Floating Point DSP Algorithms Implementation Using Dedicated Multipliers on FPGAs

By

Yasir Munir

2007-NUST-MS PhD-ComE-10

Thesis Advisor

Dr. Shoab A. Khan

Submitted to the Department of Computer Engineering
in fulfillment of the requirements for the degree of
Masters of Science

In

Computer Engineering

College of Electrical & Mechanical Engineering,

National University of Sciences and Technology, Pakistan

2009

بِسْمِ اللهِ الرَّحْمٰنِ الرَّحِيْمِ

IN THE NAME OF ALLAH ALMIGHTY, THE MOST BENEFICIENT, THE MOST MERCIFUL, AND THE MOST COMPASSIONATE

DEDICATION

This work is dedicated to my parents whose prayers and encouragement made me do this and to all those people who went off their way to help me to bring out the best in me and to make me what I am today.

Thank you

ACKNOWLEDGEMENT

In the name of *ALLAH Almighty*, Most Gracious, Most Merciful who has blessed us with the physical and mental capabilities to complete the assigned project.

I would like to thank my respected Project Supervisor*, Dr. Shoab A. Khan*, and the advisory committee members, whose special interest, kind guidance, encouragement, flexibility, and kind behavior throughout the project helped me a lot in completing the given task.

My heartiest gratitude to my parents, brother and sister whose prayers made me pass through the ups and downs in the duration of the project. I would like to admit that I owe all my achievements in my truly, sincere and most loving parents, who mean to me, and whose prayers are a source of determination for me.

It would be incomplete without thanking some of my friends whose sincere help make me succeed. Specially, I am thankful to Mr. Aqeel Ahmad, Mr. Haris Masood, and Mr. Umar Ali who helped me making out of this successfully.

YASIR MUNIR

DECLARATION

I hereby declare and affirm that the thesis titled "**Floating Point DSP Algorithms Implementation Using dedicated Multipliers on FPGAs**" is neither whole nor as a part thereof has been copied out from any source (except data,). It is further declared that I developed this report entirely on the basis of my personal efforts made under the sincere guidance of my project supervisor and due to help of ALLAH Almighty.

If any part of this project proved to be copied or found be a part of some other, I shall stand by the consequences.

No portion of this work presented in this report has been submitted in support of any application for any degree and qualification of this or any other university or institute of learning. If found I will stand responsible.

YASIR MUNIR

TABLE OF CONTENTS

List of Figures

List of Tables

Abstract

The use of FPGAs for the implementation of DSP algorithms is increasing day by day due to improvement in their size and performance. FPGAs are generally good at fixed point arithmetic but they usually prove inefficient in terms of area and space when used for implementation of floating point algorithms. This is due to the fact that floating point algorithms are complex and it has always been hard to implement these on FPGAs, especially multiplication based algorithms. To cater this problem we have proposed a new technique for implementing floating point multiplier on FPGAs in this thesis. The new design methodology helps us to reduce the area utilization on FPGAs which is a major concern while implementing a floating point algorithm. The key point is to use dedicated multipliers available on FPGAs. These multipliers are fixed point multipliers and we have used them for the simple multiplication of mantissa parts of floating point numbers. So, instead of implementing the whole floating point multiplier on FPGA, the idea is to use the built in fixed point multiplier whenever a multiplication is required. All other issues are handled separately, like sign checking, normalization of floating point numbers, pre and post adjustment of exponents, shifting of mantissas by appropriate number of bits and rounding, flags etc. The multiplier is 32-bit and is in accordance with the IEEE-754 floating point standard. This technique helps in the reduction of area utilization while implementing floating point algorithms on FPGAs. As a proof of this argument we have also implemented a FIR filter using this multiplier and got satisfactory results regarding area utilization. In the end a comparison of my results with others has also been made.

Chapter 1

Introduction

Field Programmable Gate Array (FPGA) is a set of programmable logic cells that are interconnected by means of programmable switches. These logic cells are also called Configurable Logic Blocks (CLBs). Unlike an Application Specific Integrated Circuit (ASIC) which can perform a single specific function for the lifetime of the chip an FPGA can be reprogrammed to perform a different function in a matter of microseconds [1]. Before it is programmed an FPGA knows nothing about how to communicate with the devices surrounding it. This is both a blessing and a curse as it allows a great deal of flexibility in using the FPGA while greatly increasing the complexity of programming it [1]. With the passage of time the abilities and performance of FPGAs are increasing. More and more features are being added to them every year. In 1984 when Xilinx launched the industry's first FPGA, it was able to implement just a few thousands gates. Due to the technological advancements, FPGAs are now capable to implement more than a million gates. This has drawn the attention of Digital Signal Processing (DSP) design engineers and the use of FPGAs for DSP applications is increasing day by day. FPGAs are good candidate for DSP applications due to their architecture, flexibility and pipelining capabilities. FPGAs, with their newly acquired digital signal processing capabilities, are now expanding their roles to help offload computationally intensive digital signal processing functions from the processor [2].

Some applications require a high degree of accuracy like robotic arm control or motor control. This accuracy cannot be met with the use of fixed point architecture because of the fact that in fixed point architecture the radix point is always fixed. This places a limit on the precision and magnitude of the numbers being represented. So to

avoid the loss of precision, a floating point architecture is always needed. Very large and very small numbers can be represented in a floating point format.

On the other hand FPGAs are generally good at fixed point arithmetic but they generally prove inefficient when used for floating point algorithms. The floating point algorithms usually consume more chip area which is a limited resource in FPGAs. This problem becomes even more prominent when using standard 32-bit floating point numbers. In addition to this the floating point algorithms are usually harder to implement on FPGAs. This is due to the fact that the floating point algorithms are complex in nature, particularly multiplication-based operations. The major reason for this complexity is the need for normalization of the floating point numbers. This requires the shifting of mantissas and adjustment of exponents [3], [4].

To cater this problem we have proposed a new technique for implementing floating point multiplier on FPGAs in this paper. The new design methodology helps us to reduce the area utilization on FPGAs which is a major concern while implementing a floating point algorithm. The key point is to use dedicated multipliers available on FPGAs. These multipliers are fixed point multipliers and we have used them for the simple multiplication of mantissa parts of floating point numbers. So, instead of implementing the whole floating point multiplier on FPGA, the idea is to use the built in fixed point multiplier whenever a multiplication is required. All other issues are handled separately, like sign checking, normalization of floating point numbers, pre and post adjustment of exponents, shifting of mantissas by appropriate number of bits and rounding, flags etc. The multiplier is 32-bit and is in accordance with the IEEE-754 floating point standard. This technique helps in the reduction of area utilization while implementing floating point algorithms on FPGAs. As a proof of this argument we have also implemented a FIR filter using this multiplier and got satisfactory results regarding area utilization. In the end a comparison of my results with others has also been made.

As FPGA device densities increase, field programmable gate arrays (FPGAs) are increasingly being applied to DSP applications. This is due to the enormous speed up possible over conventional digital signal processors (for some applications) usually due to parallel computation on multiple data streams. For some applications, however, FPGAs suffer in performance and logic utilization relative to digital signal processors due to the use of general purpose logic for computation as opposed to hardwired computation units. This is particularly the case for multiplication and multiply-accumulate operations. [3]

Digital Signal Processing (DSP), thanks to explosive development of wired and wireless networks and multimedia, represents one of the most fascinating areas in electronics. The applications of DSP continue to expand, driven by trends such as the increased use of video and still images and the demand for increasingly reconfigurable systems such as Software Defined Radio (SDR). Many of these applications combine the need for significant DSP processing efficiency with cost sensitivity, creating demand for high-performance, low-cost DSP solutions. Traditionally, digital signal processing algorithms are being implemented using general purpose programmable DSP chips. Alternatively, for high-performance applications, special-purpose fixed function DSP chipsets and application specific integrated circuits (ASICs) are used. Typical DSP devices are based on the concept of RISC processors with an architecture that consists of fast array multipliers. In spite of using pipeline architecture, the speed of such implementation is limited by the speed of array multiplier.

Multiplications, followed by additions, subtractions or accumulations are the basis of most DSP applications. The number of multipliers embedded in DSP processor is generally in the range of one to four. The microprocessor will sequence data to pass it through the multipliers and other functions, storing intermediate results in memories or accumulators. Performance is increased primarily by increasing the clock speed used for multiplication. Typical clock speeds are between tens of MHz to 1GHz. Performance, as measured by millions of Multiply And Accumulate (MAC) operations per second, typically ranges from 10 to 4000. The technological advancements in Field

Programmable Gate Arrays (FPGAs) in the past decade have opened new paths for DSP design engineers. FPGAs, with their newly acquired digital signal processing capabilities, are now expanding their roles to help offload computationally intensive digital signal processing functions from the processor [2].

FPGAs are an array of programmable logic cells interconnected by a matrix of programmable connections. Each cell can implement a simple logic function defined by a designer's CAD tool. Typical programmable circuit has a large number (64 to over 300,000) of such cells that can be used to form complex digital circuits. The ability to manipulate the logic at the gate level means that designer can construct a custom processor to efficiently implement the desired function. FPGAs offer performance target not achievable by DSP processors. However, to achieve the high-performance, FPGA-based designs have come at a cost. Efficient utilization of possibilities provided by modern programmable devices requires knowledge of hardware specific design methods. Designing DSP system targeted for FPGA devices is very different than designing it for DSP processors. Most algorithms being in use were developed for software implementation. Such algorithms can be difficult to translate into hardware. Thus the efficiency of FPGA-based DSP is heavily dependent on experience of the designer and his ability to tailor the algorithm to efficient hardware implementation. Moreover CAD tools for FPGA based DSP design are immature [2].

However DSP-oriented FPGAs provide the ability to implement many functions in parallel on one chip. General-purpose routing, logic and memory resources are used to interconnect the functions, perform additional functions, sequence and, as necessary, store data. This provides possibility to increase the performance of digital system by exploitation of parallelism of implemented algorithms [2].

In this thesis, FPGA based implementation of floating point algorithms are discussed. A new design of floating point multiplier is proposed which uses the dedicated multipliers available on FPGAs As the example FIR implementation on FPGA using the multiplier is used.

Digital multipliers are needed in many system applications, including digital filters, correlators and other DSP applications. Multipliers often make ineffective use of a programmable part by consuming significant logic and routing resources. One solution is to use dedicated multiplier devices connected to FPGAs, however, this often results in performance degradation due to inter-chip communication delays.

Existing Dedicated Multiplier Architectures

Lee and Flynn [16] have constructed a carry built-in architecture to implement arithmetic operations through multi-ported look up tables. The carry architecture is built-in with a multi-port LUT with additional support for carry logic by the use of a bypass multiplexer. They have achieved 5 times greater throughput density for particular applications such as variable multiplier, FIR filter, Viterbi decoder and Jacobi Iteration method. Due to the additional logic necessary they have achieved this result at the cost of a 10% area penalty, however the use of LUTs instead of dedicated arithmetic logic imposes a performance penalty.

Haynes and Cheung [11] described a technique to implement a multiplier with the aid of new flexible array block (4 x 4 multiplier). An array of these blocks is capable of being configured to perform any 4m bits × 4n bits signed/unsigned binary multiplication. The blocks are designed to be embedded within a conventional FPGA structure to increase the functionality of the device by freeing valuable general reconfigurable resources, particularly when used in the area of image processing. In general, the lack of a regular CLB structure reduces the flexibility of the design. When these blocks are used within an FPGA structure then a special routing architecture is required in order to overcome this problem.

Similar work has been done by Nabeel et al in [5]. They have presented different ways of implementing floating point adders and multipliers and have investigated suitable combination of area and speed. They have used a non-standard

representation of floating point numbers. They have used 16-bit and 18-bit instead of standard 32-bit format. So this representation is application specific and may according to their need and may cause loss of accuracy if used for other applications that demand accuracy. In [6] Loucas et al have also explored FPGA implementations of addition and multiplication for IEEE single precision floating-point numbers. The implementations tradeoff area and speed for accuracy. The adder is a bit-parallel adder, and the multiplier is a digit-serial multiplier. The multiplier takes 12 clock cycles to produce the result. In [7] the author has presented an open source floating point multiplier. We have also used this multiplier for the implementation of FIR filter and found out that it consumes a lot of area on FPGA. We have also made a comparison of its results with our work.

Floating Point Arithmetic Support

Floating point multiplication is more difficult to implement in hardware due to the need to shift the mantissa by an appropriate number of bits and adjust the exponent. In hardware, a floating point multiplier generally consists of a fixed-point multiplier and other extra circuitry for sign checking, pre and post shifting of the exponents. Shifting can be performed with barrel or logarithmic shifters. In general, barrel shifters are built with a linear chain of multiplexers with input bits and control signals.

When implemented in conventional lookup-table (LUT) based FPGAs, these multiplexers are implemented using LUTs resulting in poor utilization of the device and slow performance. To avoid this problem, an additional programmable multiplexer is provided between the multiplier unit and carry circuit for efficient implementation of barrel shifters. This enhances the area-efficiency and performance of the proposed architecture for floating-point arithmetic operations. Further investigations are planned into architectural enhancements to better support floating point arithmetic in FPGAs [3].

Organization

The thesis is organized in the following sections to easily make the readers understand;

- **Chapter 2** elaborates the IEEE-754 Floating point architecture. It discusses the standard formats of floating point numbers in use today. A brief review is presented.

- **Chapter 3** gives an overview of the FPGAs, its architecture and usability. It also elaborates the need of FPGAs and how to program it.

- **Chapter 4** provides technical details for the XtremeDSP™ Digital Signal Processing (DSP) element, the DSP48 slice. The DSP48 slices facilitate higher levels of DSP integration than previously possible in FPGAs.

- **Chapter 5** completely describes the proposed design of floating point multiplier. It completely elaborates the design of the multiplier and its usage in floating point algorithms.

- **Chapter 6** shows the results of the implementation of FIR filter using the proposed multiplier and these results have also been compared with simple implementation of FIR filter without using this multiplier.

- **Chapter 7** concludes the thesis with some suggestion of Future Work to improve the multiplier design.

Chapter 2

IEEE-754 Floating Point Standard

This section presents a brief review of IEEE-754 floating point number representation [13]. There are several ways to represent real numbers on computers, and floating point format is one of them. Floating-point representation basically represents real numbers in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

On the other hand, floating point format employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

The IEEE-754 standard [13] defines four formats for floating-point numbers. The four formats are: single precision, double precision, single-extended precision, and double-extended precision. The most commonly used floating-point formats are single precision and double precision.

The Format

The single-precision, double-precision, and extended precision formats are available to represent floating-point numbers. IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is

composed of the fraction and an implicit leading digit. The exponent base is implicit and need not be stored.

Single-Precision Format

In the single-precision format, the most significant bit (MSB) is a sign bit, followed by 8 intermediate bits to represent an exponent, and 23 least significant bits (LSBs) to represent the mantissa. As a result, the total width of single-precision numbers is 32 bits. The bias for the single-precision format is 127. Refer to Figure 2–1.

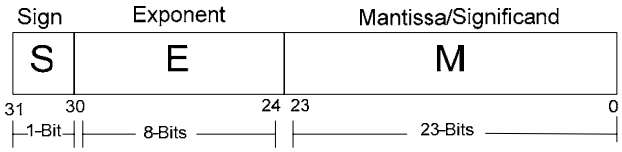

Figure 2–1: Single-Precision Representation

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. Exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers. For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa/Significand

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. To find out the value of the implicit leading bit, consider that any number can be expressed

in scientific notation in many different ways. For example, the number five can be represented as any of these:

5.00×10^{0}

0.05×10^{2}

5000×10^{-3}

In order to maximize the quantity of represent able numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×100.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

The Value

The formula for computing the value of binary floating point number is given below:

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

Double-Precision Format

In the double-precision format, the MSB is a sign bit, followed by 11 intermediate bits to represent an exponent, and 52 LSBs to represent the mantissa. As a result, the total width of double-precision numbers is 64 bits. The bias for double-precision format is 1023. Refer to Figure 2–2.

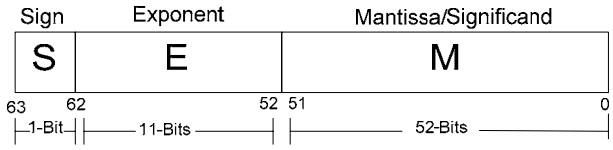

Figure 2–2: Double-Precision Representation

Extended Precision Format

In single-extended precision format, the MSB is a sign bit. However, there are no fixed widths for the exponent and mantissa fields. The exponent field has a minimum of 11 bits and its width must be less than the width of the mantissa field. The mantissa field has a minimum of 31 bits. The sum of the widths of the sign bit, the exponent field, and the mantissa field is a minimum of 43 bits and a maximum of 64 bits. The bias for the single-extended precision format is unspecified in the IEEE-754 standard.

Special Case Numbers

Table 2–1 shows the special case numbers defined by the IEEE-754 1985 Standard for Binary Arithmetic and their data bit representations.

Table 2-1: Representation of special case numbers in IEE-754 standard

Sign	Exponent	Mantissa	Representation
X	All 0's	All 0's	Zero
0	All 0's	Non-zero	Positive Denormalized
1	All 0's	Non-zero	Negative Denormalized
0	All 1's	All 0's	Positive Infinity
1	All 1's	All 0's	Negative Infinity
X	All 1's	Non-zero	Not a Number (NaN)

Rounding

In the IEEE-754 standard, there are four types of rounding modes: round-to-nearest-even, round-toward-zero, round-toward-positive-infinity, and round-toward-negative-infinity. The most commonly used rounding mode is round-to-nearest-even. This multiplier uses only the round-to-nearest-even mode. With the round-to-nearest-even mode, the result is rounded to the nearest floating-point

number. If the result is exactly halfway between two floating-point numbers, it is rounded so that the LSB becomes zero, which is even.

