uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Ben Jamil Watany
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical and Computer Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Application of FPGAs in Acceleration of Numerical Solution of Differential Equations

TITRE DE LA THÈSE / TITLE OF THESIS

Hussein Mouftah
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Emad Gad
CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Mustapha Yagoub

Tadeus Kwasniewski

Miodrag Bolic

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

APPLICATION OF FPGAS IN ACCELERATION OF NUMERICAL SOLUTION OF DIFFERENTIAL EQUATIONS

by

Watany Ben Jamil

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillement of the requirements
for the degree of
Master of Applied Science
In
Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering,
School of Information Technology and Engineering,
University of Ottawa
Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 978-0-494-65967-0
Our file Notre référence
ISBN: 978-0-494-65967-0

Abstract

This thesis proposes a new application for Field Programmable Gate Array in the acceleration of the numerical solution of differential equations. By using the FPGA as a coprocessor that can be integrated with the system main processor, certain tasks that represent a computational bottleneck can be offloaded to the FPGA coprocessor and carried out more efficiently. More specific, the domain of differential equations considered in this thesis arises during the transient simulation of nonlinear circuits. The work in this thesis investigates the various computational tasks involved in the numerical solution of differential equations. A recent approach to solve differential equations numerically has been studied that requires the computation of high-order derivatives of circuit variables (e.g. node voltages, charges) with respect to a single parameter such as time. However, complex nonlinear devices, are typically characterized by complex nonlinear functions with mathematical expressions that render such computations on conventional computing platforms very time-consuming.

This thesis demonstrates that using a computational platform with hardware-enabled accelerator can speed up the task of computing high-order derivatives by at least one-order-of-magnitude. The main idea of the thesis is based on using some recently derived formulas representing the high-order derivatives in terms of the lower-order ones to configure a Field Programmable Gate Arrays (FPGA) in a tree-like structure that represent the non-linear expression. The nodes of this tree will represent common non-linear terms such as exponential or logarithmic functions, which will programmed to propagate their own derivatives from the knowledge of their "children" nodes derivatives.

It is shown that this scheme has the potential of relieving the central processing unit in the conventional platforms from having to fetch the tree structure from the system memory and process it in computing the derivatives and will thus lead to significant acceleration.

Dedication

To my parents, my wife,

my brothers and sisters,

and my lovely kids

Acknowledgements

To Dr. Hussein Mouftah and Dr. Emad Gad for their guidance and support. To My wife for the time she spent reading this document.To Dr. Mohammed Mohammed for his help in Latex

Contents

List of Tables

List of Figures

NOMENCLATURE

$:=$	denotes definition.

ABBREVIATIONS

CTAN	Comprehensive TEX Archive Network, e.g.,
FP	Floating-Point.
ODE	Ordinary Differential Equations
DAE	Differential Algebraic Equations
CAD	Computer Aided Design
MNA	Modified Nodal Approach
LNS	Logarithmic Number Systems
FPGA	Field Programable Gate Array.
ASIC	Application Specific Integrated Circuit
GFLOPS	Giga Floating Point Operations Per Second

MFLOPS	Mega Floating Point Operations Per Second
TFLOPS	Tera Floating Point Operations Per Second
LNS	Logarithmic Number System
ULP	Unit in the Least significant Position
GPP	General Purpose Processor
HOD	High Order Derivatives
ALUT	Adaptive Look Up Table
DLR	Dedicated Logic Registers
LE	Logic Elements
API	Application Program Interface
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
IFFT	Inverse Fast Fourier Transform
DCT	Discrete Cosine Transform
CPU	Central Processing Unit
FPU	Floating Point Unit
GPU	Graphics Processing Unit
DFT	Discrete Fourier Transform

FIR	Finite Impulse Response
HCS	Hybrid Computing System
FSB	Front Side Bus
COTS	Commercial Off The Shelf
PLL	Phase Lock Loop
ALUT	Adaptive Look-Up Tables
DLR	Dedicated Logic Registers

Chapter 1

Introduction

1.1 Motivation

1.1.1 The Rationale of Using FPGAs in Acceleration Systems

Floating-point arithmetic is used in many High Performance Computing (HPC) applications and across many market segments. It is used in the simulation of many physical phenomena, financial analysis, bio-informatics, molecular dynamics, radar, seismic imaging, defence and intelligence just to name a few. Depending on the complexity of the problem, however, this can require long computational times. Although available computing power has increased considerably in the last decade, so has the complexity and the size of these problems. To accelerate these arithmetic operations three options are available [78] [74]:

- Using supercomputers: which is easy to implement although at substantial cost
- Parallel computing by running applications on clusters of workstations to realize a global system into which the overall computing power could ideally be the sum of the single ones

- Implementing dedicated hardware systems (accelerators) able to speed up those operations, which represent the core of the calculations done. These systems are then embedded in PC or workstations which drive their activity and manage the results

Field-Programmable Gate Arrays (FPGAs) are considered low cost and require little design skill in comparison to ASIC (Application Specific Integrated Circuit), and hence the great popularity of using them in implementing special purpose hardware to simulate physical or mathematical systems. FPGAs performance doubles every year and a dedicated hardware using them seems to be a competitive alternative to supercomputers and clusters. Being lower in cost, lower in power, and smaller in design form, FPGAs provide several competitive features over rival high performance computing paradigms. Yet, there are still disadvantages with these systems. First it's not easy to use, since they require knowledge of circuit design and usage of FPGA implementations and most users are software programmers with limited knowledge in this area. Secondly, it is time consuming to design and build a system. Third, for double precisions applications there are still resource problems in the FPGA.

1.1.2 Historical Background

Actual large scale development of FPGA started in 1990s. However, interest in the development of acceleration systems predates at the development of FPGA and accompanied the birth of computers through optimizing a computation features to speed up the application time. Custom hardware development in the form of ASIC was also carried out in the last decade as in the genome project. Nevertheless, it is easy to judge by now that FPGA-based acceleration systems represent the lowest cost of acceleration forms. This fact explains the observation of the rise in the number of application utilizing the FPGAs in arithmetic calculation.

1.1.3 Quick Review for HPC Applications using FPGA

FPGA advancements has enabled the emergence of many acceleration and real time applications that require floating-point arithmetic like Monte Carlo simulations, biopharma, molecular dynamics, genome research, radar, DSP , audio and medical imaging to name a few. Below are examples of these systems and their performances:

Monte Carlo Simulations Since the introduction of FPGAs, hardware acceleration systems has been tested to speed up MonteCarlo simulations in many physical phenomenons. In 1996 Adam Postula and his colleagues [73] used FPGA in the design of a specialize processor for the simulation of metallurgical sintering. He was able to achieve speedups of 50 times the workstation speed. G. Dense and his colleagues [74] [78] conducted more recent application designing application specific MonteCarlo processor for simulating dipolar systems. They used double precision floating point and were able to achieve 4 times the speedups over Intel processor (P3 and P4 at 1.7 GHz and 3GHz) with Altera Startix Pro Development Kit(EPS140). Using Startix 2 FPGA would have allowed up to 24 times the speedup due to significant improvement for floating point arithmetic's primitives. In the financial engineering fields Garry W. and his colleagues [75] used Monte Carlo simulations to benchmark the pricing of European option. They compared the acceleration that can be achieved using FPGA acceleration hardware versus the software, or using Graphic Processing Units - GPU systems, or an IBM cell broadband. The FPGA had an acceleration of 146x compared to the software implementation and 32x compared to all other acceleration systems(GPU, IBM). G.L. Zhang and his colleagues [76] built a reconfigurable acceleration for Monte Carlo based financial simulations. They used the system to simulate Brace Gatarek and Musiela(BGM) interest model for pricing derivatives. They achieved 25x times speedup over Intel Pentium machine running at 1.7 GHz. Nathan A. Wood and Tom VanCourt [77] used Quasi Monte Carlo (faster than the tra-

ditional Monte Carlo) methods in pricing derivatives securities in a FPGA acceleration systems. They achieved 50x speedup over 3GHz Intel multi core Xeon processor.

