2011

Arithmetic with the Two-Dimensional Logarithmic Number System (2DLNS)

Mahzad Azarmehr
University of Windsor

Arithmetic with the Two-Dimensional Logarithmic Number System (2DLNS)

by

Mahzad Azarmehr

A Dissertation
Submitted to the Faculty of Graduate Studies through
the Department of Electrical and Computer Engineering in Partial Fulfillment
of the Requirements for the Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada
2011

Arithmetic with the Two-Dimensional Logarithmic Number System (2DLNS)

by

Mahzad Azarmehr

APPROVED BY:

D. Al-Khalili
Department of Electrical and Computer Engineering, Royal Military College

---

A. Edrisy
Department of Mechanical, Automotive and Materials Engineering, University of Windsor

---

R. Muscedere
Department of Electrical and Computer Engineering, University of Windsor

---

M. Khalid
Department of Electrical and Computer Engineering, University of Windsor

---

M. Ahmadi
Department of Electrical and Computer Engineering, University of Windsor

---

A. T. Alpas
Department of Mechanical, Automotive and Materials Engineering, University of Windsor

September 23, 2011

Declaration of Previous Publication

This thesis includes 2 original papers that have been previously published/submitted for publication in peer reviewed journals, as follows:

Thesis Chapter	Publication title	Publication status
Chapters 4 and 5	High-Speed and Low-Power Reconfigurable Architecture of 2-digit 2DLNS-based Recursive Multipliers	Published
Chapter 6	Low-Power Finite Impulse Response (FIR) Filter Design using Two-Dimension Logarithmic Number System (2DLNS) Representations	Submitted

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor. I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been

submitted for a higher degree to any other University or Institution.

Abstract

The ever increasing demand for low power DSP applications has directed researchers to contemplate a variety of potential approaches in different contexts. Since DSP algorithms heavily rely on multiplication, there are growing demands for more efficient multiplication structures. In this regard, using some alternative number systems, which inherently are capable of reducing the hardware complexity, have been studied. The Multi-Dimensional Logarithmic Number System (MDLNS), a multi-digit and multi-base extension to the Logarithmic Number System (LNS), is considered as an alternative to the traditional binary representation for selected applications. The MDLNS provides a reduction in the size of the number representation with a non-linear mapping and promises a lower cost realization of arithmetic operations with a reduced hardware complexity. In addition, using the recursive multiplication technique, which refers to the published multiplication algorithm that uses smaller multipliers to implement a larger operation, reduces the size of operands and corresponding partial additions. As part of this research, 2DLNS-based multiplication architectures with two different levels of recursion are presented. These architectures combine some of the flexibility of software with the high performance of hardware by implementing the recursive multiplication schemes on a 2DLNS processing structure. These implementations demonstrate the efficiency of 2DLNS in DSP applications and show out-

standing results in terms of operation delay and dynamic power consumption. We also demonstrate the application of recursive 2DLNS multipliers to reconfigurable multiplication architectures. These architectures are able to perform single and double precision multiplication, as well as fault tolerant and dual throughput single precision operations. Modern DSP processors, such as those used in hand-held devices, may find considerable benefit from these high-performance, low-power, and high-speed reconfigurable architectures. In the final part of this research work, recursive 2DLNS multiplication architectures have been applied to a FIR filter structure. These implementations show considerable improvement to their binary counterparts in terms of VLSI area and power consumption.

To whom made me believe that where there is a will, there is a way.

Acknowledgments

There are several people who deserve to be acknowledged for their generous contributions to this project. I would first like to express my sincere gratitude and appreciation to Dr. Majid Ahmadi, my supervisor, for his invaluable guidance throughout the course of this thesis work. I would also like to thank my committee members: Dr. Afsaneh Edrisy, Dr. Roberto Muscedere, Dr. Mohammed Khalid, and Dr. Dhamin Al-Khalili from the Royal Military College for reviewing my thesis and their constructive comments. Special thanks to Dr. Roberto Muscedere for his expert guidance and constant support throughout my study.

I am deeply grateful to my friend and partner Hassan Haftbaradaran for his love and constant support and encouragement. I also sincerely appreciate my family and friends for their support, help and friendship.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit
BIBO	Bounded Input-Bounded Output
CMOS	Complementary Metal-Oxide-Semiconductor
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FIR	Finite Impulse Response
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response (IIR)
LNS	Logarithmic Number System
LUT	Look Up Table
MAC	Multiply and Accumulate
MDLNS	Multi-Dimensional Logarithmic Number System
RALUT	Range Addressable Look Up Table
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SBD	Single Base Domain
TSMC	Taiwan Semiconductor Manufacturing Company.
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
XOR	Exclusive OR

Chapter 1

Introduction

1.1 Introduction

In most real-time Digital Signal Processing (DSP) applications, high performance is a prime target. Here, performance may be interpreted as a combination of speed, power consumption, precision, and VLSI area efficiency.

The dynamic range of the data usually used in DSP may be best suited to floating-point representations, but the dynamic range of floating-point comes to the cost of lower precision and increased complexity.

The Logarithmic Number System (LNS) is one alternative approach to conventional floating-point for performing real arithmetic, enabling multiplication, division, exponentiation, and square root in a fixed-point representation. The use of 16-bit LNS arithmetic was originally proposed as an alternative to 16-bit fixed-point digital filtering applications by Kingsbury and Rayner in 1971 [1]. In LNS, logarithmic bit rate compression also significantly reduces the interconnections, and storage requirements [2]. However, these advantages come at the

cost of complicated and inexact addition and subtraction, as well as the need to convert between binary and logarithmic formats [3].

Using Analog to Digital Converters (ADC), that benefit from the exponential current-voltage characteristics of bipolar transistors or MOS transistors operating in weak inversion, provides a natural logarithmic input to an LNS system.

Conversion from a binary number system traditionally is accomplished by using look-up tables. These tables should cover the entire dynamic range in order to maintain conversion accuracy; this requirement can make the size of the tables unreasonably large. Therefore, researchers have been encouraged to pursue some alternative solutions. A very early approach by Mitchell finds an approximate logarithm of a binary number by a piece-wise fitting approach using a process of shifts and counting. The logarithms are accurate for powers of 2 inputs with a straight line approximation between these values for other inputs. This causes an error which can be reduced by a variety of correction schemes [4, 5].

Addition and subtraction are the bottleneck of every LNS circuit, for which there are many implementation techniques that trade off area, latency and accuracy. The central difficulty in implementing logarithmic addition and subtraction is the need to approximate two non-linear functions which have traditionally been performed using look-up tables. The algorithms presented in the literature that attempt to reduce the size and the complexity of these tables, usually cause additional overhead in critical path or hardware costs [6]. For larger word lengths, it is not efficient to store the entire range of values, and so interpolation techniques have been developed to reduce storage size using several smaller tables with some extra computation required. This procedure tends to introduce additional error, and the entire process is time consuming [7]. Efficient approximations of a non-linear function using small look-up tables, suggests the use of a Taylor series approach. In [6], a first-order Taylor series approximation is used for interpolation, with techniques being applied to eliminate the need for interpolation in singularity regions and the use of a table of derivatives.

In this regard, two techniques were presented in [8]; linear approximation by using different-sized approximation intervals and non-linear compression, which reduces table spaces by sorting the difference between the exact value of the function and a linear approximation across smaller regions where the function is more non-linear. In [9], the error caused by interpolation is simultaneously evaluated and accumulated into the result; thereby the error is corrected with a delay of one extra carry save adder stage.

Coleman et al. were the first research group to start with the design and implementation of an LNS Arithmetic and Logical Unit (ALU) [9]. They examined some practical DSP applications and then applied it to an LNS Microprocessor [10]. The essence of their work was to develop an efficient algorithm for logarithmic addition and subtraction. It has been claimed that their European Logarithmic Microprocessor (ELM), as the first microprocessor device to use the LNS instead of floating-point for real arithmetic, delivers approximately two fold improvements in speed and accuracy [11]. The main idea in ELM is that given an LNS addition and subtraction procedure capable of operating with speed and accuracy at least equivalent to that of floating-point, the system can offer an overall advantage for arithmetic operations [12].

Another LNS ALU design has been presented in [13]. This design takes benefit from an interpolator that accepts both positive and negative arguments with a new addition scheme which reduces the critical path and provides a shorter latency. The three methods of interpolation, multipartite tables and co-transformation for LNS addition and subtraction have been compared in [14]. In this research work, a library for a novel version of co-transformation was also proposed that provides more accurate results at the cost of an increase in the latency of the hardware.

There are not many publications of LNS research work that covers the 64-bit floating-point case. In [15] two algorithms have been used to implement a 64-bit LNS arithmetic unit within a conventional microprocessor. One algorithm uses higher-order Taylor series and interpolation, and the other one is based on a CORDIC engine and the authors show that

the latter design is superior.

Considering all the above, it can be concluded that LNS architectures are perfectly suited for low-power and low-precision DSP problems. The major drawback of LNS are the needs to implementing efficient techniques for data conversion and logarithmic addition and subtraction. This will be a major challenge, particularly in double-precision data widths.

The Multi-Dimensional Logarithmic Number System (MDLNS), which has some similar properties to the classical LNS, provides more degrees of freedom by the virtue of having multiple orthogonal bases, and the added advantage of the use of multiple digits. The MDLNS provides a larger dynamic range compared to binary representation with more precise mapping of binary data. Therefore, it has found initial applications in the implementation of special DSP systems, where the parallel operations on independent bases greatly reduces both the hardware and the connectivity of the architecture [16].

In MDLNS the conversions will be performed using some special look-up tables, which are considerably smaller than their LNS counterparts. By combining MDLNS architectures with "Recursive Structures" [17], both conversion and MDLNS arithmetic are efficiently accomplished, even for double-precision data widths.

In this research work new and efficient techniques to perform arithmetic operations on 2DLNS representations will be presented. The concept of recursive multiplication has been applied to 2DLNS structures, resulting in efficient digital multipliers in terms of VLSI area and power consumption. The application of these structures to $reconfigurable\ architectures$ have been implemented. The term $reconfigurable\ architecture$ in this work refers to the capability of hardware to be modularly partitioned for an optimum operation based on the setting regulated by the software. In this way, some parts of the circuit

are shut down when they are not in use, which leads to lower power consumption and, in special cases, some times less latency and better performance. Finally, a 2DLNS Finite Impulse Response (FIR) filter is implemented using the 2DLNS-based recursive multiplication structure. These applications demonstrate the superiority of 2DLNS designs in terms of VLSI area and power consumption.

In order to maintain simplicity in our designs, we have restricted our representations to 2 orthogonal bases (2DLNS), which are sufficient to provide desired precision in most applications [18].

1.2 Thesis Objectives

The work presented in this thesis conforms to the following objectives:

1. Developing new and more efficient 2DLNS-based multiplication schemes on 2DLNS platform
2. Implementing reconfigurable architectures relying on 2DLNS multiplication due to its structural modularity
3. Examining potential applications in multiplication intensive algorithms in a variety of data widths

1.3 Thesis Organization

This thesis is organized into seven chapters and one appendix. Chapter 2 provides background material on MDLNS by covering the representation, its concept and properties, and conversion techniques. Chapter 3 consists of the details of 2DLNS arithmetic, including developed algorithms for addition, subtraction , and multiplication. Chapter 4 contains the

design flow of two recursive 2DLNS-based multiplication architectures. These structures will be used as building blocks in reconfigurable multiplication architectures which are explained in Chapter 5. As another DSP application, a FIR filter is designed and implemented in Chapter 6. This chapter also provides a comparison between binary, 2DLNS-based, and recursive 2DLNS-based designs. The 2DLNS-based designs have been described in detail, including their architectures, hierarchy of components, and interfaces. The functional behavior of the components are described at the Register Transfer Level (RTL). The chapter continues with the synthesis of the HDL modules, and the final results are illustrated. Chapter 7 concludes this thesis and provides recommendations for future work. The HDL codes of one of the designs presented in Chapter 6 and corresponding auxiliary packages are attached in Appendix A.

Chapter 2

The Multi-Dimensional Logarithmic Number System

In this chapter, the most important concepts in MDLNS will be reviewed, including data representation, and mathematical operations. Conversion between MDLNS and binary representations is also discussed.

2.1 Introduction

In the area of DSP a continuing demand exists for more efficient processing algorithms. Modern processing structures should preferably be more compact, high speed, and low power. DSP systems manipulate signals as a sequence of numbers, and usually require massive arithmetic computations to perform algorithms. These kinds of applications are mostly dominated by multiplication. Multipliers are fundamental arithmetic units in modulation, filtering and in the Discrete Fourier Transform (DFT), and consume a considerable

amount of hardware and power.

An active area of research is therefore to find techniques for performing multiplication with lower hardware complexity and attendant power reduction. Furthermore, some DSP applications require a more precise mapping of data, particularly for smaller values, such as less than 1 coefficients in an FIR filter, which requires either a large integer (binary) word, or the use of a more flexible floating point arithmetic. The latter represents a non-linear approach to number representation with a finite word length, and there are alternatives to such an approach for dedicated applications [1, 19–23]. One of these alternatives, the classic Logarithmic Number System (LNS), has been extensively studied for dedicated applications such as DSP. It allows representation of a much larger dynamic range than the equivalent number of bits in integer binary, along with the property of more accurate representations of smaller values within the dynamic range. The LNS also simplifies multiplication, division and exponentiation operations, which incur a high hardware cost in the case of using binary arithmetic, and so are quite suited to low-power and low-precision DSP problems [1, 20, 21].

In some recent signal processing research works, new design approaches have been directed to provide a greater degree of modularity and parallelism compared to traditional algorithms. In this regard, the Multi-Dimensional Logarithmic Number System (MDLNS) may be considered as a multiple-digit representation of an LNS using more than one orthogonal base. Arnold et al. [22] were the first to consider the redundancy of having more than one digit in a single-base LNS. The MDLNS is a multiple-digit extension to a multiple-base logarithmic representation, which increases the redundancy even further. Although MDLNS does not always provide a reduction in the size of the number representation, due to the possibility of multiple digit representation, it definitely promises a lower cost realization of arithmetic operations. Since operation on different bases and different digits are independent, the orthogonal nature of the parallel base computations and the multi-digit extensions of the MDLNS representations decrease the complexity of computations. The

reduced hardware complexity, simplified arithmetic operations, and the non-linear nature of the representation, make MDLNS suitable for some DSP applications [23], particularly those which rely heavily on multiplication. Since the MDLNS was introduced in 1996 [24], it has been increasingly used in some DSP and cryptography applications [25–27].

2.2 Representation

The MDLNS is obtained from an index calculus implementation of the double-base number system [26]. A representation of the real number, X, in the form:

$$X = \sum_{i=1}^{n} s_i \prod_{j=1}^{b} p_j^{e_j{}^{(i)}} \tag{2.1}$$

where $s_i \in \{-1, 0, 1\}$, p_j are real numbers, and $e_j{}^{(i)}$ are integers, is called a multi-dimensional n-digit logarithmic (MDLNS) representation of X, where b is the number of bases used (at least two). In order to have hardware consistency, the first base always will be assumed to be 2 [28] and the other bases can be optimized in accordance with the specific application characteristics. Although the values of these bases do not affect the hardware structure and just specify the contents of the look-up tables, they contribute to provide the desired accuracy of data mappings as well as precision of computational results. The work of this thesis is restricted to 2DLNS representations where only two bases ($b = 2$) are used. The second base, that we refer to as the optimal base, is the value which provides the most precise mapping of its equivalent binary data over the entire dynamic range. A previously developed algorithm in [18] operates based on an exhaustive search, where a range of data is encoded with different options for the size of exponents to find a second base with the minimum overall data representation error. Previously published work [18] also shows that there is a very substantial reduction in the hardware complexity as well as more precise mapping of binary data for the 2-digit case.

The parallel calculations over two bases and digits are accomplished concurrently and the

logarithmic properties of the MDLNS allow for a reduced-complexity multiplication. The use of more than one base facilitates more precise mapping of binary data. This leads to a dramatic reduction in the size of the exponents, and consequently to hardware savings [27]. MDLNS is a redundant number system, the majority of real numbers in the dynamic range of the number system have more than one error-free representations. This redundancy can be useful in order to choose the most suitable representation for a specific application. Particularly, the redundant values of 1 in MDLNS can be used as a coefficient to decrease the values of exponents, which is an easy way to prevent overflow in calculations and consequently to save hardware.

2.3 Mathematical Operation

Every 2DLNS digit is realized in hardware with a triple, $\{s_i, a_i, b_i\}$, where s_i is the sign bit, a_i, and b_i are the exponents of the binary and non-binary bases respectively. In general, the second base, or the optimal base, is shown with *D*. *D* is a real number, but not a multiple of 2, which is chosen in a way to provide a certain accuracy for each specific application. Thus, a number *x*, can be represented in 2DLNS as:

$$x = \sum_{i=1}^{n} s_i \cdot 2^{a_i} \cdot D^{b_i} \tag{2.2}$$

where $b_i \in \{-2^{R-1}, ..., 2^{R-1} - 1\}$ and *R* is the number of bits needed to represent *b* in binary. The number of required bits to represent the binary exponent is usually shown by *B*. Although the range of *a* is self limiting, there is also some restriction based on *B* to specify the range of *a* in binary, which is $a_i \in \{-2^{B-1}, ..., 2^{B-1} - 1\}$ [29].

Exponentiation, multiplication and division are the simplest arithmetic operations in 2DLNS. Exponentiation is simply realized through consecutive shifts on exponents. The corresponding equations for multiplication and division, given a single-digit 2DLNS represen-

tation of $x = \{s_x, a_x, b_x\}$ and $y = \{s_y, a_y, b_y\}$, are:

$$x \cdot y = \{s_{x.y}, a_x + a_y, b_x + b_y\} \tag{2.3}$$

$$x/y = \{s_{x/y}, a_x - a_y, b_x - b_y\} \tag{2.4}$$

These equations show that single-digit 2DLNS multiplication / division can be implemented in hardware using two independent binary adders / subtractors and simple logic for the sign computation. For example, in the case of single-bit sign representation, 0 for positive and 1 for negative numbers, this logic is simply an XOR gate.
Unfortunately, addition and subtraction operations are not as simple as multiplication and division operations. They must be handled through a set of identities and look-up tables. The identities are:

$$\begin{aligned} 2^{a_x} \cdot D^{b_x} + 2^{a_y} \cdot D^{b_y} &= (2^{a_x} \cdot D^{b_x}) \cdot (1 + 2^{a_y - a_x} \cdot D^{b_y - b_x}) \\ &\simeq (2^{a_x} \cdot D^{b_x}) \cdot \Phi(a_y - a_x, b_y - b_x) \end{aligned} \tag{2.5}$$

$$\begin{aligned} 2^{a_x} \cdot D^{b_x} - 2^{a_y} \cdot D^{b_y} &= (2^{a_x} \cdot D^{b_x}) \cdot (1 - 2^{a_y - a_x} \cdot D^{b_y - b_x}) \\ &\simeq (2^{a_x} \cdot D^{b_x}) \cdot \Psi(a_y - a_x, b_y - b_x) \end{aligned} \tag{2.6}$$

The Φ and Ψ functions may be precomputed for all possible values of a_x, a_y, b_x, and b_y, and as 2DLNS values stored in look-up tables. Although, some algorithms have been developed to perform MDLNS addition / subtraction with reduced-size special look-up tables [29], which is applicable when the results required to be used in a further MDLNS calculation, it is still more practical to convert the MDLNS numbers to binary and perform the addition and subtraction using binary representations, as will be shown in the next chapter.

2.4 Conversion

Most often DSP data are available in a binary representation and most system requirements will also need the output data to be transferred to the external world in binary format. Therefore, some methods for converting data between binary and 2DLNS are required. Furthermore, as mentioned above, in order to perform logarithmic addition and subtraction, it is more practical to convert logarithmic numbers to binary, and perform these operations using a binary representation. The conversion techniques, which have been developed in [16], use a special memory device referred to as a Range Addressable Look-Up Table (RALUT). Conceptually, standard look-up tables use an address decoder to match an input to a series of unique address values. A RALUT differs from the classic look-up table by changing the address decoder system to provide a match on a range of values rather than exact values. Logically, the input address is compared within the range of two neighbour addresses [29].

To perform conversion to binary, in a single-digit 2DLNS number which is in this form:

$$X = s \cdot 2^a \cdot D^b \tag{2.7}$$

b is used as an index address to a RALUT to find a floating-point representation for D^b:

$$D^b = \mu(b) \cdot 2^{\epsilon(b)} \tag{2.8}$$

Here, $\mu(b)$ is the mantissa ($1 \leq \mu(b) < 2$) and $\epsilon(b)$ is the exponent. This way, the final floating-point representation of *X* is:

$$X = s \cdot \mu(b) \cdot 2^{(a+\epsilon(b))} \tag{2.9}$$

For a two-digit 2DLNS to binary conversion, both 2DLNS digits are converted separately using the single-digit method, and their results are accumulated to produce the final binary representation.

For the reverse conversion, the input to the RALUT is the mantissa, $\mu(b)$, and the outputs are b and $\epsilon(b)$. Here, the exponent remains as an output. The conversion of $|X|$ to a

mantissa is easily achieved in hardware with a conditional feedback bit-shifter and counter, or a priority encoder [29]. When the number of performed shifts, *shifts*, obtained, it is used to generate the binary exponent:

$$a = \textit{shifts} - \epsilon(mantissa) \tag{2.10}$$

For the binary to 2-digit 2DLNS conversion, four methods have been developed [16].

1. The *Quick method* chooses the first-digit nearest to the target, and generates the second-digit to reduce the error, a simple greedy algorithm.

2. The *High/Low method* chooses the two nearest approximations to the target as the first-digits, generates two associated second-digits for the error, and selects the combination with the smaller error.

3. The *Brute-Force method* operates by selecting the combination with the smallest error, but it uses all possible mantissa of D^b as the first-digits instead of just one (*Quick*) or two (*High/Low*).

4. The *Extended-Brute-Force method* improved upon the *Brute-Force method* by using first-digit approximations above 2.0 and below 1.0.

Each method ranges from simple implementations and fairly accurate approximations to difficult implementations and very accurate approximations. All of these methods have been implemented in fully parametrized Verilog HDL codes, which can be found in [29]. This reference also includes the serial and parallel implementations of the converters.

In this thesis, the modified and improved version of the *High/Low* converters have been used. Furthermore, in order to minimize both power and area, the serial implementations of this converter have been applied to the various architectures.

Chapter 3

2DLNS Arithmetic

In this chapter, we discus the main 2DLNS arithmetic operations, including addition, subtraction, multiplication, and division. In this regard, their implementation algorithms and hardware realizations will be explained.

3.1 2DLNS Addition/Subtraction

Considering single-digit representation for both operands, similar to LNS, 2DLNS addition and subtraction are computed by:

$$\begin{aligned} 2^{x_1} \cdot D^{x_2} + 2^{y_1} \cdot D^{y_2} &= (2^{x_1} \cdot D^{x_2}) \cdot (1 + 2^{y_1 - x_1} \cdot D^{y_2 - x_2}) \\ &\simeq (2^{x_1} \cdot D^{x_2}) \cdot \Phi(y_1 - x_1, y_2 - x_2) \end{aligned} \tag{3.1}$$

$$\begin{aligned} 2^{x_1} \cdot D^{x_2} - 2^{y_1} \cdot D^{y_2} &= (2^{x_1} \cdot D^{x_2}) \cdot (1 - 2^{y_1 - x_1} \cdot D^{y_2 - x_2}) \\ &\simeq (2^{x_1} \cdot D^{x_2}) \cdot \Psi(y_1 - x_1, y_2 - x_2) \end{aligned} \tag{3.2}$$

In these equations Φ and Ψ are functions of the difference of corresponding exponents. These functions may be precomputed and stored in some memory units such as look-up tables. Since these tables should include the results for all possible values of y_1, x_1, y_2, and x_2, they may become very large in size. Therefore, 2DLNS addition and subtraction are considered costly functions since they require LUTs whose sizes depend exponentially on the bit widths of the 2DLNS indices. Although converting 2DLNS data to binary for addition is a solution, when an application is mostly dominated by multiplication operation, it is preferred to continue processing in 2DLNS domain. An algorithm has been developed in [29] to perform single-digit 2DLNS addition and subtraction. The main idea in this algorithm is using the so called Slide Rule method, which is finding a multiplying factor for one of the addends, in such a way that the multiplication give the same result as addition or subtraction. By extending the Slide Rule method to operate in single-digit 2DLNS operands:

$$\begin{aligned} 2^{x_1} \cdot D^{x_2} + 2^{y_1} \cdot D^{y_2} &= 2^{x_1} \cdot D^{x_2} \cdot 2^{z_1} \cdot D^{z_2} \\ 1 + 2^{y_1 - x_1} \cdot D^{y_2 - x_2} &= 2^{z_1} \cdot D^{z_2} \end{aligned} \tag{3.3}$$

$$\begin{aligned} 2^{x_1} \cdot D^{x_2} - 2^{y_1} \cdot D^{y_2} &= 2^{x_1} \cdot D^{x_2} \cdot 2^{w_1} \cdot D^{w_2} \\ 1 - 2^{y_1 - x_1} \cdot D^{y_2 - x_2} &= 2^{w_1} \cdot D^{w_2} \end{aligned} \tag{3.4}$$

The problem is that z_1, z_2, w_1, and w_2 are not easily derivable. These values can be computed for each pair of y_1, x_1, y_2, and x_2 and stored in a look-up table. The LUT input and output number of bit widths are $1 + (B+1) + (R+1)$ and $1 + B + R$ and again it can

be very large.

To reduce the size of the LUT, a direct mapping between the representations of a single-digit 2DLNS and LNS is derived. This process starts with a single-base domain conversion for both operands. At first, each operand is represented by a power of 2, similarly to classic LNS, but this time the exponent is a real number:

$$2^v = 2^{x_1} \cdot D^{x_2} \tag{3.5}$$

where the real number v is single-base domain index:

$$v = x_1 + (x_2 \times \log_2 D) \tag{3.6}$$

By finding an appropriate way to represent and process the real exponents in hardware, again multiplying two operands leads to a simple addition. To minimize the hardware realization of v and all connecting circuits, v should be represented in an integer form. In order to make v an integer, we find an integer m so that $\log_2 D \times m$ is near an integer:

$$v \cdot m = x_1 \cdot m + (x_2 \times \log_2 D) \cdot m \tag{3.7}$$

Once m has been chosen, an efficient hardware implementation is required to convert single-digit 2DLNS indices into a single-digit base index. When data are converted to single base format, again by using the Slide Rule method, we will have:

$$\begin{aligned} D^x + D^y &= D^x \cdot D^z \\ D^z &= 1 + D^{y-x} \\ z &= \log_D(1 + D^{y-x}) \end{aligned} \tag{3.8}$$

and addition in single base domain will be obtained from:

$$\begin{aligned} 2^{x/m} \cdot 2^{z/m} &= 2^{x/m} + 2^{y/m} \\ 2^{z/m} &= 1 + 2^{(y-x)/m} \\ z &= m \cdot \log_2(1 + 2^{(y-x)/m}) \end{aligned} \tag{3.9}$$

The multiplying factor ($2^{z/m}$) is calculated from $1 + 2^{(y-x)/m}$ for all the possible combinations of addends for a particular m. It is worth mentioning that not all the possible addends need to be included into the table, because any multiple of 2 applied to both addends will also apply to the sum; furthermore, the cases with one addend considerably larger than the other one can also be omitted. Again, using RALUTs, the size of corresponding look-up tables can be kept reasonably small. The Fig. 3.1 shows the block diagram of a 2DLNS adder. In this diagram, the greater exponent is assigned to x.

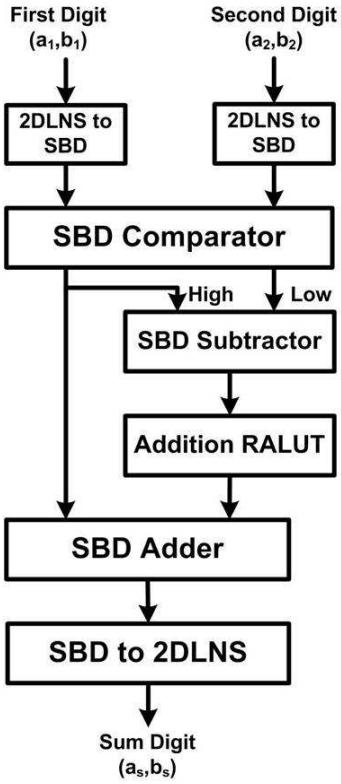

Figure 3.1: The 1-digit 2DLNS Adder

Similarly for subtraction, using the Slide Rule method, we will have:

$$D^x + D^y = D^x \cdot D^w \tag{3.10}$$

$$D^w = 1 - D^{y-x}$$

$$w = \log_D(1 - D^{y-x})$$

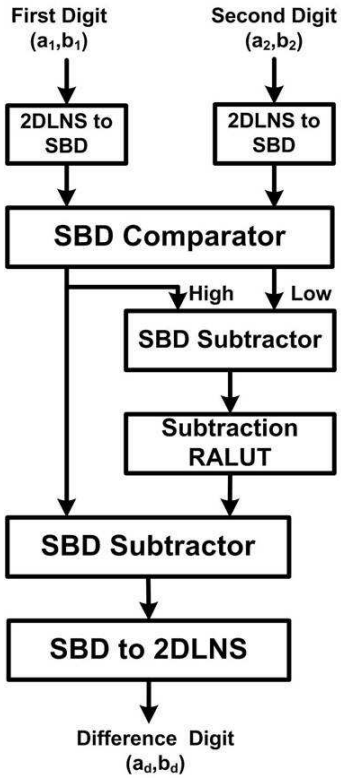

Figure 3.2: The 1-digit 2DLNS Subtractor

Subtraction in single-base domain will be:

$$\begin{aligned} 2^{x/m} \cdot 2^{w/m} &= 2^{x/m} - 2^{y/m} \\ 2^{w/m} &= 1 - 2^{(y-x)/m} \\ w &= m \cdot \log_2(1 - 2^{(y-x)/m}) \end{aligned} \tag{3.11}$$

Once the perfect addition table is found (the correct m), the subtraction table may be generated using the same m. With the same assumption of assigning the greater exponent to x, Fig. 3.2 shows the block diagram of a 2DLNS subtractor.

The 2DLNS addition / subtraction are realized in hardware through special RALUTs. These structures have been explained in [29] in more details.

3.2 2DLNS Multiplication

As mentioned before, every 2DLNS number is recognized in hardware with its sign and its first and second base exponents. The 2DLNS multiplier can be implemented in hardware using two independent small binary adders, which reduces the hardware cost of implementation even further.

As Fig. 3.3 shows, in a 1-digit 2DLNS multiplier, two parallel adders determine the corresponding exponents of the product. Since 1-bit sign representations [18] are assumed, 0 for positive and 1 for negative signs, an XOR gate is used to generate the sign of the product. As it can be seen from Fig. 3.3, the both input and output are in 2DLNS representations.

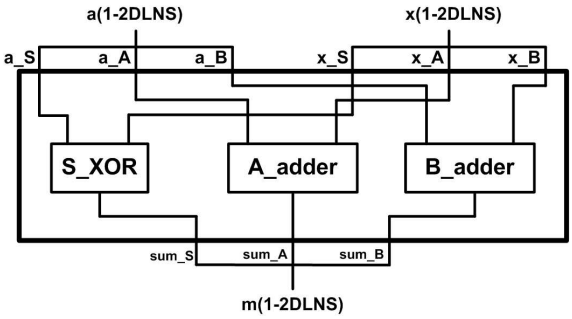

Figure 3.3: The 1-digit 2DLNS Multiplier

This is the fundamental block in a 2-digit 2DLNS multiplier structure. Having 2-digit numbers, each number is represented with a summation of 2 digits, which in hardware is shown with a set of two signs, and first and second base exponents. Therefore, in a 2-digit 2DLNS multiplication operation, there are four partial products that should be summed to form the final result. Now, there are two options to accomplish these additions, using either binary adders or 2DLNS adders. The type of these adders specifies the format of the output, as we will see in the following sections.