Algorithm for Floating-Point Calculation

Given two decimal inputs, A and B, the result R of the multiplication is as follows:

$$R = (M_a \times 2^{Ea}) \times (M_b \times 2^{Eb}) = (M_a \times M_b) \times 2^{Ea+Eb}$$

where:

- Ea is the exponent bit of A
- Eb is the exponent bit of B
- M_a is the mantissa bit of A
- M_b is the mantissa bit of B

Therefore, R has the following values:

- Sign = (sign bit of A) XOR (sign bit of B)
- Exponent = exponent of A + exponent of B – bias
- Mantissa = mantissa of A × mantissa of B

The exponent of A and the exponent of B are stored with bias adjustments in the IEEE-754 floating-point number. Therefore, when the two exponents are added together, the extra exponent must be removed.

The following calculations show how the addition of two bias numbers causes an extra bias:

- Exp_A_bias = Exp_A_actual + bias
- Exp_B_bias = Exp_B_actual + bias
- Exp_A_bias + Exp_B_bias

= Exp_A_actual + bias + Exp_B_actual + bias

= Exp_A_actual + Exp_B_actual + bias + bias

For an IEEE-754 standard floating-point number, only one bias adjustment is needed. Adding two bias numbers together causes an extra bias on the result of the calculation. The extra bias must be removed to obtain the correct result.

Common Applications

The advantage of floating-point numbers is that they can represent a much larger range of values. In a fixed-point number representation, the radix point is always at the same location. While the convention simplifies numeric operations and conserves memory, it places a limit on the magnitude and precision of the number representation. In situations that require a large range of numbers or high resolution, a relocatable radix point is desirable. Very large and very small numbers can be represented in a floating-point format. Multiplication of floating-point numbers is also commonly required in DSP-based applications.

Example

Here is an example of converting a decimal number into IEEE single precision floating point number. The number to be converted is 39887.5625

- Binary representation of it integral part is: $39887_{10} = 1001101111001111_2$
- Binary representation of its fractional part is: $0.5625 = 1001_2$
- So the binary representation of $39887.5625_{10} = 1001101111001111.1001_2$
- Now we have to normalize the number. Normalization involves the shifting of mantissa until the radix point is after the first non-zero bit. The exponent is adjusted accordingly. So after normalization we get:

$$1001101111001111.1001_2 = 1.00110111100111111001_2 \times 2^{15}$$

- Ignoring the leading 1 the mantissa is: 00110111100111110010000
- Adding bias (127) to the exponent we get: 15 + 127 = 142 = 10001110_2
- As the given number is positive so the sign bit is: 0
- So final representation of 39887.5625 in 32-bit floating point number is:

0	10001110	00110111100111110010000

Chapter 3

Introduction to FPGAs

A **field-programmable gate array** (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare). Any logical function can be implemented using FPGAs that an ASIC could perform. The ability of updating the functionality after shipping, and the low non-recurring engineering costs relative to an ASIC design (not withstanding the generally higher unit cost), proved beneficial for many applications.

FPGAs are consisted of programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—similar to a one-chip programmable breadboard. Complex combinational functions can be performed by configuring the logic blocks, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also contain memory elements, which may be simple flip-flops or more complete blocks of memory [7]. Figure 3-1 shows the general structure of FPGA.

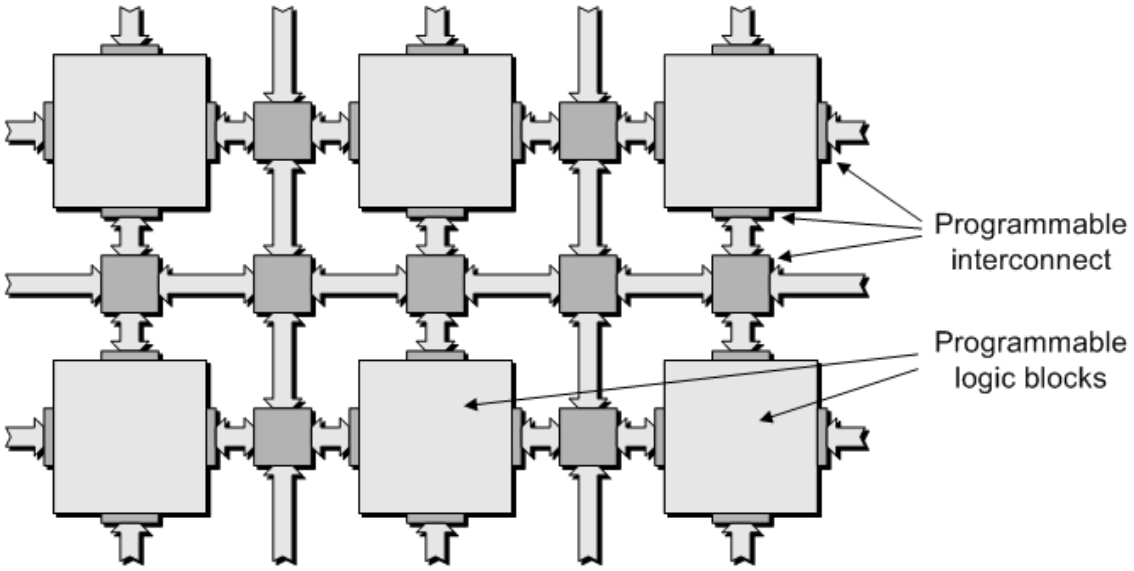

Figure 3-1: General structure of FPGA [8]

History

The FPGA industry sprouted from programmable read only memory (PROM) and programmable logic devices (PLDs). PROMs and PLDs both have an option of to be programmed in batches in a factory or in the field (field programmable); however programmable logic was hard-wired between logic gates.

Ross Freeman and Bernard Vonderschmitt, the Co-Founders of Xilinx, invented the first commercially viable field programmable gate array in 1985 – the XC2064. The XC2064 was consisted of programmable gates and programmable interconnects between gates, the beginnings of a new technology and market. The XC2064 boasted a mere 64 configurable logic blocks (CLBs), having two 3-input lookup tables (LUTs). Freeman was entered into the National Inventor's Hall of Fame for his invention after more than 20 years.

Few of the industry's foundational concepts and technologies for programmable logic arrays, gates, and logic blocks are founded in patents awarded to David W. Page and LuVerne R. Peterson in 1985.

Later in 1980s the Naval Surface Warfare Department funded an experiment proposed by Steve Casselman to develop a computer that would implement 600,000 reprogrammable gates. Casselman succeeded and his system was awarded a patent in 1992.

Xilinx continued unchallenged and rapidly growing from 1985 to the mid-1990s, when competitors sprouted up, eroding significant market-share. By 1993, Actel was serving nearly 18 percent of the market. The 1990s were the most important period of time for FPGAs, both in sophistication and the volume of production. FPGAs were primarily used in telecommunications and networking in early 1990s. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

FPGAs became very famous in 1997, when Adrian Thompson merged genetic algorithm technology and FPGAs to create a sound recognition device. Thomson's algorithm allowed the use of an array of 64 x 64 cells in a Xilinx FPGA chip to decide the configuration needed to accomplish a sound recognition task.

Modern Developments

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work reflects the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That was done in 1982. Examples of such hybrid technologies can be found in the Virtex-4 devices and Xilinx Virtex-II PRO, which include one or more PowerPC processors embedded within the FPGA's logic fabric. The Atmel FPSLIC is another device, which uses an AVR processor in combination with Atmel's programmable logic architecture.

An alternate approach of using hard-macro processors is to make use of "soft" processor cores that are implemented within the FPGA logic.

As mentioned earlier, many modern FPGAs have the ability to be reprogrammed at "run time," and this leads to the idea of reconfigurable computing or reconfigurable systems — CPUs that reconfigure themselves to suit the task at hand. The Mitrion Virtual Processor from Mitrionics is an example of a reconfigurable soft processor, implemented on FPGAs. However, instead of supporting the dynamic reconfiguration at runtime, it adapts itself to a specific program.

Additionally, new, non-FPGA architectures are beginning to emerge. Software-configurable microprocessors for example the Stretch S5000 adopt a hybrid approach by providing an array of processor cores and FPGA-like programmable cores on the same chip. Some of the major FPGA chip makers are Xilinx and Altera.

Gates

FPGA size has seen a tremendous growth in past years. In 1984, Xilinx launched the industry's first FPGA that could implement few thousands of gates. Today, an FPGA (Xilinx's Virtex) is capable of implementing one million gates [8].

- 1987: 9,000 gates, Xilinx
- 1992: 600,000, Naval Surface Warfare Department
- Early 2000s: Millions

The reasons for such a quantum leap are simple: [18]

- Technological improvements.
- Quick prototyping demands for complex digital systems targeting Application Specific Integrated Circuits (ASICs).

- Suitable for low volume, high-density and quick turn-around of complex digital systems.
- In-system reprogrammability.

FPGA Comparisons

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. A combination of fabrication improvements, volume, research and development, and the I/O capabilities of new supercomputers have largely closed the performance gap between ASICs and FPGAs.

A shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs are its advantages. Vendors can also develop their hardware on ordinary FPGAs, but manufacture their final version so it can no longer be modified after the design has been committed.

Xilinx claims that several market and technology dynamics are changing the ASIC/FPGA paradigm:

IC costs are rising aggressively [8]

- ASIC complexity has bolstered development time and costs
- R&D resources and headcount is decreasing
- Revenue losses for slow time-to-market are increasing
- Financial constraints in a poor economy are driving low-cost technologies

These trends make FPGAs a better alternative than ASICs for a growing number of higher-volume applications than they have been historically used for, to which the company attributes the growing number of FPGA design starts (see History).

Some FPGAs have the ability of partial re-configuration that lets one portion of the device can be re-programmed while other portions continue running.

Versus CPLDs

The primary differences between CPLDs and FPGAs are architectural. A CPLD has a bit restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with more predictable timing delays and a higher logic-to-interconnect ratio. On the other hand, The FPGA architectures are dominated by interconnects. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) and complex to design for.

Another significant difference between CPLDs and FPGAs is the presence in most FPGAs of higher-level embedded functions (such as adders and multipliers) and embedded memories, as well as to have logic blocks implements decoders or mathematical functions.

Security Considerations

With respect to security, FPGAs have both advantages and disadvantages as compared to ASICs or secure microprocessors. FPGAs' flexibility makes tremendous modifications during fabrication at a lower risk. The loaded design for many FPGAs is exposed while it is loaded (typically on every power-on). To deal with this issue, some FPGAs support bitstream encryption.

Applications

Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy and a growing range of other areas.

Originally, FPGAs began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their capabilities, size and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly, in the late 1990s, with the introduction of dedicated multipliers into FPGA architectures, applications, which had traditionally been the sole reserve of DSPs, began to incorporate FPGAs instead.

FPGAs especially find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. One such area is code breaking, in particular brute-force attack, of cryptographic algorithms.

FPGAs are rapidly used in conventional high performance computing applications where computational kernels such as FFT or Convolution are performed on the FPGA instead of a microprocessor.

The inherent parallelism of the logic resources on an FPGA allows a considerable computational throughput even at a low MHz clock rates. The flexibility of the FPGA allows higher performance by trading off precision and range in the number format for an increased number of parallel arithmetic units. This has given a new type of processing called reconfigurable computing, where time intensive tasks are offloaded from software to FPGAs.

The use of FPGAs in high performance computing is currently limited by the complexity of FPGA design compared to conventional software and the extremely long turn-around times of current design tools, where 4–8 hours wait is necessary after even minor changes to the source code.

FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip is more affordable than the development resources spent on creating an ASIC for a low-

volume application. The new cost and performance dynamics have broadened the range of viable applications today.

Architecture

The most common FPGA architecture consists of an array of configurable logic blocks (CLBs), I/O pads, and routing channels. All the routing channels have generally the same width (number of wires). Multiple Input/output pads may fit into the height of one row or the width of one column in the array.

An application circuit must be mapped into an FPGA with sufficient resources. The number of CLBs and I/Os required is easily determined from the design, while the number of routing tracks needed may vary considerably even among designs with the same amount of logic. (For instance, a crossbar switch requires much more routing than a systolic array with the same gate count.) Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of LUTs and IOs can be routed. This is determined by estimates like the ones derived from Rent's rule or by experiments with existing designs.

A classic FPGA logic block is consisted of a 4-input lookup table (LUT), and a flip-flop, as shown below. Now, the manufacturers have started moving to 6-input LUTs in their high performance parts, claiming increased performance.

There is only one output, which can either be the registered or the unregistered LUT output. The logic block contains four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, these and other signals are separately managed.

For the given example architecture, the locations of the FPGA logic block pins are shown below.

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channels to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels joined to it.

In the same way, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

The FPGA routing is unsegmented generally. Each wiring segment spans only one logic block before it terminates in a switch box. By switching on some of the programmable switches within a switch box, longer paths can be constructed. Some FPGA architectures use longer routing lines for higher speed interconnect that span multiple logic blocks.

Whenever a horizontal and a vertical channel intersect, there is a switch box. In such architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The topology, or pattern, of switches used in this architecture is the planar or domain-based switch box topology. In such a switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on.

Modern FPGA families are expanded upon the above capabilities to include higher level functionality fixed into the silicon. The silicon, having these common functions embedded into it, reduces the area required and gives those functions

increased speed compared to building them from primitives. Multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories are the examples of these.

FPGAs are also commonly used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, to reduce the time to market.

FPGA Design and Programming

To define the behavior of the FPGA, the user provides a hardware description language (HDL) or a schematic design. The HDL formed might be easier to work with when dealing with the large structures because it's possible to just specify them numerically rather than having to draw every piece by hand. On the other hand, schematic entry can allow for easier visualization of a design.

Using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA.

Going from schematic/HDL source files to actual configuration: The source files are fed to a software suite from the FPGA/CPLD vendor that through different steps will produce a file. This file is then transferred to the FPGA/CPLD via a serial interface (JTAG) or to an external memory device like an EEPROM.

The most common HDLs are Verilog and VHDL, although in an attempt to reduce the complexity of designing in HDLs, which have been compared to the

equivalent of assembly languages, there are moves to raise the abstraction level through the introduction of alternative languages.

For simplification of the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called *IP cores*, and are available from the third-party IP suppliers (rarely free and typically released under proprietary licenses) and FPGA vendors. Other predefined circuits are available from developer communities for example Open Cores (typically released under free and open source licenses such as the GPL, BSD or similar license), and other sources.

In a common design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. At the beginning, the RTL description in VHDL or Verilog is simulated by creating test benches to simulate the system and observe results. After the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is presented in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.

Basic Process Technology Types [8]

- **SRAM** - based on static memory technology. In-system programmable and re-programmable. Requires external boot devices. CMOS.
- **Antifuse** - One-time programmable. CMOS.
- **EPROM** - Erasable Programmable Read-Only Memory technology. Usually one-time programmable in production because of plastic packaging. Windowed devices can be erased with ultraviolet (UV) light. CMOS.
- **EEPROM** - Electrically Erasable Programmable Read-Only Memory technology. Can be erased, even in plastic packages. Some, but not all, EEPROM devices can be in-system programmed. CMOS.

- **Flash** - Flash-erase EPROM technology. It can be erased, even in plastic packages. Few flash devices can be in-system programmed. Generally, a flash cell is smaller than an equivalent EEPROM cell and is therefore less expensive to manufacture. CMOS.
- **Fuse** - One-time programmable. Bipolar.

Major Manufacturers

Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. They control over 80 percent of the market together, with Xilinx alone representing over 50 percent.

Xilinx also provides free Windows and Linux design software, while Altera provides free Windows tools; the Solaris and Linux tools are only available via a rental scheme.

Other competitors are Lattice Semiconductor (SRAM based with integrated configuration Flash, instant-on, low power, live reconfiguration), Actel (antifuse, flash-based, mixed-signal), SiliconBlue Technologies (low power), Achronix (RAM based, 1.5 GHz fabric speed), and QuickLogic (handheld focused CSSP, no general purpose FPGAs!).

Dedicated Multiplier Blocks

FPGA manufacturers have for years now been extending their chips' ability to implement digital signal processing efficiently, for example by introducing low-latency carry-chain-routing lines that speed-up addition and subtraction operations spanning multiple logic blocks. Such mechanism is relatively efficient when implementing addition and subtraction operations. However, it is not optimal in cost, performance, and power for multiplication and division functions. As a result, Altera (with Stratix), QuickLogic (with QuickDSP, now renamed Eclipse Plus) and Xilinx (with Virtex-II, Virtex-II Pro and DSP48 in Virtex-4) embedded in their chips dedicated multiplier

function blocks. Altera moved even further along the integration path, providing fully functional MAC blocks called the DSP blocks. This allows design methodologies known from DSP processors to be used [2].

Why FPGA for DSP Algorithms

Floating point operations are hard to implement on FPGAs because of the complexity of their algorithms. They usually require excessive chip area, a resource that is always limited in FPGAs. This problem becomes even harder if 32-bit floating point operations are required.

On contrary , many scientific problems require floating point arithmetic with high levels of accuracy in their calculations. Furthermore, many of these problems have a high degree of regularity that makes them good candidates for hardware accelerated implementations. Thus, the necessity for 32-bit floating point operators implemented in FPGAs arises. [6]

The architecture of FPGAs makes them suitable for dedicated functions like Digital Signal processing (DSP). Most of the DSP algorithms require multiplication and addition in real -time. The unit carrying out this function is called MAC (multiply accumulate). Three choices of technology exist for the implementation of DSP algorithms. These are [18]:

- **Programmable DSP chips** typically have only one MAC unit that can perform one MAC in less than a clock cycle. DSP processors or programmable DSP chips are flexible, but they might not be fast enough. The reason is that the DSP processor is general purpose and has architecture that constantly requires instructions to be fetched, decoded and executed.