BioPharma and Medical Applications N. Alachiotis and his colleagues [70] [71] explored FPGAs for accelerating the phylogenetic likelihood function in the field of bioinformtics. They achieved 8.3x speedups over a single-core. R.K. Snider [69] developed a reconfigurable modeling platform using FPGA to create a digital methodology combining theoretical and experimental neuroscience. This kind of an application is not even feasible using software due to the large amount of data that needs to be saved in order to process the application.

Molecular Dynamics and Genome Yongfeng Gu and his colleagues [55] reported on the integration of an FPGA accelerator into the ProtoMol molecular dynamics code. They achieved 5.5x to 15.7x speedup over 2.8 GHZ processor. Matt Chiu and his colleagues [56] used Altera Stratix III to show the performance potential of molecular dynamics simulations. They experimented both single and double precision hardware implementation. They achieved 28x speedups in double precision comparing with a single core microprocessor, and 146x times speedups in single precision compared to a single core microprocessor. Ronald Scrofano and Viktor Parsanna [57] [59] investigated advanced electrostatic in molecular dynamics using reconfigurable systems. They achieved 2.7-2.9x speedup over the corresponding software only simulation. Jung Sub Kim and his colleagues [58] developed a tool for generating accelerator for numerical computations on reconfigurable systems from Matlab. They tested the accelerators for three different applications: the calculations of gravitational potential in astrophysics, the diffusion or convolution with Gaussian kernel common in image processing, and the force calculation with vector-valued kernel function in molecular dynamics.

Radar and SAR Radar Lie Zhenyu [60] developed matched filter radar for real time systems

using FPGA. The hardware implemented parallel and FFT, IFFT. Zhi -Jian and Xui-Mei [61] developed SAR (Synthetic Aperture Radar) Radar image for real time speeds using FPGAs.

DSP applications FIR, DCT, Laplace transform In 1998 Al Walter and Peter Athanas [46] designed a scalable FIR using 32-bit floating-point on a configurable systems achieving 160 MFlops. Sherman Barganza and Miriam Leeser [44] implemented a 1D Discrete Cosine Transform (DCT) for large point sizes on a reconfigurable hardware. DCT is widely used in place of DFT (Discrete Fourier Transform) in audio and image processing due to it's energy compaction properties. Milan Tichy and his colleagues [45] implemented adaptive filters based on GSFAP on FPGAs. Andrea Suardi and his colleagues [47] implemented double precision floating point FIR on FPGA. Zhu Bo and his colleagues [48] implemented Haar wavelet transform on FPGA. All these FPGA implementations provide real time applications which would not be achievable under conventional software implementations

FFT K. Scott Hemmert and Keith D. Underwood [62] analysed the double precision floating-point FFT on FPGAs. Due to lower clock rates of FPGAs in comparison to CPUs, and higher latency, for small FFTs the FPGA seem inferior to microprocessors but for larger FFTs the FPGAs dramatically outperform microprocessors. High-speed parallel implementation of floating-point FFT on FPGAs was successfully conducted in [63] [64] [65] [66] [67] [68].

Audio and Medical Imaging Hoang C Nguyen and his colleagues [49] introduced an FPGA based implementation for processing 2D images of laser Doppler blood flow with 1024 FFT points. J. Living and colleagues [50] developed high performance integer decimators for video applications using FPGAs. Alima Damak and her colleagues [51] developed neural network edge detector used in medical imaging using FPGA and floating

point implementations. S. Kobayashi and his colleagues [52] demonstrated audio applications like MP3 decoder and surround-sound system using block floating-point DSP in an FPGA. Marek Gorgon and Jaromir Przbylo [53] designed a heterogenous image processing system using FPGA. Antoine B. and his colleagues [54] implemented a high resolution phase shift beamformer on FPGA that can be used for 2D and 3D ultrasound medical imaging.

1.2 Thesis Objectives

This thesis seeks to expand the area of applications of FPGA in numerical acceleration of floating point operations. In particular, the problem of numerical solution of Ordinary Differential Equations (ODEs) and Differential Algebraic Equations (DAEs) is almost ubiquitous in all scientific and engineering domains. For example, to understand an observable fact, an experimentalist or a theoretician typically needs to model the dynamics of the underlying phenomenon in the form of system of differential equations whose solution represents the evolution of the phenomenon over time. Indeed the spectrum in which this problem arises is vast and includes applications as far apart as the propagation of the electric pulses down a nerve fibre to the nuclear process of the cores of the stars.

The goal of this thesis is to tackle some of the critical bottlenecks encountered in the numerical solution of systems of ODEs or DAEs . More specifically, the thesis investigates one of the recent techniques used to solve differential equations numerically [2] and identifies a certain bottlenecks related to the computation of high order derivatives. The ultimate objective of this work is to deploy an FPGA as a coprocessor where this task of computing the high order derivatives is offloaded to the coprocessor, thus freeing the main processor of the system to other main tasks which may not be handled efficiently by the FPGA coprocessor.

1.3 Thesis Contributions

The work conducted in this thesis demonstrate that the task of computing high-order derivatives is better suited on a platform with FPGA as coprocessor more than on a conventional processing platform. The recursive formulation of high-order derivatives is studied and a specific hardware configuration is proposed to carry out the computations efficiently. It is demonstrated that the numerical computation of high-order derivatives can be up to one-order-of-magnitude faster on an FPGA platform than on a conventional processing platform.

1.4 Thesis Organization

In chapter 2, a discussion of the fundamentals of the floating point arithmetic, the IEEE floating point standard and its arithmetic operations add, subtract, multiply, divide, and square root, and their hardware implementations. In chapter 3, a literature review of the FPGA implementation of floating point arithmetics. In chapter 4, an overview of the proposed application is described. Chapter 5 provides a discussion of the simulation results of the FPGA implementation, a brief description of the simulation platform, and the simulations results. Chapter 6, presents some conclusion remarks and future research directions.

Chapter 2

Background to Arithmetic Processing

The chapter presents a general overview of representing real number in the form of finite precision number that can be stored and manipulated by a digital computer. Section 2.2 describes the common systems used in representing real numbers. Section 2.3 presents the basic algorithms used in floating-point manipulation. Section 2.4 describes some hardware circuit implementation for the floating-point manipulation algorithm.

2.1 Introduction

There are two kinds of methods of representing real numbers in computer numeration systems, fixed-point method and floating-point method. While both methods have a constant relative precision, the floating point method can represent a substantially larger range of numbers than the fixed point.

2.2 Representation Systems for Arithmetic Data

2.2.1 Fixed-Point Representation of Real Numbers

In the system of fixed-point, a real number is represented as follows:

$$X_{n-p-1}X_{n-p-2}...X_1X_0.X_{-1}X_{-2}...X_{-p}$$

where p is the number of digits to the right of the point and n is the total number of digits used to represent the real number. In the actual practical implementation, the point is not stored physically. Instead, a particular choice of base, B, e.g. (2 or 10) is adopted where the real number corresponding to the above representation (without the point) is given by X/B^P, where is X is the integer represented by the same sequence of digits without point.