3.2.1 2DLNS Multiplier with Binary Addition

For this architecture, as shown in Fig. 3.4, each of partial products is converted to a binary representation for the final summation. Since the binary adders act on two operands, two stages of adders are required.

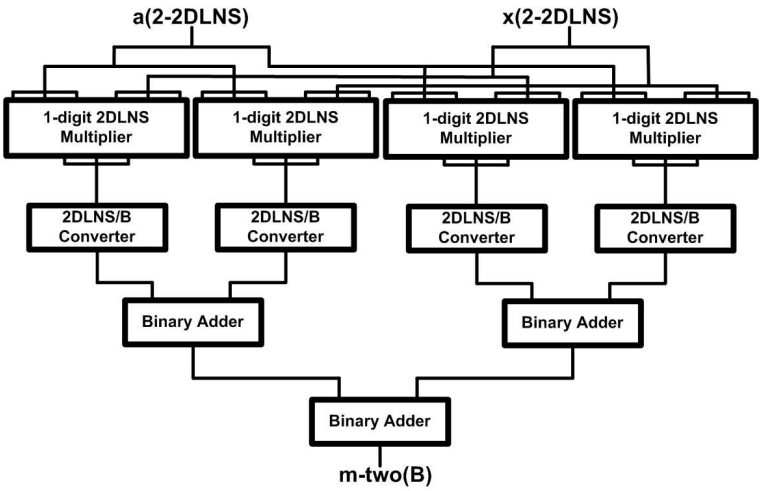

Figure 3.4: The 2-digit 2DLNS Multiplier with Binary Adders

This way, the final result is produced in binary format. Therefore, it may be more suitable for the last stage of the hardware structures.

3.2.2 2DLNS Multiplier with 2DLNS Addition

In this architecture, two stages of 2DLNS adders are used to add the partial products. The output of 2DLNS adders is also in 2DLNS format, which may be more appropriate for the multiplication dominant architectures. However, as it is shown in Fig. 3.5, a 2DLNS / Binary converter has been added to the end of this structure to produce the final product in binary format. In such a way, a fair comparison between two architectures will be accomplished.

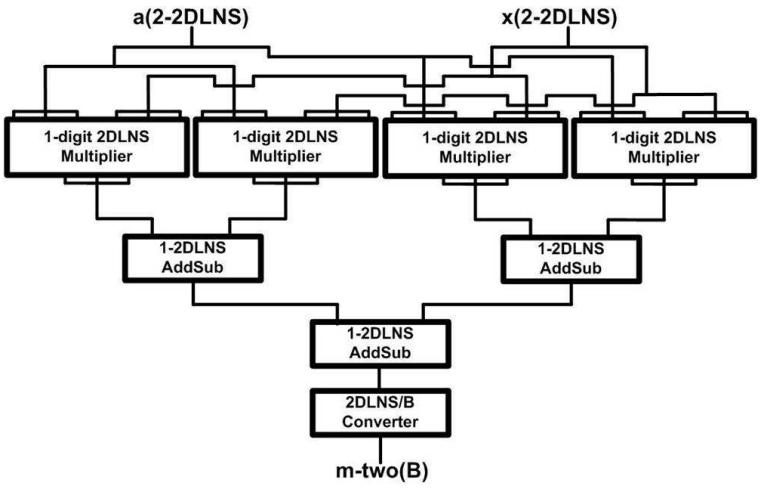

Figure 3.5: The 2-digit 2DLNS Multiplier with 2DLNS Adders

3.2.3 Synthesis Results and Comparison

In order to compare the hardware characteristics of 2-digit 2DLNS multipliers, two architectures have been implemented and synthesized using Synopsys Design Compiler in STMicroelectronics CMOS 90nm technology; one with binary adders and the other with 2DLNS adders to sum the partial products. Since the size of RALUTs used for conversion and 2DLNS addition are dependent on the size of the operands, these architectures are synthesized for different word widths. The input binary data will be firstly converted to 2-digit 2DLNS values. The size of these converters compared to the size of the architectures are also included into the table of synthesis results. Furthermore, the architecture with the binary adder is inherently a combinational circuit, but in order to make a comparison possible, a registered version of this multiplier has been realized and a clock frequency signal has been added to this structure. To examine the effect of timing constraint on the speed of the each structure, both architectures have been synthesized twice, once without a timing constraint and then with applying a clock frequency. Here, the goal is to find out whether one of these architectures is dominant in terms of performance figures for every word length.

3.2.3.1 8-bit by 8-bit Multiplier

Table 3.1 shows the synthesis results for 8-bit operands.

8×8 bit Multiplier	Binary Addition	2DLNS Addition	Binary Addition	2DLNS Addition
Clock Frequency (MHz)	-	-	250	250
Overall Area $(\mu m)^2$	17,328	29,735	18,457	33,930
Dynamic Power (mW)	0.169	0.340	0.721	1.740
Data Arrival Time (ns)	9.76	6.40	3.70	3.76
Data Converters Area	84.00%	40.70%	86.04%	40.16%

Table 3.1: 8×8 bit Multiplier Synthesis Results

As this table shows, the architecture with 2DLNS adders consume $70\% \sim 84\%$ more area and $2 \sim 2.4$ times more power. Without timing constraint, the architecture with binary adders is about 50% slower, while with applying a clock frequency of 250 MHz, the delays are almost equal. The number of binary converters in the architecture with binary adders are quadrupled, but their sizes are smaller. However, the main area of the binary structures are consumed by the converters. Meanwhile, the 2DLNS adders are the most area consuming components in the 2DLNS architectures.

3.2.3.2 16-bit by 16-bit Multiplier

In this case, as Table 3.2 shows, the architecture with 2DLNS adders consume $62\% \sim 68\%$ more area and $2.3 \sim 2.7$ times more power. Again, the binary architecture is about 70% slower without any timing constraint, but with applying a clock frequency of 250 MHz both architectures have the same speed.

16×16 bit Multiplier	Binary Addition	2DLNS Addition	Binary Addition	2DLNS Addition
Clock Frequency (MHz)	-	-	250	250
Overall Area $(\mu m)^2$	48,101	77,753	56,947	95,625
Dynamic Power (mW)	0.254	0.587	1.300	3.520
Data Arrival Time (ns)	16.07	9.28	3.75	3.74
Data Converters Area	89.25%	34.74%	91.15%	31.86%

Table 3.2: 16×16 bit Multiplier Synthesis Results

3.2.3.3 32-bit by 32-bit Multiplier

Table 3.3 shows the synthesis results for 32-bit operands. The architecture with 2DLNS adders, consume $23\% \sim 135\%$ more area and $5.8\% \sim 9.8\%$ times more power. On contrary to the previous data widths, the delays are equal for both timing conditions. However, the maximum clock frequency for both architectures is less than 250 MHz.

32×32 bit Multiplier	Binary Addition	2DLNS Addition	Binary Addition	2DLNS Addition
Clock Frequency (MHz)	-	-	189	133
Overall Area $(\mu m)^2$	1,966,867	2,419,797	1,557,885	3,663,843
Dynamic Power (mW)	1.520	8.890	7.44	72.55
Data Arrival Time (ns)	619.47	608.94	4.99	5.22
Data Converters Area	99.39%	34.98%	66.60%	30.38%

Table 3.3: 32×32 bit Multiplier Synthesis Results

3.2.3.4 Conclusions

We can conclude that the architecture with the binary adder is superior for all examined data widths, particularly when a clock frequency is applied. However, the required format of input and output data may impose some limitations. If none of the architectures were dominant in terms of performance figures, the next step would be investigating the accuracy of the product. Generally speaking, for every particular applications, based on the structure, the number of required multiplications and additions and their sequence, the proper decision should be made. For example, it seems that the multiplier architecture with binary adders is more suited to the MAC structure used in an FIR filter. Therefore, it will be our structure of choice to implement 2DLNS multipliers in the next few chapters.

Chapter 4

2DLNS Recursive Multiplication

4.1 Overview of the Recursive Multiplication Algorithm

The term "Recursive Multiplication" refers to the multiplication algorithm published by Danysh and Swartzlander [17]. The main advantage offered by this technique is the use of several small multipliers to implement larger word length multiplications. It is worth mentioning that although it seems the term "Nested Multiplication" is more suited for this algorithm, we still use the same term used in the literature. For 2DLNS recursive multiplication, the binary data is split into smaller parts, then each part is converted to its equivalent 2DLNS value. In this way, smaller size RALUTs for conversion are required, which greatly reduces the VLSI area and power consumption of the multiplier. The architectures presented in this chapter operate on 64-bit binary data. The following discussion is limited to two recursive multiplication architectures based on one and two levels of recursion using 32×32 bit and 16×16 bit base multipliers respectively. The optimal base algorithm examines all possible 2DLNS values of the second base (shown to be between $1/\sqrt{2}$ and

$\sqrt{2}$ in [18]) with a precision of 64-bit floating point, or 17 decimal digits. This process is memory intensive and time consuming. Thus, software with which we implement the algorithm has been written to run on multiple workstations in our research centre.

4.2 One-level Recursion

In a recursive multiplication scheme [17], the n-bit operands are split into two sections, then four $n/2$-bit multiplications are executed in parallel and summed to form the result. In this way, a large multiplication is carried out using a recursion of smaller multiplier modules. In a recursive multiplier the multiplicand A and the multiplier X, as two unsigned n-bit operands, can be represented as:

$$X = \sum_{k=0}^{n-1} x_k \cdot 2^k \qquad , \qquad A = \sum_{k=0}^{n-1} a_k \cdot 2^k \tag{4.1}$$

By splitting each operand into two $n/2$-bit values:

$$\begin{aligned} X &= \sum_{k=0}^{n/2-1} x_k \cdot 2^k + \sum_{k=n/2}^{n-1} x_k \cdot 2^k \\ A &= \sum_{k=0}^{n/2-1} a_k \cdot 2^k + \sum_{k=n/2}^{n-1} a_k \cdot 2^k \end{aligned} \tag{4.2}$$

Therefore, X and A may now be written as:

$$X = X_H + X_L \qquad , \qquad A = A_H + A_L \tag{4.3}$$

and the multiplication of A by X is given by:

$$\begin{aligned} P &= A \cdot X \\ &= (A_H + A_L) \cdot (X_H + X_L) \\ &= X_H \cdot A_H + X_H \cdot A_L + X_L \cdot A_H + X_L \cdot A_L \end{aligned} \tag{4.4}$$

Hence, the overall multiplication is reduced to four smaller multiplications. The four n-bit intermediary products, as shown in Fig. 4.1, are summed to give the $2n$-bit final product.

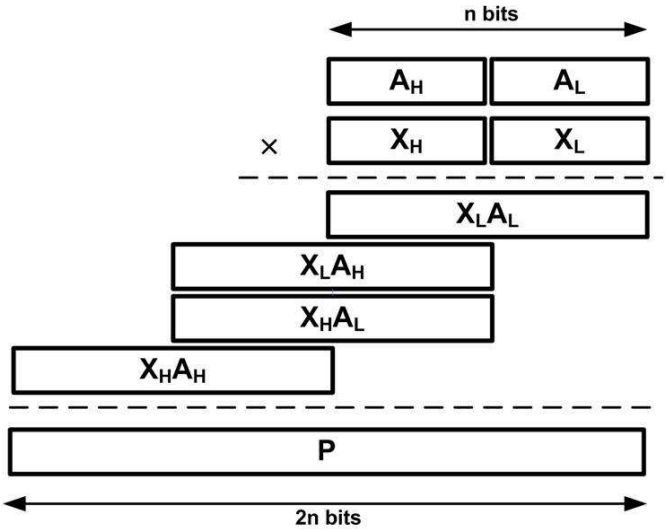

Figure 4.1: One-Level Recursive Multiplier

Classic binary multipliers have two main types of architecture. Array multipliers, which have a regular interconnection structure but a slow critical path, and compression multipliers which rely on finding low critical path delays at the expense of connection regularity. In binary multipliers with a parallel structure, a multi-operand adder is required to add the partial products. Therefore, in order to increase the efficiency of multiplier, several techniques have been developed to reduce the number of partial products. Whereas, in 2-digit 2DLNS multipliers, the operation is accomplished by two levels of parallel 2-operand adders. It has been shown that the most efficient recursive binary multiplier corresponds to only one level of recursion [30]. However, in 2DLNS, since the size of RALUTs are exponentially proportional to the size of the second base exponent, and multiplication only involves two operand adders, this process may be repeated several times using even smaller base multipliers, as is explained in the next section. In order to represent a 32-bit signed binary number in 2DLNS, the best value for the second base, found at this time, is 0.7106639580127168.

Using 11 bits to represent the binary-base exponent and 11 bits for the second-base exponent in a 2-digit 2DLNS representation, an acceptable mapping of 32-bit binary data has been achieved. In a one-level 2DLNS recursive multiplier structure, the binary operands are split into two parts, denoted by high and low sections. Each section is converted to its 2-digit 2DLNS equivalent representation and multiplication is performed by using four 2-digit 2DLNS multiplier modules. In this manner, the size of required look-up tables is smaller than would be required by a non-recursive structure, with the added advantage of fewer and shorter interconnects. The partial products are summed in two stages. In the middle adder, the two operands have the same range, while in the final adder, the operands need to be adjusted. Fig. 4.2 shows this structure. As shown, both the input and output data are in binary representations.

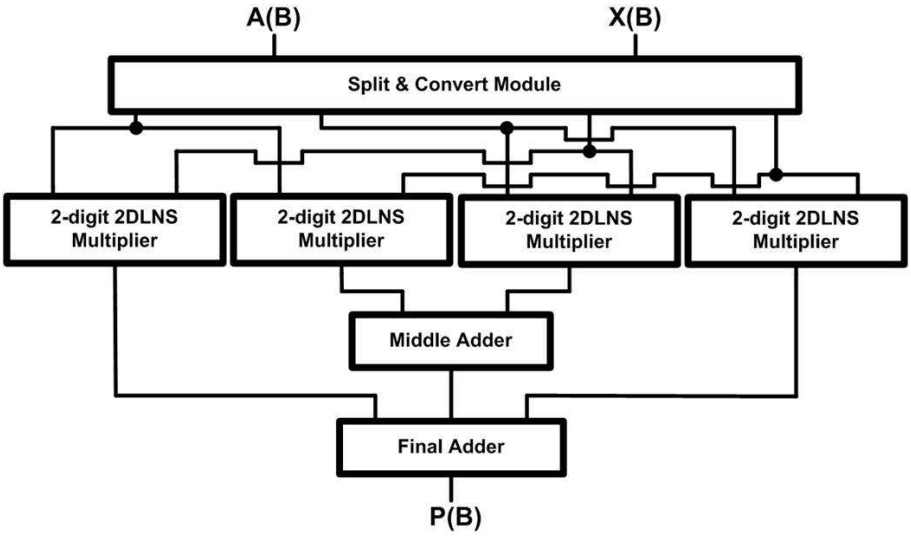

Figure 4.2: The 2-digit 2DLNS-based Recursive Multiplier (One-Level Recursion)

4.3 Two-level Recursion

Having the same n-bit operands, this time split into four $n/4$-bit values:

$$X = \sum_{k=0}^{n/4-1} x_k \cdot 2^k + \sum_{k=n/4}^{n/2-1} x_k \cdot 2^k + \sum_{k=n/2}^{3n/4-1} x_k \cdot 2^k + \sum_{k=3n/4}^{n-1} x_k \cdot 2^k \tag{4.5}$$
$$A = \sum_{k=0}^{n/4-1} a_k \cdot 2^k + \sum_{k=n/4}^{n/2-1} a_k \cdot 2^k + \sum_{k=n/2}^{3n/4-1} a_k \cdot 2^k + \sum_{k=3n/4}^{n-1} a_k \cdot 2^k$$

X and A may now be written as:

$$X = X_{HH} + X_{HL} + X_{LH} + X_{LL} \quad , \quad A = A_{HH} + A_{HL} + A_{LH} + A_{LL} \tag{4.6}$$

and the multiplication of A by X will be:

$$\begin{aligned} P &= A \cdot X \qquad (4.7)\\ &= (A_{HH} + A_{HL} + A_{LH} + A_{LL}) \cdot (X_{HH} + X_{HL} + X_{LH} + X_{LL}) \\ &= X_{HH} \cdot A_{HH} + X_{HH} \cdot A_{HL} + X_{HH} \cdot A_{LH} + X_{HH} \cdot A_{LL} \\ &+ X_{HL} \cdot A_{HH} + X_{HL} \cdot A_{HL} + X_{HL} \cdot A_{LH} + X_{HL} \cdot A_{LL} \\ &+ X_{LH} \cdot A_{HH} + X_{LH} \cdot A_{HL} + X_{LH} \cdot A_{LH} + X_{LH} \cdot A_{LL} \\ &+ X_{LL} \cdot A_{HH} + X_{LL} \cdot A_{HL} + X_{LL} \cdot A_{LH} + X_{LL} \cdot A_{LL} \end{aligned}$$

Now, the overall multiplication is reduced to sixteen smaller multiplications. As Fig. 4.3 shows, these partial products are summed to give the $2n$-bit final product. The partial products can be organized in four distinct groups, each group corresponds to one of the partial multipliers. This arrangement simplifies the overall design of the recursive multiplier architecture.

Based on the results obtained in [31], a 2-digit 2DLNS representation using 6 bits to represent the binary-base exponent and 5 bits for the second-base exponent, provides a good error-free mapping of 16-bit signed binary data. Using the exhaustive search algorithm, the optimal second base is 0.92024380912663017. A further level of recursion is used in

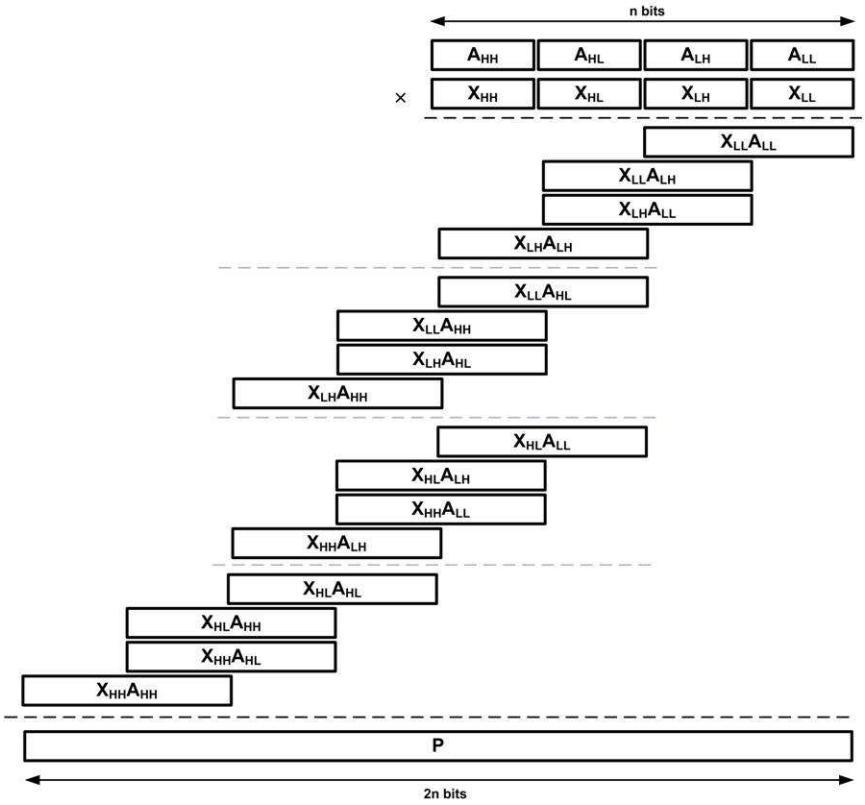

Figure 4.3: Two-Level Recursive Multiplier

this architecture by splitting the high and low sections into two parts, where each part is converted to its equivalent 2-digit 2DLNS representation. A special module, called the partial product adder, has been implemented to produce the output of four resulting partial multipliers. Again, these values are summed using two stages of adders to arrive at the final result. Fig. 4.4 illustrates this architecture.

4.4 Synthesis Results and Comparison

Two recursive 64-bit 2-digit 2DLNS based multipliers have been designed using STMicroelectronics CMOS 90nm technology. Since required look-up tables for conversions are translated to some kind of memory devices in the hardware implementation, the overall area

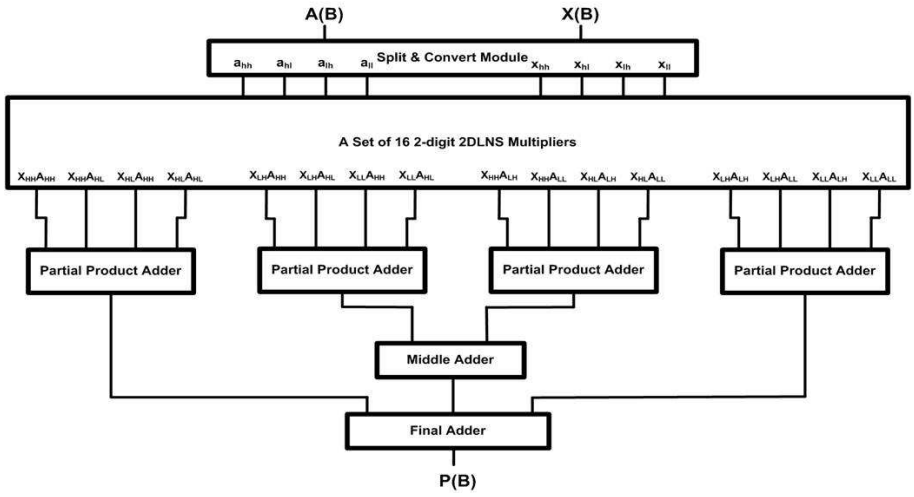

Figure 4.4: The 2-digit 2DLNS-based Recursive Multiplier (Two-Level Recursion)

is mostly consumed by converters. The first architecture (Proposed-1) applies one level of recursion, and therefore acts on 32-bit values. In the second architecture (Proposed-2), with two levels of recursion, the size of RALUTs are much smaller than the first architecture, but more of them are required to realize the fully parallel structure. In our designs, both data conversion modules (Binary-to-2DLNS and 2DLNS-to-Binary), are used in parallel to convert all operands concurrently. Since as always there is a trade off between delay and area in these designs, for each 2DLNS converter, the serial version of the structure has been used, which is more appropriate for area-limited applications. A recursive digital multiplier architecture has been previously presented in [32], and includes a new column compression scheme to alleviate interconnection irregularity. Since this design has been implemented using TSMC-CMOS 0.18 μm technology, we can use it in a comparison study with the application of suitable scaling factors. Although we were not able to fully replicate this design in CMOS 90nm technology, due to some changes in pre-designed components of the internal libraries, by scaling the technology to CMOS 90nm, a reduction in overall area is expected. Additionally, a 64-bit binary multiplier has been synthesized using Synopsys internal designs. This structure, which is not recursive and has been automatically optimized

in order to minimize the area, has been used to demonstrate the superiority of a recursive scheme (with optimum level of recursion) in terms of power consumption and speed.

Architecture	2DLNS-based Architecture (Proposed-1)	2DLNS-based Architecture (Proposed-2)	2DLNS-based Architecture (Proposed-2)	Binary-based Architecture [32]	Synopsys Multiplier
Level of Recursion	One	Two	Two	One	None
Technology	STM-CMOS 90nm	STM-CMOS 90nm	STM-CMOS 90nm	TSMC-CMOS 0.18μm	STM-CMOS 90nm
Clock Frequency (MHz)	182	355	200	200	-
Overall Area $(\mu m)^2$	4,449,996	536,648	534,715	360,417	64,308
Dynamic power (mW)	15.39	10.01	4.86	582.31	62.72
Energy (mW/MHz)	0.085	0.028	0.024	2.912	-
Data Arrival Time (ns)	5.25	2.57	4.71	8.51	35.52

Table 4.1: 64×64 bit Recursive Multipliers Synthesis Results

Our first architecture (Proposed-1), compared to its binary counterpart, reduces delay by 38% while its power consumption is reduced by 37 times. The second architecture (Proposed-2) includes two levels of recursion and acts on 16-bit words. The structure consists of the split and convert module, a set of sixteen 2-digit 2DLNS based multipliers, four partial product adders, and two 64-bit and 128-bit adders. Since the size of conversion tables depends on the size of the first and second base exponents, the RALUTs used in the binary to 2DLNS converters are almost 128 times smaller than the RALUTs used in the previous architecture. Although, the number of RALUTs in this architecture is quadrupled, we are still looking at a 32 times reduction to first order. It is worth mentioning that the delay caused by the conversion RALUTs is a weak function of the size of binary data and remains almost constant for a 2-digit 2DLNS representation; this is particularly the case when it is used in a parallel structure such as our new architectures. The fourth column shows the results for the second architecture (Proposed-2) with a clock frequency of 200 MHz, which is equal to that of the binary-based architecture. Comparing these two archi-

tectures we see that the 2DLNS-based structure, with an increase in area due to conversion, is about 45% faster while consuming 120 times less power. The clock frequency of this architecture can be increased up to 350 MHz, as shown in the third column of Table 4.1. Having this faster clock frequency adds about 0.36% to the area, which is negligible. However, with a cost of doubling the power consumption, the delay is reduced by 1.8 times to 2.57 ns. Comparing this architecture with the binary-based architecture shows that with a 48% increase in area, delay is reduced 3.3 times and dynamic power is decreased 58.2 times. Therefore, both proposed architectures show excellent results in terms of delay and power consumption comparing to their binary counterpart. The Table 4.1 summarizes the synthesis results of these architectures.

4.5 Conclusions

In this chapter, two recursive 2DLNS-based multiplier architectures along with the performance results of their 64-bit implementations have been presented. These architectures benefit from 2DLNS properties, including more precise mapping of data and smaller size of data representations that lead to a reduction in hardware. Furthermore, since every multiplication structure is conducted through 2-operand smaller size adders, no special column compression scheme is necessary. Contrary to binary structures, the recursion process in 2DLNS-based multipliers can be repeated as many times as needed to reach an optimum size for the look-up tables. In this regard, the second proposed architecture shows outstanding results as a low-power and high-speed multiplier. In recursive architectures, multiplication is carried out at maximum efficiency in terms of area, performance and power. The modern DSP processors provide optimistic context for the future of recursive architectures. The applications which will be discussed in the next chapters of this research work will benefit from these architectures.

The main essence of this chapter has already been presented in IEEE International Sympo-

sium on Circuits and Systems, ISCAS 2010 [33].

Chapter 5

Reconfigurable Multiplier Architectures

In new DSP applications, reconfigurable architectures have emerged to provide a flexible, high-performance, high-speed and low-power implementation platform for wireless embedded devices. Since DSP algorithms rely heavily on multiplication, there are still demands for more efficient multiplication structures. In this chapter, two reconfigurable recursive multipliers are presented. These architectures combine some of the flexibility of software with the high performance of hardware through implementing different levels of recursive multiplication schemes on a 2DLNS processing structure. The data is split into a number of smaller sections, where each section is converted to a 2-digit 2DLNS representation. The dynamic range reduction and logarithmic characteristics of computing with two orthogonal base exponents in this number system allows multiplication to be implemented with simple parallel small adders. These architectures are able to perform single and double precision multiplications, as well as fault tolerant and dual throughput single precision operations. Again, the implementations demonstrate the efficiency of 2DLNS in multiplication intensive DSP applications and show outstanding results in terms of operation delay

and dynamic power consumption.

5.1 Reconfigurable Architectures

We now discuss the application of two presented 2DLNS techniques in the previous chapter to the implementation of reconfigurable multiplier architectures in order to achieve improved performance under various conditions. Some modern hybrid architectures (such as controller/DSP chips) use variable width data buses which may lead to variable precision arithmetic. If we use a fixed precision arithmetic unit this will be inefficient in cases where the required precision is not the same as the fixed precision [32]. An example is the use of a fixed single precision unit to perform double precision arithmetic. Designing a reconfigurable hardware architecture definitely enhances the performance. Augmenting a data path with extra execution units provides computational parallelism, which increases processing performance and further provides an appropriate context for fault tolerant processing. The increasing complexity of electronic devices makes embedded fault detection and correction units a commercially wise decision. Checking hardware functionality is currently a more practical approach to fault tolerance [32]. The essence of fault tolerance lies in redundancy. Fault tolerance in our designs is realized through hardware redundancy when duplicate hardware implements parallel multiplication channels. Our new architectures combine the concepts of variable precision multiplication, high-speed and low-power design, fault tolerant computation, and high throughput arithmetic in one design.

5.2 Reconfigurable 2DLNS-based Multipliers

The 2-digit 2DLNS based recursive multiplier structures, presented in the previous chapter, are used as foundation blocks for the reconfigurable architectures.

There are four modes of operations in our designs and a number of multiplexers guide the appropriate signals through the architectures to realize each mode. Although the re-

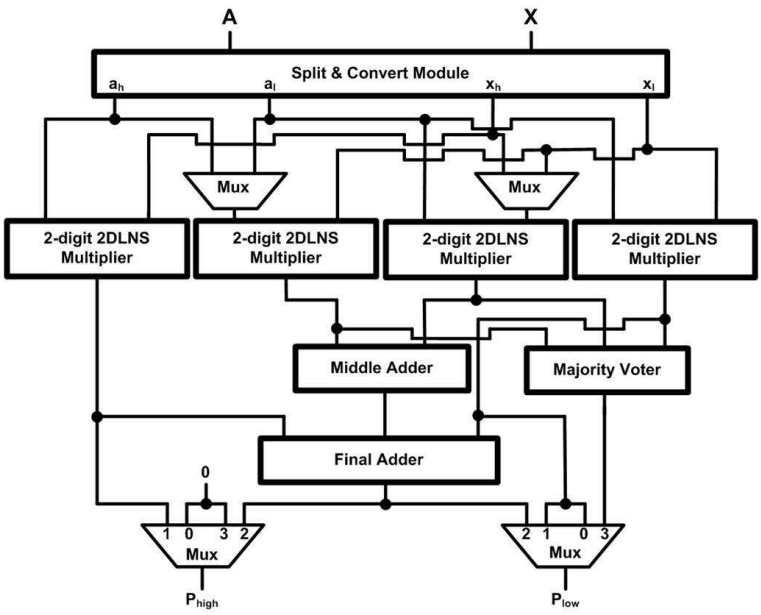

Figure 5.1: The Reconfigurable 2-digit 2DLNS based Recursive Multiplier (One-Level Recursion)

configurable architecture may be of any size, merely with the restriction that the single precision mode must be exactly one half of the double precision mode, in our architectures all data widths, data buses and look-up tables have been designed for a 64-bit multiplication in double precision mode. The schematics of the reconfigurable architectures are provided in Fig. 5.1 and Fig. 5.2 corresponding to one and two levels of recursion respectively.

The architectures are regulated by a 2-bit control signal c_1c_0 to select one of the four modes of operation. The bit c_0 acts as the partial multiplier input selector for both multiplexers, while the complete control signal is used to select the lower and higher parts of the final output:

- Double Precision (64-bit) Multiplication (c_1c_0 = “10”)

  This mode uses the recursive multiplier hardware to its fullest and performs multiplication with one or two levels of recursion. All four 2-digit 2DLNS multiplier modules are used in parallel to multiply lower and higher sections of the operands.