- **ASICs** can have multiple dedicated MACs that perform DSP functions in parallel. But, they have high cost for low volume production and the inability to make design modifications after production makes them less attractive.

- **The FPGA** architecture allows multiple MACs and pipelining. Their ability to be modified easily makes them an ideal candidate for DSP functions. The only drawback is the speed, and this can easily be overridden by using computational algorithms suitable for FPGAs.

Chapter 4

DSP48

Introduction

This chapter provides technical details for the XtremeDSP™ Digital Signal Processing (DSP) element, the DSP48 slice. The DSP48 slices facilitate higher levels of DSP integration than previously possible in FPGAs. Many DSP algorithms are supported with minimal use of the general-purpose FPGA fabric, resulting in low power, high performance, and efficient device utilization [12].

At first look, the DSP48 slice is an 18 x 18 bit two's complement multiplier followed by a 48-bit sign-extended adder/subtracter/accumulator, a function that is widely used in digital signal processing (DSP).

A second look reveals many subtle features that enhance the usefulness, versatility, and speed of this arithmetic building block.Programmable pipelining of input operands, intermediate products, and accumulator outputs enhances throughput. The 48-bit internal bus allows for practically unlimited aggregation of DSP slices.

One of the most important features is the ability to cascade a result from one XtremeDSP Slice to the next without the use of general fabric routing. This path provides high-performance and low-power post addition for many DSP filter functions of any tap length. For multi-precision arithmetic this path supports a right-wire-shift. Thus, a partial product from one XtremeDSP Slice can be right-justified and added to the next partial product computed in an adjacent such slice.

Using this technique, the XtremeDSP Slices can be configured to support any size operands.

Another key feature for filter composition is the ability to cascade an input stream from slice to slice. The C input port allows the formation of many 3-input mathematical functions, such as 3-input addition and 2-input multiplication with a single addition.

Architecture

The Virtex-4 DSP slices are organized as vertical DSP columns. Within the DSP column, two vertical DSP slices are combined with extra logic and routing to form a DSP tile. The DSP tile is four CLBs tall.

Each DSP48 slice has a two-input multiplier followed by multiplexers and a three-input adder/subtracter. The multiplier accepts two 18-bit, two's complement operands producing a 36-bit, two's complement result. The result is sign extended to 48 bits and can optionally be fed to the adder/subtracter. The adder/subtracter accepts three 48-bit, two's complement operands, and produces a 48-bit two's complement result.Higher level DSP functions are supported by cascading individual DSP48 slices in a DSP48 column. One input (cascade B input bus) and the DSP48 slice output (cascade P output bus) provide the cascade capability. For example, a Finite Impulse Response (FIR) filter design can use the cascading input to arrange a series of input data sample and the cascading output to arrange a series of partial output results.

Architecture highlights of the DSP48 slices are [12]:

- 18-bit x 18-bit, two's-complement multiplier with a full-precision 36-bit result, sign extended to 48 bits.

- Three-input, flexible 48-bit adder/subtracter with optional registered accumulation feedback.

- Dynamic user-controlled operating modes to adapt DSP48 slice functions from clock cycle to clock cycle.

- Cascading 18-bit B bus, supporting input sample propagation.

- Cascading 48-bit P bus, supporting output propagation of partial results.

- Multi-precision multiplier and arithmetic support with 17-bit operand right shift to align wide multiplier partial products (parallel or sequential multiplication).

- Performance enhancing pipeline options for control and data signals are selectable by configuration bits.

- Input port C typically used for multiply-add operation, large three-operand addition, or flexible rounding mode.

- Separate reset and clock enable for control and data registers.

- I/O registers, ensuring maximum clock performance and highest possible sample rates with no area cost.

- OPMODE multiplexers.

Number of DSP48 Slices per Virtex-4 Device

Table 4-1 shows the number of DSP48 slices for each device in the Virtex-4 families. The Virtex-4 SX family offers the highest ratio of DSP48 slices to logic, making it ideal for math-intensive applications.

Table 4-1: Number of DSP48 Slices per Virtex-4 Family Member

Device ID	No. of DSP48s	No. of Columns
XC4VLX15	32	1
XC4VLX25	48	1
XC4VLX40	64	1
XC4VLX60	64	1
XC4VLX80	80	1
XC4VLX100	96	1
XC4VLX160	96	1
XC4VLX200	96	1
XC4VSX25	128	4
XC4VSX35	192	4
XC4VSX55	512	8
XC4VFX12	32	1
XC4VFX20	32	1
XC4VFX40	48	1
XC4VFX60	128	2
XC4VFX100	160	2
XC4VFX140	192	2

DSP Slice Primitive

Figure 4-1 shows the DSP48 Slice primitive. The details of each port and its brief functionality is given in Table 4-2.

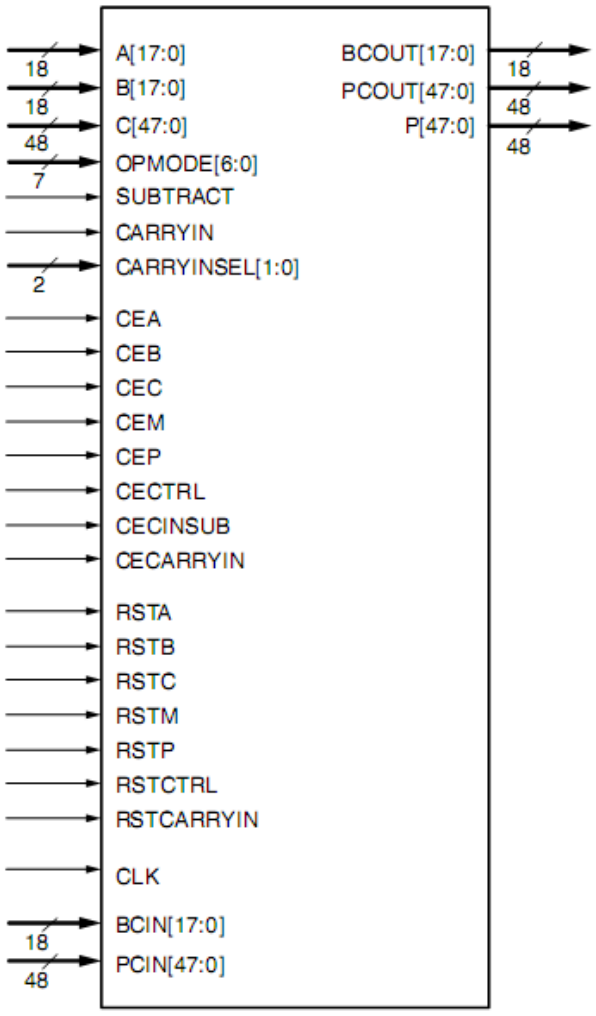

Figure 4-1: DSP48 Slice Primitive [12]

Table 4-2: The details of each port of DSP48 slice and its brief functionality

Signal Name	Direction	Size	Functionality
A	I	18	The multiplier's A input. This signal can also be used as the adder's Most Significant Word (MSW) input.
B	I	18	The multiplier's B input. This signal can also be used as the adder's Least Significant Word (LSW) input.
C	I	48	The adder's C input.
OPMODE	I	7	Controls the input to the X, Y, and Z multiplexers in the DSP48 slices
SUBTRACT	I	1	0 = add, 1 = subtract.
CARRYIN	I	1	The carry input to the carry select logic.
CARRYINSEL	I	2	Selects carry source.
CEA	I	1	Clock enable: 0 = hold, 1 = enable AREG.
CEB	I	1	Clock enable: 0 = hold, 1 = enable BREG.
CEC	I	1	Clock enable: 0 = hold, 1 = enable CREG.
CEM	I	1	Clock enable: 0 = hold, 1 = enable MREG.
CEP	I	1	Clock enable: 0 = hold, 1 = enable PREG.
CECTRL	I	1	Clock enable: 0 = hold, 1 = enable OPMODEREG, CARRYINSELREG.
CECINSUB	I	1	Clock enable: 0 = hold, 1 = enable SUBTRACTREG and general interconnect carry input.
CECARRYIN	I	1	Clock enable: 0 = hold, 1 = enable (carry input from internal paths).
RSTA	I	1	Reset: 0 = no reset, 1 = reset AREG.
RSTB	I	1	Reset: 0 = no reset, 1 = reset BREG.
RSTC	I	1	Reset: 0 = no reset, 1 = reset CREG.
RSTM	I	1	Reset: 0 = no reset, 1 = reset MREG.
RSTP	I	1	Reset: 0 = no reset, 1 = reset PREG.
RSTCTRL	I	1	Reset: 0 = no reset, 1 = reset SUBTRACTREG, OPMODEREG, CARRYINSELREG.
RSTCARRYIN	I	1	Reset: 0 = no reset, 1 = reset (carry input from general interconnect and internal paths).
CLK	I	1	The DSP48 clock.
BCIN	I	18	The multiplier's cascaded B input. This signal can also be used as the adder's LSW input.
PCIN	I	48	Cascaded adder's Z input from the previous DSP slice.
BCOUT	O	18	The B cascade output.
PCOUT	O	48	The P cascade output.
P	O	48	The Product output.

DSP48 Slice Attributes

The synthesis attributes for the DSP48 slice are described in detail throughout this section. With the exception of the B_INPUT and LEGACY_MODE attributes, all other attributes call out pipeline registers in the control and datapaths. The value of the attribute sets the number of pipeline registers[12].

The attribute settings are as follows:

- The AREG and BREG attributes can take a value of 0, 1, or 2. The values define the number of pipeline registers in the A and B input paths.

- The CREG, MREG, and PREG attributes can take a value of 0 or 1. The value defines the number of pipeline registers at the output of the multiplier (MREG) and at the output of the adder (PREG) (shown in Figure 4-9). The CREG attribute is used to select the pipeline register at the C input (shown in Figure 4-8).

- The CARRYINREG, CARRYINSELREG, OPMODEREG, and SUBTRACTREG attributes take a value of 0 if no pipelining register is on these paths, and they take a value of 1 if there is one pipeline register in their path. The CARRYINSELREG, OPMODEREG, and SUBTRACTREG paths are shown in Figure 4-10.

- The B_INPUT attribute defines whether the input to the B port is routed from the parallel input (attribute: DIRECT) or the cascaded input from the previous slice (attribute: CASCADE).

- The LEGACY_MODE attribute serves two purposes. The first purpose is similar in nature to the MREG attribute. It defines whether or not the multiplier is flow through in nature (i.e., LEGACY_MODE value equal to

MULT18x18) or contains a single pipeline register in the middle of the multiplier (i.e., LEGACY_MODE value equal to MULT18x18S is the same as MREG value equal to one.) While this is redundant to the MREG attribute, it was deemed useful for customers used to the Virtex-II and Virtex-II Pro multipliers because the DSP48 setup and hold timing most closely matches those of the Virtex-II and Virtex-II Pro MULT18x18S when the MREG is used. Any disagreement between the MREG attribute and LEGACY_MODE attribute settings are flagged as a software Design Rule Check (DRC) error. The second purpose for the attribute is to convey to the timing tools whether the A and B port through the combinatorial multiplier path (slower timing) or faster X multiplexer bypass path for A:B should be used in the timing calculations. Because the OPMODE can change dynamically, the timing tools cannot determine this without an attribute.

To summarize the timing tools behavior:

- If (attribute: NONE), then timing analysis/simulation bypasses the multiplier for the highest performance. The lowest power dissipation is achieved by setting MREG to one while CEM input is grounded.
- If (attribute: MULT18x18), then timing analysis/simulation uses the combinatorial path through the multiplier. In this case, MREG must be set to zero or a DRC error occurs.
- If (attribute: MULT18x18S), then timing analysis/simulation uses a pipelined multiplier. In this case MREG must be set to one or a DRC error occurs.

DSP48 Tile and Interconnect

Two DSP48 slices, a shared 48-bit C bus, and dedicated interconnect form a DSP48 tile. The DSP48 tiles stack vertically in a DSP48 column. The height of a DSP48 tile is the same as four CLBs and also matches the height of one block RAM. This “regularity” enhances the routing of wide datapaths. Smaller Virtex-4 family

members have one DSP48 column, while the larger Virtex-4 family members have two, four, or eight DSP48 columns.

As shown in Figure 4-2, the multipliers and block RAM share interconnect resources in the Virtex-II and Virtex-II Pro architectures. Virtex-4 devices, however, have independent routing for the DSP48 tiles and block RAM, effectively doubling the available data bandwidth between the elements.

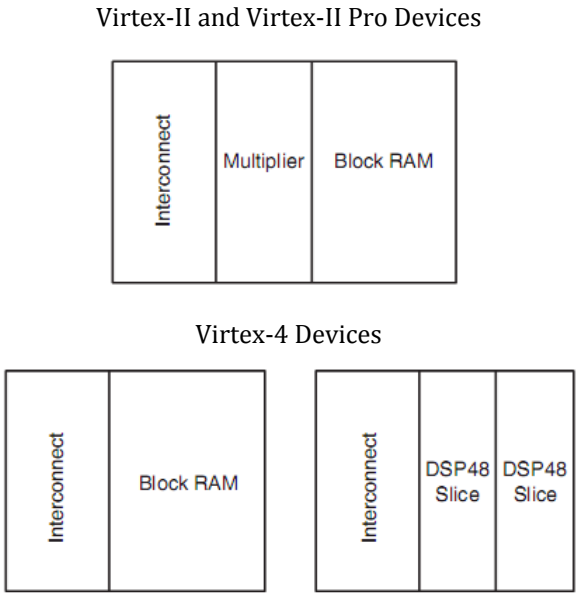

Figure 4-2: DSP48 Interconnect and Relative Dedicated Element Sizes

Figure 4-3 shows two DSP48 slices and their associated datapaths stacked vertically in a DSP48 column. The inputs to the shaded multiplexers are selected by configuration control signals. These attributes are set in the HDL source code or by the User Constraint File (UCF).

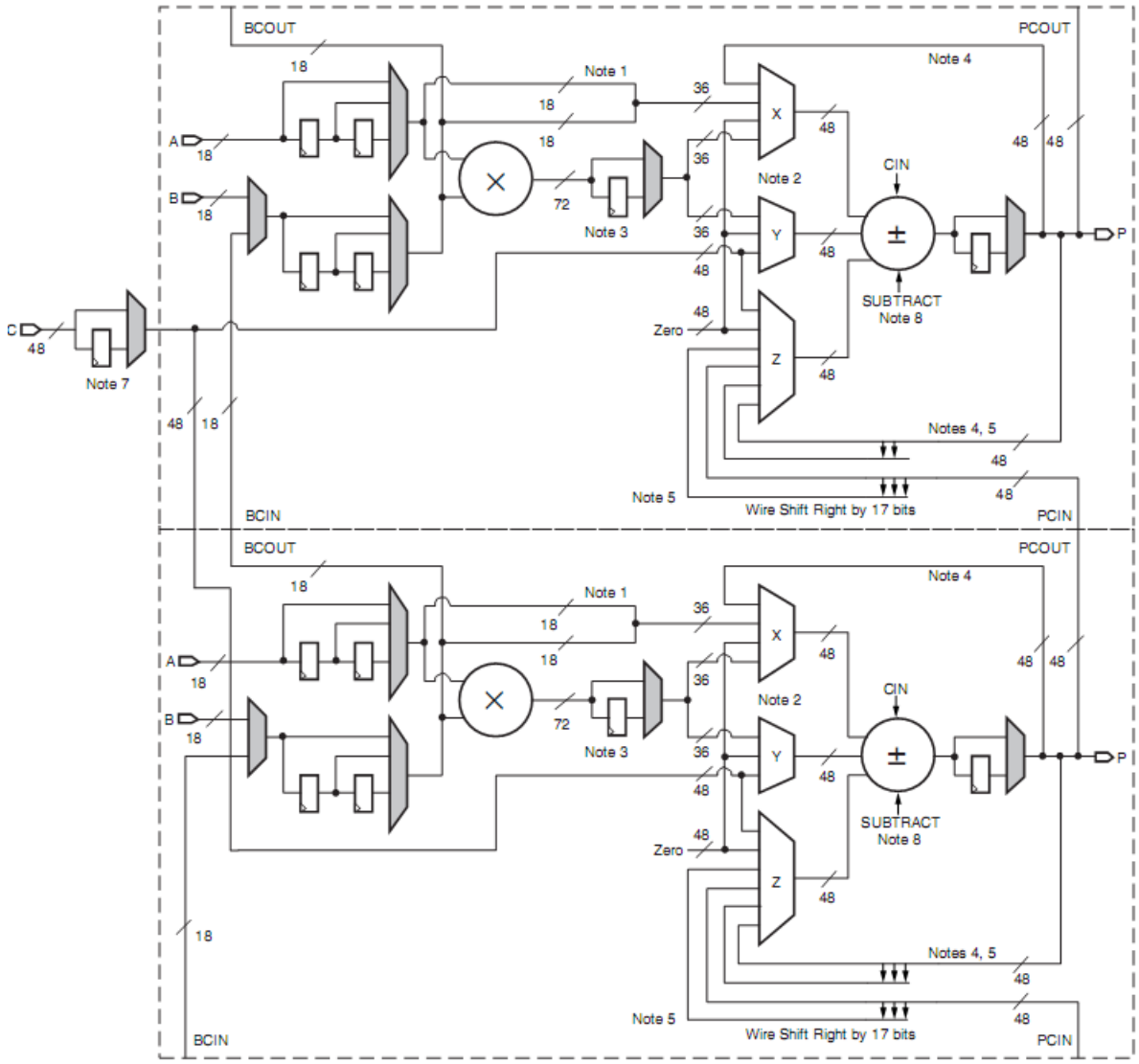

Figure 4-3: A DSP48 Tile Consisting of Two DSP48 Slices [12]

Notes:

1. The 18-bit A bus and B bus are concatenated, with the A bus being the most significant.
2. The X,Y, and Z multiplexers are 48-bit designs. Selecting any of the 36-bit inputs provides a 48-bit sign-extended output.
3. The multiplier outputs two 36-bit partial products, sign extended to 48 bits. The partial products feed the X and Y multiplexers. When OPMODE selects

the multiplier, both X and Y multiplexers are utilized and the adder/subtracter combines the partial products into a valid multiplier result.

