages does effectively increase the operating frequency. Maximum operating frequency reached about 170MHz as seen in Figure 4.20. The slowest component in the FP adder was the normalizer. Each stage of the normalizer performs a comparison, in addition to the shifting made at each stage. Also increasing the number of pipeline stages increases the consumption of FPGA resources through the slice occupation.

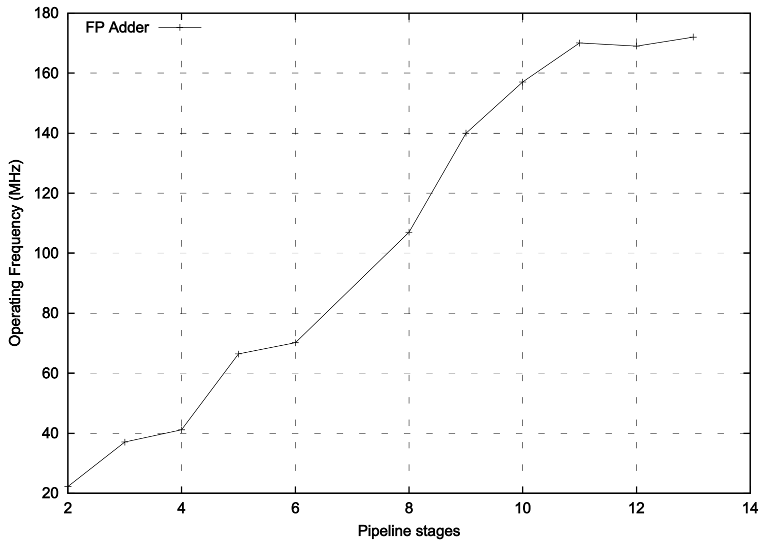

Figure 4.19: Pipelined FP Adder Operating Frequency

4.3.2 FP Multiplier

It reached a maximum operating frequency of 175 MHz, as seen en Figure 4.21. The array multiplier is the main contributor for the slice consumption behavior, as seen when comparing its slice consumption in Figure 4.22 with that of the array multiplier in Figure 4.18.

4.3.3 FP Divider

The FP divider operating frequency increases at a lower rate compared to the other units until, a high pipeline granularity value is reached. It reached a maximum of 155MHz, as seen in Figure 4.23. Additionally it presents an almost constant increase in the slice usage as seen in Figure 4.24.