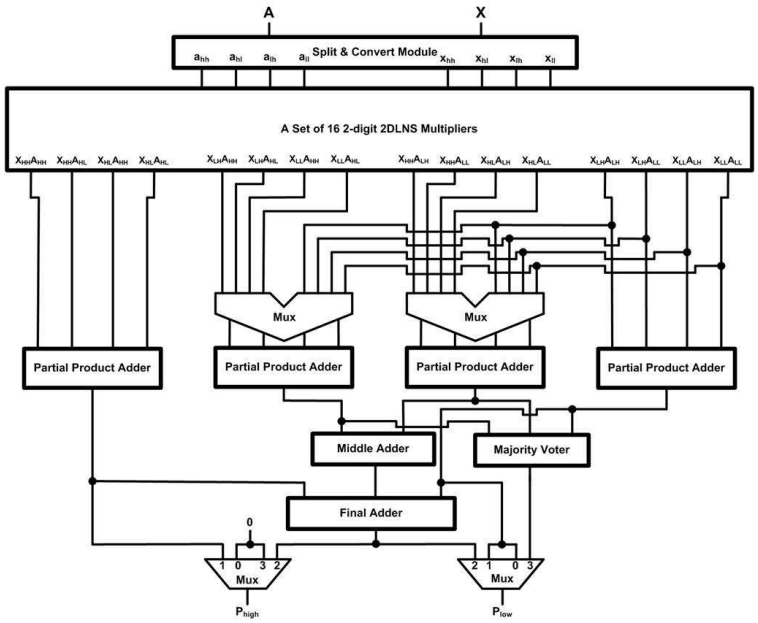

Figure 5.2: The Reconfigurable 2-digit 2DLNS based Recursive Multiplier (Two-Level Recursion)

The outputs are summed in two stages of middle and final adders.

- Single Precision (32-bit) Multiplication (c_1c_0 = "00")

  For this mode, three of the base multipliers, in addition to the middle and final adders, half of the multiplexers and the majority voter circuitry are shut down. Since the final binary output is also partitioned, the higher portion is directly set to zero to avoid unnecessary transitions. In this case, the latency of the architecture is equal to that of the 2-digit 2DLNS multiplier or the partial multiplier, which leads to faster operation than in the double precision mode. The power reduction is important here because more than 75% of the hardware is disabled.

- Dual Single Precision Multiplication (c_1c_0 = "01")

  In order to use two base multipliers of the reconfigurable architecture in parallel, the input bus is configured to allow two sets of operands to occupy the low order and high

order bits of the bus. In this case, two of the base multipliers operate on two different sets of operands concurrently, while the remainder of the circuitry is inactive. This effectively doubles the system throughput, with the same latency as that of the single precision mode.

- Single Precision Fault Tolerant Multiplication (c_1c_0 = "11")
  In this architecture, fault tolerance is realized through majority voting between three duplicate values. Here, three of the base multipliers are used in conjunction with a majority voter circuitry to form a simple single precision fault tolerant multiplier. The first level of multiplexers selects the appropriate signals as operands of the partial multipliers.

The middle and final adders are blocked in the last three modes, which leads to power savings. The two output multiplexers, addressed by the control signal, provide the corresponding output in each mode. The following figures show the recursive multiplier structures for each mode of operation corresponding to one and two levels of recursion. In each mode of operation, the idle parts are shown in grey.

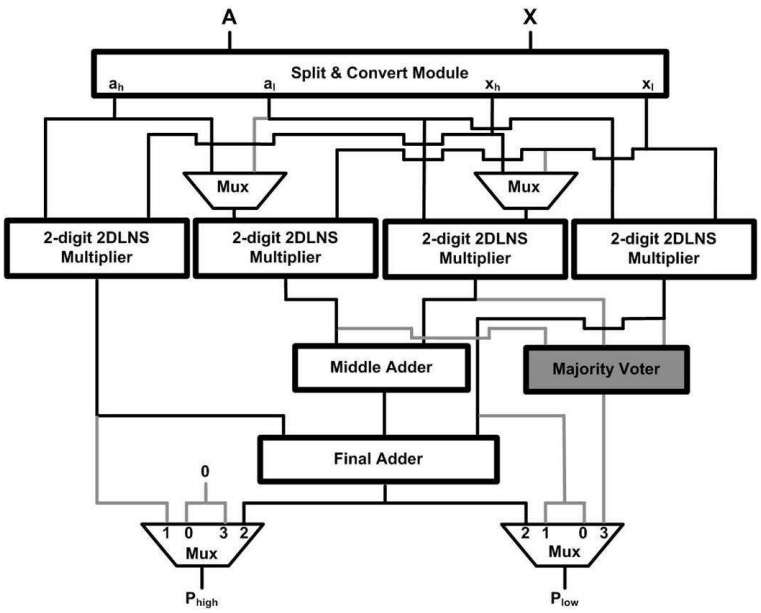

Figure 5.3: Double Precision Multiplier (One-Level Recursion)

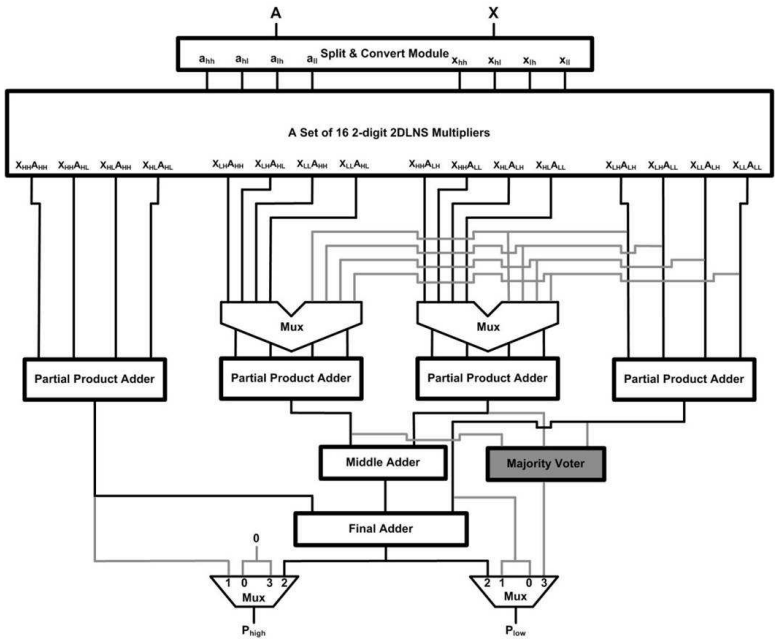

Figure 5.4: Double Precision Multiplier (Two-Level Recursion)

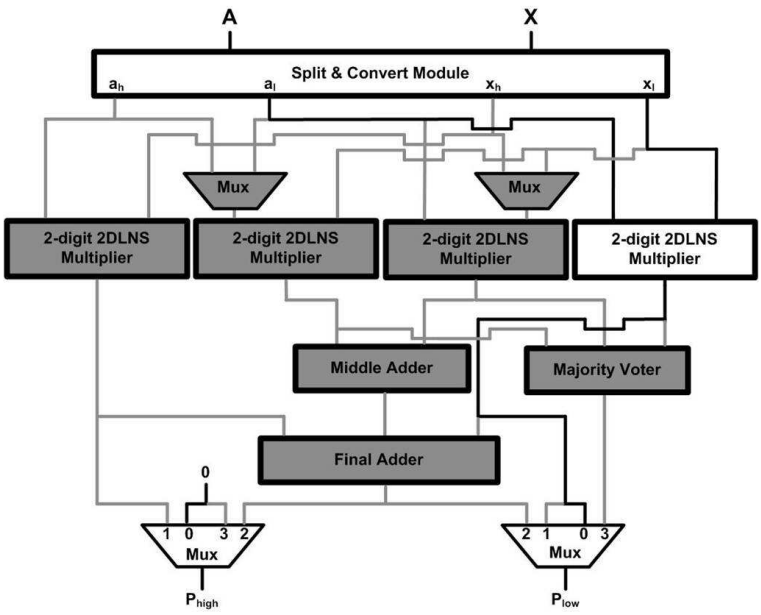

Figure 5.5: Single Precision Multiplier (One-Level Recursion)

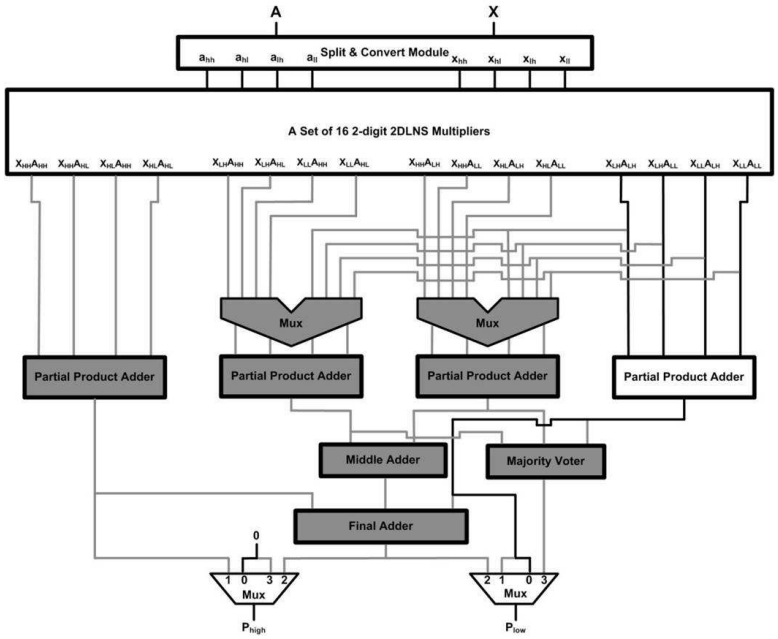

Figure 5.6: Single Precision Multiplier (Two-Level Recursion)

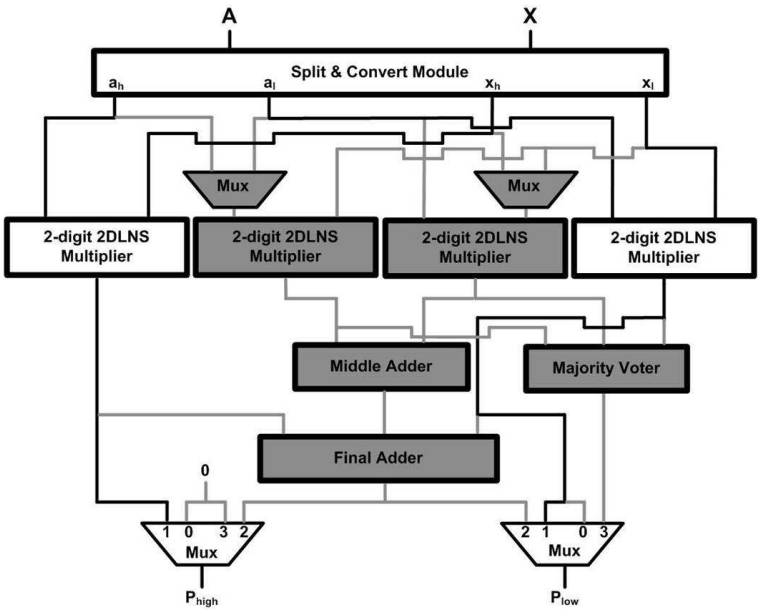

Figure 5.7: Dual Single Precision Multiplier (One-Level Recursion)

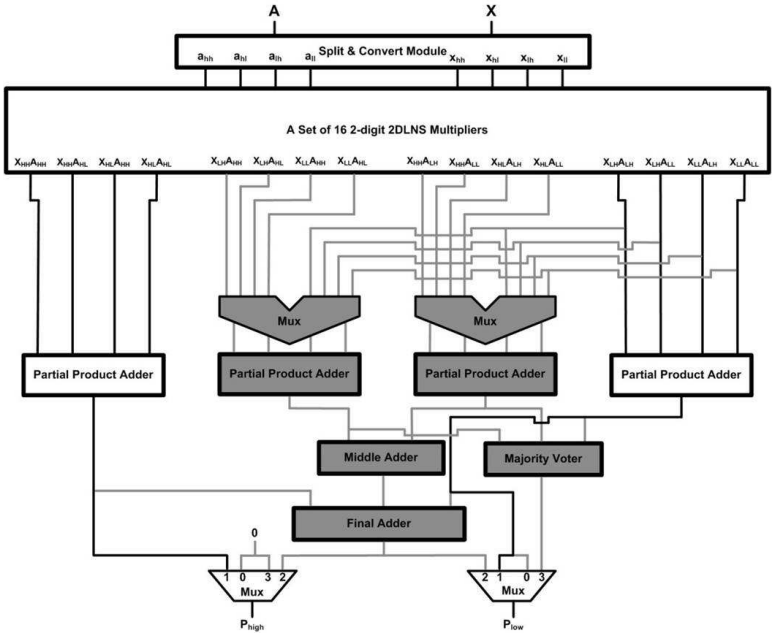

Figure 5.8: Dual Single Precision Multiplier (Two-Level Recursion)

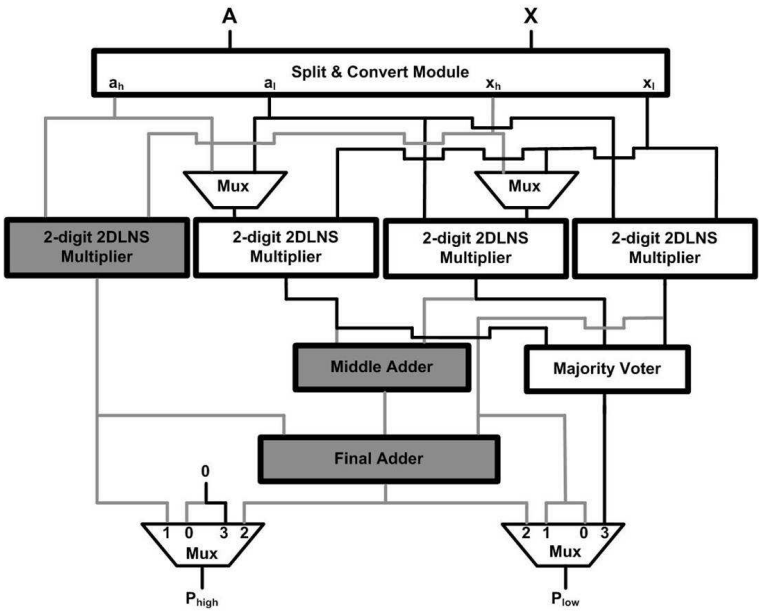

Figure 5.9: Single Precision Fault Tolerant Multiplier (One-Level Recursion)

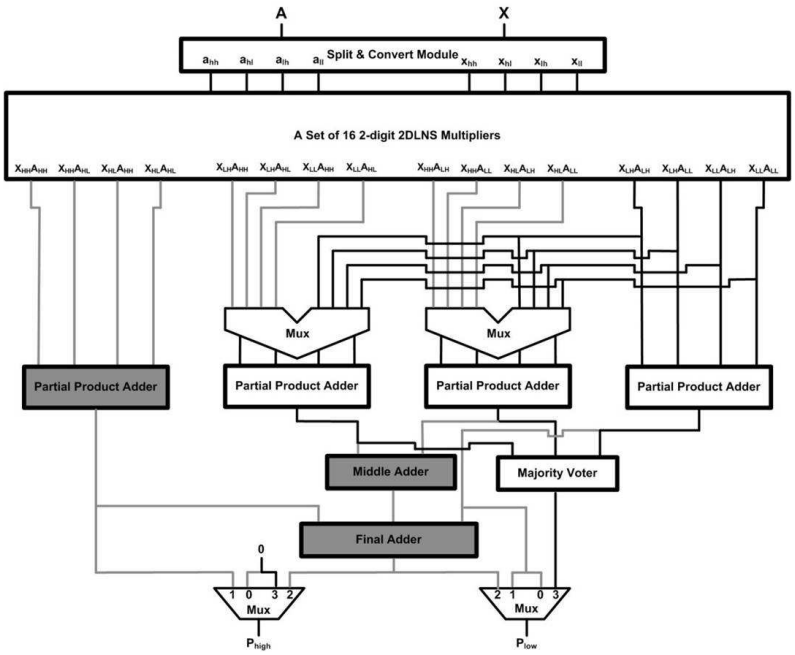

Figure 5.10: Single Precision Fault Tolerant Multiplier (Two-Level Recursion)

5.3 Synthesis Results and Comparison

Two reconfigurable 64-bit 2-digit 2DLNS-based recursive multipliers have been implemented using STMicroelectronics CMOS 90nm process, facilitated by CMC Microsystems. The first architecture (Proposed-1) applies one level of recursion, and therefore acts on 32-bit values. The design circuitry include four 2-digit 2DLNS based multipliers, four multiplexers, two 64-bit and 128-bit adders, and a special majority voter module. At first, a split and convert module divides the 64-bit binary operands to two 32-bit values and then converts them to 2-digit 2DLNS representations. The Binary/2DLNS converters use look-up tables which consume about 90% of the overall area. In this design, again both data conversion modules (Binary-to-2DLNS and 2DLNS-to-Binary), are used in parallel to convert all operands concurrently; this would appear to be a reasonable trade off between chip area (and power) and speed, where high performance is the goal.

64×64 bit Architecture	2DLNS-based Architecture (Proposed-1)	2DLNS-based Architecture (Proposed-2)	2DLNS-based Architecture (Proposed-2)	Binary-based Architecture [32]
Level of Recursion	One	Two	Two	One
Technology	90nm	90nm	90nm	$0.18\mu m$
Clock Frequency (MHz)	200	350	200	200
Overall Area $(\mu m)^2$	4,504,491	533,723	471,783	443,323
Dynamic Power (mW)	30.41	9.85	4.80	753.01
Energy (mW/MHz)	0.152	0.028	0.024	3.765
Data Arrival Time (ns)	4.75	2.63	4.73	9.00

Table 5.1: Reconfigurable Recursive Multipliers Synthesis Results

A reconfigurable digital multiplier architecture was previously published in [32]. This architecture is also centered on recursive multiplication and applies an optimized 4:2 com-

pressor distribution scheme in partial product reduction trees. This structure alleviates some of the problems associated with interconnection irregularities comparing to standard column compression multipliers, while avoiding the linear latency of array multipliers. Again, since this design has been implemented using TSMC $0.18\mu m$ CMOS standard cell libraries, applying some scaling factors may be required. Although by scaling the technology from $0.18\mu m$ to 90nm, a reduction in overall area, dynamic power and delay is generally expected, the reduction factor for each figure is highly dependent on the design characteristics. In this design, the area utilization figure is calculated as 62.15% and 6 layers of metals are used [34], which shows that a considerable portion of the chip area is occupied by interconnects. In this case, scaling technology to 90nm does not affect area, power and delay figures considerably. This design is included to Table 5.1 and is used as a reference.

Our first architecture (Proposed-1) is 47% faster than its binary counterpart, and consumes 25 times less power. The second architecture (Proposed-2) includes two levels of recursion and acts on 16-bit words. The structure consists of the split and convert module, a set of sixteen 2-digit 2DLNS based multipliers, four partial product adders, four multiplexers, two 64-bit and 128-bit adders, and again the majority voter module. Since the size of conversion tables depends on the size of the first and second base exponents, the RALUTs used in the binary to 2DLNS converters are almost 128 times smaller than the RALUTs used in the previous architecture. As we mentioned before, the number of RALUTs in this architecture is quadrupled, but we are still looking at a 32 times reduction to first order.

The fourth column shows the results for the second architecture (Proposed-2) with a clock frequency of 200 MHz, which is equal to that of the binary-based architecture. It is noticeable that the 2DLNS-based structure, without a considerable increase in area, is about 47% faster while consuming 157 times less power. The clock frequency of the second architecture (Proposed-2) can be increased up to 350 MHz, as shown in the third column of Table 5.1. Having this faster clock frequency adds about 13% to the area, with a cost

of doubling the power consumption; however, the delay is reduced by 1.8 times to 2.63 ns. Comparing this architecture with the binary-based architecture shows that with a 25% increase in area, delay is reduced by 29.2% and dynamic power is decreased 76.4 times.

Therefore, both proposed architectures show excellent results in terms of delay and power consumption comparing to their binary counterpart. Table 5.1 summarizes the synthesis results of these architectures and clearly shows the superiority of logarithmic based designs in terms of delay and power consumption. In particular, applying two levels of recursion allows the use of considerably higher clock frequencies.

5.4 Conclusions

Two reconfigurable recursive multiplier architectures have been presented, along with the performance results of their 64-bit implementations. The architectures benefit from both 2DLNS properties and recursive multiplications, which leads to a reduction in hardware. The architectures can be reconfigured in real time for both single and double precision arithmetic, as well as fault tolerant and dual single precision multiplications. Both single and double precision operations are carried out at the maximum efficiency in terms of area, performance and power. Modern DSP processors, such as those used in hand-held devices, may find considerable benefit from these high-performance, low-power, and high-speed reconfigurable architectures.

The work presented in this chapter, has already been published in IET Journal on Circuits, Devices and Systems [35].

Chapter 6

Finite Impulse Response (FIR) Filter Design

As mentioned in the previous chapters, in 2DLNS the reduced hardware complexity not only leads to power consumption savings, and a larger dynamic range, but also more precise mapping of binary data are achieved. In 2DLNS, the mathematical operations over different bases and digits are completely independent and provide more potential for parallelism and also may lead to more speed [16]. In this chapter, we will apply recursive 2DLNS multiplication to Finite Impulse Response (FIR) filter structures. A FIR filter implementation requires a digital processor to perform multiplication over sequences of sampled values of input data and filter coefficients. Therefore, it is a suitable candidate for examining the efficiency of 2DLNS arithmetic in a DSP application. In this regard, after reviewing the concept and algorithm of FIR filter design, a typical FIR filter is designed and implemented in three platforms; binary, 2DLNS, and recursive 2DLNS representations with different data widths, and their synthesis results are discussed.

6.1 FIR Filter Architecture

A digital filter uses a digital processor to perform numerical calculations on the sampled values of a signal. A FIR (non-recursive) filter is a digital filter whose output depends on the present and previous values of the input. The general difference equation for a FIR filter is:

$$Y[n] = \sum_{k=0}^{N} b_k \times X[n-k] \tag{6.1}$$

where $Y[n]$ is the filter output, $X[n-k]$ is the filter input signal delayed by k samples, and N is the order of the filter. The order of a FIR filter specifies the number of coefficients and the number of iterations in the computations. FIR filters are simple to design, linear and guaranteed to be Bounded Input-Bounded Output (BIBO) stable. The order of an FIR filter is considerably higher than that of an equivalent Infinite Impulse Response (IIR) filter meeting the same specifications, which leads to higher computational complexity for the FIR filter. As it can be seen from the difference equation, each output sample depends on the previous N input samples which are multiplied by a set of coefficients, b_ks. Fig. 6.1 shows the systolic architecture for a FIR filter:

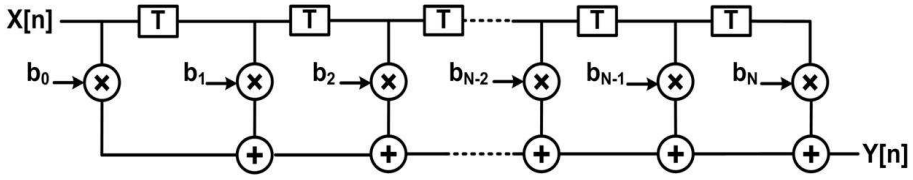

Figure 6.1: The Systolic Structure for a FIR Filter

The set of coefficients are generated based on the design specifications. These specifications are usually required in the frequency domain on magnitude response of the filter and in addition to the order of the filter and frequency range include the maximum pass band ripple and the minimum stop band attenuation, as it is shown in Fig. 6.2.

In a FIR filter the number of coefficients usually is an odd number and if symmetric

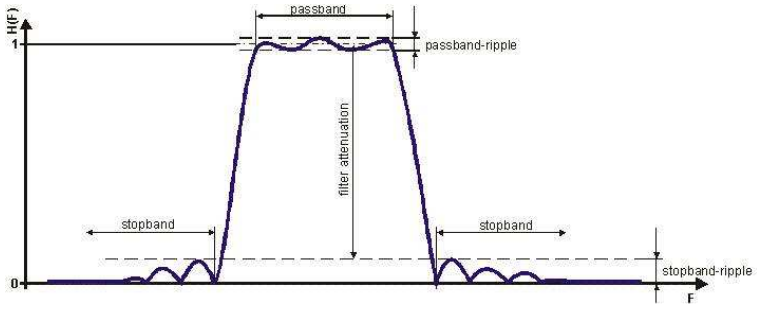

Figure 6.2: The Magnitude Response of a Band Pass Filter

coefficients around the intermediate point considered equal, a linear phase response is obtained. Furthermore, for relatively higher values of the order of filter, FIR filters have a low sensitivity to filter coefficient quantization errors. FIR filtering requires the use of inner product computations, which is based on Multiply and Accumulate (MAC) operations. When 2DLNS is applied to the implementation of a typical FIR filter, every multiplication is converted to addition or subtraction that greatly reduces the hardware complexity.

In this chapter, a band pass FIR filter with the following specifications is implemented: The order of the filter is 74 and the frequency edges of the pass band are 2857Hz and 4KHz. The maximum allowed ripple in pass band is 0.01dB and the minimum attenuation required in stop band is 60dB. Based on the above design characteristics, the filter coefficients have been generated in MATLAB and then converted to 2's complement binary representations. A chirp signal has been also generated in MATLAB to be applied to the filter architecture as the input signal. A chirp is a signal in which the frequency increases or decreases with time. In our sinusoidal chirp signal, the frequency increases from 0Hz to 8KHz in a 1 second time period.

In subsequent sections four FIR filter architectures (binary-based, 2DLNS-based and two recursive 2DLNS-based algorithms) are realized through implementing behavioral VHDL and Verilog modules. The input and output data in each case are in binary format. The simulation output of each design has been written into a file. The output data are converted to decimal values and another MATLAB program has been authored to display the filtering

results. Executing this program generates the corresponding output graphs.

6.2 16-bit Input Signal

In this case, the original values of both input signal and filter coefficients are 16-bit binary values. Here, the amplitude of the chirp signal is equivalent to 16-bit, 2's complement binary representation. Fig. 6.3 shows this signal:

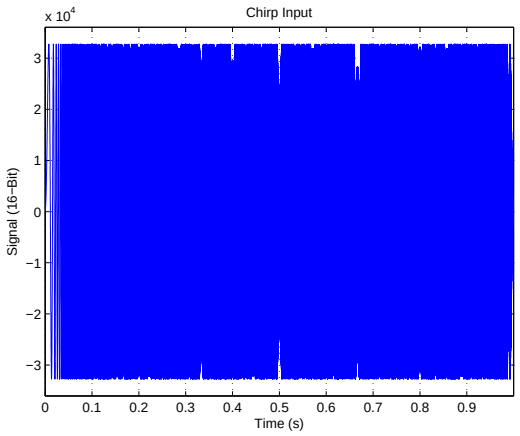

Figure 6.3: The Filter Input Signal

If the 16-bit binary coefficients generated by MATLAB program along with this chirp signal apply to the MATLAB internal FIR filter functions, the worst case stop band attenuation is -60.5116dB and pass band ripple is +0.0119dB. The obtained stop band attenuation perfectly satisfies the design criterion, but the pass band ripple is slightly more than the specified value with an acceptable precision. However, we continue with this set of filter coefficients.

6.2.1 Binary-based Design

A behavioral VHDL code has been written to implement the binary-based FIR filter design. The filter coefficients have been pre-stored into an array in the main module and the input

data are read from a file. A MAC structure operates on two sequences of data and coefficients, multiplies corresponding elements of the sequences and accumulates the sum of products in an optimized scheme. In this scheme (Optimized-1), in order to reduce the size of the product, firstly the sign bits of the operands are examined. The negative operands are converted to their positive 2's complement representations and based on the sign of the product, it will be added to or subtracted from the accumulator. A sufficient number of extra bits have been considered in order to avoid overflow in accumulation. In order to show the efficiency of our optimized design, another binary-based filter has been designed, this time with the multiplier from the Synopsys internal library cells. Both designs are synthesized in their maximum speed, our design is also synthesized with the same frequency as of Synopsys design (Optimized-2):

FIR Filter Architecture	Binary-based by Synopsys	Binary-based Optimized-1	Binary-based Optimized-2
Clock Frequency (MHz)	204	308	204
Overall Area $(\mu m)^2$	672,860	639,471	469,019
Dynamic power (mW)	11.98	18.79	11.43
Data Arrival Time (ns)	4.70	3.03	4.69

Table 6.1: 16-bit Binary-based Designs Synthesis Results

As it is noticeable from the Table 6.1, the optimized design is advantageous in terms of VLSI area, delay, and maximum applicable clock frequency. Therefore, we will compare this design with the logarithmic filtering results at the end of this chapter.

Fig. 6.4 shows the filtered output signal produced by this architecture. In this case, the worst case stop band attenuation is -60.2071dB and pass band ripple is 0.0087dB. As it can be noticed, the optimized design yields more precise results than MATLAB internal FIR filter program.

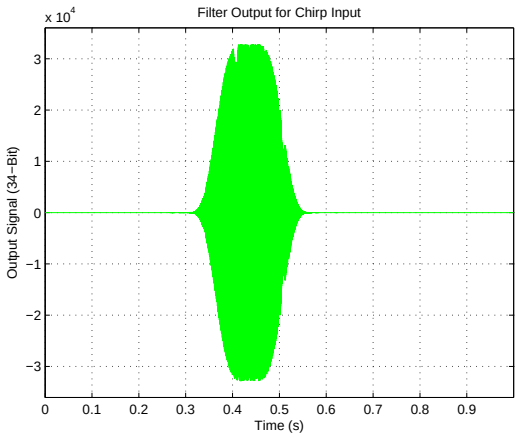

Figure 6.4: The Binary Filter Output Signal

6.2.2 2DLNS-based Design

Regarding the prior research in [31], considering 0.9202438884391765 as optimal second base in a 2-digit 2DLNS representation provides a good error-free mapping of 16-bit signed binary data. We have also considered 6 bits to represent the binary-base exponent for both input data and coefficients while to represent the second-base exponent for data and coefficients, 5 bits and 3 bits have been considered respectively. Here, the filter coefficient values are firstly converted to 2DLNS representations and then pre-stored in a coefficient memory. On the other hand, the input samples are converted to 2DLNS values during run time and then placed in a data memory. The main module in our 2DLNS design, which we will refer to as **Filter**, employs a couple of auxiliary modules through its external interface. This structure is shown in Fig. 6.5.