4. The multiply-accumulate path for P is through the Z multiplexer. The P feedback through the X multiplexer enables accumulation of P cascade when the multiplier is not used.
5. The Right Wire Shift by 17 bits path truncates the lower 17 bits, and sign extends the upper 17 bits.
6. The gray-colored multiplexers are programmed at configuration time.
7. The shared C register supports multiply-add, wide addition, or rounding.
8. Enabling SUBTRACT implements Z – (X+Y+CIN) at the output of the adder/subtracter.

Simplified DSP48 Slice Operation

The math portion of the DSP48 slice consists of an 18-bit x 18-bit, two's complement multiplier followed by three 48-bit datapath multiplexers (with outputs X, Y, and Z) followed by a three-input, 48-bit adder/subtracter.

The data and control inputs to the DSP48 slice feed the arithmetic portions directly or are optionally registered one or two times to assist the construction of different, highly pipelined, DSP application solutions. The data inputs A and B can be registered once or twice. The other data inputs and the control inputs can be registered once. Full speed operation is 500 MHz when using the pipeline registers.

In its most basic form, the output of the adder/subtracter is a function of its inputs. The inputs are driven by the upstream multiplexers, carry select logic, and multiplier array. Equation 4-1 summarizes the combination of X, Y, Z, and CIN by the adder/subtracter. The CIN, X multiplexer output, and Y multiplexer output are always added together. This combined result can be selectively added to or subtracted from the Z multiplexer output.

$$\text{Adder Out} = (Z \pm (X + Y + CIN)) \qquad \text{Equation 4-1}$$

Equation 4-2 describes a typical use where A and B are multiplied, and the result is added to or subtracted from the C register. More detailed operations based on control and data inputs are described in later sections. Selecting the multiplier function consumes both X and Y multiplexer outputs to feed the adder. The two 36-bit partial products from the multiplier are sign extended to 48 bits before being sent to the adder/subtracter.

$$\text{Adder Out} = C \pm (A \times B + CIN) \qquad \text{Equation 4-2}$$

Figure 4-4 shows the DSP48 slice in a very simplified form. The seven OPMODE bits control the selection of the 48-bit datapaths of the three multiplexers feeding each of the three inputs to the adder/subtracter. In all cases, the 36-bit input data to the multiplexers is sign extended, forming 48-bit input datapaths to the adder/subtracter. Based on 36-bit operands and a 48-bit accumulator output, the number of "guard bits" (i.e., bits available to guard against overflow) is 12. Therefore, the number of multiply accumulations possible before overflow occurs is 4096. Combinations of OPMODE, SUBTRACT, CARRYINSEL, and CIN control the function of the adder/subtracter.

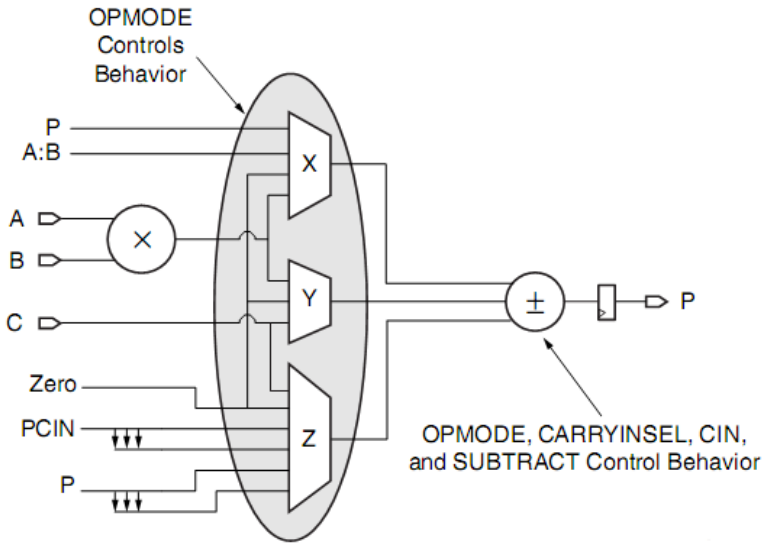

Figure 4-4: Simplified DSP48 Slice Model [12]

A, B, C, and P Port Logic

The DSP48 slice input and output data ports support many common DSP and math algorithms. The DSP48 slice has two direct 18-bit input data ports labeled A and B. Two DSP48 slices within a DSP48 tile share a direct 48-bit input data port labeled C. Each DSP48 slice has one direct 48-bit output port labeled P, a cascaded input datapath (B cascade), and a cascaded output datapath (P cascade), providing a cascaded input and output stream between adjacent DSP48 slices. The B cascade is selected via the B_INPUT attribute. The cascade is a dedicated resource that is always connected to the adjacent DSP48 and can be dynamically selected via the Z MUX (OPMODE 6:4).

Applications benefiting from this feature include FIR filters, complex multiplication, multi-precision multiplication, complex MACs, adder cascade, and adder tree (the final summation of several multiplier outputs) support.

The 18-bit A and B port can supply input data to the 18-bit x 18-bit, two's complement multiplier. When concatenated, A and B can bypass the multiplier and feed the X multiplexer input. The 48-bit C port is used as a general input to the Y and Z multiplexer to perform multiply, add, subtract, three-input add/subtract functions, or rounding.

Multiplexers controlled by configuration bits select flow-through paths, optional registers, or cascaded inputs. The data port registers allow users to typically trade off increased clock frequency (i.e., higher performance) vs. data latency. Also, a configuration controlled pipeline register between the multiplier and adder/subtracter is known as the M register. The registers have independent clock enables and resets, described in Table 4-2 and shown in Figure 4-1.

The configuration bit enables the C register to select between two potentially different clock domains, shown in Figure 4-8. The selection of the clock multiplexer is not set by user attributes. If the C register is used, the DSP48 slices packed in the same DSP48 tile must either be in the same clock domain or meet multicycle clock constraints.

The shared C input within the DSP tile can be used by the two slices within a tile in any one of the following modes:

1. **Neither DSP48 slice uses the C port**
   The C inputs in both slices are unconnected or are connected to GND, 0 in the HDL code. The place and route software maps the two slices in one tile.

2. **Both DSP48 slices use the same C port inputs**
   The C inputs in both slices are connected to C in the HDL code. The place and route software maps the two slices in one tile.

3. **Only one DSP48 slice is actually using the C port**
   There are some very specific rules when only one DSP48 slice uses the C port. The purpose of these new rules is to make sure there is "agreement" among the implementation tools, the simulation tools, and the customers' desired results.

The A, B, C, and P port logics are shown in Figure 4-6, Figure 4-7, Figure 4-8, and Figure 4-9, respectively [12].

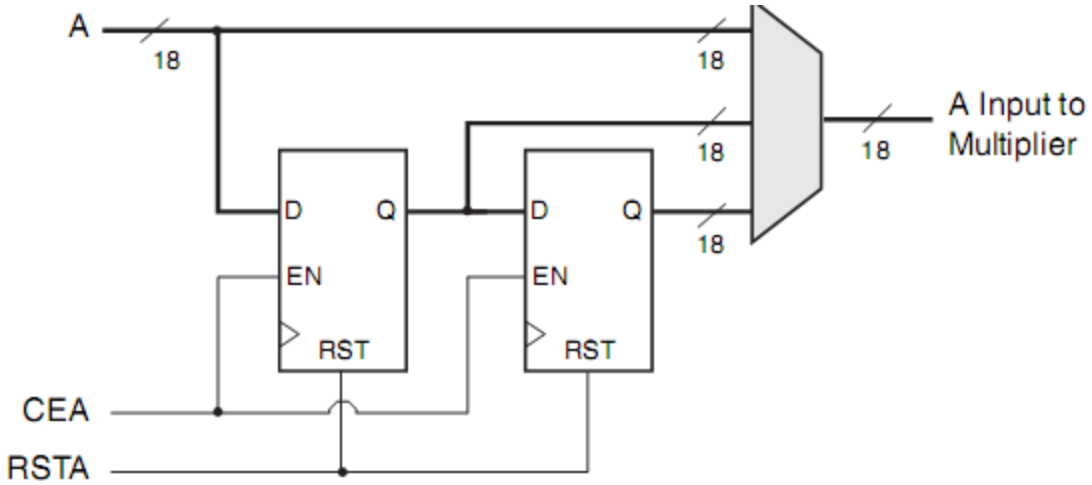

Figure 4-6: A Input Logic

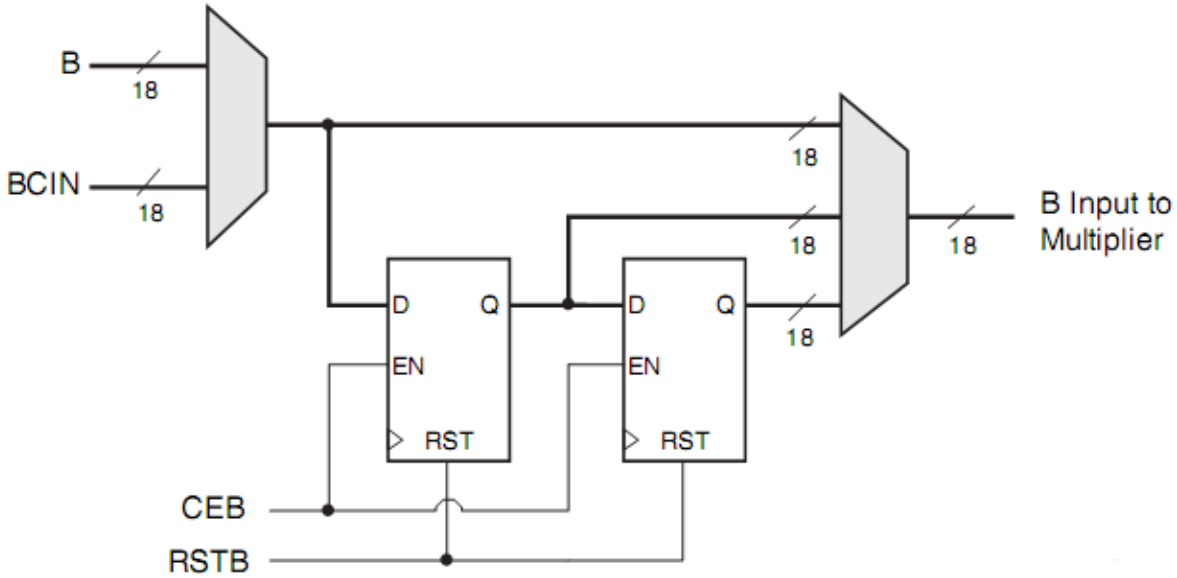

Figure 4-7: B Input Logic

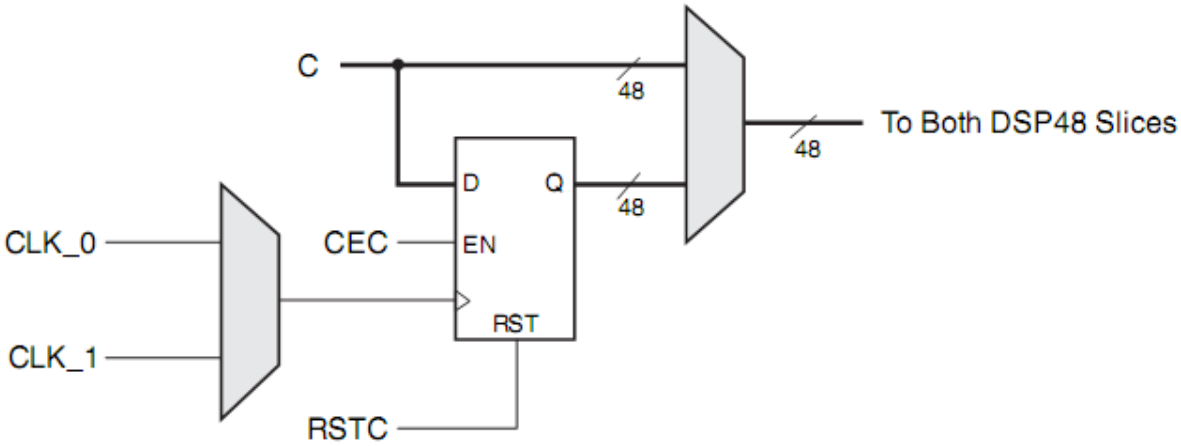

Figure 4-8: C Input Logic

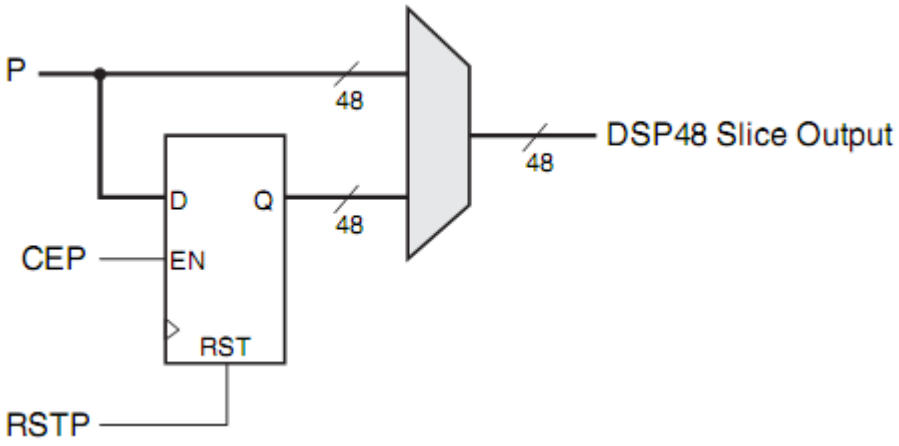

Figure 4-9: P Output Logic

OPMODE, SUBTRACT, and CARRYINSEL Port Logic

The OPMODE, SUBTRACT, and CARRYINSEL port logic supports flowthrough or registered input control signals. Similar to the datapaths, multiplexers controlled by configuration bits select flowthrough or optional registers. The control port registers allow users to trade off increased clock frequency (i.e., higher performance) vs. data latency.

The registers have independent clock enables and resets, described in Table 4-2 and shown in Figure 4-1. The OPMODE, SUBTRACT, and CARRYINSEL registers are reset by RSTCTRL. The SUBTRACT register has a separate enable labeled CECINSUB from OPMODE and CARRYINSEL. This enable signal is also used to enable the carry input from the general interconnect. Figure 4-10 shows the OPMODE, SUBTRACT, and CARRYINSEL port logic [12].

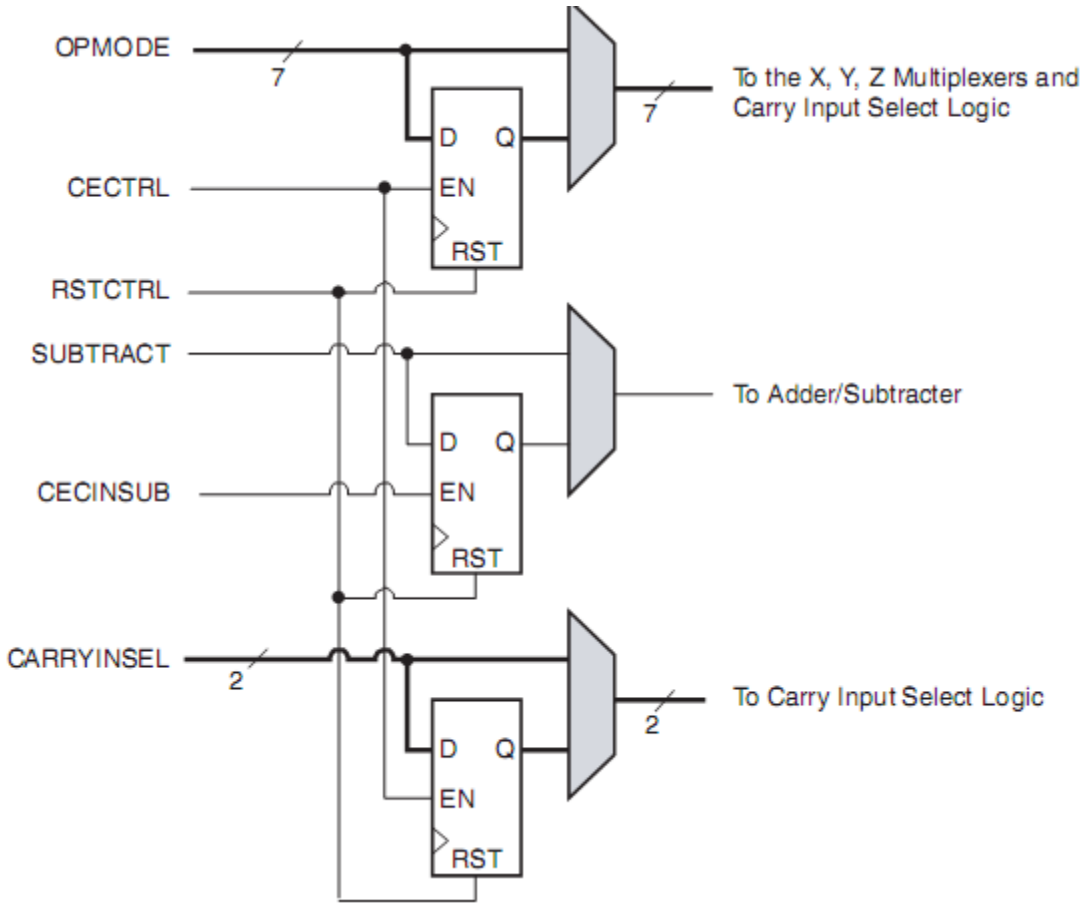

Figure 4-10: OPMODE, SUBTRACT, and CARRYINSEL Port Logic

Two's Complement Multiplier

The two's complement multiplier inside the DSP48 slice accepts two 18-bit x 18-bit, two's complement inputs and produces a 36-bit, two's complement result. Cascading of multipliers to achieve larger products is supported with a 17-bit right-shifted cascaded bus input to the adder/subtracter to right justify partial products by the correct number of bits. MACC functions can also right justify intermediate results for multi-precision. The multiplier can emulate unsigned math by setting the MSB of an 18-bit operand to zero.