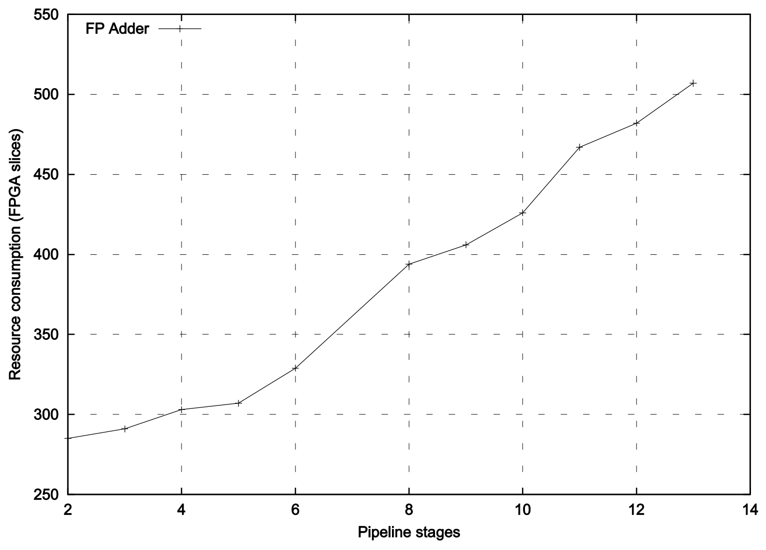

Figure 4.20: Pipelined FP Adder Slice Consumption

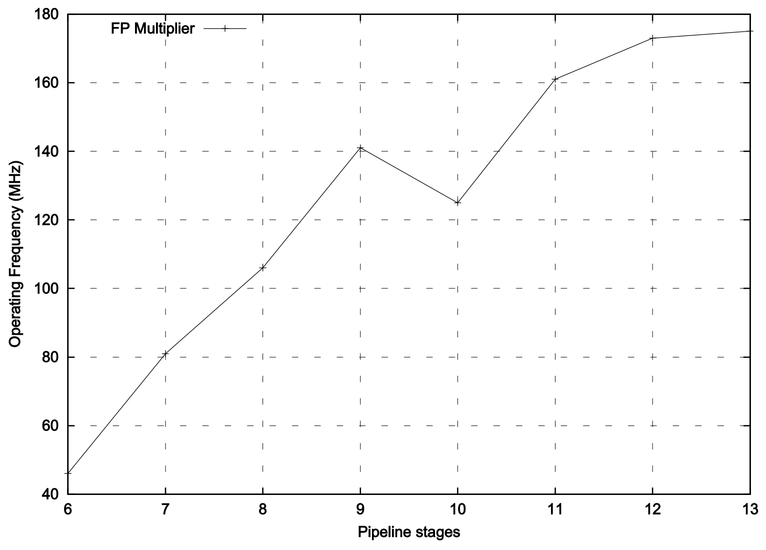

Figure 4.21: Pipelined FP Multiplier Operating Frequency

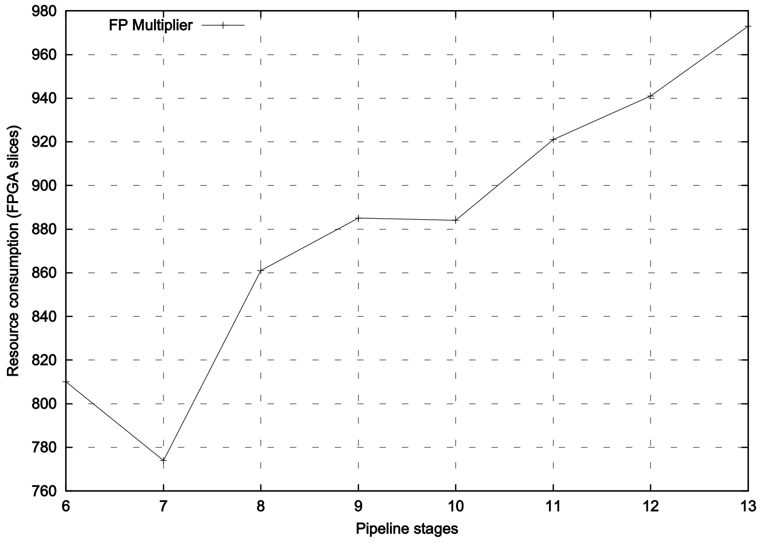

Figure 4.22: Pipelined FP Multiplier Slice Consumption

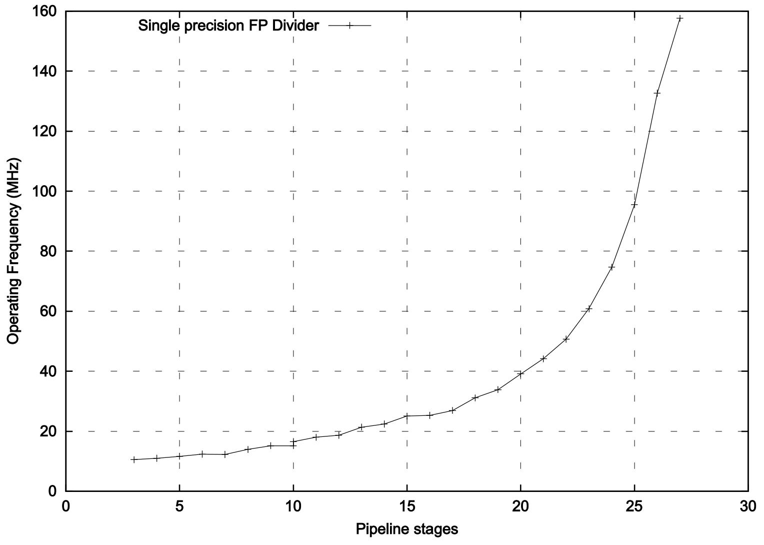

Figure 4.23: Pipelined FP Divider Operating Frequency

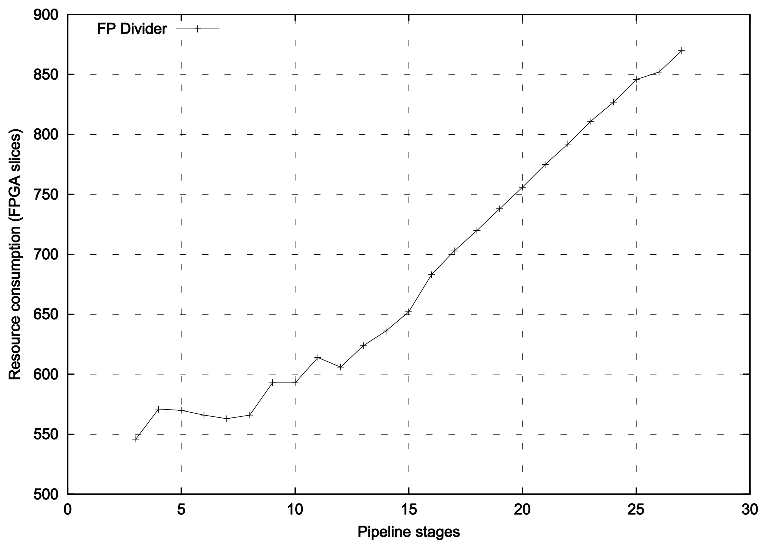

Figure 4.24: Pipelined FP Divider Slice Consumption

4.3.4 FP Square Root

This is the fastest FP unit, compared with the other developed units. Reached a maximum of 210MHz, which can be observed in Figure 4.25. Additionally it is the less resource consuming FP unit because of the improvements in the array square root, which is seen in Figure 4.26.