There are two parameters that go with every computer based system used in manipulating real numbers. The first parameter is the possible range of the values that can be represented. In the fixed-point system, the range of real numbers that can be represented falls between X_{max}/B^P and X_{min}/B^P where X_{min} and X_{max} are integers whose values depend on the representation scheme. For example, if a sign-magnitude is being adopted to represent the integer X, X_{min} and X_{max} would be given respectively by:

$$X_{min} = 1 - B^{n-1}$$
$$X_{max} = B^{n-1} - 1$$

If, on the other hand, a B's complement or excess $-B^n/2$ representation is used, the values for X_{min} and X_{max} would be:

$$X_{min} = -B^n/2$$
$$X_{max} = B^n/2 - 1$$

The other parameter necessary to characterize a number representation system is related to the accuracy of the system. This parameter refers to the smallest increment unit or alternatively

"*unit in the least significant position*" (*ulp*). $ulp/2$ represents the maximum error that can be incurred as a result of representing a real number with infinite precision in a finite-precision machine. For the system of fixed point, ulp is given by $1/B^P$, thus making the maximum error associated with that system equal to

$$\text{Maximum error} = ulp/2 = (1/B^P)/2$$

Another related parameter is the maximum relative error which is scaled by the absolute value of X, as shown next:

$$\text{Maximum relative error} = ulp/(2 \times |X|) = 1/(2 \times |X| \times B^P) \leq 1/2$$

(if $X \neq 0$ then $X \geq B^{-P}$ which leads to above conclusion)

To illustrate the fixed-point system, assume $n = 10$, $B = 10$, $p = 4$, then the parameters corresponding to this system (with B's complement representation) are as follows:

$$\begin{aligned} X_{min} &= -10^{10}/2 \times 10^{-4} = -10^6/2 \\ X_{max} &= 10^{10}/2 \times 10^{-4} = 10^6/2 \\ ulp &= 10^{-4} \end{aligned}$$

2.2.2 Floating-Point Representation of Real Numbers

The floating point system, a real number is represented by the following form:

$$\pm sb^e$$

where:

- s is known as the significand which is a non-negative number represented by fixed-point form
- e is the exponent

- b is the base chosen for representation (not necessarily the same as B used to represent the fixed-point significand s)

The range of the floating-point numbers that can be represented are:

$$-s_{max} \times b^{e_{max}} < X < s_{max} \times b^{e_{max}}$$

The minimum absolute value of a represented number is:

$$|X| \geq s_{min} \times b^{e_{min}}$$

The maximum error is equal to:

$$D_{max}/2 = ulp \times b^{e_{max}}/2$$

Where $ulp = d$, and d is the distance between two successive values of the significand. D, the distance between exactly represented numbers, equals to $D = d \times b^e$

The maximum relative error is equal to:

$$D/(2 \times |X|) \equiv ulp \times b^e/(2 \times s \times b^e) \equiv ulp/(2 \times s)$$

As in the fixed-point systems this number is less than or equal to $1/2$

The ANSI/IEEE ANSI1985 standard is presented in the next subsection as an example to illustrate a particular implementation of a floating-point system.

Floating point standard ANSI/IEEE ANSI1985

This standard [1], defines a binary-based (b=2) floating point of the form

$$(-1)^{\text{sign}} \times \underbrace{(m_0.m_1m_2...m_{p-1})}_{\text{mantissa}} \times 2^e$$

Other notation commonly used in the literature is the following one:

$$(-1)^{\text{sign}} \times (1.f).2^{e-BIAS}$$

where f stands for fractional part of the mantissa and BIAS=127

The standard defines two formats for floating point representation: the basic and the extended. Table 2.1 and 2.2 show the width of the various fields for both basic formats the single and double precision, respectively.

Table 2.1: Single 32-bit Word Format

Bit-widths	1	8	23
field	**sign**	**exponent**	**mantissa**
Bit-order		msb lsb	msb lsb

The basic single precision is 24-bits and the basic double precision is 53-bits. The single extended word is more than 32-bits and the double extended word is more than 64 bits.

In addition to the ordinary numbers, the standard also defines a set of non-ordinary numbers, which are ± 0, NaN, $\mp\infty$, and the denormalized number (when $m_0 = 0$). The values of these numbers are defined as follows:

Table 2.2: Double 64-bit Word Format

Bit-widths	1	11	52
field	**sign**	**exponent**	**mantissa**
Bit-Order		msb lsb	msb lsb

$$+0 \cong +1.0x2^{-127}$$

$$-0 \cong -1.0x2^{-127}$$

$$+\infty \cong +1.0x2^{128}$$

$$-\infty \cong -1.0x2^{128}$$

The standard also includes definitions for several arithmetic operations as well as rounding modes. The arithmetic operation include add, subtract, multiply, divide, square root, remainder, compare operations, conversions between different format of floating points and integers and decimals. Also it defines exception operations like: invalid operation, divide by zero, underflow, overflow, and inexact. Also, a trapping method for the exception is defined. On the other hand, the rounding modes include nearest, to 0, to $+\infty$, or to $-\infty$

2.2.3 Logarithmic Number System

Logarithmic Number System (LNS) have similar range and precision to the floating point systems. However, they enjoy a certain advantage compared to floating point implementation as far as hardware implementation is concerned. For example, the multiplication and division in LNS is simplified to additions and subtractions of fixed point numbers. As a result, the area requirements for those operations is significantly smaller than for floating point operations. Unfortunately, there is no available standard for LNS such as the case of floating point and the review presented in this section relies on the thorough study performed by Haselman [21].

Definition

A logarithmic number can be viewed as a special case of floating point numbers in which the mantissa is a constant number that is always equal to 1. Thus a real number, when represented by an LNS representation, takes the following form:

$$(-1)^{sign} \times 2^{e}$$

where "sign" represents the sign bit and e is a fixed point number represented in 2s complement, with negative values for e representing values that are less than 1.

In the actual hardware implementation only the word, e and $sign$, need be stored and manipulated. However, given the absence of well defined standard, one usually needs to recourse to dedicating special flag bits to signify exceptions or zeros. For example, in the work carried out in [21], it was decided that two flag bits are dedicated to code for zero, $\mp\infty$, and NaN. Table 2.3 lists the various field widths studied in [21], where K-bits are used to represent the integer part of the fixed point exponent e, whereas the fractional part of it represented by L-bits.

Table 2.3: LNS Word Format

Word		sign	exponent	
Field	flags	sign	Integer	Fraction
Bit-width	2-bits	1-bit	*K-bits*	*L-Bits*

It is interesting to note that when K is equal to the mantissa width in a given floating-point system and L is equal to its exponent width, the two systems become equivalent.

For example, the LNS equivalent system to the IEEE single precision standard has $K = 8$ and $L = 23$ bits. Likewise, the LNS equivalent to the IEEE double precision has $K = 11$ and $L = 52$

Single precision floating point equivalent: For $K = 8$ and $L = 23$ the range is:

$$\pm 2^{-129+ulp} \text{ to } 2^{128-ulp} \approx \pm 1.5x10^{-39} \text{ to } 3.4x10^{38}$$

Double precision floating point equivalent: For $K = 11$ and $L = 52$ the range is:

$$\pm 2^{-1025+ulp} \text{ to } 2^{1024-ulp} \approx \pm 2.8x10^{-309} \text{ to } 1.8x10^{308}$$

LNS Multiplication and Division

Given that in the multiplication or division process one needs to add or subtract the exponent of the same base, the multiplication and division of two LNS numbers is reduced to the process of adding or subtracting their respective exponents. This fact follows from the well known logarithmic property:

$$\log_2(A \times B) = \log_2(A) + \log_2(B)$$

Sign of the product can be implemented by an XOR of the two sings. The exceptions being zero, infinities and NaNs implementation which are done using the multiplicands sign bits and the corresponding flags in a similar manner to the one carried out in the IEEE 754 floating point standard.

The LNS multiplication and FP multiplication have the exact result without the exceptions (Overflow). With overflow the LNS flags +/-infinity. The saving in area of LNS multiplier versus FP is significant. The implementation area of LNS multiplier is 18.4x smaller for single, and 27.3x smaller for double [21].

Likewise, division in LNS is reduced to simple subtraction due to the following logarithmic property:

$$\log_2(A/B) = \log_2(A) - \log_2(B)$$

Again, the sign implementation is just an XOR of the signs. The division operation produces the same result as the FP except the possibility of underflow due to the subtraction which results in a difference equal to zero. The saving in area of LNS divider versus FP is much more significant than multiplier. The implementation area of LNS divider is 45.5x smaller for single precision and 93.8x smaller for double precision [21].