X, Y, and Z Multiplexer

The OPMODE inputs provide a way for the design to change its functionality from clock cycle to clock cycle (e.g., when altering the initial or final state of the DSP48 relative to the middle part of a given calculation). The OPMODE bits can be optionally registered under the control of the configuration memory cells (as denoted by the gray MUX symbol in Figure 4-10).

Table 4-3, Table 4-4, and Table 4-5 list the possible values of OPMODE and the resulting function at the outputs of the three multiplexers (X, Y, and Z multiplexers). The multiplexer outputs supply three operands to the following adder/subtracter. Not all possible combinations for the multiplexer select bits are allowed. Some are marked in the tables as "illegal selection" and give undefined results. If the multiplier output is selected, then both the X and Y multiplexers are consumed, supplying the multiplier output to the adder/subtracter.

There are seven possible non-zero operands for the three-input adder as selected by the three multiplexers and the 36-bit operands are sign extended to 48 bits at the multiplexer outputs:

1. Multiplier output, supplied as two 36-bit partial products
2. Multiplier bypass bus consisting of A concatenated with B
3. C bus, 48 bits, shared by two slices
4. Cascaded P bus, 48 bits, from a neighbor DSP48 slice
5. Registered P bus output, 48 bits, for accumulator functions
6. Cascaded P bus, 48 bits, right shifted by 17 bits from a neighbor DSP48 slice
7. Registered P bus output, 48 bits, right shifted by 17 bits, for accumulator functions

Table 4-3: OPMODE Control Bits Select X Multiplexer Outputs

OPMODE Binary			X Multiplexer Output
Z	Y	X	
XXX	XX	00	Zero (Default)
XXX	XX	01	Multiplier Output
XXX	XX	10	P
XXX	XX	11	A concatenate B

Table 4-4: OPMODE Control Bits Select Y Multiplexer Outputs

OPMODE Binary			Y Multiplexer Output
Z	Y	X	
XXX	00	XX	Zero (Default)
XXX	01	XX	Multiplier Output
XXX	10	XX	Illegal Selection
XXX	11	XX	C

Table 4-5: OPMODE Control Bits Select Z Multiplexer Outputs

OPMODE Binary			Z Multiplexer Output
Z	Y	X	
000	XX	XX	Zero (Default)
001	XX	XX	PCIN
010	XX	XX	P
011	XX	XX	C
100	XX	XX	Illegal Selection
101	XX	XX	Shift (PCIN)
110	XX	XX	Shift (P)
111	XX	XX	Illegal Selection

Three-Input Adder/Subtracter

The adder/subtracter output is a function of control and data inputs. OPMODE, as shown in the previous section, selects the inputs to the X, Y, Z multiplexer directed to the associated three adder/subtracter inputs. It also describes how selecting the multiplier output consumes both X and Y multiplexers.

As with the input multiplexers, the OPMODE bits specify a portion of this function. Table 4-6 shows OPMODE combinations and the resulting functions. The symbol ± in the table means either add or subtract and is specified by the state of the SUBTRACT control signal (SUBTRACT = 1 is defined as “subtraction”). The outputs of the X and Y multiplexer and CIN are always added together. This result is then added to or subtracted from the output of the Z multiplexer. When the multiplier output is selected, both X and Y multiplexers are used to feed the multiplier partial products to the adder input.

Table 4-6: OPMODE Control Bits Adder/Subtracter Function [12]

Hex OPMODE	Binary OPMODE	XYZ Multiplexer Outputs and Adder/Subtracter Output			
[6:0]	Z Y X	Z	Y	X	Adder/Subtracter Output
0x00	000 00 00	0	0	0	±CIN
0x02	000 00 10	0	0	P	±(P + CIN)
0x03	000 00 11	0	0	A:B	±(A:B + CIN)
0x05	000 01 01	0	Note 1		±(A × B + CIN)
0x0c	000 11 00	0	C	0	±(C + CIN)
0x0e	000 11 10	0	C	P	±(C + P + CIN)
0x0f	000 11 11	0	C	A:B	±(A:B + C + CIN)
0x10	001 00 00	PCIN	0	0	PCIN ± CIN
0x12	001 00 10	PCIN	0	P	PCIN ± (P + CIN)
0x13	001 00 11	PCIN	0	A:B	PCIN ± (A:B + CIN)
0x15	001 01 01	PCIN	Note 1		PCIN ± (A × B + CIN)
0x1c	001 11 00	PCIN	C	0	PCIN ± (C + CIN)
0x1e	001 11 10	PCIN	C	P	PCIN ± (C + P + CIN)
0x1f	001 11 11	PCIN	C	A:B	PCIN ± (A:B + C + CIN)
0x20	010 00 00	P	0	0	P ± CIN
0x22	010 00 10	P	0	P	P ± (P + CIN)
0x23	010 00 11	P	0	A:B	P ± (A:B + CIN)
0x25	010 01 01	P	Note 1		P ± (A × B + CIN)
0x2c	010 11 00	P	C	0	P ± (C + CIN)
0x2e	010 11 10	P	C	P	P ± (C + P + CIN)
0x2f	010 11 11	P	C	A:B	P ± (A:B + C + CIN)
0x30	011 00 00	C	0	0	C ± CIN
0x32	011 00 10	C	0	P	C ± (P + CIN)
0x33	011 00 11	C	0	A:B	C ± (A:B + CIN)
0x35	011 01 01	C	Note 1		C ± (A × B + CIN)
0x3c	011 11 00	C	C	0	C ± (C + CIN)
0x3e	011 11 10	C	C	P	C ± (C + P + CIN)
0x3f	011 11 11	C	C	A:B	C ± (A:B + C + CIN)
0x50	101 00 00	Shift (PCIN)	0	0	Shift(PCIN) ± CIN
0x52	101 00 10	Shift (PCIN)	0	P	Shift(PCIN) ± (P + CIN)
0x53	101 00 11	Shift (PCIN)	0	A:B	Shift(PCIN) ± (A:B + CIN)
0x55	101 01 01	Shift (PCIN)	Note 1		Shift(PCIN) ± (A × B + CIN)
0x5c	101 11 00	Shift (PCIN)	C	0	Shift(PCIN) ± (C + CIN)
0x5e	101 11 10	Shift (PCIN)	C	P	Shift(PCIN) ± (C + P + CIN)
0x5f	101 11 11	Shift (PCIN)	C	A:B	Shift(PCIN) ± (A:B + C + CIN)(2)
0x60	110 00 00	Shift (P)	0	0	Shift(P) ± CIN
0x62	110 00 10	Shift (P)	0	P	Shift(P) ± (P + CIN)
0x63	110 00 11	Shift (P)	0	A:B	Shift(P) ± (A:B + CIN)(2)
0x65	110 01 01	Shift (P)	Note 1		Shift(P) ± (A × B + CIN)
0x6c	110 11 00	Shift (P)	C	0	Shift(P) ± (C + CIN)
0x6e	110 11 10	Shift (P)	C	P	Shift(P) ± (C + P + CIN)
0x6f	110 11 11	Shift (P)	C	A:B	Shift(P) ± (A:B + C + CIN)

Forming Larger Multipliers

Figure 4-11 illustrates the formation of a 35 x 35-bit multiplication from smaller 18 x 18-bit multipliers. The notation "0,B[16:0]" denotes B has a leading zero followed by 17 bits, forming a positive two's complement number.

When separating two's complement numbers into two parts, only the most-significant part carries the original sign. The least-significant part must have a forced zero in the sign position meaning they are positive operands. While it seems logical to separate a positive number into the sum of two positive numbers, it can be counter intuitive to separate a negative number into a negative most-significant part and a positive least-significant part. However, after separation, the most-significant part becomes "more negative" by the amount the least-significant part becomes "more positive." The 36-bit input operands include a forced zero sign bit in the least-significant part. So the valid number of bits in the input operands is only 35-bits.

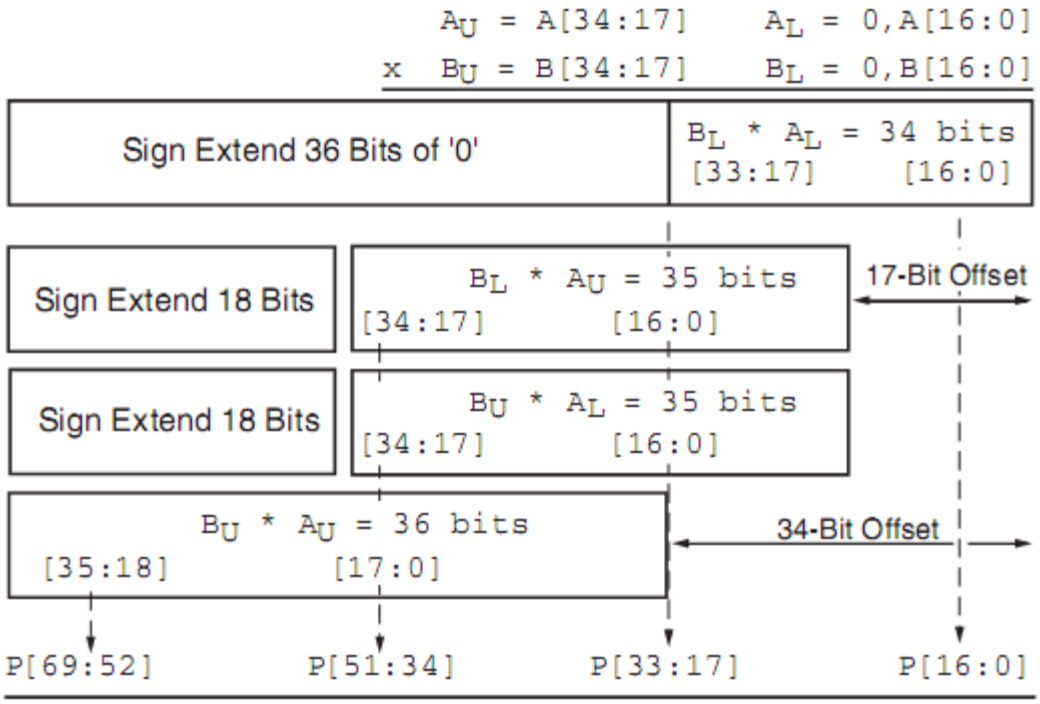

Figure 4-11: 35x35-Bit Multiplication from 18x18-Bit Multipliers

The DSP48 slices with 18 x 18 multipliers and post adder can now be used to implement the sum of the four partial products shown in Figure 4-11. The lessor significant partial products must be right-shifted by 17 bit positions before being summed with the next most-significant partial products. This is accomplished with a built in wire shift applied to PCIN supplied as one selectable Z multiplexer input. The entire process of multiplication, shifting, and addition using adder cascade to form the 70-bit result can remain in the dedicated silicon of the DSP48 slice, resulting in maximum performance with minimal power consumption.

Chapter 5

Proposed Multiplier Design

Here I am going to discuss the procedure of implementing the floating point multiplier on FPGA by using it's built in fixed point multiplier as a floating point multiplier. The new design methodology helps us to reduce the area utilization on FPGAs which is a major concern while implementing a floating point algorithm. The key point is to use dedicated multipliers available on FPGAs. These multipliers are fixed point multipliers and we have used them for the simple multiplication of mantissa parts of floating point numbers. So, instead of implementing the whole floating point multiplier on FPGA, the idea is to use the built in fixed point multiplier whenever a multiplication is required. All other issues like sign checking, normalization of floating point numbers, pre and post adjustment of exponents, shifting of mantissas by appropriate number of bits and rounding, flags etc are discussed here briefly.

The multiplier is 32-bit and is in accordance with the IEEE-754 floating point standard. This technique helps in the reduction of area utilization while implementing floating point algorithms on FPGAs.

A floating-point multiplier is nothing new. The IEEE-754 standard was published in 1985. The process of floating point multiplication is simpler and is similar to the integer multiplication as we are dealing with sign-magnitude representation. The process is shown in Figure 5-1.

Binary floating-point numbers will be split into their component parts: the sign, exponent, and mantissa. These three portions will be represented by uppercase letters: S, E, and M, respectively. As prescribed by the single precision format, E is eight bits wide [7:0] and M is twenty-three bits wide [22:0].

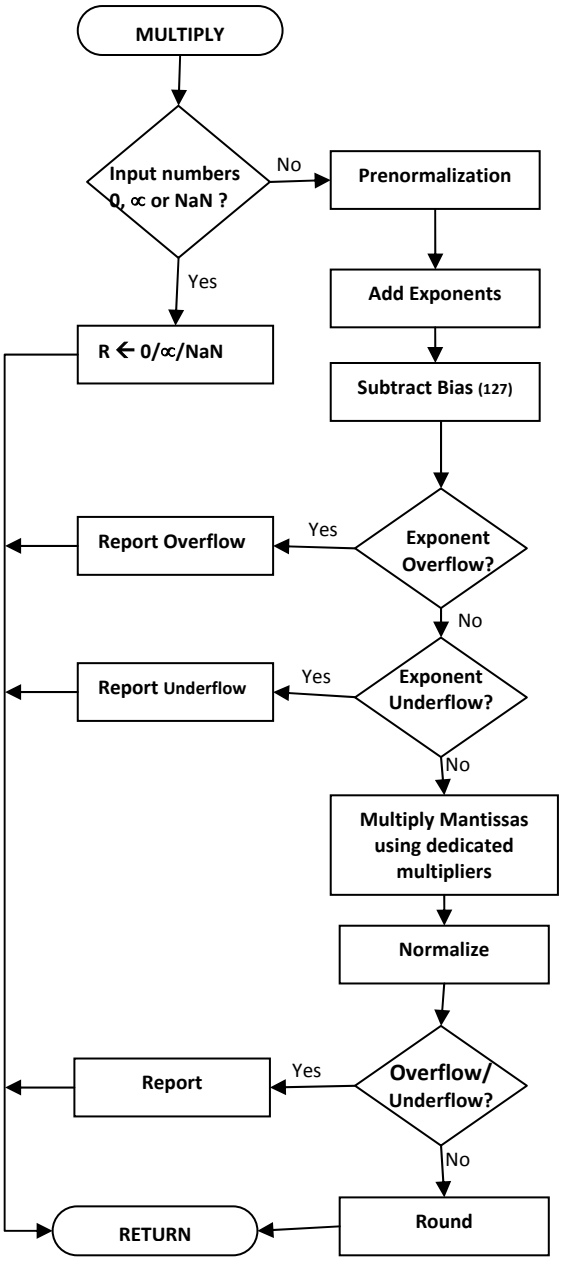

Figure 5-1: Flow chart showing floating point Multiplication

Special Case Numbers Detection

The multiplier cannot always determine a result by simply doing a multiplication. There are certain inputs that require the multiplier to take special action. Table 5–1 shows the special case numbers defined by the IEEE-754 1985 Standard for Binary Arithmetic and their data bit representations.

Table 5-1: Representation of special case numbers in IEE-754 standard

Sign	Exponent	Mantissa	Representation
X	All 0's	All 0's	Zero
0	All 0's	Non-zero	Positive Denormalized
1	All 0's	Non-zero	Negative Denormalized
0	All 1's	All 0's	Positive Infinity
1	All 1's	All 0's	Negative Infinity
X	All 1's	Non-zero	Not a Number (NaN)

First of all the inputs are checked for the special case numbers (zeros, Not a Number (NaN), infinity). These numbers are detected and appropriated action is performed by the multiplier as specified below.

Not a number (NaN) - The IEEE 754 Standard specifies that an implementation will return a NaN that is given to it as input, or either one if both inputs are NaN's. The multiplier can be configured to return either the first NaN or the higher of the two. The Intel Pentium series returns the higher of the two NaN's and, as this multiplier was tested using a processor from that series, the multiplier is by default set to do the same.

Infinity - Nearly anything multiplied by infinity is properly signed infinity, with the exception of NaN, described above, and zero, described below.

Infinity and zero - The result of the multiplication of infinity and zero is undefined. The multiplier will therefore return a predefined NaN.

If none of these cases apply, the special case path signals that the result of the standard path should be chosen.

Pre Normalizing

After this step if the input numbers are not zero, infinity or NaN, the numbers are normalized and the stage is called prenormalization. In IEEE-754 format, normalized numbers have an implicit leading 1, and that denormalized numbers do not. To keep the multiplication stage simple, both inputs are converted into the same form. In the pre normalization stage, both the input numbers are normalized so that they have an implicit leading one.

Sign Bit

If A and B be the two input numbers and R be the result of multiplying A and B, then

$$A= (S_A,E_A,M_A),$$
$$B= (S_B,E_B,M_B),$$
$$R= (S_R,E_R,M_R)$$

Where, S, E and M represent the sign, exponent, and mantissa of the respective numbers. The sign S_R of the result is calculated as:

$$\mathbf{S_R = S_A \oplus S_B}$$

Adding Exponents

The exponent E_R of the result is calculated as follows:

$$\mathbf{E_R = E_A + E_B - 127}$$

As the exponents of A and B are stored with bias, we have to subtract 127 (bias for 32-bit numbers) from their sum in order to get the right result. The result

could be either an exponent overflow or underflow which is checked and reported here.