The signal **clk** is the master clock signal that drives the **Filter** implementation. The **reset** signal is used to manage the whole operation of filtering. It resets all control signals to their default values at the beginning of filtering process. Then by applying the clock frequency, the filter operation starts. Both of these signals are generated by the Clock Generator Module. The Input Data Reader reads the data from a file and passes them to the **Filter** structure in a timely manner. The final output will also be written into a

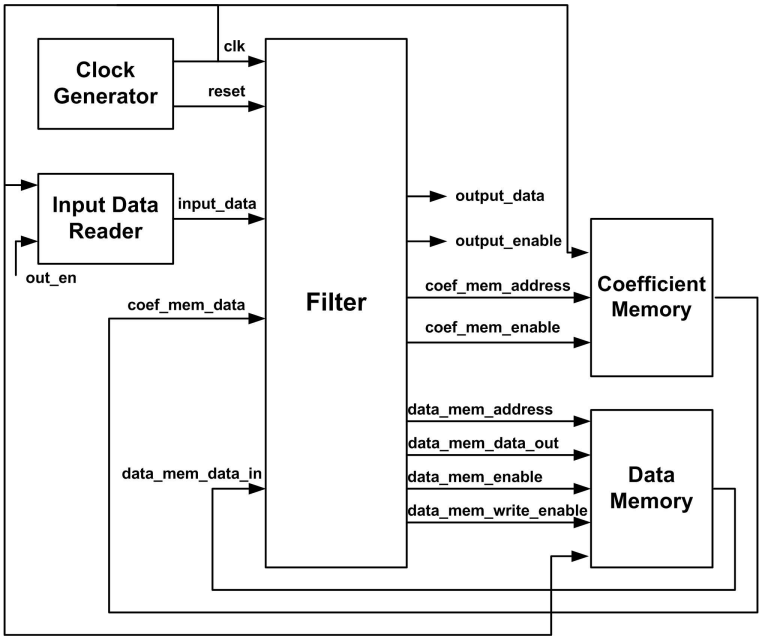

Figure 6.5: The Filter and Auxiliary Modules

file. The remaining connections of the **Filter** are its interface with the coefficient and data memories. The signals **coef-mem-address** and **data-mem-address** provide the addresses to access memories. The coefficient memory contents are pre-stored and can only be read. The Data Memory has separate data lines to read data from memory, **data-mem-data-in**, and to write data into memory, **data-mem-data-out**. Enable signals are used to enable main module to access both memories, and the signal **data-mem-write-enable** is set by the controller whenever data memory is accessed to be written.

At the Register Transfer Level (RTL) structure, the **Filter** module is composed of an Input Register, a Binary/2DLNS Converter and its corresponding output register, a Multiply and Accumulate (MAC) unit, and a sequential Controller. The RTL level organization of the **Filter** module is shown in Fig. 6.6.

Every input sample upon arrival should be converted to 2DLNS representation. Since binary to 2DLNS conversion is performed in a variable number of clock cycles, the input register is required to allow each input sample enter the structure whenever the converter is

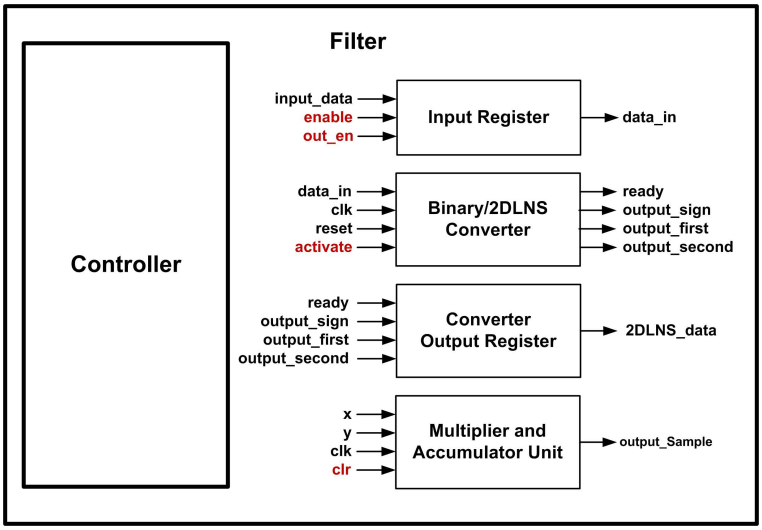

Figure 6.6: The Filter RTL Components

done with the previous one and has asserted a ready signal to accept a new input sample. The output of the converter is transformed to an appropriate format by the converter output register in order to be stored into the data memory.

At the start of a filtering iteration, the source operands which are 2-digit 2DLNS values of input samples and coefficients are placed on the input ports of the MAC unit, and the operation commences. As it is shown in Fig. 6.7, the main part of the MAC architecture actually is a sign-magnitude 2-digit 2DLNS multiplier, and another adder which is instructed by the control unit, accumulates the products in an iteration and forms the output signal. The MAC operation is accomplished in one clock cycle.

At the end of operation, the result is written into an output file. The control unit specifies the sequence and timing of the operations. The control ports of the data path component instances are connected to the control signals managed by the control unit. The output of this structure is shown in Fig. 6.8.

In this case, the worst case stop band attenuation is -57.6396dB and pass band ripple is 0.0111dB. These characteristics will be improved by using more number of bits to represent the second base index, but the costs are more required hardware and more power

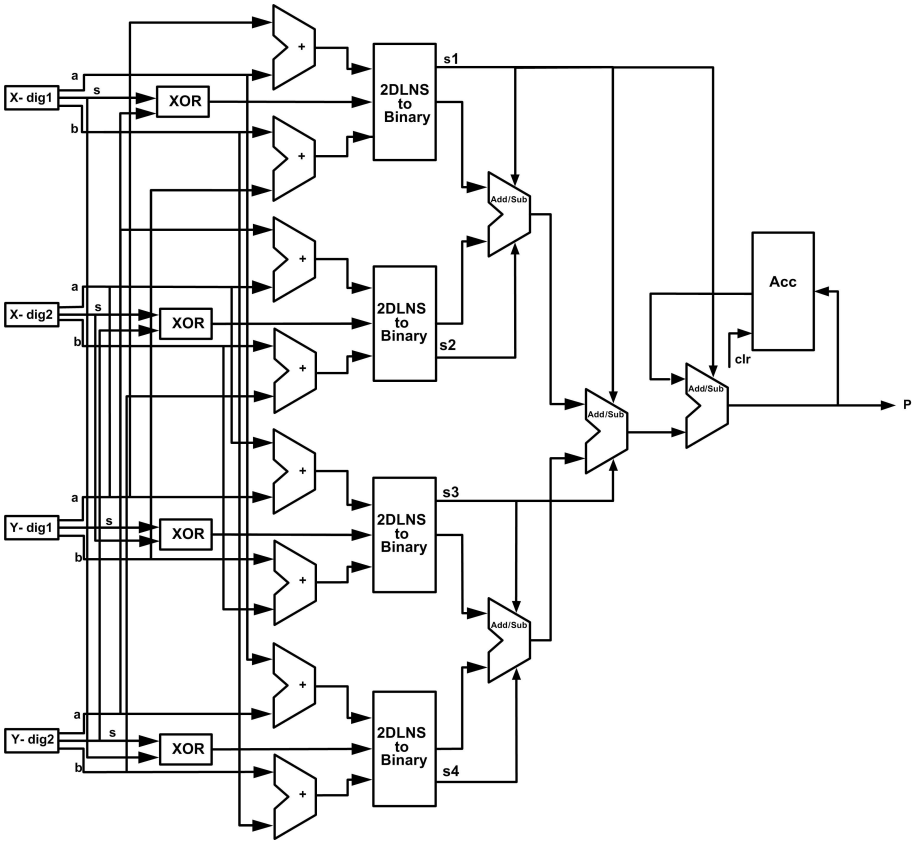

Figure 6.7: The MAC Unit Structure

consumption.

There are two 2DLNS-based FIR filterbank designs in literature [29] and [36]. In our design, the modified versions of 2DLNS/Binary conversions programs in [29] have been used. Table 6.2 shows some properties of these designs. This table also includes our 2DLNS-based design characteristics, when it is synthesized for a low clock frequency as 4.8 MHz. Since these designs both have been implemented using $0.18\mu m$ TSMC CMOS technology and furthermore, the power calculation techniques may be different, a comparison between designs may not be fair. Nevertheless, this table gives some idea about the improvements that have been made in 2DLNS algorithms and their hardware implementations.

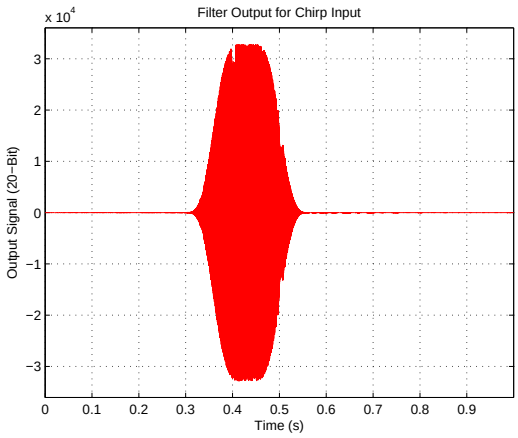

Figure 6.8: The 2DLNS Filter Output Signal

FIR Filter Architecture	[29]	[36]	Proposed
The Order of the Filter	74	74	74
Technology CMOS	TSMC 0.18μm	TSMC 0.18μm	STM 90nm
Clock Frequency (MHz)	4.8	4.8	4.8
Overall Area $(\mu m)^2$	53,716	184,965	32,132
Dynamic power (μW)	316	708	34

Table 6.2: 2DLNS-based Designs Synthesis Results

6.2.3 Recursive 2DLNS-based Design

Now, the recursive algorithm, which was explained in prior chapters, is used to reduce the size of data and coefficients in FIR filter application. In such a way, the size of required look-up tables will be reduced. When the coefficients are computed in binary format, they are split into two parts, then each part will be converted to 2-digit 2DLNS representation. These values are stored into the coefficient memory. Here, all four corresponding digits to a coefficient are included into one memory cell. Every input data sample is also split and converted to 2-digit 2DLNS values. Therefore, the number of Binary-to-2DLNS converters

are doubled in this architecture. The output of these converters are placed in one word in the corresponding output register. Similarly, data values are stored in the data memory with the same pattern, four 2DLNS digits in each memory cell. In this architecture four parallel MAC units are required to perform MAC operations on 2-digit operands.

At the start of a filtering iteration, the source operands, which are four couple of 2-digit 2DLNS values of input samples and coefficients, are placed on the input ports of four MAC units, and the operation begins. The output result of these units are summed in two stages to yield the output of the iteration. The accumulator along with its corresponding register provide the final output of the MAC operation. This structure is shown in Fig. 6.9.

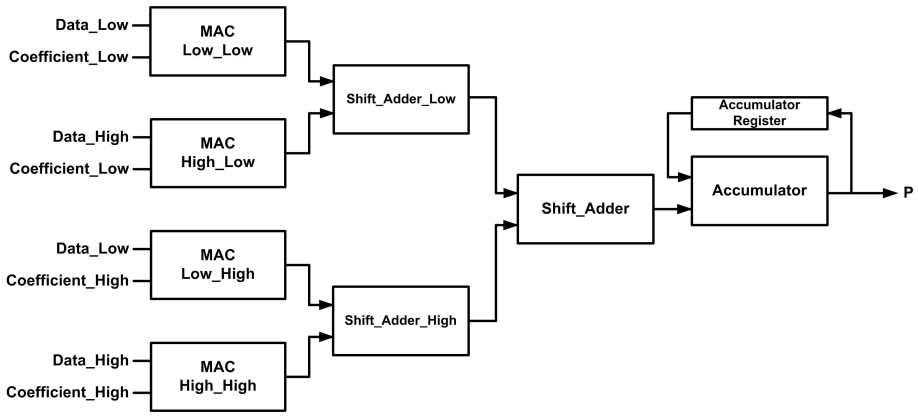

Figure 6.9: The MAC Unit Structure in Recursive Architecture

Since coefficients are pre-computed, sometimes the conversion hardware overhead regarding the split coefficients may be excluded from the recursive design considerations. This way, while the input data are split, the genuine coefficients will be still applicable. In any case, the same optimal base should be used to represent both data and coefficients. This fact may impose some limitations to the recursive filter design. However, the recursive architecture has been implemented in two approaches:

- Split input data and split coefficients

  In this approach, the filter coefficients values are firstly split into two 8-bit parts,

then converted to 2-digit 2DLNS values and pre-stored in a look-up table, which we refer to as coefficient memory. Having split data and coefficients, the full structure of the MAC unit consisting of four parallel similar channels is utilized. The outputs of these channels should be shifted properly and summed up to yield the final product of each iteration. For this structure, 5 bits to demonstrate the binary index and 2 bits to represent the second base exponent are considered. The optimal base for this case is calculated as 0.8087001487814504. After addition of corresponding exponents, we will have 3 bits to represent the second base index. This time, the optimal base is calculated as 0.808750023022608 which is close enough to the previous value. Therefore, with a good approximation, we use the same second base for conversion to binary. The output of this structure is shown in Fig. 6.10. In this case, the average stop band attenuation is -65.0279dB and maximum pass band ripple is 0.0275dB.

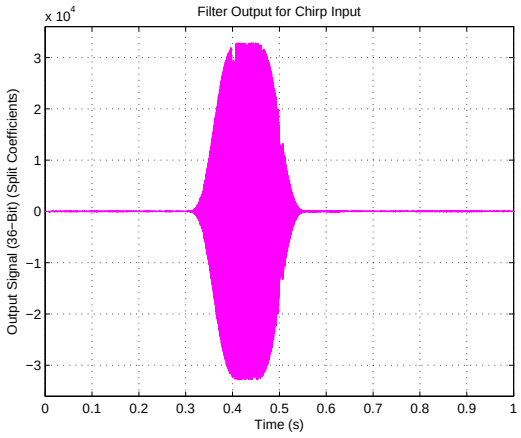

Figure 6.10: The Recursive 2DLNS Filter (Split Coefficients) Output Signal

- Split input data and genuine coefficients

  Here, the original filter coefficient values are converted to 2DLNS and pre-stored into the memory. Our filter coefficients, obtained from MATLAB, are less than 1 decimal numbers with 16 decimal digits. Therefore, in order to maintain a certain accuracy, we consider at least 14 bits for them to be represented in binary. Again

we consider the same configuration for conversion to 2DLNS values as of 16 bit case; 6 bits and 3 bits for binary and second base indices respectively. Since these values for data conversion are still 6 bits and 5 bits respectively, the same optimal base that is 0.9202438884391765 can be used for both conversions. Having genuine coefficients, two of the channels in MAC unit are eliminated. Furthermore, there are some reduction in the adder stages of the MAC unit. The output of this structure is shown in Fig. 6.11. In this case, the average stop band attenuation is -79.0385dB and maximum pass band ripple is 0.0130dB.

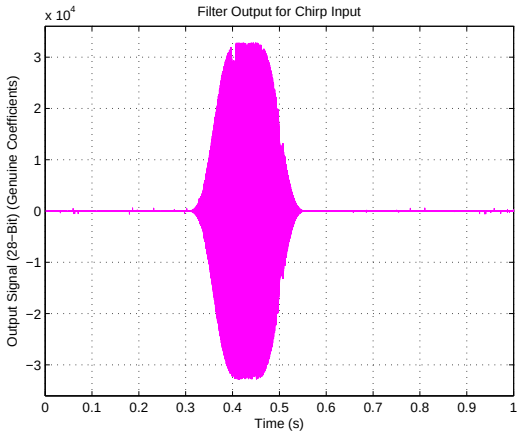

Figure 6.11: The Recursive 2DLNS Filter (Genuine Coefficients) Output Signal

6.2.4 Synthesis Results and Comparison

These four FIR filter architectures have been successfully synthesized using STMicroelectronics CMOS 90nm technology. All architectures closely follow the design specifications. As we expected, through replacing multiplication with smaller adders in a 2DLNS-based architecture, the hardware complexity is greatly simplified. In 2DLNS-based designs all required data conversions are included into the structure and are counted in the overall area. As it can be seen from the synthesis results, summarized in Table 6.3, all architectures have been synthesized in their maximum clock frequency, and the highest is in binary-based

design. If this design is synthesized with a clock frequency of 250 MHz, both area and power will be reduced to 513,743 $(\mu m)^2$ and 14.36 (mW) respectively, but delay will be increased to 3.79 ns. This way, the 2DLNS-based design consumes about 14% area and 34% power as that of the binary-based architecture, when both running in a clock frequency of 250 MHz.

FIR Filter Architecture	Binary-based	2DLNS-based	Recursive 2DLNS-based Split Coefficients	Recursive 2DLNS-based Genuine Coefficients
Clock Frequency (MHz)	308	256	188	189
Overall Area $(\mu m)^2$	639,471	72,941	65,606	118,913
Dynamic power (mW)	18.79	5.26	7.10	6.62
Energy (mW/MHz)	0.061	0.021	0.038	0.035
Data Arrival Time (ns)	3.03	3.70	5.13	5.05
Data Representation (bit)	16	2×(1,6,5)	2×2×(1,5,2)	2×2×(1,6,5)
Coefficient Representation (bit)	16	2×(1,6,3)	2×2×(1,5,2)	2×(1,6,3)

Table 6.3: 16-bit FIR Filter Designs Synthesis Results

The delay is increased in recursive 2DLNS-based architectures due to extra adder stage(s). Therefore, the maximum applicable clock frequencies are less than the other architectures. The last two rows of the table show the size of data and coefficients. The triples in 2DLNS formats contain the sign, the binary index and the second base exponent respectively. The number(s) in front of triples show the number of digits. The recursive 2DLNS-based structure with split coefficients has quadruple RALUTs, but the size of each is reduced to almost 1/6 of its size in 2DLNS design. Whereas, the recursive 2DLNS-based structure with genuine coefficients has double RALUTs, but with the same size as before. The reason is that we have considered 14 bits to represent filter coefficients in binary. This way, coefficients

and consequently data can not be converted to 2DLNS values with a smaller representation. In recursive architectures area and power are more than we expected due to an increase in interconnections. Therefore, using recursive architectures has no added advantage in this case.

Nevertheless, the synthesis results show the superiority of 2DLNS-based design in terms of VLSI area and power consumption and confirm that 2DLNS is the platform of choice for low-power FIR filter design. It seems the recursion process in 2DLNS arithmetic yields the maximum efficiency whenever it practically leads to smaller size data representations and conversion RALUTs. In order to resolve this problem with recursive designs in this application, the 32-bit architectures will be implemented in the next section.

6.3 32-bit Input Signal

Here, the amplitude of the chirp signal is equivalent to 32-bit, 2's complement binary representation. Fig. 6.12 shows this signal. This signal will be applied to the four FIR filter architectures which are modified to work on 32-bit data and coefficients.

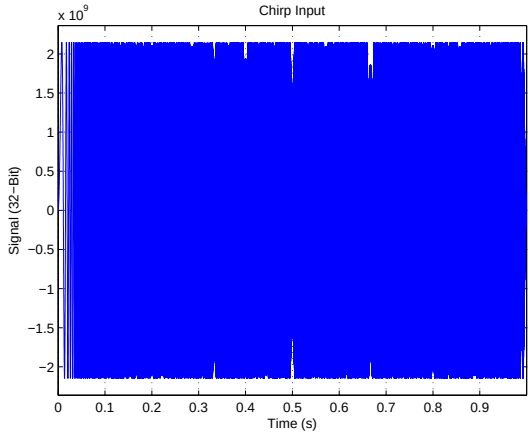

Figure 6.12: The Filter Input Signal

6.3.1 Binary-based Design

The optimized binary design has been scaled up to work on 32-bit data and coefficients (optimized-1). Similar to the 16-bit case, the binary-based filter with Synopsys internal optimized multiplier have been compared.

FIR Filter Architecture	Binary-based by Synopsys	Binary-based Optimized-1	Binary-based Optimized-2
Clock Frequency (MHz)	105	278	105
Overall Area $(\mu m)^2$	1,446,601	1,264,289	860,983
Dynamic power (mW)	12.16	32.60	11.44
Data Arrival Time (ns)	9.30	3.41	9.31

Table 6.4: 32-bit Binary-based Designs Synthesis Results

As it is noticeable from the Table 6.4, again the optimized design is advantageous in terms of VLSI area, delay, and maximum applicable clock frequency.

Fig. 6.13 shows the filtered output signal produced by this architecture. In this case, the worst case stop band attenuation is -60.2385dB and pass band ripple is 0.0091dB.

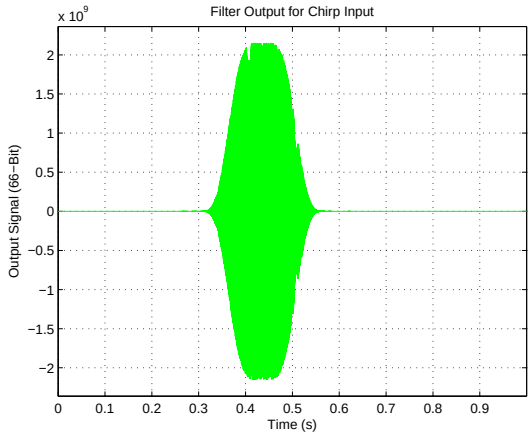

Figure 6.13: The Binary Filter Output Signal

6.3.2 2DLNS-based Design

We have computed the optimal second base of 0.7106639580127168 in a 2-digit 2DLNS representation for a good error-free mapping of 32-bit signed binary data. We have also found out that at least 11 bits to represent both the binary-base and the second-base exponents to represent both data and coefficients are required. The output of this structure is shown in Fig. 6.14.

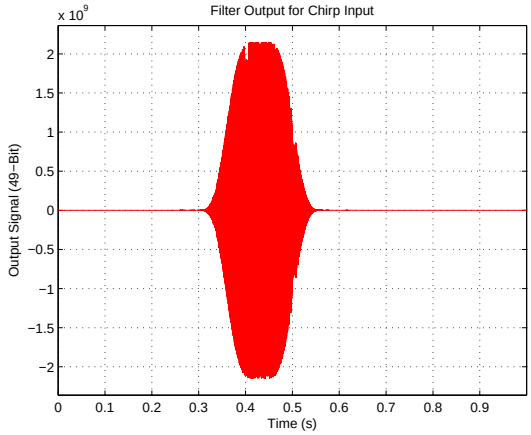

Figure 6.14: The 2DLNS Filter Output Signal

In this case, the worst case stop band attenuation is -58.8765dB and pass band ripple is 0.0091dB.

6.3.3 Recursive 2DLNS-based Design

Again, we take the same two approaches with 32-bit data and coefficients:

- Split input data and split coefficients

  After splitting both data and coefficients, we now have 16-bit operands. One extra bit is added to maintain the sign of each part correctly. In this case, considering an optimal second base of 0.737111818827987 in a 2-digit 2DLNS representation provides a good error-free mapping of binary data. Again, 6 bits are considered

to represent the binary-base exponent for both input data and coefficients while to represent the second-base exponent for data and coefficients, 5 bits and 3 bits have been considered respectively. Here, the average stop band attenuation is -74.6789dB and maximum pass band ripple is 0.0141dB. The output of this structure is shown in Fig. 6.15.

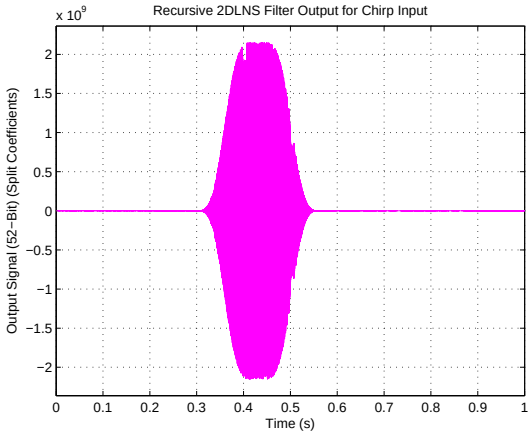

Figure 6.15: The Recursive 2DLNS Filter (Split Coefficients) Output Signal

The HDL codes written to implement this architecture and corresponding auxiliary packages are attached in Appendix A.

- Split input data and genuine coefficients

  Since genuine coefficients can be represented in 16-bit binary values, the same representations as split case will be considered for this structure. The only difference is that the coefficients are represented with two digits. Therefore, two channels of MAC units are shut down. In this case, we are able to reduce the size of data to the smaller size of genuine coefficients which makes it the most efficient architecture. The output of this structure is shown in Fig. 6.16. In this case, the average stop band attenuation is -74.3282dB and maximum pass band ripple is 0.0141dB.

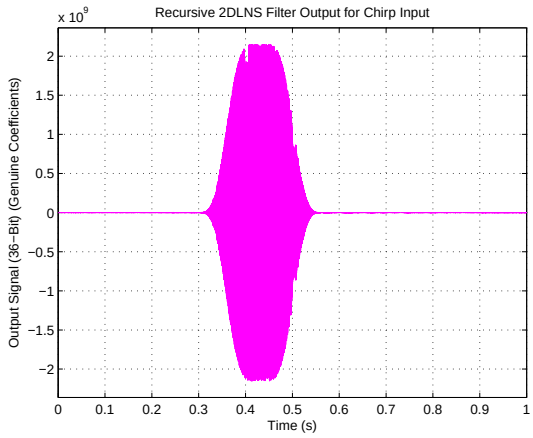

Figure 6.16: The Recursive 2DLNS Filter (Genuine Coefficients) Output Signal

6.3.4 Synthesis Results and Comparison

Table 6.5 summarizes the synthesis results of the four architectures. Again, the last two rows of the table show the size of data and coefficients. The triples in 2DLNS formats contain the sign, the binary index and the second base exponent respectively. The number(s) in front of triples show the number of digits. Again, all architectures have been synthesized in their maximum clock frequency, and the highest occurs in binary-based design. This time, the 2DLNS-based design is not efficient. The main reason is the size of conversion RALUTs, with having 11 bits to represent the second base exponent, this table has $2^{11} + 1$ rows. Although, if 32-bit binary data can be represented with a smaller 2DLNS second index, this architecture will also be improved. Here, on contrary to the 16-bit case, the recursive 2DLNS-based designs, particularly the design with genuine coefficients, are greatly advantageous in terms of VLSI area and power consumption. In this case, the recursive 2DLNS-based structure with split/genuine coefficients has quadruple/double RALUTs, but the size of each is almost 1/62 the size of the RALUT in 2DLNS design. When FIR filter characteristics lead to less than 1 coefficients, the desired accuracy can be maintained with a representation in limited number of binary bits, which in this case is 16 bits. Therefore, splitting coefficients here has no added advantage and just adds more hardware and

interconnections to the structure. The synthesis results show outstanding reduction in the area and power of the recursive design with genuine coefficients. In this design, area is decreased by 88% and power reduction is 74% as that of the binary-based architecture.

FIR Filter Architecture	Binary-based	2DLNS-based	Recursive 2DLNS-based Split Coefficients	Recursive 2DLNS-based Genuine Coefficients
Clock Frequency (MHz)	278	164	165	200
Overall Area $(\mu m)^2$	1,264,289	1,457,206	232,501	151,750
Dynamic power (mW)	32.60	59.80	11.57	8.52
Energy (mW/MHz)	0.117	0.365	0.070	0.043
Data Arrival Time (ns)	3.41	5.91	5.86	4.81
Data Representation (bit)	32	2×(1,11,11)	2×2×(1,6,5)	2×2×(1,6,5)
Coefficient Representation (bit)	32	2×(1,11,11)	2×2×(1,6,3)	2×(1,6,3)

Table 6.5: 32-bit FIR Filter Designs Synthesis Results

6.4 Conclusions

In this chapter, in order to demonstrate the efficiency of 2DLNS-based and recursive 2DLNS-based representations in signal processing, a FIR filter has been implemented. The synthesis results confirm that the 2DLNS-based designs, when structured properly, are dominant in terms of VLSI area and power consumption. This fact that coefficients are computed, converted to 2DLNS representations and stored into a memory before run time, provides flexibility for choosing the appropriate architecture. In this regard, the best architecture may be realized by reducing the size of larger data down to the size of coefficients. This may be involved with applying none to multiple level of recursion.

A part of work developed in this chapter has already been presented in IEEE International NEWCAS Conference, 2011 [37]. The more comprehensive paper authored based on this work has been submitted to Journal of Circuits, Systems and Signal Processing.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

In this dissertation, efficient techniques to perform arithmetic operations on 2DLNS representations have been developed.

We began with examining the existing 2DLNS addition/subtraction algorithms. In this regard, some multiplication architectures with binary and 2DLNS adders were implemented. Having the synthesis results for these 2DLNS multipliers, we concluded that the architecture with binary adder is privileged for all examined data widths, particularly when a clock frequency is applied. Therefore, we decided to use multiplier architecture with binary adders to realize 2DLNS multipliers in the applications that will be implemented in this research work.

Then, the concept of recursive multiplication has been applied to 2DLNS structures to reduce the size of conversion look-up tables. In this regard, two recursive 2DLNS-based multiplier architectures with one and two levels of recursion, along with their 64-bit im-

plementations have been developed. It has been also shown that the recursion process in 2DLNS-based multipliers can be repeated as many times as needed to reach an optimum size corresponding look-up tables. In this regard, the architecture with two level of recursion showed outstanding results as a low-power and high-speed multiplier. Further in this study, we examined the efficiency of recursive 2DLNS-based multiplication in two DSP applications.

As the first application, two reconfigurable recursive multiplier architectures have been developed. The architectures benefit from both 2DLNS properties and recursive multiplications, which lead to a reduction in hardware. The architectures can be reconfigured in real time for both single and double precision arithmetic, as well as fault tolerant and dual single precision multiplications. Both single and double precision operations are carried out at the maximum efficiency in terms of area, performance and power. Modern DSP processors, such as those used in hand-held devices, may find considerable benefit from these reconfigurable architectures.

Eventually, in order to demonstrate the efficiency of 2DLNS-based and recursive 2DLNS-based representations in signal processing, an FIR filter has been implemented. The synthesis results confirm that the 2DLNS-based designs are dominant in terms of VLSI area and power consumption. This fact that coefficients are precomputed, provides flexibility for choosing the appropriate architecture in recursive structures. In this regard, the best architecture may be realized by reducing the size of larger data down to the size of coefficients using the recursion techniques described in this thesis.

7.2 Suggestions for Future Works

As we realized in FIR filter designs, recursive architectures are generally slower. The reason is that multiple stages of 2-operand binary adders have been used in multipliers. Therefore, implementing more efficient adder structures, such as carry save adders, may be

a solution to reduce delay in multipliers.

In addition, some improvements may be made in the process of mapping of coefficients. More efficient mapping of coefficients, in terms of both precision and size, will definitely enhance the 2DLNS-based filter architectures.

In order to minimize the VLSI area in our architectures, the serial structure of 2DLNS / Binary conversions have been used, which has influenced the speed of the operations. In this regard, the parallel and series architectures of these converters can be substituted to provide delay and area requirements.

Furthermore, potential applications in multiplication intensive algorithms in a variety of data widths can be examined. A possible application can be design and implementation of an IIR filter. An IIR filter uses past outputs to influence the current response of the filter. Although recursive filters usually require a much lower order to produce the same magnitude response as of an FIR filter, they are not guaranteed to be stable or have a linear phase.

And finally, the ASIC implementation of recursive 2DLNS-based multipliers may be considered as an extension to this work. In such a way, the application of these architectures in modern DSP processors may be examined more realistically.

References

[1] N. G. Kingsbury and P. J. Rayner, "Digital Filtering Using Logarithmic Arithmetic," *Electronics Letters*, vol. 7, pp. 56–58, 1971.

[2] B. Hoefflinger, M. Selzer, and F. Warkowski, "Digital Logarithmic CMOS Multiplier For Very-High-Speed Signal Processing," in *IEEE Conference on Custom Integrated Circuits*, vol. 16.7, pp. 1–5, 1991.

[3] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert, "A Comparison of Floating Point and Logarithmic Number System for FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 181–190, 2005.

[4] S. Ramaswamy and R. E. Siferd, "CMOS VLSI Implementation of A Digital Logarithmic Multiplier," in *IEEE National Conference on Aerospace and Electronics*, vol. 1, pp. 291–294, 1996.