4.4 Results discussion

Implementations of single-precision, IEEE-754 compliant adder, multiplier, divider, and square root units were found to operate at 170MFLPOS, 175MFLOPS, 158MFLPOS, and 204MFLPOS, respectively. These speeds are competitive with those of highly refined, pre-routed core components commercially available from several vendors. In terms of area, it results difficult to establish meaningful comparisons since the reference implementations use dedicated Virtex-II resources other than slices, like hardware multipliers. Although a Virtex-II platform was chosen for reporting our results, the code is general enough to be easily ported to any other general FPGA platform. Our approach tries to avoid the usage of such special resources in order to keep the units portable to

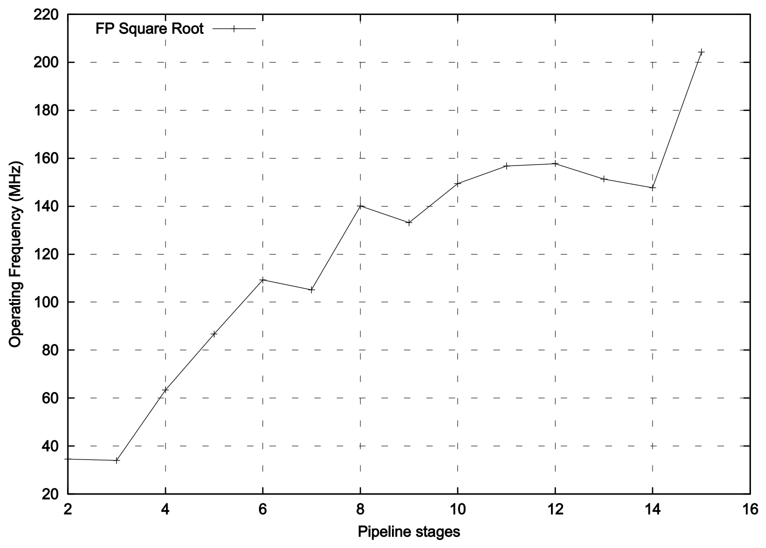

Figure 4.25: Pipelined FP Square Root Operating Frequency

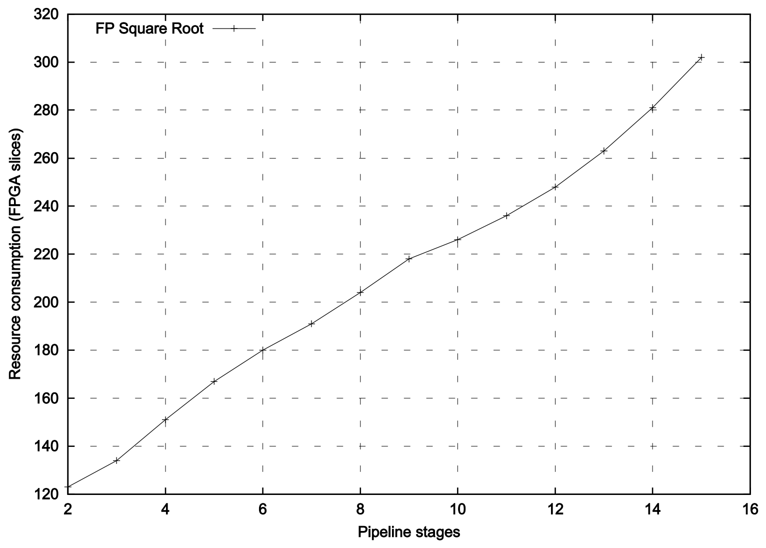

Figure 4.26: Pipelined FP Square Root Slice Consumption

other targets and to maintain the flexibility of adjustable range, precision, and pipeline granularity. Table 4.3 summarizes the obtained results along with typical speeds and resource utilization on the fastest commercial implementations found during the development of this work.