Mantissa Multiplication

Now for the multiplication of mantissas, we have to add the implicit 1to the beginning of the mantissas thus converting them from 23 bits to 24 bits as the numbers are in normalized form. The Mantissa M_R of the result is obtained as follows:

$$\mathbf{M_R = (1.M_A) * (1.M_B)}$$

For the multiplication of mantissas we propose here to use the dedicated multiplier available on the FPGA device. By doing this a great reduction in area utilization is achieved (as shown in the next section) which is always a major concern while implementing a multiplication based floating point algorithm on FGPA.For the multiplication of two mantissas the dedicated or built in multipliers available on FPGAs are used. I have used the Xilinx® Virtex4 FPGA and it has built in multiplier known as DSP48. Its functions and usages have already been discussed in chapter 4 in detail. Here is how it is used to multiply two 24bit mantissas. Figure 5-2 illustrates the process of multiplication.

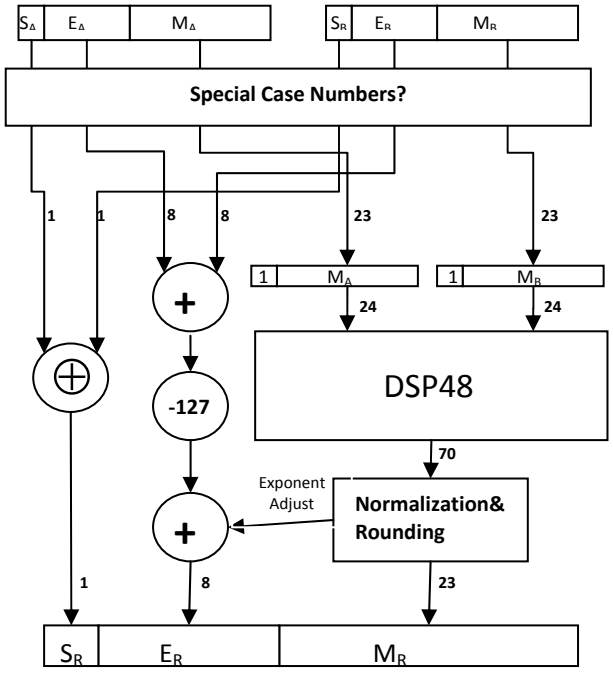

Figure 5-2: Multiplier Implementation using DSP48

As it is a 18x18 multiplier, and we need 24bit multiplier so we have to use two or more DSP48 blocks to perform the multiplication. Figure 5-3 illustrates the formation of a 35 x 35-bit multiplication from smaller 18 x 18-bit multipliers. Four DSP48 slices have been used for 35x35 multiplier.

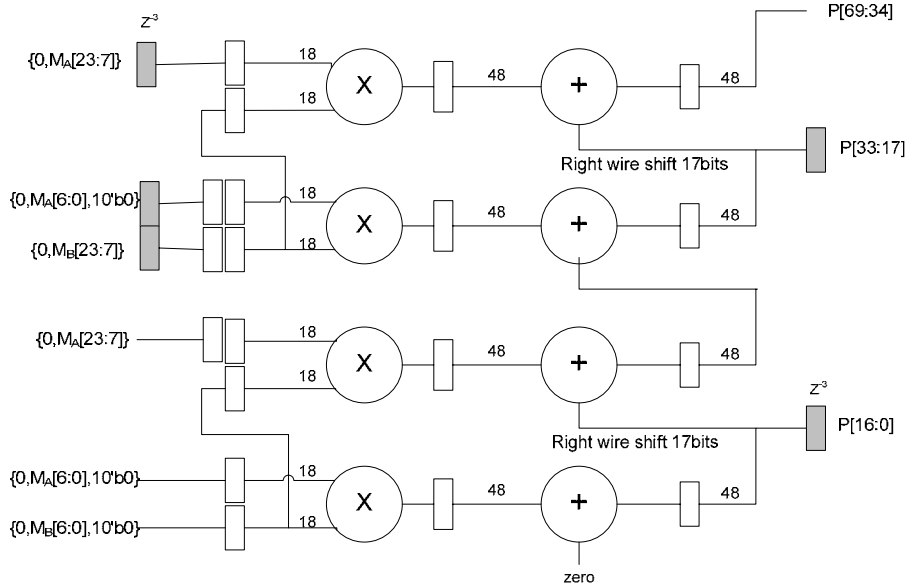

Figure 5-3: Details of DSP48 block usage in figure 5-2

Here for the multiplication of 24 bit mantissas the numbers are broken down into two parts so that they can be used with 18x18 multipliers. When separating two's complement numbers into two parts, only the most-significant part carries the original sign. The least-significant part must have a forced zero in the sign position meaning they are positive operands. As we are here dealing with the unsigned number multiplication so a zero is also placed in the sign bit of the most significant part. While it seems logical to separate a positive number into the sum of two positive numbers, it can be counter intuitive to separate a negative number into a negative most-significant part and a positive least-significant part. However, after separation, the most-significant part becomes "more negative" by the amount the least-significant part becomes "more positive." The 36-bit input operands include a forced zero sign bit in the least-significant part. So the valid number of bits in the input operands is only 35-bits.

The breakdown of the mantissa M_A is as follow:

Let A_U be the most significant part of the 35x35 multiplier. A zero is placed in the sign bit indicating unsigned multiplication. Upper 17 bits (most significant bits) of the mantissa M_A are copied to the remaining 17 bits of A_U.

Let A_L be the least significant part of the 35x35 multiplier. A zero is placed in the sign bit indicating unsigned multiplication. Remaining 7 bits (most significant bits) of the mantissa M_A are copied to the most significant 7 bits of A_L after the sign bit. The remaining 10 bits of A_L are filled with zeros.

The breakdown of the mantissa M_B is as follow:

Let B_U be the most significant part of the 35x35 multiplier. A zero is placed in the sign bit indicating unsigned multiplication. Upper 17 bits (most significant bits) of the mantissa M_B are copied to the remaining 17 bits of B_U.

Let B_L be the least significant part of the 35x35 multiplier. A zero is placed in the sign bit indicating unsigned multiplication. Remaining 7 bits (most significant bits) of the mantissa M_B are copied to the most significant 7 bits of B_L after the sign bit. The remaining 10 bits of B_L are filled with zeros.

Figure 5-4 illustrates how the multiplication of mantissas in parts is performed using the above mentioned breakdown.

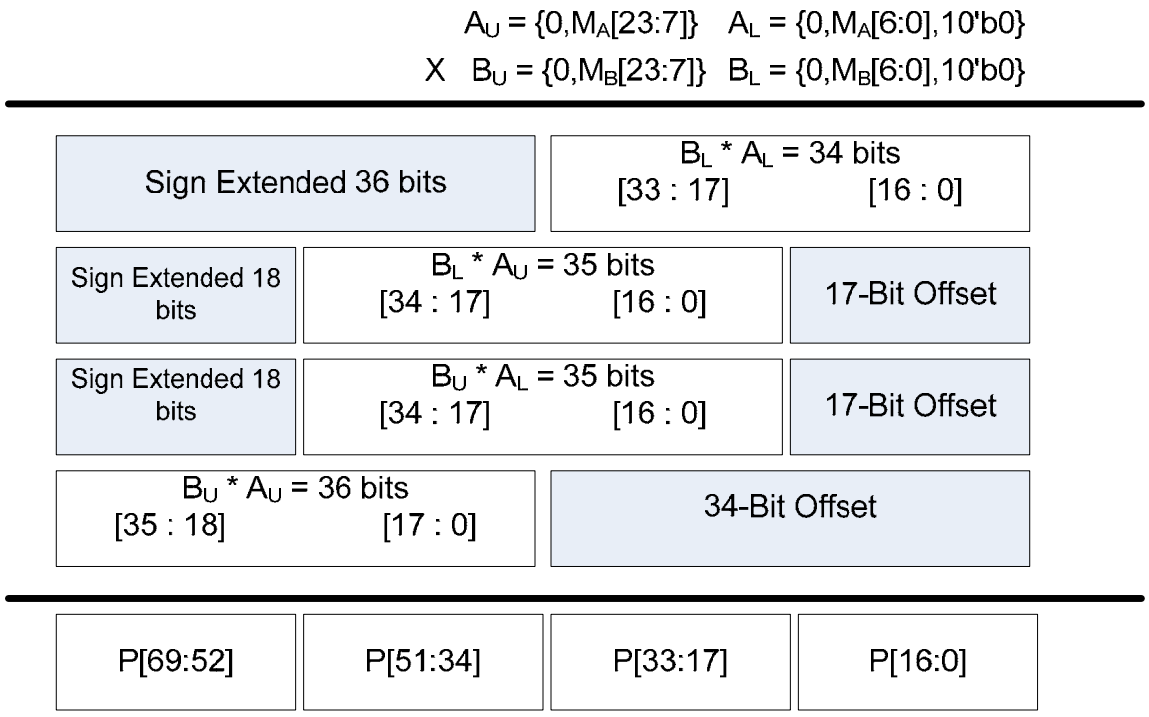

Figure 5-4: 35x35-Bit Multiplication from 18x18-Bit Multipliers

All the slices have fixed behavior as specified by the OPMODE. With reference to Figure 5-3 and Figure 5-4, slice 1 is being used for the multiplication of A_L and B_L. At this stage we obtain the lower 17 bits of the final product. Slice 2 is being used for the multiplication of A_U and B_L. The partial product from the slice 1 is being shifted right by 17 bits and the added to the multiplication result obtained in slice 2. Slice 3 is being used for the multiplication of A_L and B_U. The result from slice 2 is being added here to the multiplication result obtained in slice 3 without shifting. The slice

3 is in accumulation mode where the multiplication result is being accumulated with the partial product from slice 2 through PCIN port. Here the [33:17] bits of the final product are obtained. Slice 4 is being used for the multiplication of A_U and B_U. The partial product from slice 3 is shifted 17 bits right and added to the result of multiplication in slice 4. Here upper [69:34] bits of final product are obtained. Table 5-2 summarizes the above explained procedure and shows how 35 x 35 multiply is being used for 24 bit mantissa multiplication using four DSP48 slices.

Table 5-2: Summary of DSP48 Implementation of 24 bit mantissa multiplication

Slice	Inputs			Function	OPMODE	Output
	A	B	C			
1	A_L {0,M_A[6:0],10'b0}	B_L {0,M_B[6:0],10'b0}	X	Multiply	0x05	P[16:0]
2	A_U {0,M_A[23:7]}	B_L {0,M_B[6:0],10'b0}	X	17 bit shifted feedback Multiply Add	0x55	-
3	A_L {0,M_A[6:0],10'b0}	B_U {0,M_B[23:7]}	X	Multiply Accumulate	0x25	P[33:17]
4	A_U {0,M_A[23:7]}	B_U {0,M_B[23:7]}	X	17 bit shifted feedback Multiply Add	0x55	P[69:34]

Post Normalizing

In the post normalization stage, the multiplier normalizes the product. This phase normalizes the result. Normalization consists of shifting the mantissa digits until the most significant digit is non-zero. Each shift causes a decrement or increment of the exponent and thus could cause an exponent overflow or underflow. This is checked and reported here.

Rounding

The product of two n-bit numbers has the potential of being 2(n+1) bits wide. The result of floating point multiplication, however, must fit into the same n bits as the multiplier and the multiplicand. This, of course, often leads to loss of precision. The authors of the IEEE standard attempted to keep this loss as minimal as possible with the introduction of standard rounding modes.

According to IEEE 74 standard there are four rounding modes: round to nearest even, round to zero, round to positive infinity, and round to negative infinity.

Flags

Although the IEEE 754 standard defines five flags, only two are relevant to this implementation of a multiplier. Note that if the special case result is chosen, none of the flags will be set.

Overflow - The magnitude of the result is too large to be represented. In addition to setting the overflow flag, the multiplier will return an appropriately signed infinity.

Underflow - The result is too small to be represented as a normalized number. The multiplier will return a denormalized number, or it will return zero if the number is too small to be represented as a denormalized number. There are two conditions that lead to underflow: tininess and loss of accuracy. Each of these conditions has two methods for detecting it defined by the standard. The differences between the methods are subtle, and only matter in corner cases.

Example

Let us take two numbers for the multiplication. These numbers be:

$$A = 1.125 \times 2^4 \text{ and } B = 1.5 \times 2^3$$

Adding bias (127) to the exponents of the two numbers we get:

$$4+127 = 131 \text{ and } 3+127 = 130$$

Converting their integral and fractional parts their binary representation is as follows:

$$A = 1.001 \times 2^{10000011}$$
$$B = 1.100 \times 2^{10000010}$$

Single precision floating point representation of these two numbers is as follows. Note that the leading 1 is not stored with the mantissa:

A =

0	10000011	00100000000000000000000

B =

0	10000010	10000000000000000000000

First of all the inputs are checked for the special case numbers. As it obvious that these two numbers are normal numbers so NaN,∞ or 0 not detected and multiplication process continues

Sign of the product is calculated as $0 \oplus 0 = 0$ so the final product is positive number

Exponent of the product is calculated as

131 + 130 -127 = 134 = 10000110

No overflow or underflow detected here so multiplication process continues.

Next for the multiplication of mantissas of the two numbers first the implicit 1 is appended at the beginning as shown below:

Mantissa A: 10010000000000000000000

Mantissa B: 11000000000000000000000

For the multiplication of mantissas these are broken down into two parts as shown below and then these are given at the inputs of DSP 48 slices as explained above.

A_U = {0,10010000000000000}

A_L = {0, 0000000,0000000000}

B_U = {0,11000000000000000}

B_L = {0, 0000000,0000000000}

The result obtained after the normalization and rounding is as follows:

0	10000110	10110000000000000000000

This is same as 1.6875x 2_7.

Results of Implementing the Multiplier

The device utilization summary in Table 5-3 shows the implementation of the proposed multiplier. The multiplier is implemented using Virtex ® 4 FPGA.

Table 5-3: Device utilization summary of the proposed multiplier

Logic Utilization	Used	Available	Utilization
Number of Slices	143	6144	2%
Number of Slice Flip Flops	5	12288	0%
Number of 4 input LUTs	265	12288	2%
Number of bonded IOBs	103	240	42%
Number of DSP48s	4	32	12%

Similarly for comparison purpose the Table 5-4 shows the multiplier implementation designed by Mark in [7].

Table 5-4: Device utilization summary of the multiplier designed by Mark in [7]

Logic Utilization	Used	Available	Utilization
Number of Slices	643	6144	10%
Number of 4 input LUTs	1150	12288	9%
Number of bonded IOBs	103	240	42%

Figure 5-5 presents a comparison of the two implementations of floating point multiplier in graphical form.

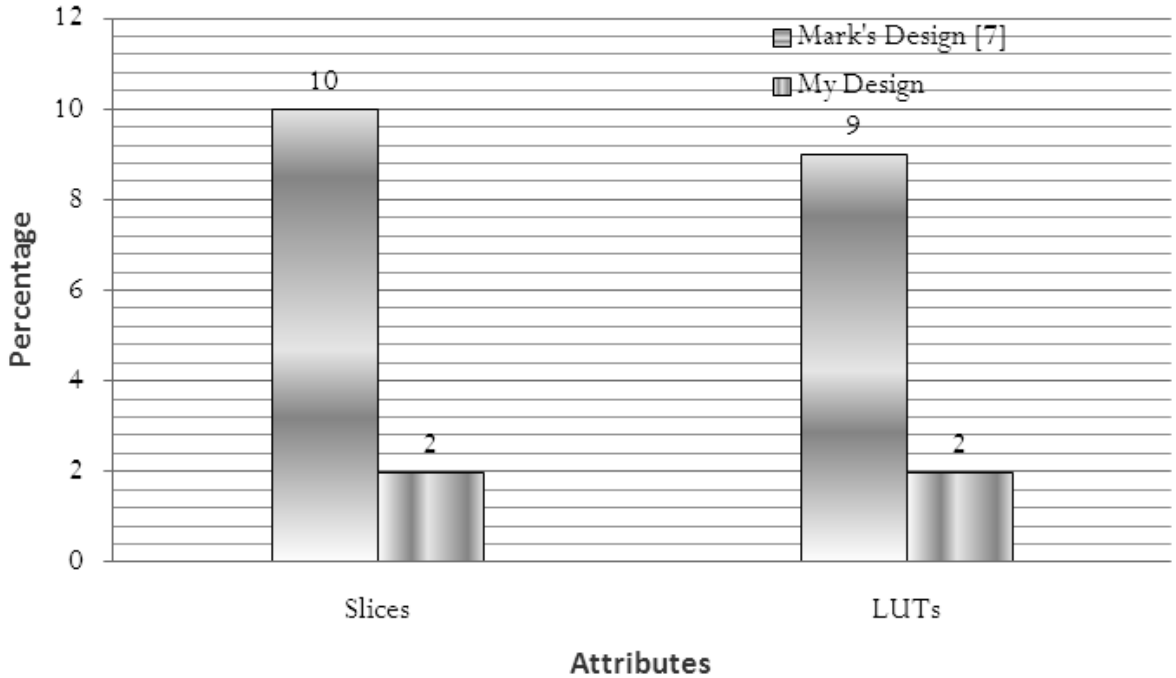

Figure 5-5: Comparison of two implementations of floating point multiplier

If we compare the two implementations we can see that the proposed multiplier is much efficient in area utilization which is the goal of this research. If we look at the frequency rate obtained the proposed multiplier is running at a speed of 409.870MHz while the multiplier in [7] is running at 23.858MHz. So the design of multiplier proposed here is not only efficient in terms of area utilization but also in terms of frequency rate obtained. By using this multiplier the floating point algorithms can be implemented efficiently both in terms of area and frequency.