[5] M. G. Arnold and P. D. Vouzis, "A Serial Logarithmic Number System ALU," in *IEEE Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp. 151–156, 2007.

[6] J. N. Coleman, "Simplification of Table Structure in Logarithmic Arithmetic," *Electronics Letters*, vol. 31, no. 22, pp. 1905–1906, 1995.

[7] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 702–715, 2000.

[8] L. K. Yu and D. M. Lewis, "A 30-b Integrated Logarithmic Number System Processor," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 10, pp. 1433–1440, 1991.

[9] J. N. Coleman and E. I. Chester, "A 32-bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point," in *IEEE Symposium on Computer Arithmetic*, pp. 142–151, 1999.

[10] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The European Logarithmic Microprocessor," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 532–546, 2008.

[11] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl, and A. Hermanek, "The European Logarithmic Microprocessor - a QR RLS Application," in *Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp. 155–159, 2001.

[12] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Licko, Z. Pohl, and A. Hermanek, " Performance of The European Logarithmic Microprocessor," in *SPIE Conference on Advanced Signal Processing Algorithms, Architectures and Implementations*, 2003.

[13] M. G. Arnold, "A Pipelined LNS ALU," in *IEEE Computer Society Workshop on VLSI*, pp. 155–161, 2001.

[14] P. D. Vouzis, S. Collange, and M. G. Arnold, "Cotransformation Provides Area and Accuracy Improvement in an HDL Library for LNS Subtraction," in *IEEE Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp. 85–93, 2007.

[15] N. Belanger and Y. Savaria, "On the Design of a Double Precision Logarithmic Number System Arithmetic Unit," in *IEEE International NEWCAS Conference*, pp. 241–244, 2006.

[16] R. Muscedere, V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "Efficient Techniques for Binary-to-Multidigit Multidimensional Logarithmic Number System Conversion Using Range Addressable Look-Up Tables," *IEEE Transactions on Computers, Special Issue on Computer Arithmetics*, vol. 54, no. 3, pp. 257–271, 2005.

[17] A. N. Danysh and E. E. J. Swartzlander, "A Recursive Fast Multiplier," in *Asilomar Conf. on Signals, Systems and Computers*, vol. 1, pp. 197–201, 1998.

[18] R. Muscedere, "Improving 2D-Log-Number-System Representations by Use of an Optimal Base," *EURASIP Journal on Advances in Signal Processing*, vol. 2008, pp. 1–14, 2008.

[19] H. Li, G. A. Jullien, V. S. Dimitrov, M. Ahmadi, and W. C. Miller, "A 2-digit Multidimensional Logarithmic Number System Filterbank for a Digital Hearing Aid Architecture," in *IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 760–763, 2002.

[20] E. E. J. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1238–1242, 1975.

[21] T. Stouraitis and V. Paliouras, "Considering the Alternatives in Low-Power Design," *IEEE Circuits and Devices Magazine*, vol. 17, no. 4, pp. 22–29, 2001.

[22] M. G. Arnold, T. A. Bailey, J. R. Cowles, and J. J. Cupal, "Redundant Logarithmic Arithmetic," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1077–1086, 1990.

[23] V. S. Dimitrov, J. Eskritt, G. A. Jullien, and W. C. Miller, "The Use of the Multi-dimensional Logarithmic Number System in DSP Applications," in *IEEE Symp. on Computer Arithmetic*, pp. 247–256, 2001.

[24] V. S. Dimitrov, S. Sadeghi-Emamchaie, G. A. Jullien, and W. C. Miller, "A Near Canonical Double-Base Number System with Applications in DSP," in *SPIE Conference on Signal Processing Algorithms*, vol. 2846, pp. 14–25, 1996.

[25] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "An Algorithm for Modular Exponentiation," in *Information Processing Letters*, vol. 36, pp. 155–159, May 1998.

[26] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "Theory and Applications of the Double-Base Number System," *IEEE Transactions on Computers*, vol. 48, pp. 1098–1106, October 1999.

[27] V. S. Dimitrov and G. A. Jullien, "A New Number Representation with Applications," *IEEE Circuits and Systems Magazine*, vol. Second Quarter, pp. 6–23, 2003.

[28] V. S. Dimitrov, G. A. Jullien, and K. Walus, "Digital filtering using the multidimensional logarithmic number system," *Advanced Signal Processing Algorithms, Architectures, and Implementations XII, Proceedings of the SPIE*, vol. 4791, pp. 412–423, December 2002.

[29] R. Muscedere, *Difficult Operations in the Multi-Dimensional Logarithmic Number System*. University of Windsor: Ph.D. Thesis, 2003.

[30] P. Mokrian, M. Ahmadi, and G. A. Jullien, "On the Reduction of Interconnect Effects in Deep Submicron Implementation of Digital Multiplication Architectures," *Journal of Circuits, Systems and Computers*, vol. 15, no. 1, pp. 83–106, 2006.

[31] R. Muscedere, V. Dimitrov, G. Jullien, and W. Miller, "A Low-Power Two-Digit Multi-Dimensional Logarithmic Number System Filterbank Architecture for a Digital Hearing Aid," *EURASIP Journal on Applied Signal Processing*, vol. 18, pp. 3015–3025, 2005.

[32] P. Mokrian, M. Ahmadi, G. A. Jullien, and W. C. Miller, "A Reconfigurable Digital Multiplier Architecture," in *IEEE CCECE Canadian Conference on Electrical and Computer Engineering*, vol. 1, pp. 125–128, 2003.

[33] M. Azarmehr, M. Ahmadi, and G. A. Jullien, "Recursive Architectures for 2DLNS Multiplication," in *IEEE International Symposium on Circuits and Systems*, pp. 3869–3872, 2010.

[34] P. Mokrian, *A Reconfigurable Digital Multiplier Architecture*. University of Windsor: M.A.Sc. Thesis, 2003.

[35] M. Azarmehr, M. Ahmadi, G. A. Jullien, and R. Muscedere, "High-Speed and Low-Power Reconfigurable Architectures of 2-digit Two-Dimensional Logarithmic Number System-based Recursive Multipliers," *IET Journal on Circuits, Devices and Systems*, vol. 4, no. 5, pp. 374–381, 2010.

[36] H. Li, *A 2-digit Multi-dimensional Logarithmic Number System Filterbank Processor for a Digital Hearing Aid*. University of Windsor: M.A.Sc. Thesis, 2003.

[37] M. Azarmehr, M. Ahmadi, and G. A. Jullien, "A Two-Dimensional Logarithmic Number System (2DLNS)-based Finite Impulse Response (FIR) Filter Design," in *IEEE International NEWCAS Conference*, pp. 37–40, 2011.

Appendix A

Hardware Description Codes

In order to show the design flow of a 2DLNS architecture and its realization in hardware, the HDL codes written to implement the "Split input data and split coefficients" design in 32-bit case, are shown here.

A.1 Packages

A VHDL package contains subprograms, constant definitions, and/or type definitions to be used through one or more design units. The filter program makes use of some IEEE standard packages.

VHDL standard packages include **STANDARD** package which contains basic type definitions and **TEXTIO** package which regards to ASCII input/output data types and subprograms. These packages are fully standard and not described here. In addition, another IEEE package, **numeric_bit** has been used in this program. This package defines numeric types and arithmetic functions for use with synthesis tools. In this research work, the original

version of this package, **Standard VHDL Synthesis Package (1076.3, NUMERIC_BIT)**, has been used with some minor changes. Since the package code is too long to be included here, just the modified portions are described. The following VHDL file shows the differences between modified **numeric_bit** package and its original copy. This file clearly shows the lines which have been changed or added to the package. Most of these changes return back to initial values of variables. In whole package, initial values have been removed from declaration statements and have been assigned to the variables later. The definition of functions RISING_EDGE and FALLING_EDGE have had conflicts with other standard libraries. Therefore, the defined functions in this package have been ignored. Some other changes are just minor corrections regard to VHDL syntax. The core shift functions, XSLL and XSRL, have been modified based on some requirements. In this regard, the code for a barrel shifter has been entirely rewritten. Finally, the function RESIZE has been changed due to some errors.

```
801,805c801,805
<   -- Id: E.1
<   function RISING_EDGE (signal S: BIT) return BOOLEAN;
<   -- Result subtype: BOOLEAN
<   -- Result: Returns TRUE if an event is detected on signal S and the
<   --         value changed from a '0' to a '1'.
---
>   --  -- Id: E.1
>   --  function RISING_EDGE (signal S: BIT) return BOOLEAN;
>   --  -- Result subtype: BOOLEAN
>   --  -- Result: Returns TRUE if an event is detected on signal S and
    the
>   --  --         value changed from a '0' to a '1'.
807,811c807,811
<   -- Id: E.2
<   function FALLING_EDGE (signal S: BIT) return BOOLEAN;
<   -- Result subtype: BOOLEAN
<   -- Result: Returns TRUE if an event is detected on signal S and the
<   --         value changed from a '1' to a '0'.
---
>   --  -- Id: E.2
>   --  function FALLING_EDGE (signal S: BIT) return BOOLEAN;
>   --  -- Result subtype: BOOLEAN
```

```
> --  -- Result: Returns TRUE if an event is detected on signal S and
    the
> --  --          value changed from a '1' to a '0'.
823,824c823,824
<    constant NAU: UNSIGNED(0 downto 1) := (others => '0');
<    constant NAS: SIGNED(0 downto 1) := (others => '0');
---
>    constant NAU: UNSIGNED(0 to 1) := (others => '0');
>    constant NAS: SIGNED(0 to 1) := (others => '0');
886c886
<      variable CBIT: BIT := C;
---
>      variable CBIT: BIT; --:= C;
887a888
>      CBIT := C;
904c905
<      variable CBIT: BIT := C;
---
>      variable CBIT: BIT; -- := C;
905a907
>      CBIT := C;
952,953c954,960
<      alias XARG: BIT_VECTOR(ARG_L downto 0) is ARG;
<      variable RESULT: BIT_VECTOR(ARG_L downto 0) := (others => '0');
---
>      variable RESULT: BIT_VECTOR(ARG_L downto 0);
>      variable temp: integer;
>      variable arg_1: bit_vector (ARG_L downto 0);
>      variable arg_12: bit_vector (ARG_L downto 0);
>      variable arg_124: bit_vector (ARG_L downto 0);
>      variable shift: bit_vector(3 downto 0);
>
955,956c962,975
<      if COUNT <= ARG_L then
<        RESULT(ARG_L downto COUNT) := XARG(ARG_L-COUNT downto 0);
---
>
>      temp := count;
>      for index in 0 to 3 loop
>        if  (temp rem 2) = 0 then shift(index) := '0';
>                             else shift(index) := '1';
>        end if;
>          temp := temp / 2 ;
>      end loop;
>
>      if shift(0) = '0' then arg_1 := arg ;
>                        else arg_1 := arg(ARG_L-1 downto 0) & "0" ;
>      end if;
>      if shift(1) = '0' then arg_12 := arg_1 ;
```

```
>                     else arg_12 := arg_1(ARG_L-2 downto 0) & "00" ;
957a977,983
>     if shift(2) = '0' then arg_124 := arg_12 ;
>                       else arg_124 := arg_12(ARG_L-4 downto 0) & "0000"
    ;
>     end if;
>     if shift(3) = '0' then RESULT := arg_124 ;
>                       else RESULT := arg_124(ARG_L-8 downto 0) &
                                        "00000000" ;
>     end if;
>
963,964c989,995
<     alias XARG: BIT_VECTOR(ARG_L downto 0) is ARG;
<     variable RESULT: BIT_VECTOR(ARG_L downto 0) := (others => '0');
---
>     variable RESULT: BIT_VECTOR(ARG_L downto 0);
>     variable temp: integer;
>     variable arg_1: bit_vector (ARG_L downto 0);
>     variable arg_12: bit_vector (ARG_L downto 0);
>     variable arg_124: bit_vector (ARG_L downto 0);
>     variable shift: bit_vector(3 downto 0);
>
966,967c997,1010
<     if COUNT <= ARG_L then
<       RESULT(ARG_L-COUNT downto 0) := XARG(ARG_L downto COUNT);
---
>
>     temp := count;
>     for index in 0 to 3 loop
>       if  (temp rem 2) = 0 then shift(index) := '0';
>                            else shift(index) := '1';
>       end if;
>         temp := temp / 2 ;
>     end loop;
>
>     if shift(0) = '0' then arg_1 := arg ;
>                       else arg_1 := "0" & arg(ARG_L downto 1) ;
>     end if;
>     if shift(1) = '0' then arg_12 := arg_1 ;
>                       else arg_12 := "00" & arg_1(ARG_L downto 2) ;
968a1012,1018
>     if shift(2) = '0' then arg_124 := arg_12 ;
>                       else arg_124 := "0000" & arg_12(ARG_L downto 4) ;
>     end if;
>     if shift(3) = '0' then RESULT := arg_124 ;
>                       else RESULT := "00000000" &
                                        arg_124(ARG_L downto 8) ;
>     end if;
>
```

```
976c1026
<       variable XCOUNT: NATURAL := COUNT;
---
>       variable XCOUNT: NATURAL; -- := COUNT;
977a1028
>       XCOUNT := COUNT;
991c1042
<       variable RESULT: BIT_VECTOR(ARG_L downto 0) := XARG;
---
>       variable RESULT: BIT_VECTOR(ARG_L downto 0); --:= XARG;
993a1045
>       RESULT := XARG;
1005c1057
<       variable RESULT: BIT_VECTOR(ARG_L downto 0) := XARG;
---
>       variable RESULT: BIT_VECTOR(ARG_L downto 0); -- := XARG;
1007a1060
>       RESULT := XARG;
1100c1153
<       variable CBIT: BIT := '1';
---
>       variable CBIT: BIT; --:= '1';
1101a1155
>       CBIT := '1';
1219c1273
<       variable RESULT: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0) :=
                         (others => '0');
---
>       variable RESULT: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0);
                         -- := (others => '0');
1221a1276
>       RESULT := (others => '0');
1239c1294
<       variable RESULT: SIGNED((L_LEFT+R_LEFT+1) downto 0) :=
                         (others => '0');
---
>       variable RESULT: SIGNED((L_LEFT+R_LEFT+1) downto 0);
                         -- := (others => '0');
1241a1297
>       RESULT := (others => '0');
1301c1357
<       variable QNEG: BOOLEAN := FALSE;
---
>       variable QNEG: BOOLEAN; --:= FALSE;
1302a1359
>       QNEG := FALSE;
1353a1411
>       null;
1387a1446
```

```
>       null;
1411c1470
<       variable RNEG: BOOLEAN := FALSE;
---
>       variable RNEG: BOOLEAN; --:= FALSE;
1412a1472
>       RNEG := FALSE;
1446a1507
>       null;
1482a1544
>       null;
1500a1563
>       null;
1524c1587
<       variable RNEG: BOOLEAN := FALSE;
---
>       variable RNEG: BOOLEAN; -- := FALSE;
1525a1589
>       RNEG := FALSE;
1581a1646
>       null;
1599a1665
>       null;
1617a1684
>       null;
1626c1693
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1627a1695
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1639c1707
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1640a1709
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1710c1779
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1711a1781
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1723c1793
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1724a1795
>       SIZE := MAX(L'LENGTH, R'LENGTH);
```

```
1794c1865
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1795a1867
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1807c1879
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1808a1881
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1878c1951
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1879a1953
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1891c1965
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1892a1967
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1962c2037
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1963a2039
>       SIZE := MAX(L'LENGTH, R'LENGTH);
1975c2051
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
1976a2053
>       SIZE := MAX(L'LENGTH, R'LENGTH);
2046c2123
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
2047a2125
>       SIZE := MAX(L'LENGTH, R'LENGTH);
2059c2137
<       variable SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
---
>       variable SIZE: NATURAL; -- := MAX(L'LENGTH, R'LENGTH);
2060a2139
>       SIZE := MAX(L'LENGTH, R'LENGTH);
2314c2393
```

```
<       variable RESULT: NATURAL := 0;
---
>       variable RESULT: NATURAL; -- := 0;
2315a2395
>       RESULT := 0;
2350c2430
<       variable I_VAL: NATURAL := ARG;
---
>       variable I_VAL: NATURAL; -- := ARG;
2351a2432
>       I_VAL := ARG;
2364a2446
>       null;
2372,2373c2454,2455
<       variable B_VAL: BIT := '0';
<       variable I_VAL: INTEGER := ARG;
---
>       variable B_VAL: BIT; -- := '0';
>       variable I_VAL: INTEGER; -- := ARG;
2374a2457,2458
>       B_VAL := '0';
>       I_VAL := ARG;
2392a2477
>       null;
2402c2487
<       variable RESULT: SIGNED(NEW_SIZE-1 downto 0) := (others => '0');
---
>       variable RESULT: SIGNED(NEW_SIZE-1 downto 0); -- := (others
    => '0');
2404a2490
>       RESULT := (others => '0');
2418,2420c2504,2506
<       constant ARG_LEFT: INTEGER := ARG'LENGTH-1;
<       alias XARG: UNSIGNED(ARG_LEFT downto 0) is ARG;
<       variable RESULT: UNSIGNED(NEW_SIZE-1 downto 0) := (others => '0');
---
>       alias INVEC: UNSIGNED(ARG'LENGTH-1 downto 0) is ARG;
>       variable RESULT: UNSIGNED(NEW_SIZE-1 downto 0);
                         -- := (others => '0');
>       constant BOUND: INTEGER := MIN(ARG'LENGTH, RESULT'LENGTH)-1;
2421a2508
>       RESULT := (others => '0');
2424c2511
<       if XARG'LENGTH =0 then return RESULT;
---
>       if (ARG'LENGTH =0) then return RESULT;
2426,2430c2513,2514
<       if (RESULT'LENGTH < ARG'LENGTH) then
<         RESULT(RESULT'LEFT downto 0) := XARG(RESULT'LEFT downto 0);
```

```
<       else
<         RESULT(RESULT'LEFT downto XARG'LEFT+1) := (others => '0');
<         RESULT(XARG'LEFT downto 0) := XARG;
---
>       if BOUND >= 0 then
>         RESULT(BOUND downto 0) := INVEC(BOUND downto 0);
2434c2518
<
---
>
2559,2563c2643,2651
<   -- Id: E.1
<   function RISING_EDGE (signal S: BIT) return BOOLEAN is
<   begin
<     return S'EVENT and S = '1';
<   end RISING_EDGE;
---
>   -- Id: E.1
>   function RISING_EDGE (signal S: BIT) return BOOLEAN is
>   begin
>         return S'EVENT and S = '1';
>    if (S'EVENT and S = '1')
>    then  return true;
>    else return  false;
>    end if;
>   end RISING_EDGE;
2565,2569c2653,2657
<   -- Id: E.2
<   function FALLING_EDGE (signal S: BIT) return BOOLEAN is
<   begin
<     return S'EVENT and S = '0';
<   end FALLING_EDGE;
---
>   -- Id: E.2
>   function FALLING_EDGE (signal S: BIT) return BOOLEAN is
>   begin
>     return S'EVENT and S = '0';
>   end FALLING_EDGE;
```

A.2 The Filter Modules

The **Filter** data path consists of several features. All features are declared as components in the **Filter** top level file. The RTL architecture body of the **Filter** is constructed using these data path components.

A.2.1 The Filter

The VHDL description of "filter" entity and its RTL level architecture body is shown in this section. This is the top level file in this design. All components are declared and instantiated in this file.

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.numeric_bit.all;

entity filter is
  port ( clk : in std_logic;
         reset : in std_logic;
         input_data : in std_logic_vector(31 downto 0);
         output_data : out std_logic_vector(51 downto 0);
         a : out std_logic_vector(9 downto 0);
         d : in std_logic_vector(39 downto 0);
         ir_mem_enable : out std_logic;
         mem_a : out std_logic_vector(9 downto 0);
         mem_d_out : in std_logic_vector(47 downto 0);
         mem_d_in : out std_logic_vector(47 downto 0);
         mem_write_en : out std_logic;
         output_enable : out std_logic;
         mem_enable : out std_logic);
end entity filter;


architecture rtl of filter is

  -- The FIR filter Input Register
  component input_reg is
    port (d : in std_logic_vector(31 downto 0);
          q_low : out std_logic_vector(15 downto 0);
          q_high : out std_logic_vector(15 downto 0);
          enable : in std_logic;
          out_en : in std_logic);
  end component input_reg;

  -- The Binary / 2-digit 2DLNS Converter
  component  serial2digithighlow17
    port (CK : in std_logic;
          reset : in std_logic;
          activate : in std_logic;
          i : in std_logic_vector(16 downto 0);
```

```
        ready : out std_logic;
        output_sign : out std_logic_vector(1 downto 0);
        output_first : out std_logic_vector(11 downto 0);
        output_second : out std_logic_vector(9 downto 0));
end component;

-- The FIR filter Binary / 2DLNS Conversion Register
component conv_out_reg is
  port ( enable : in std_logic;
         ready_low : in std_logic;
         ready_high : in std_logic;
         output_sign_low : in std_logic_vector(1 downto 0);
         output_first_low : in std_logic_vector(11 downto 0);
         output_second_low : in std_logic_vector(9 downto 0);
         output_sign_high : in std_logic_vector(1 downto 0);
         output_first_high : in std_logic_vector(11 downto 0);
         output_second_high : in std_logic_vector(9 downto 0);
         output : out std_logic_vector(47 downto 0));
end component;

-- The FIR filter Multiply and Accumulate unit
component top_mac is
  port (clk,clr : in std_logic;
        x : in std_logic_vector(47 downto 0);
        y : in std_logic_vector(39 downto 0);
        p : out std_logic_vector(51 downto 0));
end component top_mac;

-- The Controller
component controller is
  port ( clk : in std_logic;
         reset : in std_logic;
         ir_mem_enable : out std_logic;
         mem_write_en : out std_logic;
         mem_enable : out std_logic;
         top_mac_clr : out std_logic;
         btc_reset_low : out std_logic;
         btc_ready_low : in std_logic;
         btc_activate_low : out  std_logic;
         btc_reset_high : out std_logic;
         btc_ready_high : in std_logic;
         btc_activate_high : out  std_logic;
         in_reg_enable : out std_logic;
         in_reg_out_en : out std_logic;
         out_reg_enable : out std_logic;
         ctrl_mem_a : out std_logic_vector(9 downto 0);
         ctrl_ir_mem_a : out std_logic_vector(9 downto 0));
end component controller;
```

```
  signal data_out : std_logic_vector(51 downto 0);
  signal data_in_low : std_logic_vector(15 downto 0);
  signal data_in_high : std_logic_vector(15 downto 0);
  signal data_in_low_17 : std_logic_vector(16 downto 0);
  signal data_in_high_17 : std_logic_vector(16 downto 0);
  signal a_out : std_logic_vector(9 downto 0);
  signal top_mac_clr : std_logic;
  signal btc_reset_low : std_logic;
  signal btc_ready_low : std_logic;
  signal btc_activate_low : std_logic;
  signal btc_reset_high : std_logic;
  signal btc_ready_high : std_logic;
  signal btc_activate_high : std_logic;
  signal ready : std_logic;
  signal in_reg_enable : std_logic;
  signal in_reg_out_en : std_logic;
  signal out_reg_enable : std_logic;
  signal ctrl_mem_a : std_logic_vector(9 downto 0);
  signal mem_data_in : std_logic_vector(47 downto 0);
  signal mem_data_out : std_logic_vector(47 downto 0);
  signal ctrl_ir_mem_a : std_logic_vector(9 downto 0);
  signal output_sign_low : std_logic_vector(1 downto 0);
  signal output_first_low : std_logic_vector(11 downto 0);
  signal output_second_low : std_logic_vector(9 downto 0);
  signal output_sign_high : std_logic_vector(1 downto 0);
  signal output_first_high : std_logic_vector(11 downto 0);
  signal output_second_high : std_logic_vector(9 downto 0);

begin

  -- Input / Output ports connections
  a <= ctrl_ir_mem_a;
  mem_a <=  ctrl_mem_a;
  output_enable <= out_reg_enable;
  ready <= btc_ready_low and btc_ready_high;

  data_in_low_17 <= "0" & data_in_low(15 downto 0) ;
  data_in_high_17 <= data_in_high(15) & data_in_high(15 downto 0) ;

  input_register : input_reg
    port map ( d => input_data , q_low => data_in_low , q_high =>
               data_in_high , enable => in_reg_enable , out_en =>
               in_reg_out_en );

  BTC_converter_low : serial2digithighlow17
    port map ( CK => clk , reset => btc_reset_low , activate =>
               btc_activate_low , i => data_in_low_17 , ready =>
               btc_ready_low , output_sign => output_sign_low ,
```

```
                output_first => output_first_low,
                output_second => output_second_low );

  BTC_converter_high : serial2digithighlow17
    port map ( CK => clk, reset => btc_reset_high, activate =>
                btc_activate_high, i => data_in_high_17, ready =>
                btc_ready_high, output_sign => output_sign_high,
                output_first => output_first_high,
                output_second => output_second_high );

  BTC_reg : conv_out_reg
    port map ( enable => ready,
                ready_low => btc_ready_low,
                output_sign_low => output_sign_low,
                output_first_low => output_first_low,
                output_second_low => output_second_low,
                ready_high => btc_ready_high,
                output_sign_high => output_sign_high,
                output_first_high => output_first_high,
                output_second_high => output_second_high,
                output => mem_d_in);

  multiplier_accumulater : top_mac
    port map ( clk => clk, clr => top_mac_clr, x => mem_d_out, y => d,
               p => output_data );

  the_controller : controller
    port map ( clk => clk, reset => reset,
                ir_mem_enable => ir_mem_enable,
                mem_write_en => mem_write_en,
                mem_enable => mem_enable,
                top_mac_clr => top_mac_clr,
                btc_reset_low => btc_reset_low,
                btc_ready_low => btc_ready_low,
                btc_activate_low => btc_activate_low,
                btc_reset_high => btc_reset_high,
                btc_ready_high => btc_ready_high,
                btc_activate_high => btc_activate_high,
                in_reg_enable => in_reg_enable,
                in_reg_out_en => in_reg_out_en,
                out_reg_enable => out_reg_enable,
                ctrl_mem_a => ctrl_mem_a,
                ctrl_ir_mem_a => ctrl_ir_mem_a);


end architecture rtl;
```

A.2.2 The Input Register

The input data is written from a file and stored into this register.

```
library ieee;
use ieee.std_logic_1164.all;

use work.numeric_bit.all;

entity input_reg is
  port ( d : in std_logic_vector(31 downto 0);
         q_low : out std_logic_vector(15 downto 0);
         q_high : out std_logic_vector(15 downto 0);
         enable : in std_logic;
         out_en : in std_logic);
end entity input_reg;


architecture behavior of input_reg is

  signal stored_value_low : std_logic_vector(15 downto 0);
  signal stored_value_high : std_logic_vector(15 downto 0);

begin

  -- data is stored when input is enabled
  input : process ( d, enable) is

  begin
    if enable = '1' then
        stored_value_low <= d(15 downto 0);
        stored_value_high <= d(31 downto 16);
    end if;

  end process input;

  -- stored data is written to output when output is enabled
  q_low <= stored_value_low when out_en = '1' else
        "ZZZZZZZZZZZZZZZZ" ;
  q_high <= stored_value_high when out_en = '1' else
        "ZZZZZZZZZZZZZZZZ" ;

end architecture behavior;
```

A.2.3 The Binary / 2DLNS Converter

For the binary / 2DLNS conversion, the modified version of the Verilog code in [29] is used. The HDL code has been written fully parametrized and the parameters should be set when the shell script which generates Verilog module is run. The definition of these parameters were also included in [29]. Before running this script, the optimal base has been computed and stored in an ASCII file, 13-65536-nzsmn2-00. The binary / 2DLNS converter module in **Filter** has been generated by setting the parameters as:

makeserial2digithighlow.sh 13-65536-nzsmn2-00 17 6 5 3 2 1 -nz > converter.v

The name of generated module is **serial2digithighlow17** which is instantiated, as an component, in the **Filter** top module. This Verilog file is shown here.