FP unit	Source	Freq.	Slices	Latency	mbit	ebit
Adder	Nallatech [15]	184	290	14	24	8
	Quixilica [23]	147	121	11	20	6
	Ours	170	467	11	24	8
Multiplier	Nallatech [15]	188	126	6	24	8
	Quixilica [23]	122	326	6	24	8
	Ours	175	973	13	24	8
Divider	Nallatech [15]	179	730	26	24	8
	Quixilica [23]	176	738	27	24	8
	Ours	158	870	24	24	8
Square Root	Nallatech [15]	181	330	27	24	8
	Quixilica [23]	222	675	27	24	8
	Ours	204	302	15	24	8

Table 4.3: FP units' comparation

The slowest components in the FP Adder are the normalizer and in the FP multiplier is the array multiplier. The slowest one in the FP Divider is the array divider, while in the FP square root the bottleneck is created by the array square root. These components have priority in the assignment of pipeline parameters in order to achieve a higher throughput. The throughput increase has a variable rate mainly due to the routing delay, which sometimes achieved values over 50% for the worst delay path, just like pipeline integer units in Figure 4.17. Also, it can be noticed that a unitary increase in a pipeline parameter does not necessarily increase the operating frequency of an operator. An example of this can be founf in Figure 4.25 for pipeline values of 7 and 9. This is seen in many of the frequency Vs Pipeline stages plots. Note also that FP operands work at a lower frequency than its components because of the extra logic needed for FP arithmetic. Aditionally, increasing the number of pipeline stages increases the consumption of FPGA resources. This increase is mainly due to the increased usage of latches.

Selecting the appropriate pipeline depth of an FP unit has a trade off. Higher operating frequencies are achieved by incrementing the pipeline granularity, but the FPGA slices usage increases too. Depending the target application, the user can choose high throughput or low resources consumption.

Chapter 5

Conclusion and Recommendations

A set of FP, scalable operators, adjustable to any custom data size (mantissa and exponent fields) and pipeline depth has been developed. Several fixed-point operators were built using this algorithm, achieving operating frequencies well above 200MHz. The developed integer structures were integrated to the FP Adder, Multiplier, Divider, and Square Root, obtaining scalable pipelined FP units.

Pipeline insertion was accomplished using a pipeline insertion algorithm. This algorithm works on regular structures such as adders, multipliers, and multi-stage operators. Frequency increase was achieved in all the units in which it was applied.

Operating frequency resulted competitive with commercially available implementations. Implementations of single-precision FP, IEEE-754 compliant adder, multiplier, divider and square root units were found to operate at 170MFLPOS, 175MFLOPS, 158MFLPOS and 204MFLOPS, respectively. Their advantage is the flexibility of scalable pipeline, mantissa, and exponent fields as well as portability to a wide range of FPGA targets. This flexibility is helpful for rapid prototyping. In the case of the divider, there is room for improvement in terms resources consumption.

Improvements were obtained in the integer adder, divider, and square root. These helped the development of more efficient FP units. New topologies for several integer units were developed, like the shifter and the normalizer.

A shifter with shifting time independent from the shifting positions was developed. Also, its resource consumption grows logarithmically with the data size. Also, a new topology for a mantissa normalizer was developed, which performs leading-zero detection and mantissa normalization in a single step without requiring an extra unit. Scaling the number of stages in the normalizer and shifter was found to have minimal effect on the overall area due to suitability of their structures to support pipelined operation.

Pipeline techniques have been used extensively for the development of high speed arithmetic hardware, with main application on rapid system prototyping.

Although pipeline insertion was not applied to the comparators, the modular comparator structure is suitable for it. Latches can be inserted at each module's output. This would allow the FP Adder to increase its performance, easing the bottleneck created by the comparator.

The FP divider and square root could also be further improved. This can be obtained by inserting pipeline stages in each row, after all the rows are pipelined. This has the potential of doubling the divider performance.

An important extension of this project would be the automation of the pipeline insertion process for FP units. In this process, the end-user would specify the desired operating frequency. The FP unit would be synthesized ans iteratively improved, letting the pipelining algorithm determine the necessary pipeline depth.

Bibliography

[1] J. Allan and W. Luk. Parameterised floating-point arithmetic on FPGAs. *IEEE InternationalConference on Acoustics, Speech, and Signal Processing, 2001. Proceedings.*, pages 897 – 900 vol.2, 2001.

[2] C. Asato, C. Ditzen, and S. Dholakia. A data-path multiplier with automatic insertion of pipeline stages. *IEEE Journal of Solid-State Circuits*, pages 383 – 387, April 1990.

[3] J. Bhasker. *A VHDL Primer.* Prentice Hall PTR, Upper Saddle River, New Jersey 07458, third edition edition, 1999.

[4] S. Chen and C. Li. An efficient division algorithm and its architecture. *IEEE TENCON'93*, pages 24–27, 1993.