Chapter 6

FIR Filter Implementation

Digital filters are typically used to modify attributes of signal in the time or frequency domain trough the process called linear convolution. There are only a few applications (e.g. adaptive filters) where general programmable filter architecture is required. In many cases the coefficients do not change over time - linear time-invariant filters (LTI). Digital filters are generally classified as being finite impulse response (FIR) or infinite impulse response (IIR). According to the names, an FIR filter consists of a finite number of samples values, reducing the above presented convolution to a finite sum per output sample. An IIR filter requires that an infinite sum has to be performed. In this paper implementation of the LTI FIR filters will be discussed [2].

Basic FIR Filters

FIR filters are used extensively in video broadcasting and wireless communications. DSP filter applications include, but are not limited to, the following [12]:

- Wireless Communications
- Image Processing
- Video Filtering
- Multimedia Applications
- Portable Electrocardiogram (ECG) Displays
- Global Positioning Systems (GPS)

Equation 6-1 shows the basic equation for a single-channel FIR filter.

$$y[n] = \sum_{k=0}^{k=N-1} h(k)x(n-k) \qquad \text{Equation 6-1}$$

The terms in the equation can be described as input samples, output samples, and coefficients. Imagine x as a continuous stream of input samples and y as a resulting stream (i.e., a filtered stream) of output samples. The n and k in the equation correspond to a particular instant in time, so to compute the output sample y(n) at time n, a group of input samples at N different points in time, or x(n), x(n-1), x(n-2), … x(n-N+1) is required. The group of N input samples are multiplied by N coefficients and summed together to form the final result y.

The main components used to implement a digital filter algorithm include adders, multipliers, storage, and delay elements. The DSP48 slice includes all of the above elements, making it ideal to implement digital filter functions. All of the input samples from the set of n samples are present at the input of each DSP48 slice. Each slice multiplies the samples with the corresponding coefficients within the DSP48 slice. The outputs of the multipliers are combined in the cascaded adders.

In Figure 6-1, the sample delay logic is denoted by Z^{-1}, where the –1 represents a single clock delay. The delayed input samples are supplied to one input of the multiplier. The coefficients (denoted by h(0) to h(N-1)) are supplied to the other input of the multiplier through individual ROMs, RAMs, registers, or constants. Y(n) is merely the summation of a set of input samples, and in time, multiplied by their respective coefficients.

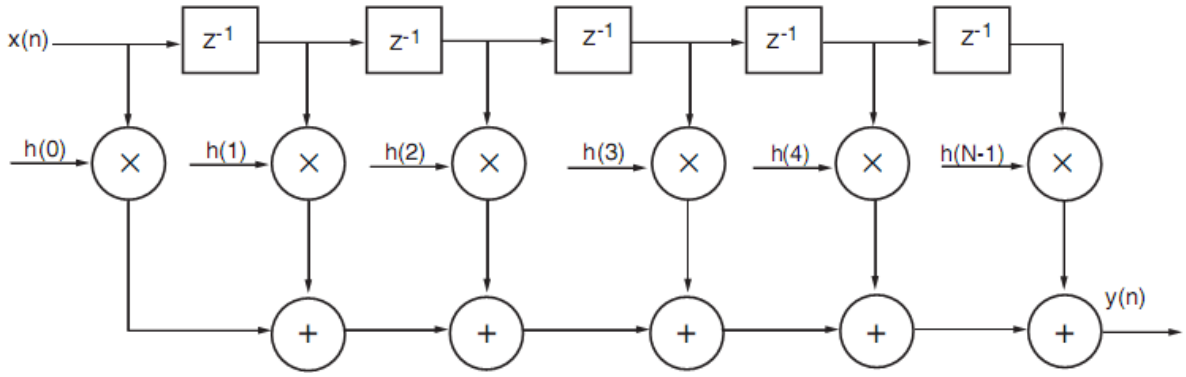

Figure 6-1: Conventional Tapped Delay Line FIR Filter

Implementation of 3-Tap FIR Filter on FPGA

Available digital filter software allows for very easy computation of coefficients of given filter. However, the challenge is in mapping the FIR structure into suitable architecture. Digital filters are typically implemented as multiply-accumulate algorithms with use of special DSP devices. In case of programmable structures direct or transposed forms are preferred for maximum speed and lowest resource utilization. Efficient hardware implementation of filter's structure is possible by optimization of multipliers and adders implementation [2].

I have implemented 3-tap filter on FPGA using the multiplier which I have designed and has been discussed in the previous chapter. This multiplier is used whenever a multiplication is required. Talbe 6-1 shows the device utilization of 3-tap FIR filter using multiplier designed by [7]

Table 6-1: Device utilization summary of 3-tap FIR filter implementation using multiplier in [7]

Logic Utilization	Used	Available	Utilization
Number of Slices	3496	6144	56%
Number of Slice Flip Flops	135	12288	1%
Number of 4 input LUTs	6774	12288	55%
Number of bonded IOBs	67	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	16	32	50%

Similarly, Table 6-2 shows the device utilization when the 3-tap FIR filter is implemented on FPGA using the multiplier that is described in the previous chapter. By comparing these two results it is clear that the 2nd implementation is far better than the first one in terms of area utilization. There is a significant decrease in the area utilization when my multiplier is used.

Table 6-2: Device utilization summary of 3-tap filter using proposed multiplier

Logic Utilization	Used	Available	Utilization
Number of Slices	162	6144	2%
Number of Slice Flip Flops	149	12288	1%
Number of 4 input LUTs	293	12288	2%
Number of bonded IOBs	67	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	7	32	21%

If we look at the timing summary it is obvious that the second implementation is also better in terms of frequency. The implementation of the filter using the multiplier designed by [7] is running at a lower frequency (11.754MHz) while the other implementation using my multiplier design is running at higher frequency (125.469MHz). So there is a significant improvement in my design in terms of both frequency and the area. A complete comparison and analysis in terms of area of the two implementations is shown in Figure 6-2.

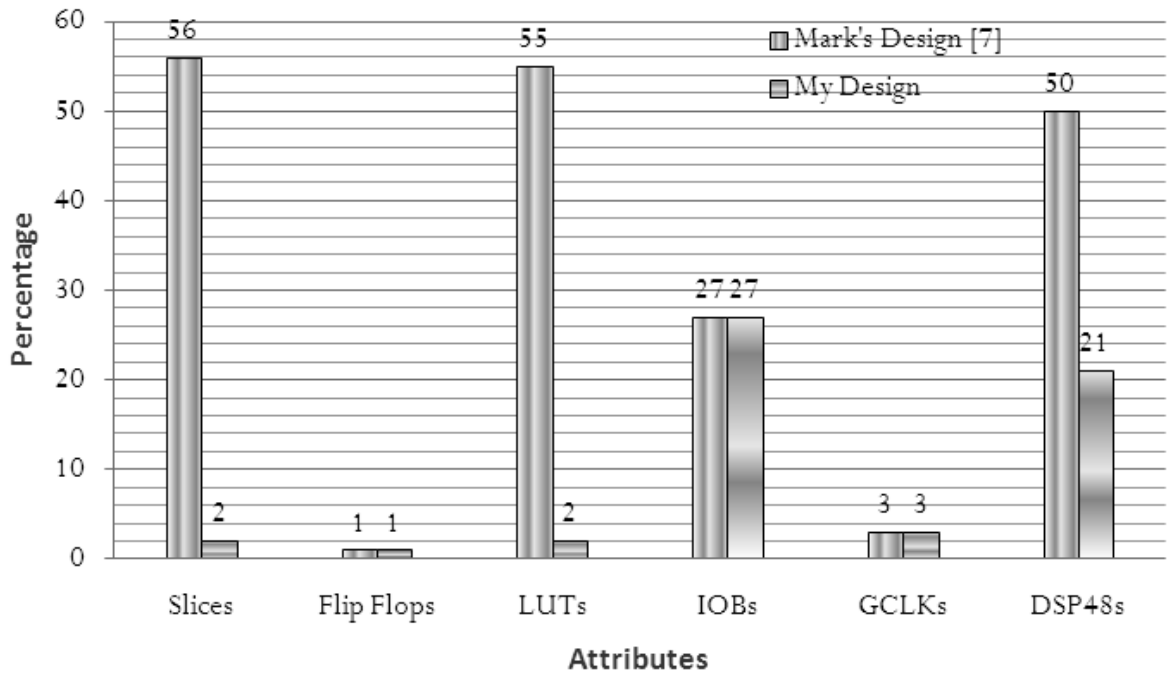

Figure 6-2: Graphical comparison of two implementations of 3-Tap FIR filter

I have also analyzed another aspect of the design that if we increase the number of the taps of the filter then what happens. The results and comparisons of the two implementations are presented below. These results have been prepared for 5-tap, 7-tap, 9-tap and 10-tap FIR filters.

5-Tap FIR Filter

Talbe 6-3 shows the device utilization of 5-tap FIR filter using multiplier designed by [7].

Table 6-3: Device utilization summary of 5-tap FIR filter implementation using multiplier in [7]

Logic Utilization	Used	Available	Utilization
Number of Slices	5461	6144	88%
Number of Slice Flip Flops	199	12288	1%
Number of 4 input LUTs	10554	12288	85%
Number of bonded IOBs	67	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	24	32	75%

Similarly, Table 6-4 shows the device utilization when the 5-tap FIR filter is implemented on FPGA using the multiplier that is described in the previous chapter. By comparing these two results it is clear that the 2nd implementation is far better than the first one in terms of area utilization. There is a significant decrease in the area utilization when my multiplier is used.

Table 6-4: Device utilization summary of 5-tap filter using proposed multiplier

Logic Utilization	Used	Available	Utilization
Number of Slices	207	6144	3%
Number of Slice Flip Flops	223	12288	1%
Number of 4 input LUTs	375	12288	3%
Number of bonded IOBs	67	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	11	32	34%

If we look at the timing summary it is obvious that the second implementation is also better in terms of frequency. The implementation of the filter using the multiplier designed by [7] is running at a lower frequency (8.014MHz) while the other implementation using my multiplier design is running at higher frequency (118.344MHz). So there is a significant improvement in my design in terms of both frequency and the area. A complete comparison and analysis in terms of area of the two implementations is shown in Figure 6-3.

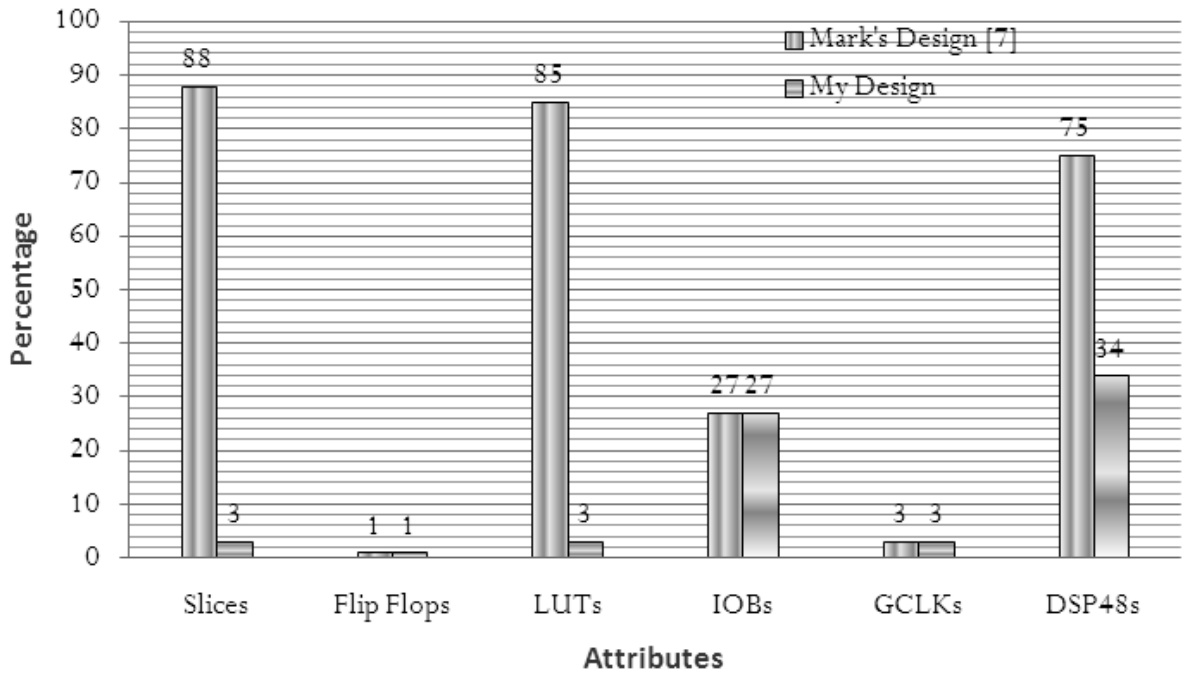

Figure 6-3: Graphical comparison of two implementations of 5-Tap FIR filter

7-Tap FIR Filter

Talbe 6-5 shows the device utilization of 7-tap FIR filter using multiplier designed by [7]

Table 6-5: Device utilization summary of 7-tap FIR filter implementation using multiplier in [7]

Logic Utilization	Used	Available	Utilization
Number of Slices	5881	6144	95%
Number of Slice Flip Flops	249	12288	2%
Number of 4 input LUTs	11339	12288	92%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	32	32	100%

Similarly, table 6-6 shows the device utilization when the 7-tap FIR filter is implemented on FPGA using the multiplier that is described in the previous chapter. By comparing these two results it is clear that the 2nd implementation is far better than the first one in terms of area utilization. There is a significant decrease in the area utilization when my multiplier is used.

Table 6-6: Device utilization summary of 7-tap FIR filter using proposed multiplier			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	255	6144	4%
Number of Slice Flip Flops	289	12288	2%
Number of 4 input LUTs	432	12288	3%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	15	32	46%

If we look at the timing summary it is obvious that the second implementation is also better in terms of frequency. The implementation of the filter using the multiplier designed by [7] is running at a lower frequency (5.770MHz) while the other implementation using my multiplier design is running at higher frequency (123.384MHz). So there is a significant improvement in my design in terms of both frequency and the area. A complete comparison and analysis in terms of area of the two implementations is shown in Figure 6-4.

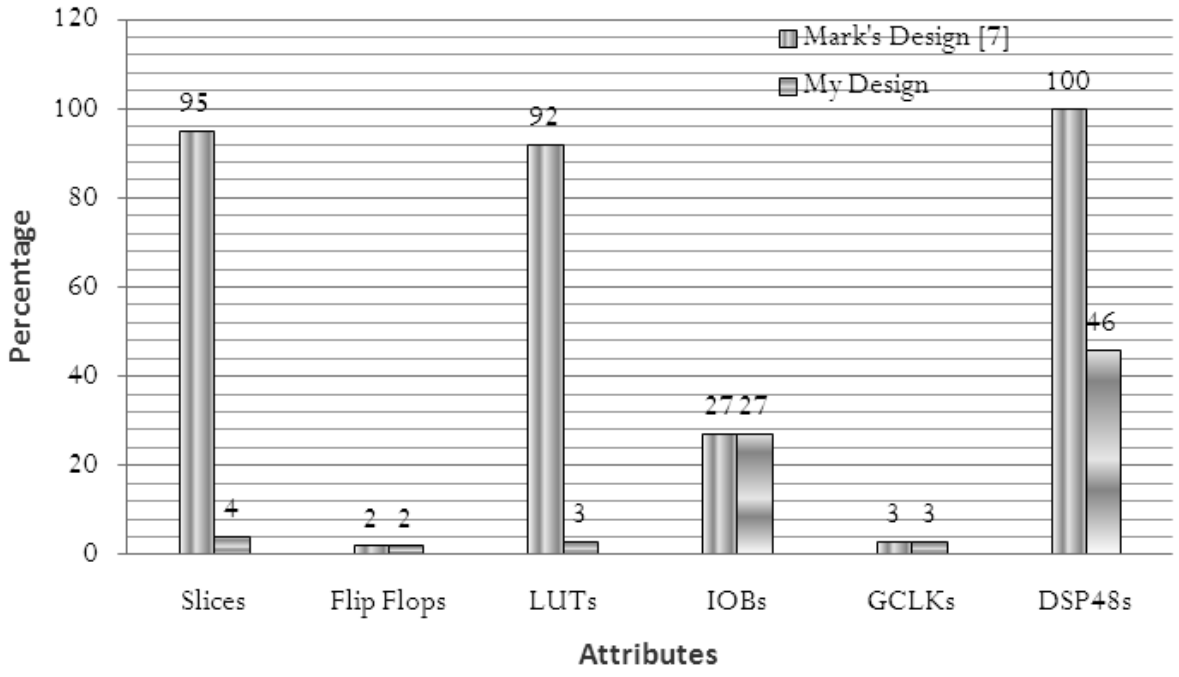

Figure 6-4: Graphical comparison of two implementations of 7-Tap FIR filter

9-Tap FIR Filter

Talbe 6-7 shows the device utilization of 9-tap FIR filter using multiplier designed by [7]

Table 6-7: Device utilization summary of 9-tap FIR filter implementation using multiplier in [7]			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	9010	6144	146%
Number of Slice Flip Flops	318	12288	2%
Number of 4 input LUTs	17425	12288	141%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	32	32	100%

Similarly, Table 6-8 shows the device utilization when the 9-tap FIR filter is implemented on FPGA using the multiplier that is described in the previous chapter. By comparing these two results it is clear that the 2nd implementation is far better than the first one in terms of area utilization. There is a significant decrease in the area utilization when my multiplier is used.

Table 6-8: Device utilization summary of 9-tap filter using proposed multiplier			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	308	6144	5%
Number of Slice Flip Flops	361	12288	2%
Number of 4 input LUTs	510	12288	4%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	19	32	59%

If we look at the timing summary it is obvious that the second implementation is also better in terms of frequency. The implementation of the filter using the multiplier designed by [7] is running at a lower frequency (5.024MHz) while the other implementation using my multiplier design is running at higher frequency (125.436MHz). So there is a significant improvement in my design in terms of both

frequency and the area. A complete comparison and analysis in terms of area of the two implementations is shown in Figure 6-5.