LNS Addition and Subtraction

The simple implementation of the multiplication and division operations give the LNS advantage over FP for FPGA implementation. Unfortunately this does not extend to the addition and subtraction operation. The calculation of addition and subtraction is much more complex and requires significant memory storage. Here is the mathematical problem of addition and subtraction:

Assume that we have two numbers A and B and $A > B$: $A = (-1)^{S_A} \times 2^{E_A}$, and $B = (-1)^{S_B} \times 2^{E_B}$

We need to calculate the operation $C = A \pm B$, where $C = (-1)^{S_C} \times 2^{E_C}$

The sign of the number C depends on the assumption $A > B$ hence:

$$S_C = S_A$$

The power of the number C $C = E_C$ is given by:

$$\begin{aligned} E_C &= \log_2(A \pm B) \\ &= \log_2(A(1 \pm B/A)) \\ &= \log_2(|A|) + \log_2(|1 \pm B/A|) \\ &= E_A + F(E_B - E_A) \end{aligned}$$

where:

$$F(E_B - E_A) = \log_2(|1 \pm B/A|) = \log_2(|1 \pm 2^{E_B - E_A}|)$$

It is evident that the difficultly in addition or subtraction stems from the difficulty in calculating the nonlinear function $F(x)$ defined above. For that purpose, values of F(x) must be calculated offline and stored up in a ROM in the form of a lookup table. Some implementation of calculating the $F(x)$ value is to interpolate the nonlinear function like polynomial interpolation. Also, when the word size increases this implementation is not even feasible.

There is no saving in area of LNS adder/subtractor versus FP. On the contrary, the implementation area of LNS adder or subtractor is 3-4.x bigger for single precision and 3.3-4.3x bigger for double precision [21].

LNS Conversion to/from FP

There are some methods of conversion between LNS and FP numbers refer to [21] for an example. These conversions are not exact and error can accumulate for multiple conversions. In spite of this fact, sometimes it is worth it to implement an application in LNS where division and multiplication are dominant, and do the conversion to FP when addition and subtraction are required or when interfacing to FP system.

2.3 Algorithms for FP Arithmetic Manipulations

There two conditions that must be satisfied in order to make arithmetic operation in floating point easier:

1. The significand s is represented in the base B=b

2. The significand falls in the following range $1 \leq s \leq B - ulp$

2.3.1 Addition of FP Positive Numbers

The algorithm of adding two positive numbers

$$m1B^{e1} + m2B^{e2} = mtB^{e}$$

can be summarized into three procedures: Alignment, Rounding and Normalization as required. Note, that in this operation an overflow might occur. An overflow happens when the exponent value is bigger than the allowed value. Here is the algorithm in Table 2.4 followed by a numerical examples and then explanation of the procedures:

Table 2.4: Floating-Point Addition Algorithm

if $e1 \geq e2$ then $e = e1$; $mt := m1 + (m2/B^{e1-e2})$; – This stage is called Alignment
else
$e := e2$; $mt := m2 + m1/b^{e2-e1}$; – Alignment but rearrange the numbers
end if;
if $mt \geq B$ then $e := e + 1$; $mt := mt/B$; –Normalize
$mt := round(mt)$; – Round
if $mt \geq B$ then $e := e + 1$; $mt := mt/B$; – Normalize

Here are few numerical example to illustrate the steps of addition:

Example 1 : Alignment and Rounding Steps Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to add the following two numbers?

$M = 5.123 \times 10^{3} + 4.1234 \times 10^{-1}$ – Adding the following two numbers

$M = (5.1234 + 0.00041234) \times 10^{3}$ – Aligning the exponents

$M \cong 5.1239 \times 10^{3}$ – Rounding the numbers $(5.1239 < 10)$

Example 2 : Alignment, Normalization and Rounding Steps Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to add the following two numbers?:

$M = 5.9876 \times 10^3 + 6.1234 \times 10^2$ – Adding the following two numbers

$M = (5.9876 + 0.61234) \times 10^3 = 15.59994 \times 10^3$ – Alignment

$15.59994 > 10$ – Must Normalize

$M = 1.559994 \times 10^4$ – Normalize

$M = 1.5600 \times 10^4$ – Rounding

As we can see from the algorithm and the examples, in order to add two positive numbers we must first align the numbers such that we have the same exponent. As a result of this step, the significand s is now in the range of $1 \leq m \leq 2-2ulp$. If $m > B$ we must do normalization to keep m in the required range of $1 \leq m \leq B - ulp$. The normalization step is just dividing s by B, i.e. m/B. This step will ensure that the range rule satisfaction:

$$(1 \leq m \leq (2B - 2ulp)/B \leq B - ulp \text{ when } B \geq 2$$

When $e1$ and $e2$ are different, there is a good chance that the new significand after alignment of normalization is no longer a multiple of ulp. Hence the Rounding function would ensure that the new significand is a multiple of ulp. The rounding function can use different methods like truncation, rounding up, rounding down, or round to the nearest.

2.3.2 Difference Between Two FP Positive Numbers

The algorithm of finding the difference between two positive numbers:

$$m1B^{e1} - m2B^{e2} = mtB^{e}$$

is similar to the addition. The difference is instead of addition of significands we do complement subtraction, which might result that the significand values get out of the limits allowed. A new procedure called leading-zeros is introduced which makes a shift of the significand value until it is in the range allowed ($mt > 1$). The procedure counts the number of zeros in the most signfincant locations; it stops counting and shifting when the most significant bit is 1. Also, the use of the sign bit is introduced here to change a negative value back to positive value. Here is the algorithm in Table 2.5 followed by a numerical example and then explanation of the procedures:

Table 2.5: Floating-Point Subtraction Algorithm

if $e1 \geq e2$ then $e := e1$; $mt := m1 - (m2/B^{e1-e2})$; – This stage is called Alignment
else
$e := e2$; $mt := m1/b^{e2-e1} - m2$; – Alignment but rearrange the numbers
end if;
if $mt < 0$ then $mt := -mt$; $sign := 1$; end if; –Flip the sign when the result is negative
$Leading - zeros(mt, k)$; –if the significant bits are zero count them, number of zeros=k
$s := s \times B^K$;$e := e - k$;
$mt := round(mt)$; – Round
if $mt \geq B$ then $e := e + 1$; $mt := mt/B$; – Normalize

As a result of the alignment the value of the significand is :

$$-(B - ulp) \leq mt \leq B - ulp$$

If it is negative, we change the sign bit and keep it positive. If the value is zero, then we use the leading-zero function which count the number of zeros after each shift left of the s digits and adjusts the exponent as a result of this shift. We must note that in this operation an

underflow might occur. An underflow is when the exponent value is less than the minimum allowed.

Here are few numerical example to illustrate the steps of subtraction:

Example 1: Alignment , Sign Change, Rounding Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to subtract the following two numbers?:

$M = 4.1234 \times 10^{-1} - 5.1234 \times 10^{3}$ $M = (0.00041234 - 5.1234) \times 10^{3}$ – Alignment

$M = -5.1229877 \times 10^{3}$ – Sign Change

$M = -5.1230 \times 10^{3}$ –Rounding

Example 2: Alignment , Leading Zeros, Rounding Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to subtract the following two numbers?:

$M = 1.1234 \times 10^{3} - 9.9934 \times 10^{2}$ – Subtract these two numbers

$M = (1.0004 - 0.9994) \times 10^{3}$ – Alignment

$M = 0.0046 \times 10^{3}$ – Leading zeros, we must count them and deduct them from e

$M = 4.6 \times 10^{0}$ –Leading zeros deducted final result.

2.3.3 FP Addition and Subtraction

The addition and subtraction algorithm is a combination of the addition algorithm and the difference algorithm presented in the previous two sections. We add to the algorithm the variable operation 0 for addition and 1 for subtraction.

Given two numbers $(-1)^{sign1} m1B^{e1}$ and $(-1)^{sign2} m1B^{e2}$ and the operation variable, a summary Table 2.6 lists the arithmetic operation required after alignment.