[5] L. Dadda and V. Piuri. Pipelined adders. *IEEE Transactions on Computers*, pages 348 – 356, March 1996.

[6] A. Dawood, Z. Asdani, and B. Bravo. FIR filter design and implementation on reconfigurable computing technology. *Proceedings of the Fifth International Symposium on Signal Processing and Its Applications, ISSPA '99.*, pages 383 – 386 vol.1, Aug. 1999.

[7] J. Evans. Efficient FIR filter architectures suitable for FPGA implementation. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, pages 490 – 493, July 1994.

[8] R. Hashemian. Square rooting algorithms for integer and floating-point numbers. *IEEE Transactions on Computers*, pages 1025 – 1029, Aug. 1990.

[9] S. Hauck, L. Zhiyuan, and E. Schwabe. Configuration compression for the xilinx XC6200 FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems Volume: 18 Issue: 8 , Aug.*

1999, pages 1107 –1113, 1999.

[10] S. Heo. A low-power 32-bit datapath design. *S. Heo. A low-power 32-bit datapath design. Master's thesis, Massachusetts Institute of Technology, August*, pages 66–76, 2000.

[11] D. E. Inc. Floating point library IP core v1.1. *www.dilloneng.com*, 2001.

[12] M. Jiménez, N. Santiago, and D. Rover. Development of a scalable FPGA-based floating point multiplier. *In Proceedings of The FIfth Canadian Workshop on Field-Programable Devices Workshop (FPD'98)*, pages 145–150, 1998. citeseer.nj.nec.com/42441.html.

[13] I. Klotchkov and S. Pedersen. A codesign case study: Implementing arithmetic functions in FPGAs. *Symposium and Workshop on Engineering of Computer-Based Systems*, pages 389 – 394, 1996.

[14] W. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206 – 215, April 1998.

[15] N. Limited. *IEEE 754 Floating Point Core.* www.nallatech.com, 2001.

[16] L. Louca, T. Cook, and W. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. *Symposium on FPGAs for Custom Computing Machines*, pages 107 – 116, April 1996.

[17] M. Louie and M. Ercegovac. Mapping division algorithms to field programmable gate arrays. *Computer Science Deparment University of California, Los Angeles, CA*, pages 371–375, 1992.

[18] M. Louie and M. Ercegovac. On digit-recurrence division implementations for field programmable gate arrays. *11th Symposium on Computer Arithmetic*, pages 202 – 209, 1993.

[19] D. Narasimhan, D. Fernandes, V. Raj, J. Dorenbosch, M. Bowden, and V. Kapoor. A 100 MHz FPGA based floating point adder. *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, pages 3.1.1 – 3.1.4, 9-12 May 1993.

[20] S. Oberman and M. Flynn. Division algorithms and implementations. *IEEE TRANSACTIONS ON COMPUTERS, VOL. 46, NO. 8,*, pages 833–854, AUGUST 1997.

[21] I. Ortiz and M. Jimenez. Scalable pipeline insertion in floating point units for FPGA synthesis. *Proceedings of the IASTED International Conference on Circuit, Signals, and Systems*, pages 421–426, May 19-21 2003.

[22] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*, chapter 18, pages 300–303. Oxford, 2000.

[23] QinetiQ, www.quixilica.com. *Quixilica Floating Point Cores*, 2002.

[24] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155 – 162, April 1995.

[25] C. Souani, M. Abid, and R. Tourki. An FPGA implementation of the floating point addition. *Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society,IECON '98*, pages 1644 – 1648 vol.3, Sept. 1998.

[26] V. Tchoumatchenko, T. Vassileva, and P. Gurov. A FPGA based square-root coprocessor. *Proceedings of the 22nd EUROMICRO Conference EUROMICRO 96*, pages 520 – 525, Sept. 1996.

[27] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003*, pages 1–23, 2000.

[28] H. Verma. Field programmable gate arrays. *IEEE Potentials , Volume: 18 Issue: 4 ,Oct.-Nov. 1999*, pages 34 –36, 1999.

[29] R. Walke, R. Smith, and G. Lightbody. 20 GFLOPS QR processor on a xilinx virtex-e FPGA. pages 1–11, 2000.

[30] A. Walters and P. Athanas. A scaleable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 333 – 334, April 1998.

Appendix A

FP Units's Manual

These FP units were designed following several guidelines to assure the easiness on reuse. They offer high flexibility for rapid prototyping. Basic VHDL knowledge is needed to use these units. Some VHDL tools like Active HDL[1] allow for design entry using a GUI based environment. The design entry is similar to a schematic entry application in which the user graphically instantiates and interconnects the component. Here we describe the VHDL entry method.