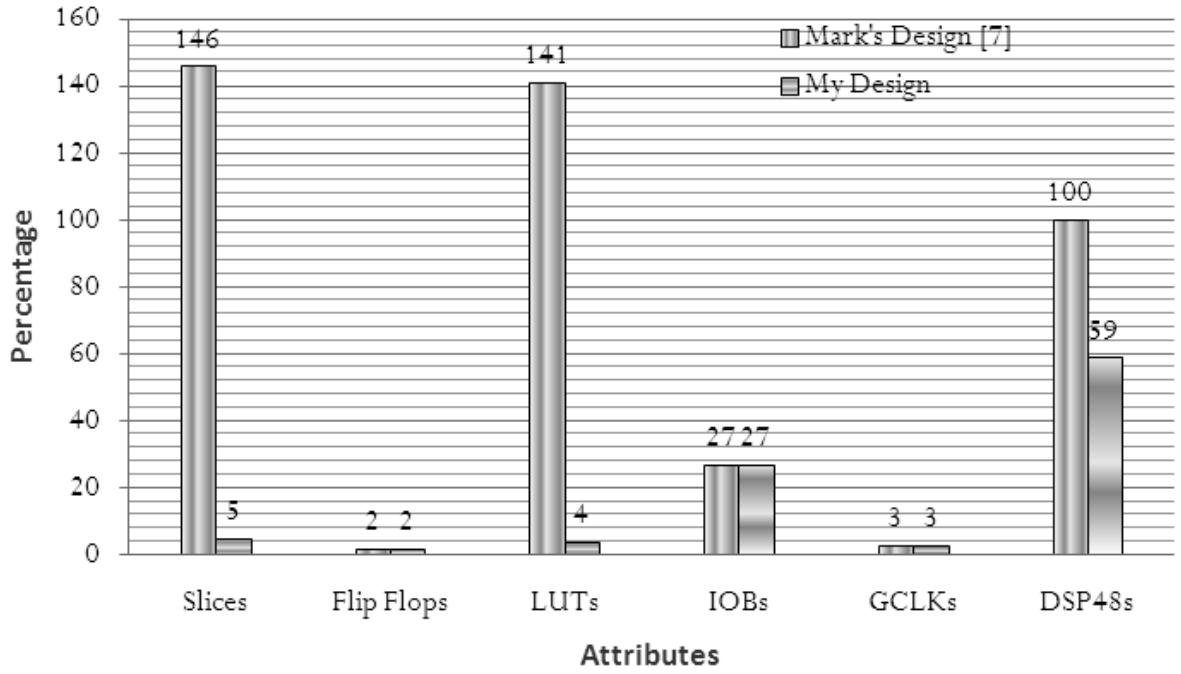

Figure 6-5: Graphical comparison of two implementations of 9-Tap FIR filter

10-Tap FIR Filter

Talbe 6-9 shows the device utilization of 10-tap FIR filter using multiplier designed by [7]

Table 6-9: Device utilization summary of 10-tap FIR filter implementation using multiplier in [7]

Logic Utilization	Used	Available	Utilization
Number of Slices	9829	6144	159%
Number of Slice Flip Flops	348	12288	2%
Number of 4 input LUTs	19010	12288	154%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	32	32	100%

Similarly, Table 6-10 shows the device utilization when the 10-tap FIR filter is implemented on FPGA using the multiplier that is described in the previous chapter. By comparing these two results it is clear that the 2nd implementation is far better than

the first one in terms of area utilization. There is a significant decrease in the area utilization when my multiplier is used.

Table 6-10 Device utilization summary of 10-tap filter using proposed multiplier			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	328	6144	5%
Number of Slice Flip Flops	397	12288	3%
Number of 4 input LUTs	543	12288	4%
Number of bonded IOBs	66	240	27%
Number of GCLKs	1	32	3%
Number of DSP48s	21	32	65%

If we look at the timing summary it is obvious that the second implementation is also better in terms of frequency. The implementation of the filter using the multiplier designed by [7] is running at a lower frequency (4.667MHz) while the other implementation using my multiplier design is running at higher frequency (131.356MHz). So there is a significant improvement in my design in terms of both frequency and the area. A complete comparison and analysis in terms of area of the two implementations is shown in Figure 6-6.

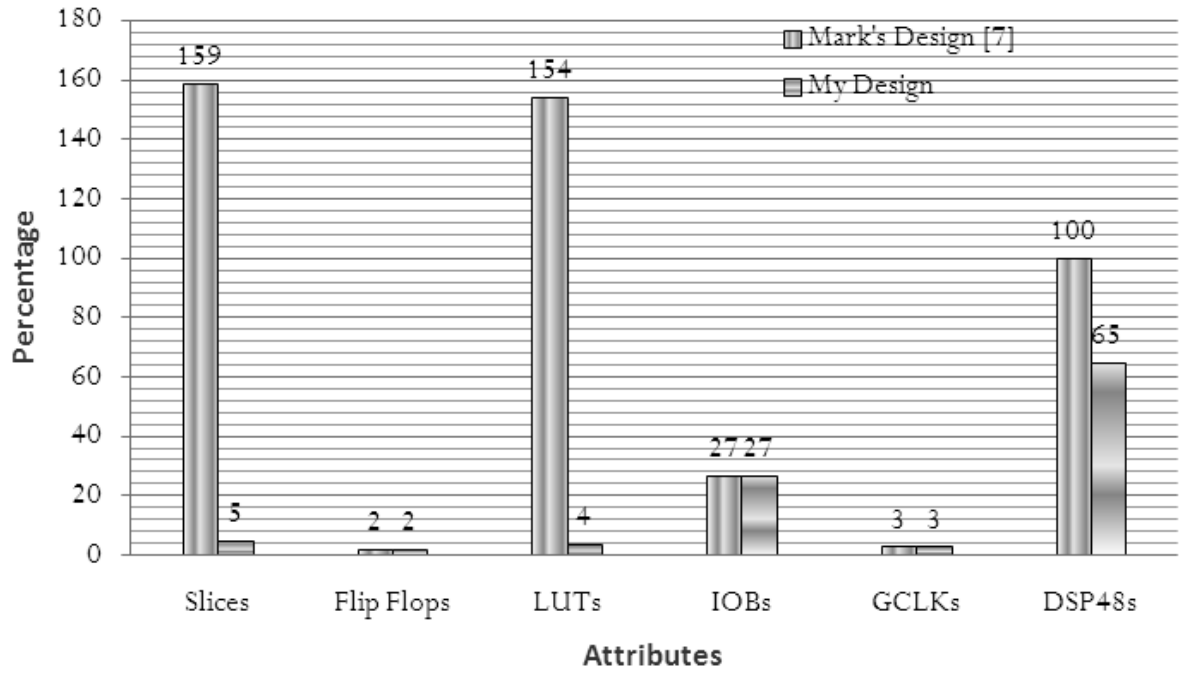

Figure 6-6: Graphical comparison of two implementations of 10-Tap FIR filter

Figure 6-7 shows the comparison of two implementations of FIF filters in terms of frequency rate obtained. These comparisons are of 3-Tap, 5-Tap, 7-Tap, 9-Tap and 10-Tap FIR filters. It is obvious from the graph that the implementations that used the proposed multipliers are far better than the other one.

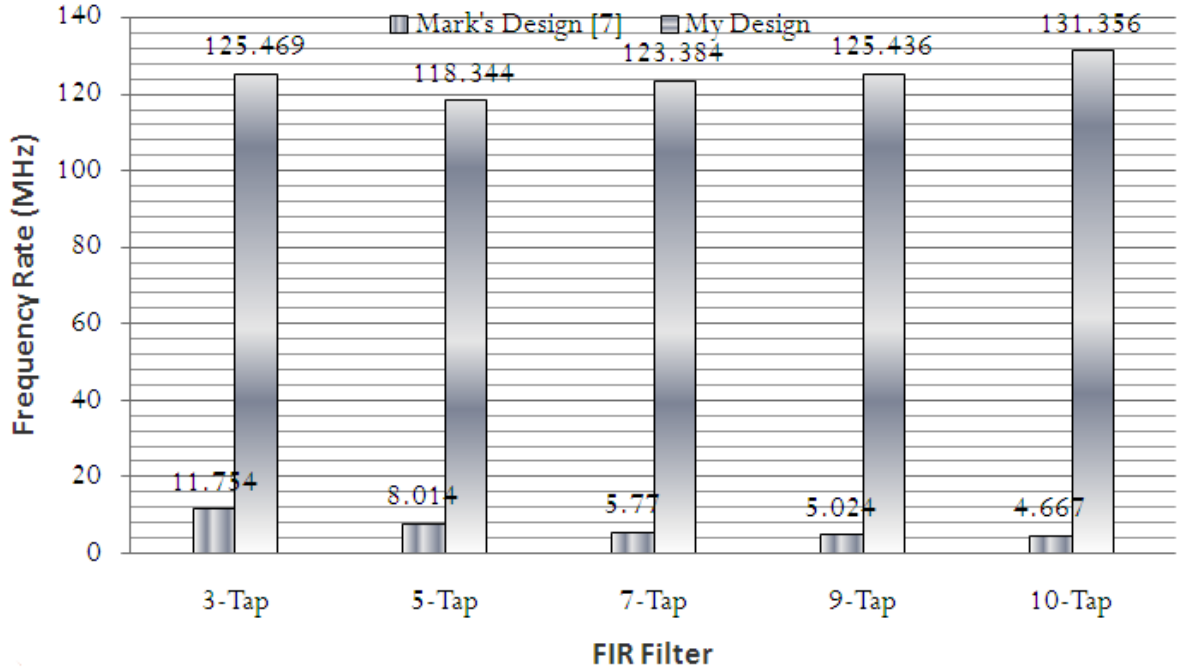

Figure 6-7: Graphical comparison of frequency rate obtained of two implementations of 3, 5, 7, 9 and10-Tap FIR filter

If we look at the percentage improvement obtained in each case than it is observed that the improvement in each case is almost the same. Percentage improvements obtained in each case is summarized in Table 6-11 and graphically shown in Figure 6-8.

Table 6-11: The percentage improvement obtained after the comparison of the two implementations of 5,7,9 & 10-Tap FIR filter

Attributes	Percentage Improvement (%)				
	3-Tap	5-Tap	7-Tap	9-Tap	10-Tap
Number of Slices	96.42	96.59	95.78	95.20	96.85
No. of Slice Flip Flops	0	0	0	0	0
Number of 4 input LUTs	96.33	96.47	96.73	97.16	97.40
Number of bonded IOBs	0	0	0	0	0
Number of GCLKs	0	0	0	0	0
Number of DSP48s	58	54.66	54	41	35
Frequency Rate	90.63	93.22	95.32	95.99	96.44

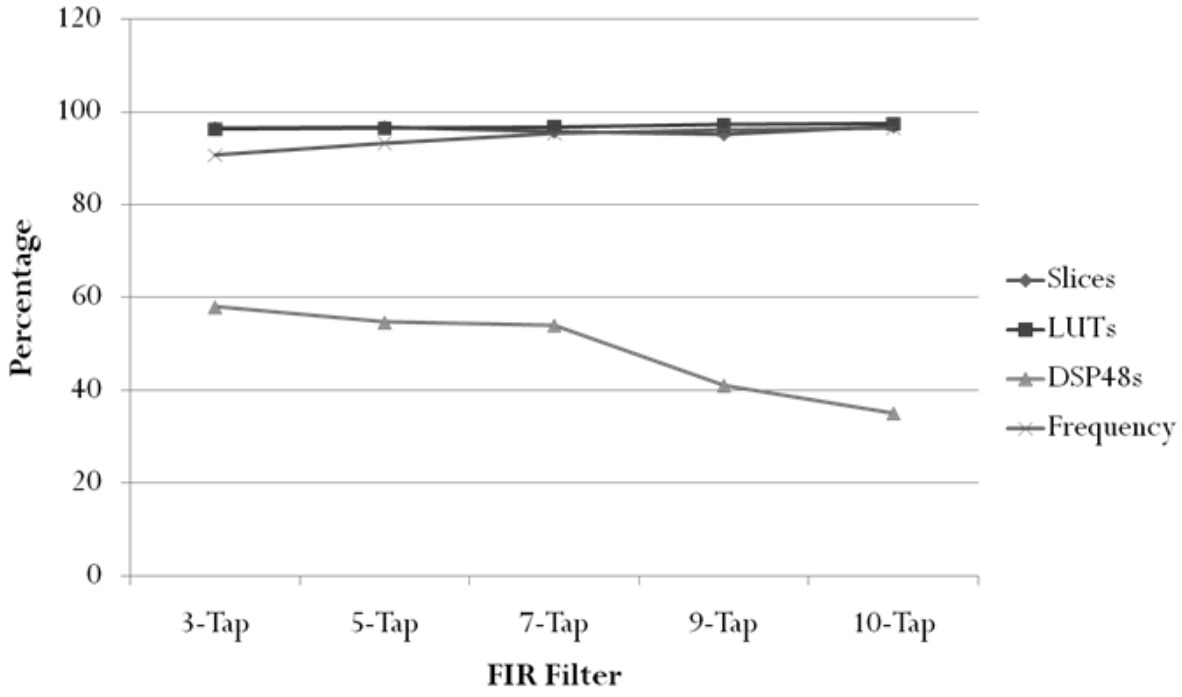

Figure 6-8: Percentage improvement obtained after the comparison of the two implementations of 3, 5,7,9 & 10-Tap FIR filter

Chapter 7

Conclusion and Future Work

The presented results lead to the conclusion that if the designer decides to use the methodology discussed above, they can get a significant improvement in terms of area utilization. They will b able to implement their designs using 32-bit IEEE standard format on FPGAs. However, best results can be obtained by utilizing the parallelism in implemented algorithms and by applying advanced synthesis methods based on decomposition. We have shown that IEEE single precision floating point arithmetic can be successfully implemented on FPGAs. Our implementations give respectable performance both in terms of area and frequency. The main objectives throughout our work were to minimize the area utilization for implementation of floating point algorithms, while at the same time keeping the speed of the operations at a reasonable level and maintaining IEEE 32-bit accuracy. The results presented above show that these requirements have been satisfied to a great extent; however, this does not mean that further improvements are not possible. The design can be pipelined to get more benefits from it. Furthermore, it can be designed in a way that the other formats can also be implemented. By changing the number of bits from 32 to other formats can also be useful. Analysis can also be made by using other devices instead of Xilinx's Virtex-4.

References

[1] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin and Christine Kitchen "An overview of FPGAs and FPGA programming; Initial experiences at Daresbury" *Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, UK* November 2006

[2] Mariusz Rawski, Bogdan J. Falkowski, and Tadeusz Łuba "Digital Signal Processing Designing for FPGA Architectures" *ELEC. ENERG. vol. 20, no. 3, December 2007, 437-459*

[3] Kamal Rajagopalan, Peter Sutton School of Computer Science and Electrical Engineering "A Flexible Multiplication Unit For An FFPGA Logic Block" *The University of Queensland Brisbane Queensland Australia*

[4] Per Karlstrom Andreas Ehliar Dake Liu "High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4" *Department of Electrical Engineering Linkping University*

[5] Nabeel Shirazi, A1 Walters, and Peter Athanas "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines" *In IEEE Symposium on FPGAs for Custom Computing Machines, pages 155-162, April 1995*.

[6] Loucas Louca, Todd A. Cook, William H. Johnson "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs" *Dept. of Electrical and Computer Engineering Rutgers University*

[7] Free Floating-Point Madness: Multiplier Mark E. Phair, Harvey Mudd College For Dr. David Harris, Harvey Mudd College May 14, 2002

[8] http://en.wikipedia.org/wiki/Field-programmable_gate_array

[9] U. Meyer-Baese, "Digital Signal Processing with Field Programmable Gate Arrays" *Berlin: Springer-Verlag, 2004.*

[10] M. Rawski, P. Tomaszewicz, and T. Łuba, "Logic synthesis importance in FPGA based designing of information and signal processing systems," *in Proc. of International Conference on Signal and Electronics Systems, Pozna´n, Poland, 2004, pp. 425–428.*

[11] Haynes, S.D., and Cheung, P.Y. "Configurable Multiplier Blocks for use within a FPGA", *IEEE Trans.Computers, Vol .3, No.1, 1998, pp 638-639*

[12] Xilinx Corporation Inc, "XtremeDSP for Virtex-4 FPGAs", 2008.

[13] IEEE Standards Board. "IEEE Standard for Binary Floating-point Arithmetic". *Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.*

[14] John L. Hennessy and David A. Patterson, "Computer Architecture A quantitative Approach", Second Edition. Morgan Kaufmann, 1996.

[15] M. Rawski, P. Tomaszewicz, H. Selvaraj, and T. Łuba, "Efficient implementation of digital filters with use of advanced synthesis methods targeted fpga architectures," *in Proc. of Eighth Euromicro Conference on Digital System Design (DSD 2005), Porto, Portugal, Aug. 2005, pp. 460–466.*

[16] Lee, H. and Flynn, M. “Coarse Grained Carry Architecture for FPGA”, *Proceedings of the ACM/SIGDA international symposium on FPGAs, Feb 10 2000, Monterey, CA.*

[17] Ramy Raafat, Amira Mohamed, Rodina Samy “A Decimal Fully Parallel and pipelined Floating Point Multiplier “*Electronics and Communication Department, Cairo University, Egypt.*

[18] Syed Shahzad Shah, Saqib Yaqub, and Faisal Suleman “Distributed Arithmetic for the Design of High Speed FIR Filter using FPGAs“ Chameleon Logics, #301, Kiran Plaza F-8 Markaz, Islamabad.