```
`ifdef DC

`else

`timescale 1ns/10ps

// Define serial simulation module
module simulate_serial;

parameter inputbits = 17;
parameter firstbasebits = 6;
parameter secondbasebits = 5;
parameter twobitmode = 0;
parameter digits = 2;
parameter starter = -65536;
parameter stopper = 65535;

// Define systm clock
reg CK;

// Test bench registers, just one
reg [63:0] i;

// Interfacing registers
reg reset;
```

```
reg activate;
reg [ inputbits - 1 : 0 ] i0;
wire ready;
wire [ ( digits * ( twobitmode + 1 ) ) - 1 : 0 ] output_sign;
wire [ ( firstbasebits * digits ) - 1 : 0 ] output_first;
wire [ ( secondbasebits * digits ) - 1 : 0 ] output_second;

// Reordered converter results
reg [ twobitmode : 0 ] final_sign[ 0 : digits - 1];
reg [ firstbasebits - 1 : 0 ] final_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 : 0 ] final_second[ 0 : digits - 1 ];

// Temporary registers
reg [ ( digits * ( twobitmode + 1 ) ) - 1 : 0 ] output_sign2;
reg [ ( firstbasebits * digits ) - 1 : 0 ] output_first2;
reg [ ( secondbasebits * digits ) - 1 : 0 ] output_second2;

// Conversion operates on rising edge
serial2digithighlow17
  serial(
    CK,
    reset,
    activate,
    i0,
    ready,
    output_sign,
    output_first,
    output_second
  );

integer q;

time starttime;

// Initialization routine
initial
begin

  // Disable verilog input logging
  $nokey;

  // Disable verilog output logging
  $nolog;

  // Initialize clock
  CK = 0;

  // Set initial input
  i = starter - 1;
```

```
  // Disable converter
  activate = 0;

  // Reset converter
  reset = 1;

end

// Set up the clock to pulse every 10 units
always #10 CK = !CK;

// On the rising edge
always @( posedge CK )
begin

  // When conversion is done, read converter results
  if( ready == 1 && activate == 0 )
  begin

    $display( "!total_time=%t", $time - starttime );

    // Copy the conerter results
    output_sign2 = output_sign;
    output_first2 = output_first;
    output_second2 = output_second;

    // Write out the input for comparison
    $write( "+%b", i0 );

    // Loop through all the digits to extract the sign and indices
    // We can not use a variable in the indexing, so we use shifts
       instead
    for( q = 0 ; q < digits ; q = q + 1 )
    begin

      // Extract the sign
      final_sign[ q ] = output_sign2[ twobitmode : 0 ];
      // Shift it down for the next extraction
      output_sign2 = output_sign2 >> ( twobitmode + 1 );

      // Extract the first index
      final_first[ q ] = output_first2[ firstbasebits - 1 : 0 ];
      // Shift it down for the next extraction
      output_first2 = output_first2 >> firstbasebits;

      // Extract the second index
      final_second[ q ] = output_second2[ secondbasebits - 1 : 0 ];
      // Shift it down for the next extraction
```

```
      output_second2 = output_second2 >> secondbasebits;

      // Write out the sign and indices
      $write( "\t%b\t%b\t%b", final_sign[ q ], final_first[ q ],
          final_second[ q ] );

    end

    // Write a new line
    $write( "\n" );

    // Increment the test input
    i = i + 1;

    // Stop simulations when it reaches the last input
    if( i - 1 == stopper )
    begin

      $finish;

    end

    // Set the input
    i0 <= i[ inputbits - 1 : 0 ];

    // Request a conversion
    activate <= 1;

    // Record start time
    starttime = $time;

  end
  else
  begin

    // Conversion is either busy or in reset

    // Turn off conversion
    activate <= 0;

    // Take out of reset
    reset <= 0;

  end

end

endmodule
```

```
`endif

// Define Serial 2 Digit High/Low Conversion Module

module serial2digithighlow17(
  CK,
  reset,
  activate,
  i,
  ready,
  output_sign,
  output_first,
  output_second
);

// Define parameters

// inputbits: Input word size in bits
// internalbits: Internal working bit size (>= inputbits)
// firstbasebits: Number of bits for the first base index
// secondbasebits: Number of bits for the second base index
// secondbasereservedbits: Number of bits for exclusion on the
   secondbase index
// normalizerbits: Number of bits for shift from normalizer
// twobitmode: Sign mode (1 for two-bits, 0 for one-bit)

parameter inputbits = 17;
parameter internalbits = 18;
parameter firstbasebits = 6;
parameter secondbasebits = 5;
parameter secondbasereservedbits = 3;
parameter normalizerbits = 5;
parameter twobitmode = 0;

// Dummy parameter, should be for the number of digits
parameter dummy = 2;

// This is a two digit 2DLNS converter
parameter digits = 2;

// Internal comparitor accuracy, this should be adjusted after
   simulation
parameter shiftdifference = 20;

// Define ports

// Data is processed on rising edge
input CK;
```

```
// Reset is active high
input reset;

// Input word in 2's complement
input [ inputbits - 1 : 0 ] i;

// Set to 1 to run converstion on input
input activate;

// Is set to 1 when output data is ready
output ready;
reg ready;

// Output Signs (concatenated)
output [ ( twobitmode + 1 ) * digits - 1 : 0 ] output_sign;
reg [ ( twobitmode + 1 ) * digits - 1 : 0 ] output_sign;

// Output Binary exponent (concatenated)
output [ ( firstbasebits * digits ) - 1 : 0 ] output_first;
reg [ ( firstbasebits * digits ) - 1 : 0 ] output_first;

// Output OtherBase exponent (concatenated)
output [ ( secondbasebits * digits ) - 1 : 0 ] output_second;
reg [ ( secondbasebits * digits ) - 1 : 0 ] output_second;

reg [ inputbits - 1 : 0 ] sep_manin;
wire [ internalbits - 1 : 0 ] sep_manout;
wire [ twobitmode : 0 ] sep_signout;

separatesign_noclk
  #(
    inputbits,
    internalbits,
    twobitmode
  )
  ss
  (
    sep_manin,
    sep_manout,
    sep_signout
  );

reg [ internalbits - 1 : 0 ] norm_manin;
reg [ twobitmode : 0 ] norm_signin;

wire [ internalbits - 1 - twobitmode : 0 ] norm_manout;
wire [ twobitmode : 0 ] norm_signout;
wire [ normalizerbits - 1 : 0 ] norm_shift;
```

```
normalizer_noclk
  #(
    internalbits,
    normalizerbits,
    twobitmode
  )
  no
  (
    norm_manin,
    norm_signin,
    norm_manout,
    norm_signout,
    norm_shift
  );

reg [ internalbits - 1 - twobitmode : 0 ] ralut_manin;
wire [ internalbits - 1 - twobitmode : 0 ] ralut_manlow;
wire [ internalbits : 0 ] ralut_manhigh;
wire [ firstbasebits - 1 : 0 ] ralut_firstlow, ralut_firsthigh;
wire [ secondbasebits - 1 : 0 ] ralut_secondlow, ralut_secondhigh;

ralut6_473192379_2184_noclk
  #(
    internalbits - twobitmode,
    firstbasebits,
    secondbasebits,
    internalbits + 1,
    firstbasebits,
    secondbasebits,
    internalbits - twobitmode
  )
  ralut
  (
    ralut_manlow,
    ralut_firstlow,
    ralut_secondlow,
    ralut_manhigh,
    ralut_firsthigh,
    ralut_secondhigh,
    ralut_manin
  );

reg [ twobitmode : 0 ] final_sign[ 0 : digits - 1 ];
reg [ firstbasebits - 1 : 0 ] final_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 : 0 ] final_second[ 0 : digits - 1 ];

reg [ twobitmode : 0 ] other_sign;
reg [ firstbasebits - 1 :0 ] other_first[ 0 : digits - 1 ];
reg [ secondbasebits - 1 :0 ] other_second[ 0 : digits - 1 ];
```

```
reg [ internalbits : 0 ] error_low, error_high, other_error;

reg [ normalizerbits : 0 ] other_shiftdifference, best_shiftdifference;
reg [ normalizerbits - 1 : 0 ] other_lastshift, best_lastshift;
reg [ internalbits - 1 + shiftdifference : 0 ] other_errorcompare,
    best_errorcompare;

reg [ internalbits - 1 : 0 ] best_error;
reg [ normalizerbits : 0 ] best_shift, other_shift;

// State machine register
reg [ 3 : 0 ] state;

`ifdef DC

`else

// For simulation optimize the barrel shifter
integer max_shiftdifference;

// Initialization routine
initial
begin

  // Reset the shiftdifference maximum
  max_shiftdifference = 0;

  // Stop if more than 2 digits were passed
  if( dummy != digits ) $stop;

end

`endif

// Reset integer
integer j;

// Give Synopsys some hints on the synthesis
//synopsys state_vector state
//synopsys sync_set_reset "reset"

always @( posedge CK )
begin: main

  // Synchronous Reset active high
  if( reset )
  begin
```

```
    // Reset state machine
    state <= 0;

    // Set output to invalid
    ready <= 0;

    // Set all MDLNS output to zeros
    for( j = 0 ; j < 2 ; j = j + 1)
    begin

      final_sign[ j ] <= 0;
      final_first[ j ] <= 0;
      final_second[ j ] <= 0;

    end

  end
  else
  begin

    // Process each state
    case( state )

      0:
      begin

        // Conversion starts if activate line is high
        if( activate )
        begin

          // Load input data into sign separator
          sep_manin <= i;

          // Invalidate output
          ready <= 0;

          $display( "Conversion starts for %d", i );

          // Set all MDLNS output to zeros incase conversion finishes
              early
          for( j = 0 ; j < 2 ; j = j + 1)
          begin

            final_sign[ j ] <= 0;
            final_first[ j ] <= 0;
            final_second[ j ] <= 0;

          end
```

```
      // Move to next state
      state <= state + 1;

    end
    else
    begin

      // No conversion signal, continue looping
      ready <= 1;

    end

  end

  1: begin

    // Get sign and data from sign separator, put it in to the
       normalizer
    norm_signin <= sep_signout;
    norm_manin <= sep_manout;

    // Move to next state
    state <= state + 1;

  end

  2:
  begin

    if( twobitmode && norm_signout == 0 )
    begin

      // input is zero, end conversion
      ready <= 1;
      state <= 0;

    end
    else
    begin

      // Get shift and sign from normalizer
      final_sign[0] <= norm_signout;
      other_shift <= norm_shift;

      // Load RALUT with the mantissa
      ralut_manin <= norm_manout;

      // Move to next state
      state <= state + 1;
```

```
  end

end

3:
begin

  // Determine the error between the two RALUT outputs and the
     input
  if( twobitmode )
  begin

    error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
    other_error <= ralut_manhigh - { 2'b01, ralut_manin };

  end
  else
  begin

    error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
    other_error <= ralut_manhigh - { 1'b0, ralut_manin };

  end

  // Load normalizer with low error and sign of input (to
     determine if it is zero)
  norm_manin <= error_low[ internalbits - 1 : 0 ];
  norm_signin <= final_sign[ 0 ];

  // Save the low approximation as the current result
  final_first[ 0 ] <= ralut_firstlow - other_shift;
  final_second[ 0 ] <= ralut_secondlow;

  // Remember the high approximation for later
  other_first[ 0 ] <= ralut_firsthigh - other_shift;
  other_second[ 0 ] <= ralut_secondhigh;

  // Move to next state
  state <= state + 1;

end

4:
begin

  // Find the second digit for the low approximation
  if( twobitmode && norm_signout == 0 )
  begin
```

```
      // Error is zero, end conversion
      ready <= 1;
      state <= 0;

    end
    else
    begin

      // Set output sign
      final_sign[1] <= norm_signout;
      // Save accumulated shift
      best_shift <= other_shift + norm_shift;
      // Load RALUT
      ralut_manin <= norm_manout;

      // Move to next state
      state <= state + 1;

    end

  end

  5:
  begin

    // Determine the error between the two RALUT outputs and the
    // input We do this since we only have one RALUT, we have to
    // find if the low  or the high approximation is best
    if( twobitmode )
    begin

      error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
      error_high = ralut_manhigh - { 2'b01, ralut_manin };

    end
    else
    begin

      error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
      error_high = ralut_manhigh - { 1'b0, ralut_manin };

    end

    // Set the current result as the best approximation
    if( error_low < error_high )
    begin

      final_first[1] <= ralut_firstlow - best_shift;
```

```
      final_second[ 1 ] <= ralut_secondlow;
      best_error <= error_low;

    end
    else
    begin

      final_first[ 1 ] <= ralut_firsthigh - best_shift;
      final_second[ 1 ] <= ralut_secondhigh;
      best_error <= error_high;

    end

    // Load normalizer with high error from before and sign of -1
    norm_manin <= other_error[ internalbits - 1 : 0 ];

    if( twobitmode )
    begin

      if( final_sign[ 0 ] == 1 )
      begin

        norm_signin <= 2'b11;

      end
      else
      begin

        norm_signin <= 2'b01;

      end

    end
    else
    begin

      norm_signin <= ~final_sign[ 0 ];

    end

    // Move to next state
    state <= state + 1;

  end

  6:
  begin

    // Save accumulated shift and sign
```

```
    other_shift <= other_shift + norm_shift;
    other_sign <= norm_signout;

    // Load RALUT
    ralut_manin <= norm_manout;

    // Move to next state
    state <= state + 1;

  end

  7:
  begin

    // Determine the error between the two RALUT outputs and the
       input
    if( twobitmode )
    begin

      error_low = { 2'b01, ralut_manin } - { 2'b01, ralut_manlow };
      error_high = ralut_manhigh - { 2'b01, ralut_manin };

    end
    else
    begin

      error_low = { 1'b0, ralut_manin } - { 1'b0, ralut_manlow };
      error_high = ralut_manhigh - { 1'b0, ralut_manin };

    end

    // Store this result temporarily
    if ( error_low < error_high )
    begin

      other_first[1] <= ralut_firstlow - other_shift;
      other_second[1] <= ralut_secondlow;
      other_error <= error_low;

    end
    else
    begin

      other_first[1] <= ralut_firsthigh - other_shift;
      other_second[1] <= ralut_secondhigh;
      other_error <= error_high;

    end
```

```
        // Move to next state
        state <= state + 1;

      end

      8:
      begin

        // Find the difference in shifts between the two approximations
        other_shiftdifference = other_shift - best_shift;
        best_shiftdifference = best_shift - other_shift;

        if( other_shiftdifference[ normalizerbits ] == 1'b0 )
        begin

          // high approximation must be shifted right
          other_lastshift <= other_shiftdifference;
          best_lastshift <= 0;

        end
        else
        begin

          // low approximation must be shifted right
          other_lastshift <= 0;
          best_lastshift <= best_shiftdifference;

        end

        // Move to next state
        state <= state + 1;

      end

      9:
      begin

`ifdef DC

// For synthsis do not do this

`else

        // Find the maximum shift to minimize the hardware

        if( best_shift != 0 )
        begin

          if( other_lastshift > max_shiftdifference )
```

```
            begin

              max_shiftdifference = other_lastshift;
              $display( "!max_shiftdifference=%d", max_shiftdifference );

            end
            if( best_lastshift > max_shiftdifference )
            begin

              max_shiftdifference = best_lastshift;
              $display( "!max_shiftdifference=%d", max_shiftdifference );

            end

          end

`endif

          // Shift both errors accordingly
          other_errorcompare = ( other_error << shiftdifference ) >>
              other_lastshift;
          best_errorcompare = ( best_error << shiftdifference ) >>
              best_lastshift;

          // Compare
          if( other_errorcompare < best_errorcompare )
          begin

            // Higher approximation is better, move into output results
            final_first[ 0 ] <= other_first[ 0 ];
            final_second[ 0 ] <= other_second[ 0 ];
            final_sign[ 1 ] <= other_sign;
            final_first[ 1 ] <= other_first[ 1 ];
            final_second[ 1 ] <= other_second[ 1 ];

          end

          // Conversion complete
          ready <= 1;
          state <= 0;

        end

        // All other cases, reset the state machine
        default:
        begin

          state <= 0;
          ready <= 0;
```

```
      end

    endcase

  end

end // always

`ifdef DC

// For synthesis
always

`else

// For simulation
always @( posedge ready )

`endif

begin

// Concatinate outputs for universal access
  output_sign = { final_sign[ 1 ], final_sign[ 0 ] };
  output_first = { final_first[ 1 ], final_first[ 0 ] };
  output_second = { final_second[ 1 ], final_second[ 0 ] };

end

endmodule

// Module for separating an integer into a number and a sign
//  This module doesn't generate a zero sign, simply a 1 or -1
module separatesign_noclk ( i, o, os );

// Default parameters

// Input word size in bits
parameter inputbits = 16;

// Output word size in bits (must be equal to or greater than inputbits)
parameter outputbits = 15;

// twobit sign indicator
parameter twobitmode = 1;

// Define ports
```

```
// Input of module in 2's complements form
input [ inputbits - 1 : 0 ] i;

// Output number of module in binary form
output [ outputbits - 1 : 0 ] o;
reg [ outputbits - 1 : 0 ] o;

// Output sign of module
output [ twobitmode : 0 ] os;
reg [ twobitmode : 0 ] os;

`ifdef DC

`else

initial
begin

  // Stop simulation if output is smaller than input
  if( outputbits < inputbits ) $stop;

end

`endif

// Two intermediate registers to expand word length
reg [ 63 : 0 ] t1, t2;

always @( i )
begin

  // Generate negative value of input, but extend the sign too
  t1 = 0 - { { ( 63 - inputbits ) { i[ inputbits - 1 ] } }, i[ inputbits
       - 2 : 0 ] };

  // Copy input
  t2 = i;

  // Check the high bit of the input, if it is one, then input
  //  is negative
  if( i[ inputbits - 1 ] == 1'b1 )
  begin

    // Set output to negated input
    o = t1[ outputbits - 1 : 0 ];

    // Set sign to -1, 1 for one bit sign mode
    if( twobitmode )
    begin
```

```
      os = 2'b11;

    end
    else
    begin

      os = 1'b1;

    end

  end
  else
  begin

    // Set output to extended input
    o = t2[ outputbits - 1 : 0 ];

    // Set sign to 1, 0 for one bit sign mode
    if( twobitmode )
    begin

      os = 2'b01;

    end
    else
    begin

      os = 1'b0;

    end

  end

end

endmodule

// Module for normalizing and integer and possibly setting the sign to
    zero
module normalizer_noclk( i, is, o, os, s );

// Default parameters

// inputbits is the input number of bits, the output will be inputbits
    -1 bits since
//  the first "1" will be omitted
parameter inputbits = 16;
```

```
// shiftbits is the number of bits to describe the shift of the
    normalization
parameter shiftbits = 4;

// two bit mode
parameter twobitmode = 1;

// Define ports

// input
input [ inputbits - 1 : 0 ] i;

// input sign
input [ twobitmode : 0 ] is;

// output of normalization,
output [ inputbits - 1 - twobitmode : 0 ] o;
reg [ inputbits - 1 - twobitmode : 0 ] o;

// output sign, will be zero is input sign is also
output [ twobitmode : 0 ] os;
reg [ twobitmode : 0 ] os;

// output shift
output [ shiftbits - 1 : 0 ] s;
reg [ shiftbits - 1 : 0 ] s;

// temporary variables
reg [ inputbits - 1 : 0 ] t;
reg z;

// internal counter
integer c;

always @( i or is )
begin

  // there are other ways to do this procedure, but through synthesis
  //  this turns out to be the smallest

  // set shift to zero
  s = 0;

  // set loop control
  z = 0;

  // loop through all bits from left to right
  for( c = inputbits - 1 ; c >= 0 ; c = c - 1 )
  begin
```

```
    // if the bit is zero and no ones have been encountered,
    //  keep updating this shift
    if( i[ c ] == 1'b0 && z == 1'b0 )
    begin

      // record the size
      s = inputbits - c;

      // set the loop to keep going
      z = 0;

    end
    else
    begin

      // don't infer any latches
      s = s;
      // a one has been spotted, no more updates
      z = 1;

    end

  end

  // check for twobit mode
  if( twobitmode )
  begin

    // check the loop control bit
    if( z == 0 )
    begin

      // if zero, set the output sign to zero
      os = 0;
      s = 0;

    end
    else
    begin

      // if not, pass the input sign through
      os = is;

    end

  end
  else
  begin
```

```
    // pass the input sign through
    os = is;

    // check the loop control bit
    if( z == 0 )
    begin

      s = 0;

    end

  end

  // shift temporary register
  t = i << s;

  // set output from temporary register (we can not do this in one step)
  o = t[ inputbits - 1 - twobitmode : 0 ];

end

endmodule

module ralut6_473192379_2184_noclk(
        o1,
        o2,
        o3,
        o4,
        o5,
        o6,
        i );

// Default parameters
parameter o1bits = 18;
parameter o2bits = 6;
parameter o3bits = 5;
parameter o4bits = 19;
parameter o5bits = 6;
parameter o6bits = 5;
parameter ibits = 18;
parameter rasize = 26;

// Define ports

// data to match
input [ ibits - 1 : 0 ] i;

// datas to output
```

```
output [ o1bits - 1 : 0 ] o1;
output [ o2bits - 1 : 0 ] o2;
output [ o3bits - 1 : 0 ] o3;
output [ o4bits - 1 : 0 ] o4;
output [ o5bits - 1 : 0 ] o5;
output [ o6bits - 1 : 0 ] o6;
reg [ o1bits - 1: 0 ] o1;
reg [ o2bits - 1: 0 ] o2;
reg [ o3bits - 1: 0 ] o3;
reg [ o4bits - 1: 0 ] o4;
reg [ o5bits - 1: 0 ] o5;
reg [ o6bits - 1: 0 ] o6;
reg [ o1bits - 1: 0 ] t1;
reg [ o2bits - 1: 0 ] t2;
reg [ o3bits - 1: 0 ] t3;
reg [ o4bits - 1: 0 ] t4;
reg [ o5bits - 1: 0 ] t5;
reg [ o6bits - 1: 0 ] t6;

reg [ ibits - 1 : 0 ] romAddr[ rasize - 1 : 0 ];
reg [ o1bits - 1 : 0 ] romOut1[ rasize - 1 : 0 ];
reg [ o2bits - 1 : 0 ] romOut2[ rasize - 1 : 0 ];
reg [ o3bits - 1 : 0 ] romOut3[ rasize - 1 : 0 ];
reg [ o4bits - 1 : 0 ] romOut4[ rasize - 1 : 0 ];
reg [ o5bits - 1 : 0 ] romOut5[ rasize - 1 : 0 ];
reg [ o6bits - 1 : 0 ] romOut6[ rasize - 1 : 0 ];

integer p;

`ifdef DC
`else

initial
begin

// include data information here

romAddr[0]=18'b000000000000000000;
romAddr[1]=18'b100000000000000000;
romAddr[2]=18'b100000111000111111;
romAddr[3]=18'b100001110101001111;
romAddr[4]=18'b100010110001011111;
romAddr[5]=18'b100011101111011011;
romAddr[6]=18'b100100110000111001;
romAddr[7]=18'b100101110010010111;
romAddr[8]=18'b100110110111100110;
romAddr[9]=18'b100111111100110100;
romAddr[10]=18'b101001000011111101;
romAddr[11]=18'b101010001111001100;
```

```
romAddr[12]=18'b101011011010011010;
romAddr[13]=18'b101100100111101110;
romAddr[14]=18'b101101111001011101;
romAddr[15]=18'b101111001011001101;
romAddr[16]=18'b110000011111001110;
romAddr[17]=18'b110001111000000011;
romAddr[18]=18'b110011010000111000;
romAddr[19]=18'b110100101100001011;
romAddr[20]=18'b110110001100101100;
romAddr[21]=18'b110111101101001110;
romAddr[22]=18'b111001010011010001;
romAddr[23]=18'b111010111001010100;
romAddr[24]=18'b111100100010001101;
romAddr[25]=18'b111110010001000110;

romOut1[0]=18'b000000000000000000;
romOut2[0]=6'b010001;
romOut3[0]=5'b10000;
romOut4[0]=19'b0000000000000000000;
romOut5[0]=6'b010001;
romOut6[0]=5'b10000;
romOut1[1]=18'b100000000000000000;
romOut2[1]=6'b010001;
romOut3[1]=5'b00000;
romOut4[1]=19'b0100000111000111111;
romOut5[1]=6'b010101;
romOut6[1]=5'b01001;
romOut1[2]=18'b100000111000111111;
romOut2[2]=6'b010101;
romOut3[2]=5'b01001;
romOut4[2]=19'b0100001110101001111;
romOut5[2]=6'b001110;
romOut6[2]=5'b11001;
romOut1[3]=18'b100001110101001111;
romOut2[3]=6'b001110;
romOut3[3]=5'b11001;
romOut4[3]=19'b0100010110001011111;
romOut5[3]=6'b010010;
romOut6[3]=5'b00010;
romOut1[4]=18'b100010110001011111;
romOut2[4]=6'b010010;
romOut3[4]=5'b00010;
romOut4[4]=19'b0100011101111011011;
romOut5[4]=6'b010110;
romOut6[4]=5'b01011;
romOut1[5]=18'b100011101111011011;
romOut2[5]=6'b010110;
romOut3[5]=5'b01011;
romOut4[5]=19'b0100100110000111001;
```

```
romOut5[5]=6'b001111;
romOut6[5]=5'b11011;
romOut1[6]=18'b100100110000111001;
romOut2[6]=6'b001111;
romOut3[6]=5'b11011;
romOut4[6]=19'b0100101110010010111;
romOut5[6]=6'b010011;
romOut6[6]=5'b00100;
romOut1[7]=18'b100101110010010111;
romOut2[7]=6'b010011;
romOut3[7]=5'b00100;
romOut4[7]=19'b0100110110111100110;
romOut5[7]=6'b001100;
romOut6[7]=5'b10100;
romOut1[8]=18'b100110110111100110;
romOut2[8]=6'b001100;
romOut3[8]=5'b10100;
romOut4[8]=19'b0100111111100110100;
romOut5[8]=6'b010000;
romOut6[8]=5'b11101;
romOut1[9]=18'b100111111100110100;
romOut2[9]=6'b010000;
romOut3[9]=5'b11101;
romOut4[9]=19'b0101001000011111101;
romOut5[9]=6'b010100;
romOut6[9]=5'b00110;
romOut1[10]=18'b101001000011111101;
romOut2[10]=6'b010100;
romOut3[10]=5'b00110;
romOut4[10]=19'b0101010001111001100;
romOut5[10]=6'b001101;
romOut6[10]=5'b10110;
romOut1[11]=18'b101010001111001100;
romOut2[11]=6'b001101;
romOut3[11]=5'b10110;
romOut4[11]=19'b0101011011010011010;
romOut5[11]=6'b010001;
romOut6[11]=5'b11111;
romOut1[12]=18'b101011011010011010;
romOut2[12]=6'b010001;
romOut3[12]=5'b11111;
romOut4[12]=19'b0101100100111101110;
romOut5[12]=6'b010101;
romOut6[12]=5'b01000;
romOut1[13]=18'b101100100111101110;
romOut2[13]=6'b010101;
romOut3[13]=5'b01000;
romOut4[13]=19'b0101101111001011101;
romOut5[13]=6'b001110;
```

```
romOut6[13]=5'b11000;
romOut1[14]=18'b101101111001011101;
romOut2[14]=6'b001110;
romOut3[14]=5'b11000;
romOut4[14]=19'b0101111001011001101;
romOut5[14]=6'b010010;
romOut6[14]=5'b00001;
romOut1[15]=18'b101111001011001101;
romOut2[15]=6'b010010;
romOut3[15]=5'b00001;
romOut4[15]=19'b0110000011111001110;
romOut5[15]=6'b010110;
romOut6[15]=5'b01010;
romOut1[16]=18'b110000011111001110;
romOut2[16]=6'b010110;
romOut3[16]=5'b01010;
romOut4[16]=19'b0110001111000000011;
romOut5[16]=6'b001111;
romOut6[16]=5'b11010;
romOut1[17]=18'b110001111000000011;
romOut2[17]=6'b001111;
romOut3[17]=5'b11010;
romOut4[17]=19'b0110011010000111000;
romOut5[17]=6'b010011;
romOut6[17]=5'b00011;
romOut1[18]=18'b110011010000111000;
romOut2[18]=6'b010011;
romOut3[18]=5'b00011;
romOut4[18]=19'b0110100101100001011;
romOut5[18]=6'b010111;
romOut6[18]=5'b01100;
romOut1[19]=18'b110100101100001011;
romOut2[19]=6'b010111;
romOut3[19]=5'b01100;
romOut4[19]=19'b0110110001100101100;
romOut5[19]=6'b010000;
romOut6[19]=5'b11100;
romOut1[20]=18'b110110001100101100;
romOut2[20]=6'b010000;
romOut3[20]=5'b11100;
romOut4[20]=19'b0110111101101001110;
romOut5[20]=6'b010100;
romOut6[20]=5'b00101;
romOut1[21]=18'b110111101101001110;
romOut2[21]=6'b010100;
romOut3[21]=5'b00101;
romOut4[21]=19'b0111001010011010001;
romOut5[21]=6'b001101;
romOut6[21]=5'b10101;
```

```
romOut1[22]=18'b111001010011010001;
romOut2[22]=6'b001101;
romOut3[22]=5'b10101;
romOut4[22]=19'b0111010111001010100;
romOut5[22]=6'b010001;
romOut6[22]=5'b11110;
romOut1[23]=18'b111010111001010100;
romOut2[23]=6'b010001;
romOut3[23]=5'b11110;
romOut4[23]=19'b0111100100010001101;
romOut5[23]=6'b010101;
romOut6[23]=5'b00111;
romOut1[24]=18'b111100100010001101;
romOut2[24]=6'b010101;
romOut3[24]=5'b00111;
romOut4[24]=19'b0111110010001000110;
romOut5[24]=6'b001110;
romOut6[24]=5'b10111;
romOut1[25]=18'b111110010001000110;
romOut2[25]=6'b001110;
romOut3[25]=5'b10111;
romOut4[25]=19'b1000000000000000000;
romOut5[25]=6'b010010;
romOut6[25]=5'b00000;

end
`endif

// include line shorting here

always @( i )
begin

`ifdef DC

romAddr[0]=18'b000000000000000000;
romAddr[1]=18'b100000000000000000;
romAddr[2]=18'b100001110001111111;
romAddr[3]=18'b100001110101001111;
romAddr[4]=18'b100010110001011111;
romAddr[5]=18'b100011101111011011;
romAddr[6]=18'b100100110000111001;
romAddr[7]=18'b100101110010010111;
romAddr[8]=18'b100110110111100110;
romAddr[9]=18'b100111111100110100;
romAddr[10]=18'b101001000011111101;
romAddr[11]=18'b101010001111001100;
romAddr[12]=18'b101011011010011010;
romAddr[13]=18'b101100100111101110;
```

```
romAddr[14]=18'b101101111001011101;
romAddr[15]=18'b101111001011001101;
romAddr[16]=18'b110000011111001110;
romAddr[17]=18'b110001111000000011;
romAddr[18]=18'b110011010000111000;
romAddr[19]=18'b110100101100001011;
romAddr[20]=18'b110110001100101100;
romAddr[21]=18'b110111101101001110;
romAddr[22]=18'b111001010011010001;
romAddr[23]=18'b111010111001010100;
romAddr[24]=18'b111100100010001101;
romAddr[25]=18'b111110010001000110;

romOut1[0]=18'b000000000000000000;
romOut2[0]=6'b010001;
romOut3[0]=5'b10000;
romOut4[0]=19'b0000000000000000000;
romOut5[0]=6'b010001;
romOut6[0]=5'b10000;
romOut1[1]=18'b100000000000000000;
romOut2[1]=6'b010001;
romOut3[1]=5'b00000;
romOut4[1]=19'b0100000111000111111;
romOut5[1]=6'b010101;
romOut6[1]=5'b01001;
romOut1[2]=18'b100000111000111111;
romOut2[2]=6'b010101;
romOut3[2]=5'b01001;
romOut4[2]=19'b0100001110101001111;
romOut5[2]=6'b001110;
romOut6[2]=5'b11001;
romOut1[3]=18'b100001110101001111;
romOut2[3]=6'b001110;
romOut3[3]=5'b11001;
romOut4[3]=19'b0100010110001011111;
romOut5[3]=6'b010010;
romOut6[3]=5'b00010;
romOut1[4]=18'b100010110001011111;
romOut2[4]=6'b010010;
romOut3[4]=5'b00010;
romOut4[4]=19'b0100011101111011011;
romOut5[4]=6'b010110;
romOut6[4]=5'b01011;
romOut1[5]=18'b100011101111011011;
romOut2[5]=6'b010110;
romOut3[5]=5'b01011;
romOut4[5]=19'b0100100110000111001;
romOut5[5]=6'b001111;
romOut6[5]=5'b11011;
```

```
romOut1[6]=18'b100100110000111001;
romOut2[6]=6'b001111;
romOut3[6]=5'b11011;
romOut4[6]=19'b0100101110010010111;
romOut5[6]=6'b010011;
romOut6[6]=5'b00100;
romOut1[7]=18'b100101110010010111;
romOut2[7]=6'b010011;
romOut3[7]=5'b00100;
romOut4[7]=19'b0100110110111100110;
romOut5[7]=6'b001100;
romOut6[7]=5'b10100;
romOut1[8]=18'b100110110111100110;
romOut2[8]=6'b001100;
romOut3[8]=5'b10100;
romOut4[8]=19'b0100111111100110100;
romOut5[8]=6'b010000;
romOut6[8]=5'b11101;
romOut1[9]=18'b100111111100110100;
romOut2[9]=6'b010000;
romOut3[9]=5'b11101;
romOut4[9]=19'b0101001000011111101;
romOut5[9]=6'b010100;
romOut6[9]=5'b00110;
romOut1[10]=18'b101001000011111101;
romOut2[10]=6'b010100;
romOut3[10]=5'b00110;
romOut4[10]=19'b0101010001111001100;
romOut5[10]=6'b001101;
romOut6[10]=5'b10110;
romOut1[11]=18'b101010001111001100;
romOut2[11]=6'b001101;
romOut3[11]=5'b10110;
romOut4[11]=19'b0101011011010011010;
romOut5[11]=6'b010001;
romOut6[11]=5'b11111;
romOut1[12]=18'b101011011010011010;
romOut2[12]=6'b010001;
romOut3[12]=5'b11111;
romOut4[12]=19'b0101100100111101110;
romOut5[12]=6'b010101;
romOut6[12]=5'b01000;
romOut1[13]=18'b101100100111101110;
romOut2[13]=6'b010101;
romOut3[13]=5'b01000;
romOut4[13]=19'b0101101111001011101;
romOut5[13]=6'b001110;
romOut6[13]=5'b11000;
romOut1[14]=18'b101101111001011101;
```