The FP units can be used as stand alone units and/or as components for more complex designs. If the units are used alone, the user only has to specify the input/output pins for the FPGA. If the units are used as a components, the user should specify the unit's interface and set the generic parameters which control the mantissa and exponent width, and the pipeline granularity.

A.1 General Use Guide

All FP units accept normalized data according the IEEE-754 standard. Each FP unit has generic parameters controlling the mantissa (*mbit*) and exponent (*ebit*) width, and its pipeline granularity. Increasing the mantissa and exponent widths will also increase the numeric precision and dynamic range, respectively. Increasing any of these two parameters will also increase the resource consumption. Internal adjustments to meet the specified *mbit* and *ebit* are performed automatically.

[1]Active HDL is a trademark of ALDEC Inc.

Additional generic parameters are used to increase the pipeline granularity of the unit, and therefore its operating frequency. These parameters will also increase resource consumption. The number of pipeline parameters varies between one and five, depending on the FP unit. Pipeline parameters should be increased carefully because of speed-area trade offs. These parameters are modified near the beginning of the FP unit's VHDL code inside the entity declaration when used as a stand alone unit. When used as a component, these parameters are modified in the component instantiation section of the entity which uses the FP unit.

A.2 Parameter Specification for Floating Point Units

This section explains how to select the values for the pipeline parameters associated to each FP unit. It also specifies which internal components are affected by each pipeline parameter along with other usage hints. Most of the FP units have more than one generic parameter to control the pipeline granularity. For those units, the procedure specified in Figure 3.1 is recommended for the selection of the pipeline parameters to satisfy a desired speed specification.

A.2.1 Floating Point Adder

This FP operator uses fixed-point adders, a logarithmic shifter, and a normalizer (which also performs leading zero detection). The total amount of pipeline stages is given by the sum of five pipeline parameters plus two stages provided by two VHDL processes. These VHDL processes set a lower bound of two pipeline stages for the FP unit. The following list describes the function of the components affected by these pipeline parameters.

- *pip*1: Controls the number of pipeline stages of the exponent subtractor, which determines how many shiftings are needed for the mantissa.
- *pip*2*a*: Controls the number of pipeline stages of the shifter. It right shifts the smaller numbers mantissa to compensate for exponent equalization.
- *pip*2*b*: Pipeline stages of the mantissa adder.

- *pip3a*: Controls the number of pipeline stages of the normalizer and leading zero detector. Detects leading zeros of the mantissa result and normalizes it.
- *pip3b*: Controls the number of pipeline stages of the post-normalization exponent subtractor. Uses the amount of leading zeros detected to adjustment the exponent.

When using the FP adder as a stand alone unit, the pipeline parameters can be modified in the line which begins with the **generic** VHDL command. The following VHDL code correspond to the entity declaration section of this FP unit, where generic parameters can be modified.

```
entity fp_adder_pip is
   generic(ebit:Integer:=8; mbit:Integer:=23; pip1:Integer:=0 ; pip2a:Integer:=0 ;
           pip2b:Integer:=0 ; pip3a:Integer:=0 ; pip3b:Integer:=0);
   port(fp1,fp2 : in std_logic_vector(ebit+mbit downto 0);
        clk : in std_logic;
        stat :out std_logic_vector(1 downto 0);
        fpr : out std_logic_vector(ebit+mbit downto 0));
end fp_adder_pip;
```

When using the FP adder as a component, the pipeline parameters can be modified in the component instantiation section of the top level unit which uses the FP units as a component. The following VHDL code presents an example of an FP Adder instantiation. Generic parameters can be modified in the line which begins with the **generic map** VHDL command. This example assigns 8, 23, 1, 2, 3, 4, and 5 to *ebit*, *mbit*, *pip*1, *pip2a*, *pip2b*, *pip*3*a*, and pip3b, respectively.

```
fpadd: fp_adder_pip
generic map (8,23,1,2,3,4,5)
port map(to_fp1,to_fp2,to_clk,to_stat,to_fpr);
```

The following VHDL files are used by the FP adder (fp_adder_pip) as VHDL components and should be placed in the same VHDL design folder.

- **bus_latches.vhd :** Row of latches of variable width.
- **bus_mux.vhd :** Two-input multiplexer which accepts variable width input operand.
- **bus_mux_pip.vhd :** Two-input multiplexer which accept variable width input operand. Its output is latched.