Here is the algorithm in Table 2.7 for addition and subtraction:

Table 2.6: Addition and Subtraction Operation Summary

Operation	*Sign1*	*Sign2*	Actual Operation
0	0	0	$m1 + m2$
0	0	1	$m1 - m2$
0	1	0	$-(m1 - m2)$
0	1	1	$-(m1 + m2)$
1	0	0	$m1 - m2$
1	0	1	$m1 + m2$
1	1	0	$-(m1 + m2)$
1	1	1	$-(m1 - m2)$

2.3.4 FP Multiplication

Given two numbers $(-1)^{sign1}.m1.B^{e1}$ and $(-1)^{sign2}.m2.B^{e2}$, the multiplication result $(-1)^{sign}.mt.B^{e}$ can be obtained by the following algorithm in Table 2.8:

The multiplication is straightforward, multiply the fixed-point mantissas, add the exponents, and xor the signs. The only requirement is to keep the mantissa constraints to:

$$(1 \leq mt \leq (2B - 2ulp)2$$

if it is bigger than B (i.e. $B \leq m \leq (2B - 2ulp)2$ then normalize and round it. Here is an illustration numerical example:

Example: Multiplicities Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to multiply the following two numbers?

$M := 4.1234 \times 10^{-1} \times 5.1234 \times 10^{3}$ – Multiply

$M = 21.12582756 \times 10^{2}$ – $Mantissa > B$ need to Normalize

Table 2.7: Floating-Point Addition and Subtraction Algorithm

if $e1 \geq e2$ then $e := e1$; $mt := m1 - (m2/B^{e1-e2})$; – This stage is called Alignment

else

$e := e2$; $mt := m1/b^{e2-e1} - m2$; – Alignment but rearrange the numbers

end if;

if operation xor sing1 xor $sign2 = 0$ then

$mt := m1 + m2$; – it's obvious from looking at the table

if $mt \geq B$ then $e := e + 1$; $mt := mt/B$;end if; – Normalize

$mt := round(mt)$; – Round

if $mt \geq B$ then $e := e + 1$; $mt := mt/B$;end if; – Normalize

else

$mt := m1 - m2$; – it's obvious from looking at the table

if $mt < 0$ then $mt := -mt$; $sign := 1$; end if; – Flip the sign when negative result

$Leading - zeros(mt, k)$; –if the significant bits are zero count them, number of zeros=k

$s := s \times B^K$;$e := e - k$;

$mt := round(mt)$; – Round

if $mt \geq B$ then $e := e + 1$; $mt := mt/B$; end if; – Normalize

end if;

Table 2.8: Floating-Point Multiplication Algorithm

sign:=sign1 xor sign2; $mt := m1 * m2$; $e := e1 + e2$;
if $mt \geq B$ then $e := e + 1$; $mt := mt/B$;end if; – Normalize
$mt := round(mt)$; – Round
if $mt \geq B$ then $e := e + 1$; $mt := mt/B$;end if; – Normalize

$M = 2.112582756 \times 10^2$ – Normalize

$M = 2.1126 \times 10^2$ – Rounding

2.3.5 FP Division

Given two numbers $(-1)^{sign1}.m1.B^{e1}$ and $(-1)^{sign2}.m2.B^{e2}$, the division $(-1)^{sign1}.m1.B^{e1}/(-1)^{sign2}.m2.B^{e}$ result $(-1)^{sign}.mt.B^{e}$ can be obtained by the following algorithm in Table 2.9 :

Table 2.9: Floating-Point Division Algorithm

sign:=sign1 xor sign2; $mt := m1/m2$; $e := e1 - e2$;
if $mt < 1$ then $e := e - 1$; $mt := mt * B$;end if; – Normalize
$mt := round(mt)$; – Round

In the division case we must maintain :

$$1/B \leq mt \leq B - ulp$$

if mt is less than 1, we normalize it by multiplying by B to keep $1 < mt < B - ulp$ and decrease the exponent by one as a result.

Here is a proof :

$m1 < m2 \Rightarrow m1 <= m2 - ulp \Rightarrow m1/m2 \leq 1 - ulp/m2 < 1 - ulp/B \Rightarrow 1/B < m <$

$1 - ulp/B$

Here is an illustration numerical examples:

Example1: Division Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to divide the following two numbers?

$M = 4.1234 \times 10^{-1}/5.1234 \times 10^{3}$ – Divide these two numbers

$M = 0.80481711 \times 10^{-4}$ – subtract exponents, multiply mantissas, and xor the signs

$M = 8.0481711 \times 10^{-4}$ – Normalize

$M = 8.0482 \times 10^{-5}$ – Round

Example2 : Division Assume $B = 10$ and $ulp = 10^{-4}$

What are the steps to divide the following two numbers?

$M = 5.1234 \times 10^{3}/4.1234 \times 10^{-1}$– Divide these two numbers

$M = 1.24251831 \times 10^{4}$ – subtract exponents, multiply mantissas, and xor the signs

$M = 1.2425 \times 10^{4}$ –Round

2.3.6 FP Square Root

The number mB^e is the square root of the positive number $m1B^{e1}$. The calculations of it's value depends on exponent e1 if it's odd or even:

If $e1$ **is even** $m = (m1)^{1/2}, e = (e1)/2$

If $e1$ **is odd** $m = (m1/B)^{1/2}, e = (e1 + 1)/2$

Here is the full algorithm in Table 2.10:

Table 2.10: Floating-Point Square Root Algorithm

If (e1 mod 2) =1 then $e1 := e1 + 1$; $m1 := m1/B$; end if;
–Similar to Normalize increase the exponent and divide mantissa by B
$m = square_root(m1)$; $e = e1/2$;
If $m < 1$ then $e := e1 - 1$; $m := m \times B$; end if; –Normalize
$m := round(m)$; – Round
If $m \geq B$ then $e := e + 1$; $m := m/B$; end if; –Normalize

2.4 Circuit Implementations of FP Arithmetic Operations and Algorithms

A typical addition and subtraction circuit diagram is shown in Fig. 2.1 [40] [41] [42] . The first step is to unpack the input operands A and B to Mantissa A, Mantissa B, Exponent A, Exponent B, Sign A and Sign B circuit inputs. Then, check for zero, infinity or NaN are performed in the *Exception handling* block. Assuming all operands are normal we can proceed to the next step of performing the actual operation. Based on the sign bits SignA and SignB and the original operation, the effective operation when both operands are made positive is determined refer to Table 2.6. From this point, it can be assumed that both A and B are positive.

The first step of the calculation of the sum or difference of floating-point numbers A and B are as follows. Calculate the absolute value of the difference of the two exponents i.e. (ExponentA- ExponentB) and select the larger number to be the exponent result. Then, select the mantissa that belongs to the larger exponent to be in *Max* output in the Mantissa Selection block, and the mantissa that belongs to the smaller exponent to be in *Min* output in the Mantissa Selection block. Then, shift right the *Min* mantissa by the absolute difference value of the two exponents to *Align* it with bigger exponent.