```
romOut2[14]=6'b001110;
romOut3[14]=5'b11000;
romOut4[14]=19'b0101111001011001101;
romOut5[14]=6'b010010;
romOut6[14]=5'b00001;
romOut1[15]=18'b101111001011001101;
romOut2[15]=6'b010010;
romOut3[15]=5'b00001;
romOut4[15]=19'b0110000011111001110;
romOut5[15]=6'b010110;
romOut6[15]=5'b01010;
romOut1[16]=18'b110000011111001110;
romOut2[16]=6'b010110;
romOut3[16]=5'b01010;
romOut4[16]=19'b0110001111000000011;
romOut5[16]=6'b001111;
romOut6[16]=5'b11010;
romOut1[17]=18'b110001111000000011;
romOut2[17]=6'b001111;
romOut3[17]=5'b11010;
romOut4[17]=19'b0110011010000111000;
romOut5[17]=6'b010011;
romOut6[17]=5'b00011;
romOut1[18]=18'b110011010000111000;
romOut2[18]=6'b010011;
romOut3[18]=5'b00011;
romOut4[18]=19'b0110100101100001011;
romOut5[18]=6'b010111;
romOut6[18]=5'b01100;
romOut1[19]=18'b110100101100001011;
romOut2[19]=6'b010111;
romOut3[19]=5'b01100;
romOut4[19]=19'b0110110001100101100;
romOut5[19]=6'b010000;
romOut6[19]=5'b11100;
romOut1[20]=18'b110110001100101100;
romOut2[20]=6'b010000;
romOut3[20]=5'b11100;
romOut4[20]=19'b0110111101101001110;
romOut5[20]=6'b010100;
romOut6[20]=5'b00101;
romOut1[21]=18'b110111101101001110;
romOut2[21]=6'b010100;
romOut3[21]=5'b00101;
romOut4[21]=19'b0111001010011010001;
romOut5[21]=6'b001101;
romOut6[21]=5'b10101;
romOut1[22]=18'b111001010011010001;
romOut2[22]=6'b001101;
```

```
romOut3[22]=5'b10101;
romOut4[22]=19'b01110101111001010100;
romOut5[22]=6'b010001;
romOut6[22]=5'b11110;
romOut1[23]=18'b111010111001010100;
romOut2[23]=6'b010001;
romOut3[23]=5'b11110;
romOut4[23]=19'b01111001000010001101;
romOut5[23]=6'b010101;
romOut6[23]=5'b00111;
romOut1[24]=18'b111001000010001101;
romOut2[24]=6'b010101;
romOut3[24]=5'b00111;
romOut4[24]=19'b01111100100010000110;
romOut5[24]=6'b001110;
romOut6[24]=5'b10111;
romOut1[25]=18'b111110010001000110;
romOut2[25]=6'b001110;
romOut3[25]=5'b10111;
romOut4[25]=19'b1000000000000000000;
romOut5[25]=6'b010010;
romOut6[25]=5'b00000;

`endif

        t1 = romOut1[ 0 ];
        t2 = romOut2[ 0 ];
        t3 = romOut3[ 0 ];
        t4 = romOut4[ 0 ];
        t5 = romOut5[ 0 ];
        t6 = romOut6[ 0 ];

        for( p = 1 ; p < rasize ; p = p + 1 )
        begin: main

                if (p < rasize-1)
                begin
                        if ( i >= romAddr[ p ] && i < romAddr[ p+1 ] )
                        begin
                                t1 = romOut1[ p ];
                                t2 = romOut2[ p ];
                                t3 = romOut3[ p ];
                                t4 = romOut4[ p ];
                                t5 = romOut5[ p ];
                                t6 = romOut6[ p ];
//                              disable main;
                        end
                end
                else
```

```
                begin
                        if ( i >= romAddr[ p ] )
                        begin
                                t1 = romOut1[ p ];
                                t2 = romOut2[ p ];
                                t3 = romOut3[ p ];
                                t4 = romOut4[ p ];
                                t5 = romOut5[ p ];
                                t6 = romOut6[ p ];
//                              disable main;
                        end
                end

        end

        o1 <= t1;
        o2 <= t2;
        o3 <= t3;
        o4 <= t4;
        o5 <= t5;
        o6 <= t6;

end

endmodule

// RALUT contents ( address, output(s) ... )

/*

000000000000000000      000000000000000000      010001  10000
    000000000000000000      010001  10000
100000000000000000      100000000000000000      010001  00000
    010000011110001111111       010101  01001
100000111000111111      100000111000111111      010101  01001
    010000111010100111111       001110  11001
100001110101001111      100001110101001111      001110  11001
    010001011000101111111       010010  00010
100010110001011111      100010110001011111      010010  00010
    010001110111101101111       010110  01011
100011101111011011      100011101111011011      010110  01011
    010010011000011100111       001111  11011
100100110000111001      100100110000111001      001111  11011
    010010111001001011111       010011  00100
100101110010010111      100101110010010111      010011  00100
    010011010111110011000       001100  10100
100110110111100110      100110110111100110      001100  10100
    010011111110011010000       010000  11101
100111111100110100      100111111100110100      010000  11101
```

```
    0101001000011111101      010100  00110
10100100001111101       1010010000111111101      010100  00110
    0101010001111001100      001101  10110
10101000111100 1100      1010100011110011 00      001101  10110
    0101011011010011010      010001  11111
10101101101001 1010      1010110110100110 10      010001  11111
    0101100100111101110      010101  01000
10110010011110 1110      1011001001111011 10      010101  01000
    0101101111001011101      001110  11000
10110111100101 1101      1011011110010111 01      001110  11000
    0101111001011001101      010010  00001
10111100101100 1101      1011110010110011 01      010010  00001
    0110000011111001110      010110  01010
11000001111100 1110      1100000111110011 10      010110  01010
    0110001111000000011      001111  11010
11000111100000 0011      1100011110000000 11      001111  11010
    0110011010000111000      010011  00011
11001101000011 1000      1100110100001110 00      010011  00011
    0110100101100001011      010111  01100
11010010110000 1011      1101001011000010 11      010111  01100
    0110110001100101100      010000  11100
11011000110010 1100      1101100011001011 00      010000  11100
    0110111101101001110      010100  00101
11011110110100 1110      1101111011010011 10      010100  00101
    0111001010011010001      001101  10101
11100101001101 0001      1110010100110100 01      001101  10101
    0111010111001010100      010001  11110
11101011100101 0100      1110101110010101 00      010001  11110
    0111100100010001101      010101  00111
11110010001000 1101      1111001000100011 01      010101  00111
    0111110010001000110      001110  10111
11111001000100 0110      1111100100010001 10      001110  10111
    1000000000000000000      010010  00000

*/
```

A.2.4 The Binary / 2DLNS Conversion Register

This register reorders the output of the binary / 2DLNS converter.

```
library ieee;
use ieee.std_logic_1164.all;

use work.numeric_bit.all;
```

```
entity conv_out_reg is
  port ( enable : in std_logic;
         ready_low : in std_logic;
         ready_high : in std_logic;
         output_sign_low : in std_logic_vector(1 downto 0);
         output_first_low : in std_logic_vector(11 downto 0);
         output_second_low : in std_logic_vector(9 downto 0);
         output_sign_high : in std_logic_vector(1 downto 0);
         output_first_high : in std_logic_vector(11 downto 0);
         output_second_high : in std_logic_vector(9 downto 0);
         output : out std_logic_vector(47 downto 0));
end entity conv_out_reg;


architecture behavior of conv_out_reg is

  signal stored_value : std_logic_vector(47 downto 0);

begin

  -- extracts the data related to each digit from inputs
  -- and merges them as an stored value
  conv_output : process ( enable ) is
  begin
    stored_value <= (output_sign_high(1) & output_first_high(11 downto
        6)
                  & output_second_high(9 downto 5) & output_sign_high(0)
                  & output_first_high(5 downto 0)
                  & output_second_high(4 downto 0) & output_sign_low(1)
                  & output_first_low(11 downto 6)
                  & output_second_low(9 downto 5) & output_sign_low(0)
                  & output_first_low(5 downto 0)
                  & output_second_low(4 downto 0)) ;
  end process conv_output;

  -- stored value enabled to output based on ready signal
  output <= stored_value when enable = '1' ;

end architecture behavior;
```

A.2.5 The Multiply and Accumulate unit (MAC)

The MAC unit consists of several components. All components are instantiated in a top module which is shown here. The consecutive subsections include the HDL codes of all components.

```
library ieee;
use ieee.std_logic_1164.all;

use work.numeric_bit.all;

entity top_mac is
  port (clk,clr : in std_logic;
        x : in std_logic_vector(47 downto 0);
        y : in std_logic_vector(39 downto 0);
        p : out std_logic_vector(51 downto 0));
end entity top_mac;


architecture rtl of top_mac is

  -- The FIR filter Multiplier unit
  component mul is
    port (clk,clr : in std_logic;
          x : in std_logic_vector(23 downto 0);
          y : in std_logic_vector(19 downto 0);
          sign : out std_logic;
          p : out std_logic_vector(17 downto 0));
  end component mul;

  -- The 18-bit adder_subtractor
  component adder_subtractor_18
    port ( a, b : in std_logic_vector(17 downto 0);
           s : out std_logic_vector(33 downto 0);
           sign1, sign2 : in std_logic );
   end component;

  -- The 34-bit adder_subtractor
  component adder_subtractor_34
    port ( a, b : in std_logic_vector(33 downto 0);
           s : out std_logic_vector(49 downto 0);
           sign1, sign2 : in std_logic );
   end component;

  -- The 50-bit adder_subtractor
  component adder_subtractor_50
    port ( a : in std_logic_vector(49 downto 0);
           b : in std_logic_vector(51 downto 0);
           s : out std_logic_vector(51 downto 0);
           sign1 : in std_logic );
    end component;

  -- The register in feedback loop of accumulator
  component accumulator_regp
```

```
    port ( clk, clr : in std_logic;
           d : in std_logic_vector(51 downto 0);
           q : out std_logic_vector(51 downto 0));
  end component;


  signal sign_low_low : std_logic;
  signal sign_low_high : std_logic;
  signal sign_high_low : std_logic;
  signal sign_high_high : std_logic;
  signal p_low_low : std_logic_vector(17 downto 0);
  signal p_low_high : std_logic_vector(17 downto 0);
  signal p_high_low : std_logic_vector(17 downto 0);
  signal p_high_high : std_logic_vector(17 downto 0);
  signal p_29_low : std_logic_vector(33 downto 0);
  signal p_29_high : std_logic_vector(33 downto 0);
  signal p_37 : std_logic_vector(49 downto 0);
  signal accumulator : std_logic_vector(51 downto 0);
  signal acc : std_logic_vector(51 downto 0);

begin

  -- The component instantiations

   mul_low_low :  mul
    port map ( clk => clk,
               clr => clr,
               x => x(23 downto 0),
               y => y(19 downto 0),
               sign => sign_low_low,
               p => p_low_low);

   mul_low_high :  mul
    port map ( clk => clk,
               clr => clr,
               x => x(23 downto 0),
               y => y(39 downto 20),
               sign => sign_low_high,
               p => p_low_high);

  mul_high_low :  mul
    port map ( clk => clk,
               clr => clr,
               x => x(47 downto 24),
               y => y(19 downto 0),
               sign => sign_high_low,
               p => p_high_low);

  mul_high_high :  mul
```

```
    port map ( clk => clk,
               clr => clr,
               x => x(47 downto 24),
               y => y(39 downto 20),
               sign => sign_high_high,
               p => p_high_high);

  add_sub_18_low : adder_subtractor_18
    port map ( a => p_low_low, b => p_high_low, s => p_29_low, sign1 =>
               sign_low_low, sign2 => sign_high_low);

  add_sub_18_high : adder_subtractor_18
    port map ( a => p_low_high, b => p_high_high, s => p_29_high, sign1
               => sign_low_high, sign2 => sign_high_high);

  add_sub_34 : adder_subtractor_34
    port map ( a => p_29_low, b => p_29_high, s => p_37,
               sign1 => sign_low_low, sign2 => sign_low_high);

  add_sub_50 : adder_subtractor_50
    port map ( a => p_37, b => accumulator, s => acc,
               sign1 => sign_low_low);

  accumulator_reg : accumulator_regp
    port map ( clk => clk, clr => clr, d => acc,
               q => accumulator);


  -- disables output based on clr signal
  with clr select
    p <= acc(51 downto 0) when '0',
         "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ" when
           others;

end architecture rtl;
```

The multiplier is the main module in the top MAC unit:

```
library ieee;
use ieee.std_logic_1164.all;

use work.numeric_bit.all;

entity mul is
  port (clk,clr : in std_logic;
        x : in std_logic_vector(23 downto 0);
        y : in std_logic_vector(19 downto 0);
        sign : out std_logic;
```

```
        p : out std_logic_vector(17 downto 0));
end entity mul;


architecture rtl of mul is

  -- The xor unit
  component x_or
    port ( a, b : in std_logic;
           c : out std_logic );
    end component ;

  -- The first exponent adder
  component adder
    port ( a : in std_logic_vector(5 downto 0);
           b : in std_logic_vector(5 downto 0);
           s : out std_logic_vector(6 downto 0) );
    end component;

  -- The second exponent adder
  component adder_r
    port ( a : in std_logic_vector(4 downto 0);
           b : in std_logic_vector(2 downto 0);
           s : out std_logic_vector(4 downto 0) );
  end component;

  -- The 2DLNS / Binary Converter
  component  convert2dlnstobinary
    port ( signin : in std_logic;
           firstbaseindex : in std_logic_vector(6 downto 0);
           secondbaseindex: in std_logic_vector(4 downto 0);
           binaryout : out std_logic_vector(23 downto 0);
           signout : out std_logic );
  end component;

  -- The 24-bit adder_subtractor
  component adder_subtractor_24
    port ( a, b : in std_logic_vector(23 downto 0);
           s : out std_logic_vector(24 downto 0);
           sign1, sign2 : in std_logic );
    end component;

  -- The 25-bit adder_subtractor
  component adder_subtractor_25
    port ( a, b : in std_logic_vector(24 downto 0);
           s : out std_logic_vector(25 downto 0);
           sign1, sign2 : in std_logic );
   end component;
```

```
  signal p_p1_dig1 : std_logic_vector(12 downto 0);
  signal p_p1_dig2 : std_logic_vector(12 downto 0);
  signal p_p2_dig1 : std_logic_vector(12 downto 0);
  signal p_p2_dig2 : std_logic_vector(12 downto 0);
  signal p_p1_b1 : std_logic_vector(23 downto 0);
  signal p_p1_b2 : std_logic_vector(23 downto 0);
  signal p_p2_b1 : std_logic_vector(23 downto 0);
  signal p_p2_b2 : std_logic_vector(23 downto 0);
  signal p_p1_b : std_logic_vector(24 downto 0);
  signal p_p2_b : std_logic_vector(24 downto 0);
  signal p_b : std_logic_vector(25 downto 0);
  signal s1, s2, s3, s4 : std_logic;
  signal dummy_s1, dummy_s2, dummy_s3, dummy_s4 : std_logic;

begin

  -- Output port connection
  sign <= s1;

  -- The component instantiations

  p_p1_dig1_s :  x_or
    port map ( a => x(23), b => y(19), c => p_p1_dig1(12));

  s1 <= p_p1_dig1(12);


  p_p1_dig2_s :  x_or
    port map ( a => x(11), b => y(9), c => p_p1_dig2(12));

  s2 <= p_p1_dig2(12);


  p_p2_dig1_s :  x_or
    port map ( a => x(11), b => y(19), c => p_p2_dig1(12));

  s3 <= p_p2_dig1(12);


  p_p2_dig2_s :  x_or
    port map ( a => x(23), b => y(9), c => p_p2_dig2(12));

  s4 <= p_p2_dig2(12);


  p_p1_dig1_a :  adder
    port map ( a => x(22 downto 17), b => y(18 downto 13),
               s => p_p1_dig1(11 downto 5));
```

```
p_p1_dig1_b :  adder_r
  port map ( a => x(16 downto 12), b => y(12 downto 10),
             s => p_p1_dig1(4 downto 0));

p_p1_dig2_a :  adder
  port map ( a => x(10 downto 5), b => y(8 downto 3),
             s => p_p1_dig2(11 downto 5));

p_p1_dig2_b :  adder_r
  port map ( a => x(4 downto 0), b => y(2 downto 0),
             s => p_p1_dig2(4 downto 0));

p_p2_dig1_a :  adder
  port map ( a => x(10 downto 5), b => y(18 downto 13),
             s => p_p2_dig1(11 downto 5));

p_p2_dig1_b :  adder_r
  port map ( a => x(4 downto 0), b => y(12 downto 10),
             s => p_p2_dig1(4 downto 0));

p_p2_dig2_a :  adder
  port map ( a => x(22 downto 17), b => y(8 downto 3),
             s => p_p2_dig2(11 downto 5));

p_p2_dig2_b :  adder_r
  port map ( a => x(16 downto 12), b => y(2 downto 0),
             s => p_p2_dig2(4 downto 0));


p_p1_b1_con :  convert2dlnstobinary
  port map ( signin => p_p1_dig1(12),
             firstbaseindex => p_p1_dig1(11 downto 5),
             secondbaseindex => p_p1_dig1(4 downto 0),
             binaryout => p_p1_b1,
             signout => dummy_s1);


p_p1_b2_con :  convert2dlnstobinary
  port map ( signin => p_p1_dig2(12),
             firstbaseindex => p_p1_dig2(11 downto 5),
             secondbaseindex => p_p1_dig2(4 downto 0),
             binaryout => p_p1_b2,
             signout => dummy_s2);


p_p2_b1_con :  convert2dlnstobinary
  port map ( signin => p_p2_dig1(12),
             firstbaseindex => p_p2_dig1(11 downto 5),
```

```
                secondbaseindex => p_p2_dig1(4 downto 0),
                binaryout => p_p2_b1,
                signout => dummy_s3);


  p_p2_b2_con :  convert2dlnstobinary
    port map ( signin => p_p2_dig2(12),
                firstbaseindex => p_p2_dig2(11 downto 5),
                secondbaseindex => p_p2_dig2(4 downto 0),
                binaryout => p_p2_b2,
                signout => dummy_s4);


  p1_bin_add_sub :  adder_subtractor_24
    port map ( a => p_p1_b1, b => p_p1_b2,
                s => p_p1_b, sign1 => s1, sign2 => s2);

  p2_bin_add_sub :  adder_subtractor_24
    port map ( a => p_p2_b1, b => p_p2_b2,
                s => p_p2_b, sign1 => s3, sign2 => s4);


  p_bin_add_sub :  adder_subtractor_25
    port map ( a => p_p1_b, b => p_p2_b, s => p_b, sign1 => s1,
                sign2 => s3);



  -- removes the repeatative bits and disables output based on clr
  -- signal
  with clr select
    p <=  p_b(25 downto 8) when '0',
          "ZZZZZZZZZZZZZZZZZZ"  when others;

end architecture rtl;
```

And it consists of a hierarchy of components as included in the consecutive subsections.

A.2.5.1 The Exclusive-or unit

This code, simply makes an exclusive-or of its inputs.

```
library ieee;
use ieee.std_logic_1164.all;

entity x_or is
```

```
  port ( a,b :   in std_logic;
         c :     out std_logic );
end entity x_or;


architecture behavioral of x_or is
begin
      c <= a xor b;

end architecture behavioral;
```

A.2.5.2 The First Exponent Adders

This code shows a two's-complement bit by bit adder. The multiplier unit makes use of this adder to add the first base exponents.

```
library ieee;
use ieee.std_logic_1164.all;

entity adder is
  port ( a : in std_logic_vector(5 downto 0);
         b : in std_logic_vector(5 downto 0);
         s : out std_logic_vector(6 downto 0) );
end entity adder;


architecture behavioral of adder is

begin

  behavior : process (a, b) is

    variable carry_in : std_logic;
    variable carry_out : std_logic;
    variable ext_a : std_logic_vector(5 downto 0);

  begin
    -- carry in to the first bit
    carry_out := '0';

    -- operand a is sign extended
    ext_a(5 downto 0) := a(5 downto 0);

    -- computes sum and carry for all bits
    -- carry out of each order is carry in for the next one
```

```
    for index in 0 to b'left loop
      carry_in := carry_out;
      s(index) <= ext_a(index) xor b(index) xor carry_in ;
      carry_out := (ext_a(index) and b(index))
                   or (carry_in and (ext_a(index) xor b(index)));
    end loop;

    -- one extra bit is considered for sum
    s(ext_a'left + 1) <= ext_a(ext_a'left) xor b(ext_a'left) xor
                         carry_out ;
  end process behavior;

end architecture behavioral;
```

A.2.5.3 The Second Exponent Adders

Since the size of operands of this adder are different, the smaller operand is sign extended before addition. In our design, the most negative second base exponent is used to represent zero. Therefore, if either multiplicand or multiplier are zero, the product will be set to zero.

```
library ieee;
use ieee.std_logic_1164.all;

entity adder_r is
  port ( a : in std_logic_vector(4 downto 0);
         b : in std_logic_vector(2 downto 0);
         s : out std_logic_vector(4 downto 0) );
end entity adder_r;


architecture behavioral of adder_r is

begin

  behavior : process (a, b) is

    variable carry_in : std_logic;
    variable carry_out : std_logic;
    variable ext_b : std_logic_vector(4 downto 0);

  begin

  -- checks if either of inputs represents zero,
  -- the product should also be set to zero
```

```
  if a = "10000" or b = "100" then s <= "10000" ;
  else
    -- carry in to the first bit
    carry_out := '0';

    -- operand b is sign extended
    ext_b(4 downto 0) := b(2) & b(2) & b(2 downto 0);

    -- computes sum and carry for all bits
    -- carry out of each order is carry in for the next one
    for index in 0 to a'left loop
      carry_in := carry_out;
      s(index) <= a(index) xor ext_b(index) xor carry_in ;
      carry_out := (a(index) and ext_b(index))
                   or (carry_in and (a(index) xor ext_b(index)));
    end loop;

  end if;

  end process behavior;

end architecture behavioral;
```

A.2.5.4 The 2DLNS / Binary Converter

The modified version of the Verilog code in [29], is used for the 2DLNS / binary conversion. This HDL code has also been written fully parametrized and the parameters should be set when the shell script which generates Verilog module is run. The definition of these parameters were also included in [29]. Before running this script, the optimal base has been computed and stored in an ASCII file, 13-65536-nzsmn2-00. The 2DLNS / binary converter module has been generated by setting the parameters as:

makeconvert2dlnstobinary.sh 13-65536-nzsmn2-00 16 7 5 4 -nz -ns > bconverter.v

The name of generated module is **convert2dlnstobinary** which is instantiated, as an component, in the multiplier module. This Verilog file is shown here.

```
`timescale 1ns/10ps
```

```
module convert2dlnstobinary(
signin,
firstbaseindex,
secondbaseindex,
binaryout,
signout
);
// Default parameters
parameter firstbasebits = 7;
parameter secondbasebits = 5;
parameter outputbits = 16;
parameter extrabits = 8;
parameter memfirstbasebits = 6;
parameter subbits = 8;
parameter twobitmode = 0;
parameter nosignmode = 0;
// Define ports
input [ twobitmode : 0 ] signin;
input [ firstbasebits - 1 : 0 ] firstbaseindex;
input [ secondbasebits - 1 : 0 ] secondbaseindex;
output [ nosignmode + outputbits + extrabits - 1 : 0 ] binaryout;
reg [ nosignmode + outputbits + extrabits - 1 : 0 ] binaryout;
output signout;
reg signout;
reg [ subbits - 1 : 0 ] arg1, arg2, sum;
reg [ outputbits + extrabits - 1 : 0 ] absolute;
wire [ outputbits + extrabits - 1 : 0 ] mantissa;
wire [ memfirstbasebits - 1 : 0 ] shift;
lut2_3481113873_1216_noclk
lut
(
mantissa,
shift,
secondbaseindex
);
// Perform shift
always @( mantissa or shift or firstbaseindex )
begin
// Extend signs on each argument
if( subbits > memfirstbasebits )

begin
arg1 = { { ( subbits - memfirstbasebits ) { shift[ memfirstbasebits
 - 1 ] } }, shift[ memfirstbasebits - 1 : 0 ] };
end
else
begin
arg1 = shift;
```

```
end
if( subbits > firstbasebits )
begin
arg2 = { { ( subbits - firstbasebits ) { firstbaseindex[ firstbasebits
 - 1 ] } }, firstbaseindex[ firstbasebits - 1 : 0 ] };
end
else
begin
arg2 = firstbaseindex;
end
sum = arg1 - arg2;
absolute = mantissa >> sum;
if( twobitmode )
begin
if( signin == 0 )
begin
absolute = 0;
end
end
if( nosignmode )
begin
if( signin[ twobitmode ] )
begin
binaryout <= - absolute;
end
else
begin
binaryout <= absolute;
end

end
else
begin
binaryout <= absolute;
signout <= signin[ twobitmode ];
end
// $display("@@ %b %b %b %b %b %b %b",mantissa,shift,arg1,firstbaseindex
    ,
// arg2,sum,binaryout);
end
endmodule
module lut2_3481113873_1216_noclk(
        o1,
        o2,
        id
);

// Default parameters
parameter o1bits = 24;
```

```
parameter o2bits = 6;
parameter idbits = 5;
parameter rlsize = 5'b00000;
parameter rhsize = 5'b11111;

// Define ports

// direct data access line
input [ idbits - 1 : 0 ] id;

// datas to output
output [ o1bits - 1 : 0 ] o1;
output [ o2bits - 1 : 0 ] o2;
reg [ o1bits - 1 : 0 ] o1;
reg [ o2bits - 1 : 0 ] o2;

// reg [ idbits - 1 : 0 ] romAddr[ rlsize : rhsize ];
reg [ o1bits - 1 : 0 ] romOut1[ rlsize : rhsize ];
reg [ o2bits - 1 : 0 ] romOut2[ rlsize : rhsize ];

`ifdef DC
`else

initial
begin

// include data information here

romOut1[5'b00000]=24'b100000000000000000000000;
romOut2[5'b00000]=6'b001111;
romOut1[5'b00001]=24'b101111001011001101011100;
romOut2[5'b00001]=6'b010000;
romOut1[5'b00010]=24'b100010110001011111101101;
romOut2[5'b00010]=6'b010000;
romOut1[5'b00011]=24'b110011010000111000001011;
romOut2[5'b00011]=6'b010001;
romOut1[5'b00100]=24'b100101110010010111111011;
romOut2[5'b00100]=6'b010001;
romOut1[5'b00101]=24'b110111101101001110010101;
romOut2[5'b00101]=6'b010010;
romOut1[5'b00110]=24'b101001000011111101111111;
romOut2[5'b00110]=6'b010010;
romOut1[5'b00111]=24'b111100100010001101101010;
romOut2[5'b00111]=6'b010011;
romOut1[5'b01000]=24'b101100100111101110101000;
romOut2[5'b01000]=6'b010011;
romOut1[5'b01001]=24'b100000111000111111011100;
romOut2[5'b01001]=6'b010011;
romOut1[5'b01010]=24'b110000011111001110100101;
```

```
romOut2[5'b01010]=6'b010100;
romOut1[5'b01011]=24'b100011101111011011010000;
romOut2[5'b01011]=6'b010100;
romOut1[5'b01100]=24'b110100101100001011010110;
romOut2[5'b01100]=6'b010101;
romOut1[5'b01101]=24'b100110110101101010111111;
romOut2[5'b01101]=6'b010101;
romOut1[5'b01110]=24'b111001010000011011111010;
romOut2[5'b01110]=6'b010110;
romOut1[5'b01111]=24'b101010001101000110010110;
romOut2[5'b01111]=6'b010110;
romOut1[5'b10000]=24'b000000000000000000000000;
romOut2[5'b10000]=6'b001111;
romOut1[5'b10001]=24'b110000100001101000001010;
romOut2[5'b10001]=6'b001001;
romOut1[5'b10010]=24'b100011110001001100011101;
romOut2[5'b10010]=6'b001001;
romOut1[5'b10011]=24'b110100101110110010001111;
romOut2[5'b10011]=6'b001010;
romOut1[5'b10100]=24'b100110110111100110000000;
romOut2[5'b10100]=6'b001010;
romOut1[5'b10101]=24'b111001010011010001010000;
romOut2[5'b10101]=6'b001011;
romOut1[5'b10110]=24'b101010001111001100000001;
romOut2[5'b10110]=6'b001011;
romOut1[5'b10111]=24'b111110010001000110100111;
romOut2[5'b10111]=6'b001100;
romOut1[5'b11000]=24'b101101111001011101111000;
romOut2[5'b11000]=6'b001100;
romOut1[5'b11001]=24'b100001110101001111011101;
romOut2[5'b11001]=6'b001100;
romOut1[5'b11010]=24'b110001111000000011001101;
romOut2[5'b11010]=6'b001101;
romOut1[5'b11011]=24'b100100110000111001011101;
romOut2[5'b11011]=6'b001101;
romOut1[5'b11100]=24'b110110001100101100101001;
romOut2[5'b11100]=6'b001110;
romOut1[5'b11101]=24'b100111111100110100010110;
romOut2[5'b11101]=6'b001110;
romOut1[5'b11110]=24'b111010111001010100100101;
romOut2[5'b11110]=6'b001111;
romOut1[5'b11111]=24'b101011011010011010010101;
romOut2[5'b11111]=6'b001111;

end
`endif

always @( id )
begin
```

```
`ifdef DC

romOut1[5'b00000]=24'b100000000000000000000000;
romOut2[5'b00000]=6'b001111;
romOut1[5'b00001]=24'b101111001011001101011100;
romOut2[5'b00001]=6'b010000;
romOut1[5'b00010]=24'b100010110001011111101101;
romOut2[5'b00010]=6'b010000;
romOut1[5'b00011]=24'b110011010000111000001011;
romOut2[5'b00011]=6'b010001;
romOut1[5'b00100]=24'b100101110010010111111011;
romOut2[5'b00100]=6'b010001;
romOut1[5'b00101]=24'b110111101101001110010101;
romOut2[5'b00101]=6'b010010;
romOut1[5'b00110]=24'b101001000011111101111111;
romOut2[5'b00110]=6'b010010;
romOut1[5'b00111]=24'b111100100010001101101010;
romOut2[5'b00111]=6'b010011;
romOut1[5'b01000]=24'b101100100111101110101000;
romOut2[5'b01000]=6'b010011;
romOut1[5'b01001]=24'b100000111000111111011100;
romOut2[5'b01001]=6'b010011;
romOut1[5'b01010]=24'b110000011111001110100101;
romOut2[5'b01010]=6'b010100;
romOut1[5'b01011]=24'b100011101111011011010000;
romOut2[5'b01011]=6'b010100;
romOut1[5'b01100]=24'b110100101100001011010110;
romOut2[5'b01100]=6'b010101;
romOut1[5'b01101]=24'b100110110101101010111111;
romOut2[5'b01101]=6'b010101;
romOut1[5'b01110]=24'b111001010000011011111010;
romOut2[5'b01110]=6'b010110;
romOut1[5'b01111]=24'b101010001101000110010110;
romOut2[5'b01111]=6'b010110;
romOut1[5'b10000]=24'b000000000000000000000000;
romOut2[5'b10000]=6'b001111;
romOut1[5'b10001]=24'b110000100001101000001010;
romOut2[5'b10001]=6'b001001;
romOut1[5'b10010]=24'b100011110001001100011101;
romOut2[5'b10010]=6'b001001;
romOut1[5'b10011]=24'b110100101110110010001111;
romOut2[5'b10011]=6'b001010;
romOut1[5'b10100]=24'b100110110111100110000000;
romOut2[5'b10100]=6'b001010;
romOut1[5'b10101]=24'b111001010011010001010000;
romOut2[5'b10101]=6'b001011;
romOut1[5'b10110]=24'b101010001111001100000001;
romOut2[5'b10110]=6'b001011;
```

```
romOut1[5'b10111]=24'b111110010001000110100111;
romOut2[5'b10111]=6'b001100;
romOut1[5'b11000]=24'b101101111001011101111000;
romOut2[5'b11000]=6'b001100;
romOut1[5'b11001]=24'b100001110101001111011101;
romOut2[5'b11001]=6'b001100;
romOut1[5'b11010]=24'b110001111000000011001101;
romOut2[5'b11010]=6'b001101;
romOut1[5'b11011]=24'b100100110000111001011101;
romOut2[5'b11011]=6'b001101;
romOut1[5'b11100]=24'b110110001100101100101001;
romOut2[5'b11100]=6'b001110;
romOut1[5'b11101]=24'b100111111100110100010110;
romOut2[5'b11101]=6'b001110;
romOut1[5'b11110]=24'b111010111001010100100101;
romOut2[5'b11110]=6'b001111;
romOut1[5'b11111]=24'b101011011010011010010101;
romOut2[5'b11111]=6'b001111;

`endif

        o1 = romOut1[ id ];
        o2 = romOut2[ id ];

end

endmodule

// LUT contents ( address, output(s) ... )

/*

00000   100000000000000000000000        001111
00001   101111001011001101011100        010000
00010   100010110001011111101101        010000
00011   110011010000111000001011        010001
00100   100101110010010111111011        010001
00101   110111101101001110010101        010010
00110   101001000011111101111111        010010
00111   111100100010001101101010        010011
01000   101100100111101110101000        010011
01001   100000111000111111011100        010011
01010   110000011111001110100101        010100
01011   100011101111011011010000        010100
01100   110100101100001011010110        010101
01101   100110110101101010111111        010101
01110   111001010000011011111010        010110
01111   101010001101000110010110        010110
10000   000000000000000000000000        001111
```

```
10001    110000100001101000001010          001001
10010    100011110001001100011101          001001
10011    110100101110110010001111          001010
10100    100110110111100110000000          001010
10101    111001010011010001010000          001011
10110    101010001111001100000001          001011
10111    111110010001000110100111          001100
11000    101101111001011101111000          001100
11001    100001110101001111011101          001100
11010    110001111000000011001101          001101
11011    100100110000111001011101          001101
11100    110110001100101100101001          001110
11101    100111111100110100010110          001110
11110    111010111001010100100101          001111
11111    101011011010011010010101          001111

*/
```

A.2.5.5 24-bit Adder / Subtractor

In this module, the type of operands are converted to **unsigned**. Therefore, addition or subtraction of unsigned values, are performed using the functions which have been defined in the **numeric_bit** package.