- **ff_bus_reset.vhd :** Flip-flop with reset, also manages variable data width.
- **latch_chain.vhd :** Array of latched buses used for pipelined structures. Pipeline stages for each data-bit is variable.
- **latch_chain_down.vhd :**Array of latched buses used for pipelined structures. Pipeline stages for each data-bit is variable.
- **latch_chain_norm.vhd :**Array of latched buses used for pipelined structures. Pipeline stages for each data-bit is variable.
- **latch_chain_shift.vhd :**Array of latched buses used for pipelined structures. Pipeline stages for each data-bit is variable.
- **normalizer.vhd :** Detects leading zeros on it input operand and normalizes it.
- **shift_pip.vhd :** Right shift the input operand as specified on its other input.
- **sum_chain.vhd :** It is the pipelined fixed point adder without the synchronizing input and output array of latches.
- **sum_pipeline.vhd :** Pipelined fixed point adder with variable data width. Add additional latches to the output when pipeline capacityis exhausted.
- **sum_pipeline_arr.vhd :** Pipelined fixed point adder with variable data width.
- **sumherr.vhd :** Fixed point adder with variable data width.
- **sumherrpip.vhd :**Fixed point adder with variable data width with latched output.
- **sumres_pip.vhd :**Unit which performs addition or subtraction. Includes one pipeline stage.
- **utility.vhd :** Utility package used by some components to calculate design parameters. It must be compiled before the others components.

A.2.2 Floating Point Multiplier

This FP operator uses fixed-point adders, and an array multiplier. The total amount of pipeline stages is given by the sum of three pipeline parameters plus three stages provided by three VHDL processes. These VHDL processes set a lower bound of three pipeline stages for the FP unit. The following list describes the function of the components affected by these pipeline parameters.

- $pip1a$,$pip1b$: The parameter $pip1a$ controls the pipeline stages of the top half of the array multiplier while ($pip1b$) controls the lower half of the array multiplier. This array multiplier performs mantissa multiplication. The pipeline stages of the exponent addition are controlled by $pip1a + pi1b$.

- $pip2$: Determine the amount of pipeline stages of the exponent bias subtractor. This compensates for the exponent addition section.

When using the FP multiplier as a stand alone unit, the pipeline parameters can be modified in the line which begins with the **generic** VHDL command. The following VHDL code correspond to the entity declaration section of this FP unit, where generic parameters can be modified.

```
entity fp_mult is
   generic (ebit : integer := 8; mbit : integer :=23; pip1a: integer := 0;
            pip1b: integer := 0; pip2: integer := 0);
   port(fp1,fp2 : in std_logic_vector(ebit+mbit downto 0);
        clk  : in std_logic;
        stat : out std_logic_vector (1 downto 0);
        fpr : out std_logic_vector(ebit+mbit downto 0));
end fp_mult;
```

When using the FP multiplier as a component, the pipeline parameters can be modified in the component instantiation section of the top level unit which uses the FP units as a component. The following VHDL code presents an example of an FP multiplier instantiation. Generic parameters can be modified in the line which begins with the **generic map** VHDL command. This example assigns 8, 23, 1, 2, and 3 to *ebit*, *mbit*, $pip1a$, $pip1b$, and $pip2$, respectively.

```
fpadd: fp_mult
generic map (8,23,1,2,3)
port map(to_fp1,to_fp2,to_clk,to_stat,to_fpr);
```

The following VHDL files are used by the FP multiplier (fp_mult) as VHDL components and should be placed in the same VHDL design folder.

- **arr_cell.vhd :** Array multiplier cell
- **arr_cell_carrpip.vhd :**Array multiplier cell with pipelined carry output
- **arr_cell_pip.vhd :**Array multiplier cell with pipelined output

- **arraymult_pipe.vhd :** Pipelined array multiplier
- **arrmult_sum.vhd :** It is the adder at he final stage of the array multiplier
- **dff_arr.vhd :** Rectangular array of flip-flops
- **fa_latch.vhd :** Full adder cell with latched output
- **full_adder.vhd :** Full adder cell
- **pipe_latchout.vhd :** Aray of latches for the final stage of the array multiplier.

These other components are also used by the FP multiplier; they are defined in the last part of Section A.2.1

- **bus_latches.vhd**
- **ff_bus_reset.vhd**
- **latch_chain.vhd**
- **latch_chain_down.vhd**
- **sum_chain.vhd**
- **sum_pipeline.vhd**
- **sum_pipeline_arr.vhd**
- **sumherr.vhd**
- **sumherrpip.vhd**
- **utility.vhd**

A.2.3 Floating Point Divider

This FP operator uses fixed-point adders, and an array divider. The total amount of pipeline stages is given by the sum of two pipeline parameters plus three stages provided by three VHDL processes. These VHDL processes set a lower bound of three pipeline stages for the FP unit. The following list describes the function of the components affected by these pipeline parameters.

- Pip1 : Specifies the number of pipeline stages mantissa divider and exponent subtractor.
- Pip2 : Specifies the number of pipeline stages of the exponent bias adder. This compensates for the exponent subtraction section.