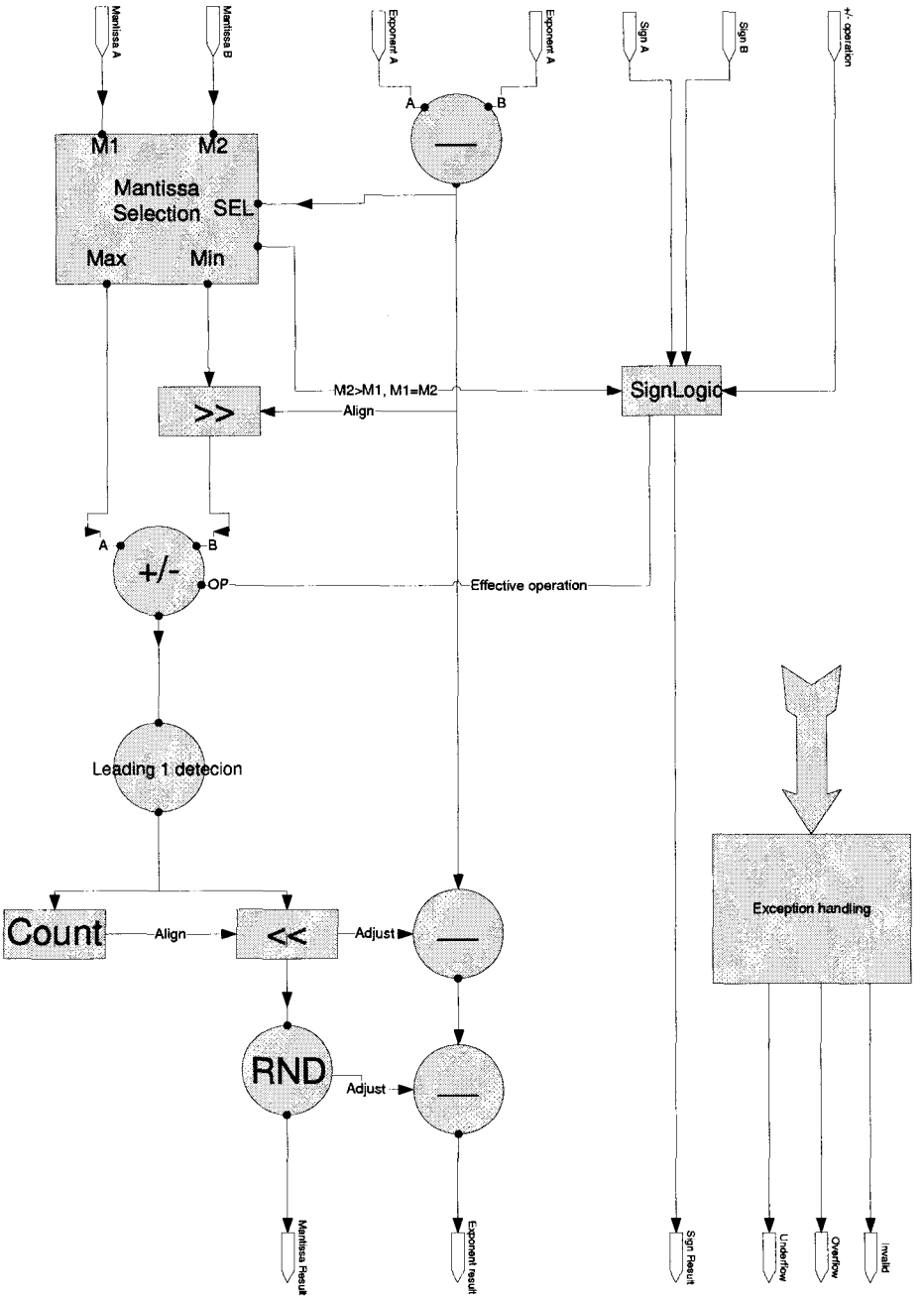

Figure 2.1: Floating Point Add or Subtract Circuit Diagram.

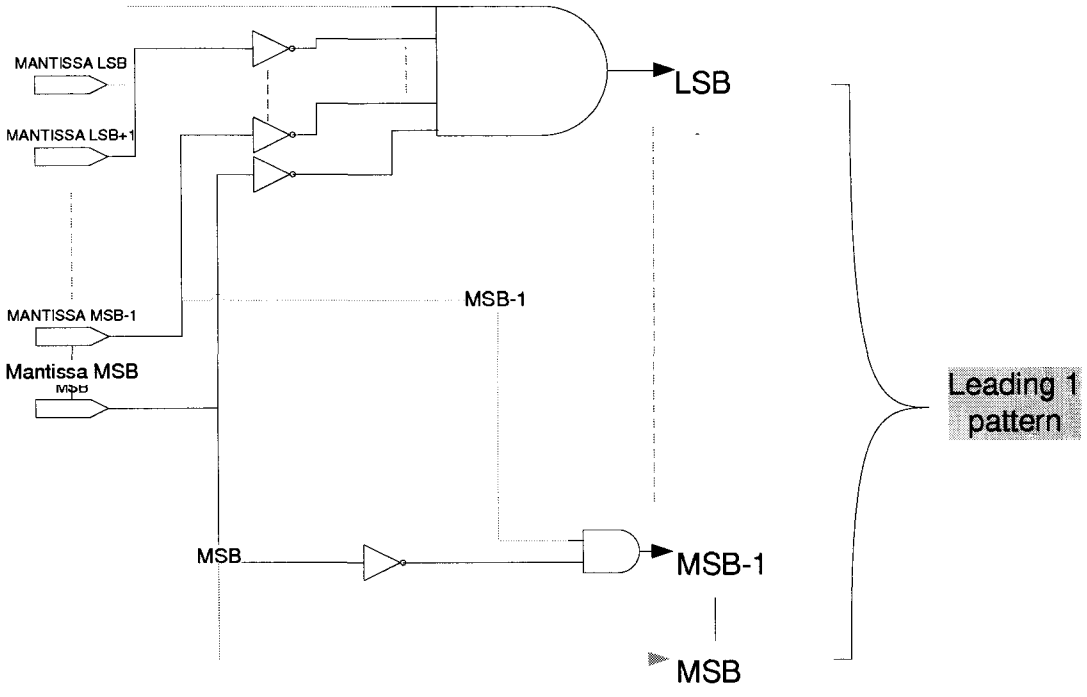

Figure 2.2: Leading 1 Circuit Detect.

The actual add or subtract the two mantissas can now take place according to the *effective operation* and make the result positive if it is negative and feedback to the *SignLogic* block. Next step is to normalize the result of ($\pm$) block. The *Leading-1-detecion* block 2.2 counts the number of zeros from the left until the first 1 in the result mantissa, and then sets counter value of this value for the shift left block ($\ll$). After each shift, the result exponent gets *adjusted* i.e. decremented by one and the counter decremented. The last step in the mantissa calculation is rounding which is done *RND* block. Rounding can cause another adjustment of the exponent.

The hardware of the absolute difference of the exponents can be implemented using two cascaded adders and a multiplexer. The selection of the larger of the two exponents, the expo-