```
library ieee;
--use ieee.std_logic_signed.all;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtractor_24 is
  port ( a, b : in std_logic_vector(23 downto 0);
         s : out std_logic_vector(24 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtractor_24;


architecture behavioral of adder_subtractor_24 is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(24 downto 0);
    variable ext_b : unsigned(24 downto 0);
```

```
    variable result : unsigned(24 downto 0);


  begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are unsigned extended by one zero bit
    ext_a := unsigned(to_bitvector("0" & a(23 downto 0)));
    ext_b := unsigned(to_bitvector("0" & b(23 downto 0)));

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.5.6 25-bit Adder / Subtractor

The only difference with the previous module is the size of operands and result.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtractor_25 is
  port ( a, b : in std_logic_vector(24 downto 0);
         s : out std_logic_vector(25 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtractor_25;


architecture behavioral of adder_subtractor_25 is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
```

```
    variable ext_a : unsigned(25 downto 0);
    variable ext_b : unsigned(25 downto 0);
    variable result : unsigned(25 downto 0);

 begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are signed extended by one bit
    ext_a := unsigned(to_bitvector(a(24) & a(24 downto 0)));
    ext_b := unsigned(to_bitvector(b(24) & b(24 downto 0)));

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

The consecutive subsections include the HDL codes of all other components of MAC unit.

A.2.5.7 18-bit Adder / Subtractor

There are two 18-bit adder / subtracter in the MAC design. They add the output of multiplier channels. This module also shifts the output of higher order channel by 16 bits.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtractor_18 is
  port ( a, b : in std_logic_vector(17 downto 0);
         s : out std_logic_vector(33 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtractor_18;
```

```
architecture behavioral of adder_subtractor_18 is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(33 downto 0);
    variable ext_b : unsigned(33 downto 0);
    variable result : unsigned(33 downto 0);

  begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are signed extended by one bit
   ext_a := unsigned(to_bitvector(a(17) & a(17) & a(17) & a(17)& a(17)
       & a(17) & a(17) & a(17) & a(17) & a(17) & a(17) & a(17)& a(17)
       & a(17) & a(17) & a(17) & a(17 downto 0)));
   ext_b := unsigned(to_bitvector(b(17 downto 0) & "0000000000000000"));


    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.5.8 34-bit Adder / Subtractor

This adder operates on the output of two 18-bit adders. Again, it shifts the output of higher order partial product by 16 bits.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;
```

```
entity adder_subtractor_34 is
  port ( a, b : in std_logic_vector(33 downto 0);
         s : out std_logic_vector(49 downto 0);
         sign1, sign2 : in std_logic );
end entity adder_subtractor_34;


architecture behavioral of adder_subtractor_34 is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(49 downto 0);
    variable ext_b : unsigned(49 downto 0);
    variable result : unsigned(49 downto 0);

  begin

    -- the sign is xor of input signs
    sign := sign1 xor sign2;

    -- the operands are signed extended and shifted
    ext_a := unsigned(to_bitvector(a(33) & a(33) & a(33) & a(33) & a(33)
        & a(33) & a(33) & a(33) & a(33) & a(33) & a(33) & a(33)
        & a(33) & a(33) & a(33) & a(33 downto 0)));
    ext_b := unsigned(to_bitvector(b(33 downto 0) & "0000000000000000"))
       ;

    -- addition or subtraction is done based on the computed sign
    if sign = '1' then
       result := ext_a - ext_b ;
    else
       result := ext_a + ext_b ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;

end architecture behavioral;
```

A.2.5.9 50-bit Adder / Subtractor

This module accumulates the final product of MAC unit in a filtering iteration.

```
library ieee;
use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity adder_subtractor_50 is
  port ( a : in std_logic_vector(49 downto 0);
         b : in std_logic_vector(51 downto 0);
         s : out std_logic_vector(51 downto 0);
         sign1 : in std_logic );
end entity adder_subtractor_50;


architecture behavioral of adder_subtractor_50 is
begin

  behavior : process (a, b) is

    variable sign : std_logic;
    variable ext_a : unsigned(51 downto 0);
    variable ext_b : unsigned(51 downto 0);
    variable result : unsigned(51 downto 0);

  begin

    -- the sign is determined by input sign
    sign := sign1;

    -- this operand is signed extended by two bits
    ext_a := unsigned(to_bitvector(a(49) & a(49) & a(49 downto 0)));

    -- this operand does not need to be extended
    ext_b := unsigned(to_bitvector(b(51 downto 0)));

    -- addition or subtraction is done based on the sign
    if sign = '1' then
       result := ext_b - ext_a ;
    else
       result := ext_b + ext_a ;
    end if;

  s <= to_x01(bit_vector(result)) ;

  end process behavior;
```

```
end architecture behavioral;
```

A.2.5.10 Accumulator Register

This register should be cleared whenever a MAC operation commences.

```
library ieee;
use ieee.std_logic_1164.all;

entity accumulator_regp is
  port ( clk : in std_logic;
         clr : in std_logic;
         d : in std_logic_vector(51 downto 0);
         q : out std_logic_vector(51 downto 0));
end entity accumulator_regp;

architecture behavioral of accumulator_regp is
begin

  -- when register is cleared the output is zero
  behavior : process (clk) is

  begin
    if rising_edge(clk) then
      if To_X01(clr) = '1' then
        q <= (others => '0') ;
      else
        q <= d ;
      end if;
    end if;
  end process behavior;

end architecture behavioral;
```

A.2.6 The Controller

The VHDL code for this unit, realizes a complicated state machine. This file is well documented to be self explanatory.

```
library ieee;
```

```
use ieee.std_logic_1164.all,
    ieee.std_logic_unsigned.all;

use work.numeric_bit.all;

entity controller is
  port ( clk : in std_logic;
         reset : in std_logic;
         ir_mem_enable : out std_logic;
         mem_write_en : out std_logic;
         mem_enable : out std_logic;
         top_mac_clr : out std_logic;
         btc_reset_low : out std_logic;
         btc_ready_low : in std_logic;
         btc_activate_low : out  std_logic;
         btc_reset_high : out std_logic;
         btc_ready_high : in std_logic;
         btc_activate_high : out  std_logic;
         in_reg_enable : out std_logic;
         in_reg_out_en : out std_logic;
         out_reg_enable : out std_logic;
         ctrl_mem_a : out std_logic_vector(9 downto 0);
         ctrl_ir_mem_a : out std_logic_vector(9 downto 0));
end entity controller;


architecture behavior of controller is

    type state_type is ( s0,S1,S2,S3,S4,S4_a,S5,S6,S7 );
    signal state : state_type;

begin

  sequencer : process(clk, reset ) is

    variable filter_tap : unsigned(6 downto 0);
    variable filter_order : unsigned(6 downto 0);
    variable first_half : bit;
    variable coef_address : unsigned(9 downto 0);
    variable coef_end_address : unsigned(9 downto 0);
    variable top_coef_address : unsigned(6 downto 0);
    variable filt_data_address : std_logic_vector(9 downto 0);
    variable filt_coef_address : std_logic_vector(9 downto 0);
    variable filt_data_start : std_logic_vector(9 downto 0);
    variable filt_data_end : std_logic_vector(9 downto 0);
    variable current_data_address : std_logic_vector(9 downto 0);


    -- sets addresses for data and coefficient memories
```

```
procedure do_EX_settings is
begin
  filter_order := B"1001010";

  -- writes addresses into variables
  current_data_address := B"0001001001" ;
  filt_coef_address := B"0000000000" ;
  filt_data_start := B"0000000000" ;
  filt_data_end := B"1111111111" ;
end procedure do_EX_settings;

-- reads external data into input register
procedure do_EX_input_1 is
begin
  -- input register is enabled to receive data
  in_reg_enable <= '1' ;
end procedure do_EX_input_1;

-- resets control signal
procedure do_EX_input_2 is
begin
  in_reg_enable <= '0' ;
end procedure do_EX_input_2;

-- performs binary / 2DLNS conversion
procedure do_EX_btconvert_1 is
begin
  -- reads binary data from register
  in_reg_out_en <= '1' ;

  -- activates converters
  btc_reset_low <= '0' ;
  btc_activate_low <= '1' ;
  btc_reset_high <= '0' ;
  btc_activate_high <= '1' ;

  current_data_address := current_data_address + 1 ;
  ctrl_mem_a <= current_data_address ;

end procedure do_EX_btconvert_1;

-- prepares destination memory
procedure do_EX_btconvert_2 is
begin
  in_reg_out_en <= '0' ;
  btc_activate_low <= '0';
  btc_activate_high <= '0';
  mem_write_en <= '1' ;
  mem_enable <= '1' ;
```

```
end procedure do_EX_btconvert_2;

-- disables memory and resets control signals
procedure do_EX_btconvert_3 is
begin
  mem_write_en <= '0';
  mem_enable <= '0';
  ctrl_mem_a <= "ZZZZZZZZZZ";
  btc_reset_low <= '1';
  btc_reset_high <= '1';
end procedure do_EX_btconvert_3;

-- specifies addresses for filtering
procedure do_EX_filt_start is
begin
  -- writes addresses into variables
  filt_data_address := current_data_address;

  -- enables both memories
  ir_mem_enable <= '1';
  mem_enable <= '1';

  -- determines the coefficients' address range
  coef_address := unsigned(to_bitvector(filt_coef_address));
  top_coef_address := filter_order srl 1;
  coef_end_address := coef_address + top_coef_address;

  filter_tap := B"0000000";
  first_half := '1';

  -- addresses memories
  ctrl_mem_a <= filt_data_address;
  ctrl_ir_mem_a <= to_x01(bit_vector(coef_address));
end procedure do_EX_filt_start;

-- performs MAC operations
procedure do_EX_filt_mac is
begin
  -- one MAC operation is performed
  top_mac_clr <= '0';

  filter_tap := filter_tap + 1;

  -- specifies the next coefficient address
    if coef_address < coef_end_address then
      if first_half = '1' then
        coef_address := coef_address + 1;
      else
        coef_address := coef_address - 1;
```

```
        end if;
      else
        coef_address := coef_address - 1;
        first_half := '0';
      end if;

    -- determines the next data address
    filt_data_address := filt_data_address - 1 ;
    if filt_data_address = filt_data_start - 1 then
       filt_data_address := filt_data_end;
    end if;

    -- addresses memories
    ctrl_mem_a <= filt_data_address;
    ctrl_ir_mem_a <= to_x01(bit_vector(coef_address));
  end procedure do_EX_filt_mac;

  -- performs the last iteration of MAC operation
  procedure do_EX_filt_last is
  begin
    -- The output register is written
    out_reg_enable <= '1' ;

    -- memories are disabled
    ir_mem_enable <= '0' ;
    mem_enable <= '0' ;
    ctrl_mem_a <= "ZZZZZZZZZZ" ;
    ctrl_ir_mem_a <= "ZZZZZZZZZZ" ;
  end procedure do_EX_filt_last;

  -- all control signals are reset to their defaults
  procedure do_EX_filt_out is
  begin
    out_reg_enable <= '0' ;
    top_mac_clr <= '1' ;
  end procedure do_EX_filt_out;


begin -- sequencer
  if reset = '1' then

  -- initialize all control signals
  ir_mem_enable <= '0' ;
  mem_write_en <= '0' ;
  mem_enable <= '0' ;
  ctrl_mem_a <= "ZZZZZZZZZZ" ;
  top_mac_clr <= '1';
  btc_reset_low <= '1';
  btc_activate_low <= '0';
```

```
btc_reset_high <= '1';
btc_activate_high <= '0';
in_reg_enable <= '0';
in_reg_out_en <= '0';
out_reg_enable <= '0';
ctrl_ir_mem_a <= "ZZZZZZZZZZ" ;
state <= s0 ;

elsif rising_edge(clk) then

  -- control loop
  case state is


      when s0       => do_EX_settings;
                       state <= s1 ;

      when s1       => do_EX_input_1;
                       state <= s2 ;

      when s2       => do_EX_input_2;
                       state <= s3 ;

      when s3       => do_EX_btconvert_1;
                       state <= s4 ;

      when s4       => do_EX_btconvert_2;
                       state <= s4_a;

      when s4_a     => -- remains here while conversion is in process
                       if (btc_ready_low and btc_ready_high) = '1'
                          then
                          do_EX_btconvert_3;
                          state <= s5;
                       else
                          state <= s4_a;
                       end if;

      when s5       => do_EX_filt_start;
                       state <= s6 ;

      when s6       => do_EX_filt_mac;
                          -- MAC operation continues for all
                             coefficients
                       if filter_tap < filter_order then
                          state <= s6 ;
                       else
                          do_EX_filt_last;
                          state <= s7 ;
```

```
                        end if;

          when s7       => do_Ex_filt_out;
                           state <= s1 ;

          when others   => null;
                           state <= s1 ;
      end case;

    end if;

  end process sequencer;

end architecture behavior;
```

A.3 The Filter Test

This section contains the VHDL files of **Filter** test bench module and its components.

A.3.1 The Filter Test Bench

This code is the top module of test bench model. The test bench includes instances of the **Filter**, the coefficients and data memories, the clock generator, and the input data reader.

```
library ieee;
use ieee.std_logic_1164.all,
    ieee.std_logic_textio.all;

library work;
use work.numeric_bit.all;

use std.textio.all;

entity filter_test is

end entity filter_test;


architecture bench of filter_test is

  -- The Input Data Reader
```

```
component input_gen is
  port ( clk : in std_logic;
         data_in : out std_logic_vector(31 downto 0);
         out_en : in std_logic );
end component input_gen;

-- The Clock Generator
component clock_gen is
  generic ( Tclk : delay_length := 20 ns );
  port ( clk : out std_logic;
         reset : out std_logic );
end component clock_gen;

-- The Instruction Memory
component memory is
  generic ( mem_size : positive := 1024;
            Tac_first : delay_length := 70 ns;
            Tpd_clk_out : delay_length := 2 ns );
  port ( clk : in std_logic;
           a : in std_logic_vector(9 downto 0);
           d : out std_logic_vector(39 downto 0);
           ir_mem_enable : in std_logic );
end component memory;

-- The Data Memory
component data_memory is
    generic ( data_memory_size : positive := 1024);
    port ( clk : in std_logic;
         mem_a : in std_logic_vector(9 downto 0);
         mem_d_in : in std_logic_vector(47 downto 0);
         mem_d_out : out std_logic_vector(47 downto 0);
         mem_write_en : in std_logic;
         mem_enable : in std_logic );
end component data_memory;

-- The Filter
component filter is
  port ( clk : in std_logic;
         reset : in std_logic;
         input_data : in std_logic_vector(31 downto 0);
         output_data : out std_logic_vector(51 downto 0);
         a : out std_logic_vector(9 downto 0);
         d : in std_logic_vector(39 downto 0);
         ir_mem_enable : out std_logic;
         mem_a : out std_logic_vector(9 downto 0);
         mem_d_out : in std_logic_vector(47 downto 0);
         mem_d_in : out std_logic_vector(47 downto 0);
         mem_write_en : out std_logic;
         output_enable : out std_logic;
```

```
        mem_enable : out std_logic );
  end component filter;

  signal clk : std_logic;
  signal reset : std_logic;
  signal a : std_logic_vector(9 downto 0);
  signal d : std_logic_vector(39 downto 0);
  signal ir_mem_enable: std_logic;
  signal mem_a : std_logic_vector(9 downto 0);
  signal mem_d_in : std_logic_vector(47 downto 0);
  signal mem_d_out : std_logic_vector(47 downto 0);
  signal mem_enable, mem_write_en : std_logic;
  signal data_in : std_logic_vector(31 downto 0);
  signal data_out : std_logic_vector(51 downto 0);
  signal out_en : std_logic;
  signal data_in_en: std_logic := '0';

begin

  -- The component instantiations

  input : component input_gen
     port map (  clk => clk, data_in => data_in, out_en => data_in_en );

  cg : component clock_gen
     port map ( clk => clk, reset => reset );

  mem : component memory
    port map ( clk => clk,
               a => a, d => d,
               ir_mem_enable => ir_mem_enable );

  data_mem : component data_memory
    port map ( clk => clk,
               mem_a => mem_a,
               mem_d_in => mem_d_in,
               mem_d_out => mem_d_out,
               mem_write_en => mem_write_en,
               mem_enable => mem_enable );

  proc : component filter
    port map ( clk => clk,
               reset => reset,
               input_data => data_in,
               output_data => data_out,
               a => a,
               d => d,
               ir_mem_enable => ir_mem_enable,
               mem_a => mem_a,
```

```
                mem_d_in => mem_d_in,
                mem_d_out => mem_d_out,
                mem_write_en => mem_write_en,
                output_enable => out_en,
                mem_enable => mem_enable );

  -- this process writes the output into a file
  write_output: process( clk)

    file output_file : text open write_mode is "filter_output";
    variable line_out : line;

  begin
   if rising_edge(clk) then
     -- writes the output when output is enabled
     if out_en = '1' then
        write(line_out, data_out);
        writeline(output_file, line_out);
        data_in_en <= '1';
      else
        data_in_en <= '0';
      end if;

    end if;

  end process write_output;

end architecture bench;
```

A.3.2 The Test Bench Clock Generator

The VHDL code for this module, generates clock and reset signals for the **Filter**.

```
library ieee;
use ieee.std_logic_1164.all;

entity clock_gen is
  generic ( Tclk : delay_length );
  port ( clk : out std_logic;
         reset : out std_logic );
end entity clock_gen;


architecture behavior of clock_gen is
begin
```

```
  -- sets reset port of the Filter
  reset_driver :
    reset <= '1', '0' after 3.5 * Tclk;

  -- generates the clock signal
  clock_driver : process is
  begin
    clk <= '0';
    wait for Tclk;
    loop
      clk <= '1', '0' after Tclk / 2;
      wait for Tclk;
    end loop;
  end process clock_driver;

end architecture behavior;
```

A.3.3 The Test Bench Coeffi cient Memory

This file shows the memory which is preloaded with the coefficients of the filter.

```
library ieee;
library work;

use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity memory is
  generic ( mem_size : positive := 1024;
            Tac_first : delay_length;
            Tpd_clk_out : delay_length);
  port ( clk : in std_logic;
         a : in std_logic_vector(9 downto 0);
         d : out std_logic_vector(39 downto 0);
         ir_mem_enable : in std_logic);
end entity memory;


architecture preloaded of memory is
begin

  mem_behavior : process is

    constant high_address : natural := mem_size - 1;
```

```
type memory_array is array ( natural range <> ) of
                              unsigned(39 downto 0);
variable mem : memory_array(0 to high_address)
  := ( others => X"0000000000" );

variable byte_address, word_address : natural;
variable write_access : boolean;


procedure load is

  constant program : memory_array
    := ( B"0000000100000000010001100100110110101010",
         B"0000000100000000010011100111001110011000",
         B"0000000100000000010011100001101110110001",
         B"0000000100000000010001100000001110100000",
         B"0000000100000000010001100001010110111010",
         B"0000000100000000010011100100010110101110",
         B"0000000100000000010001100100011110101110",
         B"0000000100000000010011101110110111011010",
         B"0000000100000000010011100010110110100001",
         B"0000000100000000010000000011000000001100",
         B"0000000100000000010011101001001110100000",
         B"0000000100000000010011101010110110110101",
         B"0000000100000000010011010111010110111111",
         B"0000000100000000010001100010001111000000",
         B"0000000100000000010011101001111111000110",
         B"0000000100000000010011101000110110110101",
         B"0000000100000000010011100110000111011011",
         B"0000000100000000010011100100010111001011",
         B"0000000100000000010011101100011111011011",
         B"0000000100000000010001101011111111000101",
         B"0000000100000000010011101000000111010010",
         B"0000000100000000010011101001110111000101",
         B"0000000100000000010011100100101110110111",
         B"0000000100000000010000000011000000001100",
         B"0000000100000000010011100011110110110110",
         B"0000000100000000010011100101101111010000",
         B"0000000100000000010011101011101111011010",
         B"0000000100000000010001101001010111100011",
         B"0000000100000000010011101110110111101011",
         B"0000000100000000010001101001111111010000",
         B"0000000100000000010011101111011111011101",
         B"0000000100000000010011100111011111010111",
         B"0000000100000000010011110000110111110011",
         B"0000000100000000010011101111110111011101",
         B"0000000100000000010011110010111111011101",
         B"0000000100000000010011110100101111100110",
```

```
                B"00000001000000000100011001111001110110000",
                B"00000001000000000100011100011001111110010"
                        );
    begin
      mem(program'range) := program;
    end load;


  begin
    load;
    -- initialize outputs
    d <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

    -- process memory cycles
    loop
      -- wait for a command, valid on leading edge of clk
      wait on clk until rising_edge(clk);
      -- decode address and perform command if selected
      word_address := to_integer(unsigned(to_bitvector(a)));
      if word_address <= high_address then
        if(ir_mem_enable = '1') then
            d <= To_X01( bit_vector(mem(word_address)) );
        end if;
      end if;
    end loop;
  end process mem_behavior;

end architecture preloaded;
```

A.3.4 The Test Bench Data Memory

The VHDL code for the data memory includes the input data samples. Since the order of the filter is 74, the last 75 locations of memory before the incoming input data, is filled with 0. Therefore, it is guaranteed that for the first 75 iteration of MAC operations, no overflow occurs, although they are not valid values and should be ignored.

```
library ieee;
library work;

use ieee.std_logic_1164.all,
    work.numeric_bit.all;

entity data_memory is
```

```
    generic ( data_memory_size : positive := 1024);
    port ( clk : in std_logic;
         mem_a : in std_logic_vector(9 downto 0);
         mem_d_in : in std_logic_vector(47 downto 0);
         mem_d_out : out std_logic_vector(47 downto 0);
         mem_write_en : in std_logic;
         mem_enable : in std_logic);
end entity data_memory;


architecture behavior of data_memory is
begin

  data_memory_behavior : process is

    constant high_address : natural := data_memory_size - 1;

    type data_memory_array is array ( natural range <>) of
                                      std_logic_vector(47 downto 0);

    variable row_address : natural;
    variable write_access : boolean;
    variable data_memory : data_memory_array(0 to high_address)
      := ( others => X"000000000000" );

    -- The filterbank input data  are loaded into memory
    procedure load is

      constant data : data_memory_array
        := ( B"000000010000000000010000000000010000000000010000", -- 0
             B"000000010000000000010000000000010000000000010000", -- 1
             B"000000010000000000010000000000010000000000010000", -- 2
             B"000000010000000000010000000000010000000000010000", -- 3
             B"000000010000000000010000000000010000000000010000", -- 4
             B"000000010000000000010000000000010000000000010000", -- 5
             B"000000010000000000010000000000010000000000010000", -- 6
             B"000000010000000000010000000000010000000000010000", -- 7
             B"000000010000000000010000000000010000000000010000", -- 8
             B"000000010000000000010000000000010000000000010000", -- 9
             B"000000010000000000010000000000010000000000010000", -- a
             B"000000010000000000010000000000010000000000010000", -- b
             B"000000010000000000010000000000010000000000010000", -- c
             B"000000010000000000010000000000010000000000010000", -- d
             B"000000010000000000010000000000010000000000010000", -- e
             B"000000010000000000010000000000010000000000010000", -- f
             B"000000010000000000010000000000010000000000010000", -- 10
             B"000000010000000000010000000000010000000000010000", -- 11
             B"000000010000000000010000000000010000000000010000", -- 12
             B"000000010000000000010000000000010000000000010000", -- 13
```

```
B"00000001000000000001000000000001000000000010000" , -- 14
B"00000001000000000001000000000001000000000010000" , -- 15
B"00000001000000000001000000000001000000000010000" , -- 16
B"00000001000000000001000000000001000000000010000" , -- 17
B"00000001000000000001000000000001000000000010000" , -- 18
B"00000001000000000001000000000001000000000010000" , -- 19
B"00000001000000000001000000000001000000000010000" , -- 1a
B"00000001000000000001000000000001000000000010000" , -- 1b
B"00000001000000000001000000000001000000000010000" , -- 1c
B"00000001000000000001000000000001000000000010000" , -- 1d
B"00000001000000000001000000000001000000000010000" , -- 1e
B"00000001000000000001000000000001000000000010000" , -- 1f
B"00000001000000000001000000000001000000000010000" , -- 20
B"00000001000000000001000000000001000000000010000" , -- 21
B"00000001000000000001000000000001000000000010000" , -- 22
B"00000001000000000001000000000001000000000010000" , -- 23
B"00000001000000000001000000000001000000000010000" , -- 24
B"00000001000000000001000000000001000000000010000" , -- 25
B"00000001000000000001000000000001000000000010000" , -- 26
B"00000001000000000001000000000001000000000010000" , -- 27
B"00000001000000000001000000000001000000000010000" , -- 28
B"00000001000000000001000000000001000000000010000" , -- 29
B"00000001000000000001000000000001000000000010000" , -- 2a
B"00000001000000000001000000000001000000000010000" , -- 2b
B"00000001000000000001000000000001000000000010000" , -- 2c
B"00000001000000000001000000000001000000000010000" , -- 2d
B"00000001000000000001000000000001000000000010000" , -- 2e
B"00000001000000000001000000000001000000000010000" , -- 2f
B"00000001000000000001000000000001000000000010000" , -- 30
B"00000001000000000001000000000001000000000010000" , -- 31
B"00000001000000000001000000000001000000000010000" , -- 32
B"00000001000000000001000000000001000000000010000" , -- 33
B"00000001000000000001000000000001000000000010000" , -- 34
B"00000001000000000001000000000001000000000010000" , -- 35
B"00000001000000000001000000000001000000000010000" , -- 36
B"00000001000000000001000000000001000000000010000" , -- 37
B"00000001000000000001000000000001000000000010000" , -- 38
B"00000001000000000001000000000001000000000010000" , -- 39
B"00000001000000000001000000000001000000000010000" , -- 3a
B"00000001000000000001000000000001000000000010000" , -- 3b
B"00000001000000000001000000000001000000000010000" , -- 3c
B"00000001000000000001000000000001000000000010000" , -- 3d
B"00000001000000000001000000000001000000000010000" , -- 3e
B"00000001000000000001000000000001000000000010000" , -- 3f
B"00000001000000000001000000000001000000000010000" , -- 40
B"00000001000000000001000000000001000000000010000" , -- 41
B"00000001000000000001000000000001000000000010000" , -- 42
B"00000001000000000001000000000001000000000010000" , -- 43
B"00000001000000000001000000000001000000000010000" , -- 44
```

```
                B"00000001000000000001000000000001000000000010000" , -- 45
                B"00000001000000000001000000000001000000000010000" , -- 46
                B"00000001000000000001000000000001000000000010000" , -- 47
                B"00000001000000000001000000000001000000000010000" , -- 48
                B"00000001000000000001000000000001000000000010000"   -- 49
                                                    );
    begin
      data_memory(data'range) := data;
    end load;

  begin
    load;
    -- initialize output
    mem_d_out <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

    -- process memory cycles
    loop
      -- wait for a read or write command, valid on leading edge of clk
      wait on clk until rising_edge(clk);
        -- decode address and perform command if selected
        row_address := to_integer(unsigned(to_bitvector(mem_a)));
        write_access := mem_write_en = '1';
        if row_address <= high_address then
          if(mem_enable = '1') then
            if write_access then
              -- write cycle
              data_memory(row_address) := mem_d_in ;
              mem_d_out <= "
                  ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
            else
              -- read cycle
              mem_d_out <= data_memory(row_address);
            end if;
          else
            mem_d_out <= "
                ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
          end if;
        end if;
    end loop;

  end process data_memory_behavior;

end architecture behavior;
```

A.3.5 The Test Bench Input Data Reader

This module reads the input data for the filter application from a file.

```
library ieee;

use ieee.std_logic_1164.all,
    ieee.std_logic_textio.all,
    work.numeric_bit.all;

use std.textio.all;

entity input_gen is
  port ( clk : in std_logic;
         data_in : out std_logic_vector(31 downto 0);
         out_en : in std_logic);
end entity input_gen;

architecture behavior of input_gen is
begin

  -- this process reads the input from a file
  read_input : process(clk) is

    file input_file : text open read_mode is "filter_input";
    variable line_in : line;
    variable data : std_logic_vector(31 downto 0);


  begin
    if rising_edge(clk) then
     -- reads input when an output is completed
     if out_en = '1' then
        readline(input_file, line_in);
        read(line_in, data);
        data_in <= data;
        if endfile(input_file) then
          assert false report "Simulation is complete. End of Stimulus
              File"
                  severity note;
        end if;
     end if;
    end if;
  end process read_input;

end architecture behavior;
```

VITA AUCTORIS

Mahzad Azarmehr was born in Esfahan, Iran, in 1965. She received her B.A.Sc. degree from the University of Tehran,Tehran, Iran, in 1990, and the M.A.Sc. degree from University of Windsor, Windsor, Canada in 2007, both in Electrical Engineering. She is currently in the Electrical and Computer Engineering Ph.D. program at the University of Windsor. Her research interests include VLSI circuit design, computer arithmetic, HDL synthesis and digital signal processing.