When using the FP divider as a stand alone unit, the pipeline parameters can be modified in the line which begins with the **generic** VHDL command. The following VHDL code correspond to the entity declaration section of this FP unit, where generic parameters can be modified.

```
entity fp_divisor_pip is
   generic(mbit: integer := 23; ebit: integer := 8; pip1: integer := 24;
           pip2: integer :=1);
   port(fp1,fp2 : in std_logic_vector(ebit+mbit downto 0);
        clk : in std_logic;
        fpr : out std_logic_vector(ebit+mbit downto 0);
        stat : out std_logic_vector(1 downto 0));
end fp_divisor_pip;
```

When using the FP divider as a component, the pipeline parameters can be modified in the component instantiation section of the top level unit which uses the FP units as a component. The following VHDL code presents an example of an FP divider instantiation. Generic parameters can be modified in the line which begins with the **generic map** VHDL command. This example assigns 8, 23, 1, and 2 *mbit*, *ebit*, *pip*1, *pip*2, respectively.

```
fpadd: fp_divisor_pip
generic map (8,23,1,2,3)
port map(to_fp1,to_fp2,to_clk,to_stat,to_fpr);
```

The following VHDL files are used by the FP divider (fp_divisor_pip) as VHDL components and should be placed in the same VHDL design folder.

- **array_divider.vhd :** Fixed point divider
- **sumherr_half_pip.vhd :** Fixed point adder with latched sum output
- **xorarray.vhd :** Arraqy of three input XORs
- **xorcell.vhd :** Three input XOR
- **xorcellpip.vhd :** Pipelined three input XOR

These VHDL files are also used by the FP multiplier and defined in the last part of Section A.2.1

- **bus_latches.vhd**

- **ff_bus_reset.vhd**
- **latch_chain.vhd**
- **latch_chain_down.vhd**
- **sum_chain.vhd**
- **latch_chain_shift.vhd**
- **sum_pipeline.vhd**
- **sum_pipeline_arr.vhd**
- **sumherr.vhd**
- **sumherrpip.vhd**
- **utility.vhd**

This component is also used by the FP multiplier and defined in the last part of Section A.2.2

- **pipe_latchout.vhd**

A.2.4 Floating Point Square Root

This FP operator uses fixed-point adders, and an array square root unit. The total amount of pipeline stages is given by the sum of one pipeline parameters plus two stage provided by two VHDL processes. These VHDL processes set a lower bound of two pipeline stages for the FP unit. The following list describes the function of the components affected by these pipeline parameters.

- Pip1 : Specifies the amount of pipeline stages of the mantissa square root unit.

When using the FP square root as a stand alone unit, the pipeline parameters can be modified in the line which begins with the **generic** VHDL command. The following VHDL code correspond to the entity declaration section of this FP unit, where generic parameters can be modified.

```
entity fp_sqrt_pip is
   generic(mbit: integer := 23; ebit: integer := 8; pip1: integer := 13);
   port(fp1 : in std_logic_vector(ebit+mbit downto 0);
        clk,reset : in std_logic;
        fpr : out std_logic_vector(ebit+mbit downto 0);
        stat : out std_logic_vector(1 downto 0));
end fp_sqrt_pip;
```

When using the FP square root as a component, the pipeline parameters can be modified in the component instantiation section of the top level unit which uses the FP units as a component. The following VHDL code presents an example of an FP square root instantiation. Generic parameters can be modified in the line which begins with the **generic map** VHDL command. This example assigns 8, 23, and 1 to *mbit*, *ebit*, and *pip*1, respectively.

```
fpadd: fp_divisor_pip
generic map (8,23,1,2,3)
port map(to_fp1,to_fp2,to_clk,to_stat,to_fpr);
```

The following VHDL files are used by the FP square root (fp_sqrt_pip) as VHDL components and should be placed in the same VHDL design folder.

- **latch_chain_sqr.vhd :** Array of latches for the pipelines fixed point square root.
- **latch_down_sqrt.vhd :** Array of latches for the pipelines fixed point square root.
- **pipsqrt.vhd :** Fixed point square root unit.

These other VHDL files are also used by the FP multiplier and defined in the last part of Section A.2.1

- **bus_latches.vhd**
- **ff_bus_reset.vhd**
- **latch_chain_down.vhd**
- **sumherr.vhd**
- **utility.vhd**

This component is also used by the FP square root and defined in the last part of Section A.2.3

- **sumherr_half_pip.vhd**

A.3 VHDL source codes

The VHDL codes for the developed FP units and their components are recorded in the attached CD. It has four folders. Each one of them contains the required VHDL code for each FP unit. In order to use an FP unit, it is only required to paste the folder's content into a VHDL project and to set the FP unit's VHDL code as the top level entity.