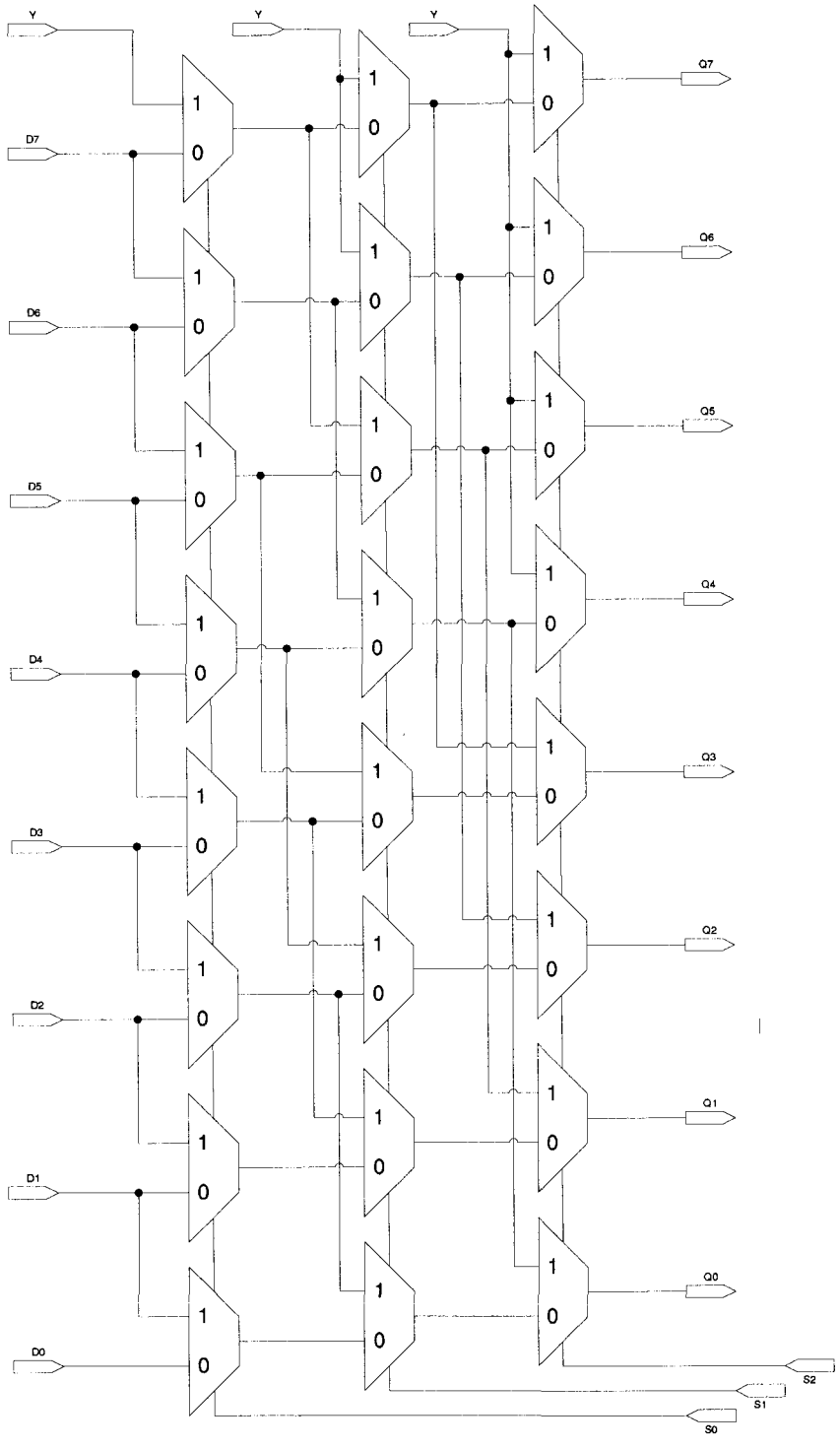

Figure 2.3: Typical Shift Right Barrel Shifter for 8-bits

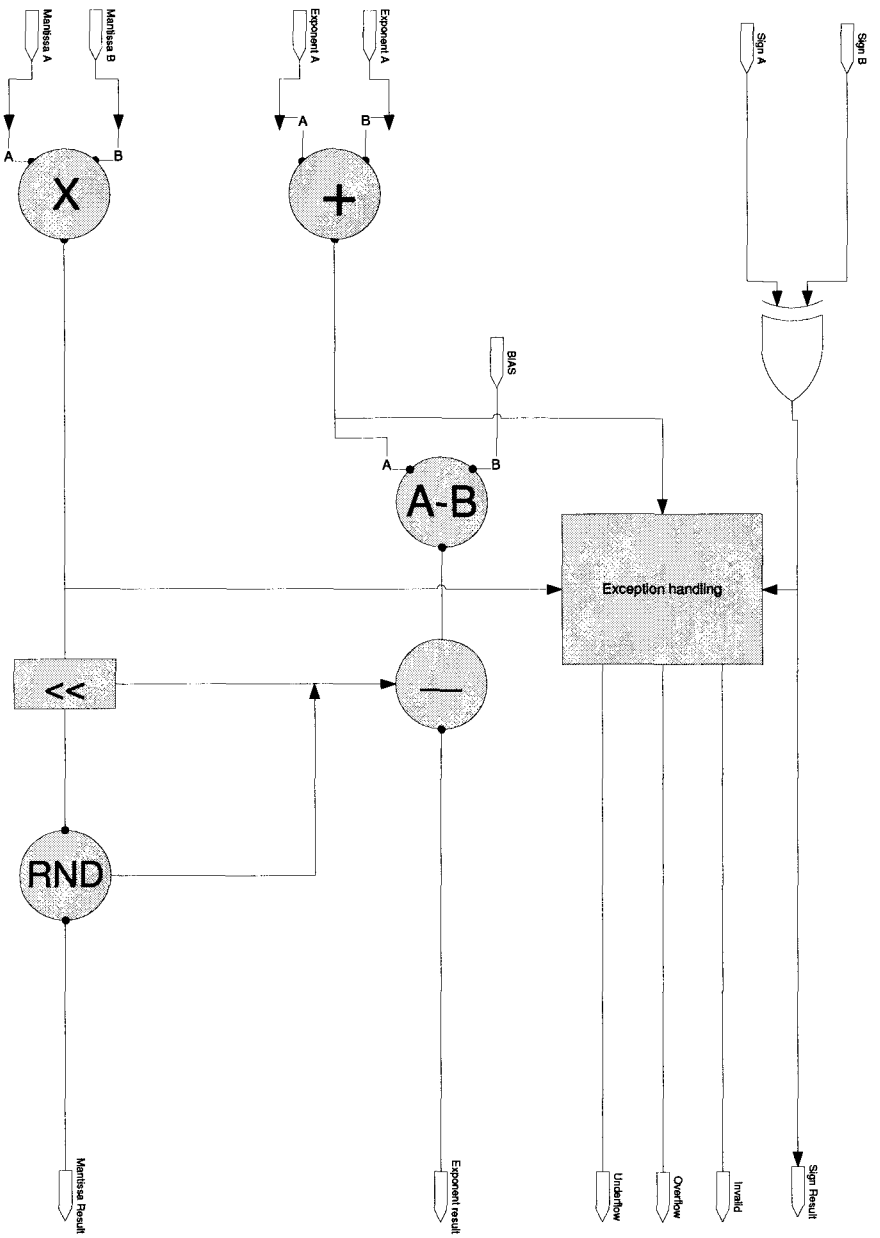

Figure 2.4: Floating Multiply Circuit Diagram

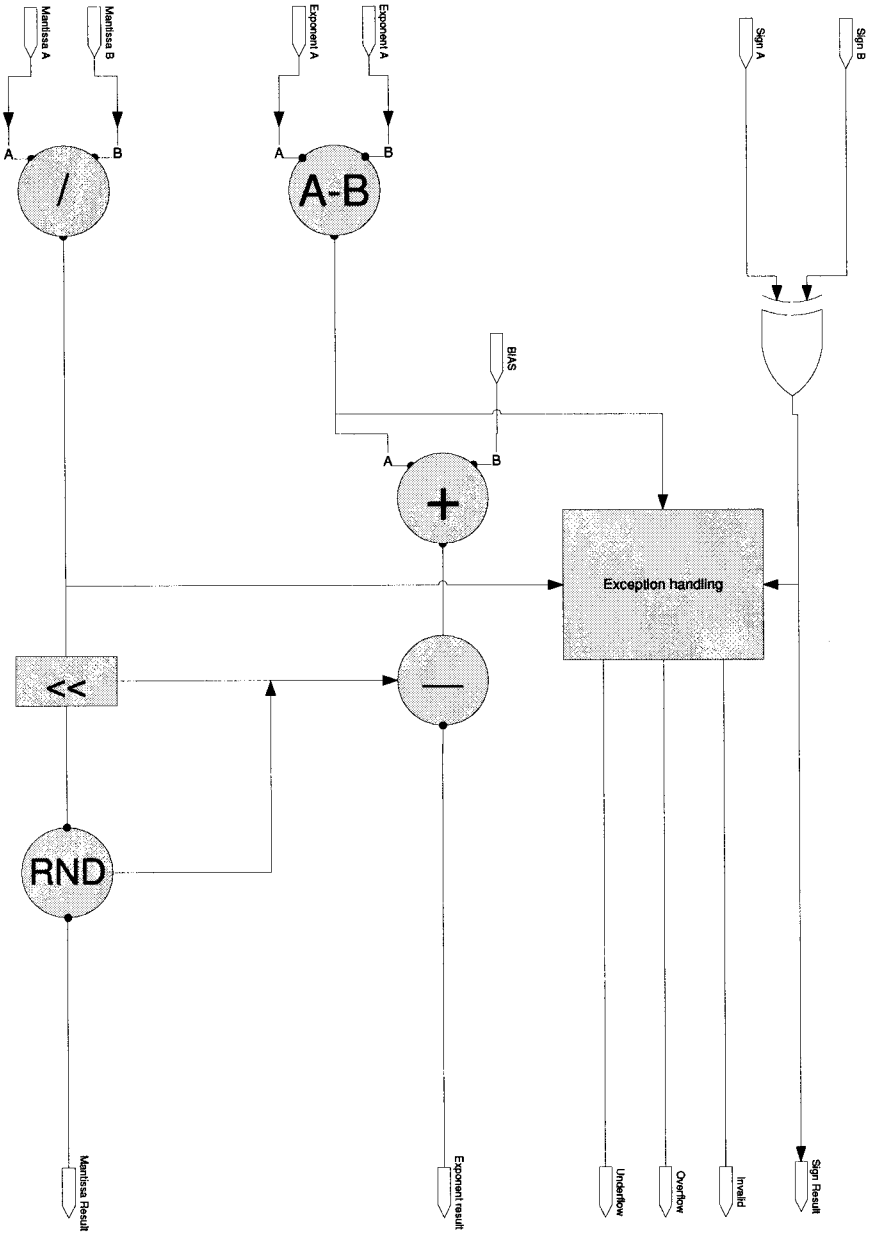

Figure 2.5: Floating Point Division Circuit Diagram

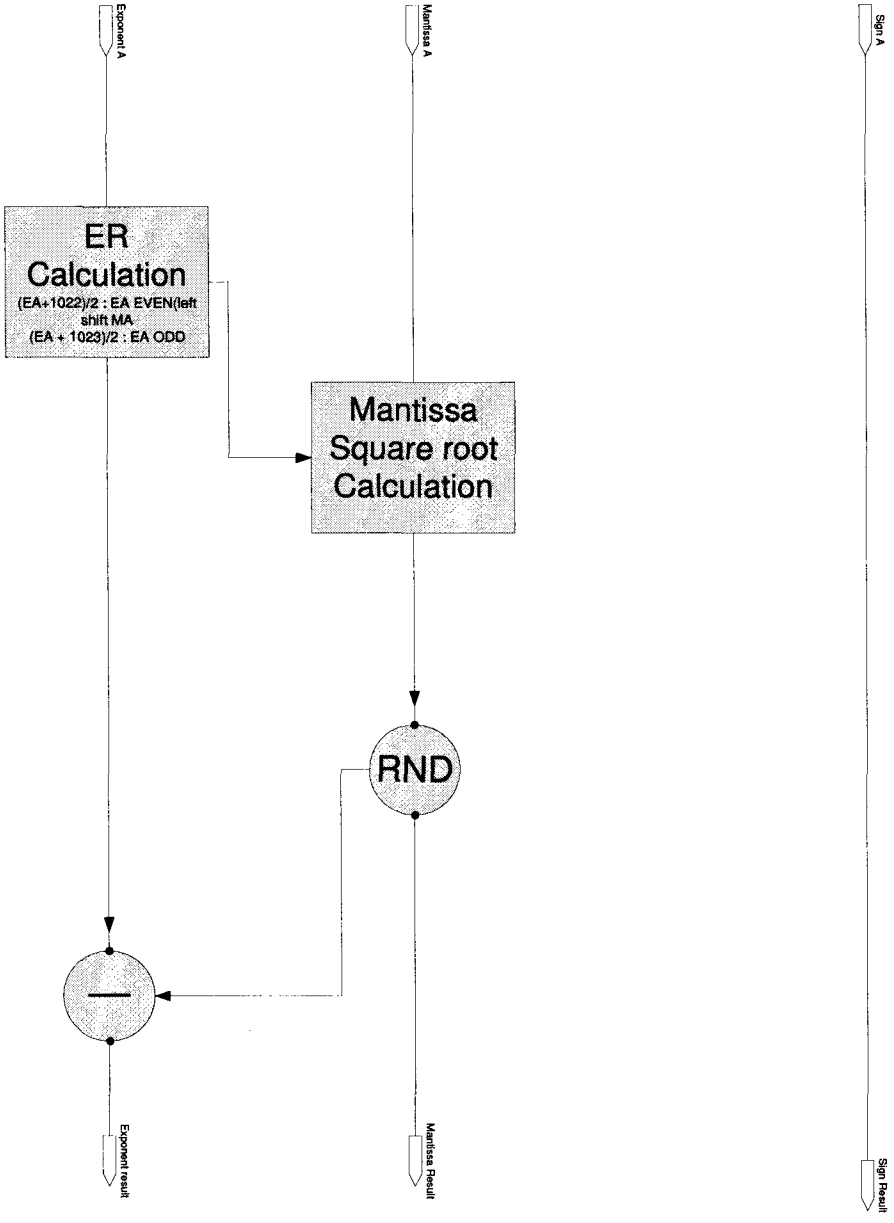

Figure 2.6: Floating Point Square Root Circuit Diagram

nent result depends on exponent A, Exponent B, and mantissa A and mantissa B when Exponent A and B are equal. A double precision mantissa comparison hardware can be implemented by a comparator implemented using seven 8-bit comparators that operate in parallel and an additional 8-bit comparator which processes their outputs. The exponent result can be further adjusted after normalization or rounding by adders circuits. The Mantissa Selection block requires only multiplexors. The mantissa *Min* output from the *Mantissa Selection* block goes through right alignment shifter $\gg$ block can be implemented using several stages of a barrel shifter 2.3. Also, the left alignment shifter which normalizes the output of $\pm$ block is also implemented using a barrel shifter. The $\pm$ block can be implemented using fast ripple-carry adder.The *SignLogic* block is trivial and can be implemented a few logic gates. The *RND* is implemented using adder as well.

Figure 2.3 shows a typical shift right barrel shifter [43]. A programmable shifter capable of shifting arbitrary number of bits in a single clock cycle was first introduced by Intel. Barrel shifter hardware implementation is very important since it could use substantial amount of the FPGA resources. Several types of shifting operations exist depending on the target application, including shift logical, shift arithmetic and rotate. A barrel shifter has n data inputs, n data outputs and a set of control inputs that specify how to shift the data inputs. The control inputs specify the type of shift (logical, arithmetic or circular - circular shift is usually designated rotation), the direction of the shift (left or right), and the amount of shift (from 0 to n - 1).

The following are the most used shifting oeprations:

SRL-Shift Right Logical Shift the n-bits right by m-bits and set the upper m-bits to zeros

SRA-Shift Right Arithmetic Shift the n-bits right by m-bits and set the upper m-bits to the most significant bit to implement sign extension

SRC-Shift Right Circular Shift the n-bits right by m-bits and set the upper m-bits to the lower bits of the input

SLL-Shift Left Logical Shift the n-bits left by m-bits and set the lower m-bits to zeros

SLA-Shift Left Arithmetic Shift the n-bits left by m-bits and set the lower m-bits to zeros. The sign bit does not get shifted

SLC-Shift Left Circular Shift the n-bits left by m-bits and set the lower m-bits to the upper bits of the input

A logarithmic shifter is the most common way to implement barrel shifters. Logarithmic shifter uses $m = \log_2 n$ levels, and each level provides a fixed shifting of 2^x, where $x \in [0, m-1]$. Each shifting level is implemented with n two-input multiplexors (see Figure 2.3 for n=8) where each receives a non-shifted input and a shifted input. The shifted input depends on the type of shifting.

The typical multiplicities circuit is shown in Figure 2.4. First, unpacks the operands A and B to Exponent A, Exponent B, Mantissa A, Mantissa B, and SignA and SignB. Checks are performed for zero, infinity or NaN. Assuming all operands are normal, then we add Exponent A and Exponent B in the $+$ block, then deduct the BIAS value in $A-B$ block, and lastly adjust the exponent result due to the normalization ($\ll$) block or the rounding *RND* block. The mantissa A and B are multiplied and then normalized in the shifter ($\ll$) block, lastly the mantissa result is rounded in the *RND* block. The sign of the result is a simple xor function. Finally, checks are also performed on the exponent result to detect an overflow or underflow. Lastly, zero, infinity or a NaN operands result to a zero.

The addition of Exponent A and Exponent B can be done using fast ripple-carry adder.

The excess BIAS can be removed using the same circuit in a different cycle to keep the circuit utilization low. The mantissas multiplication can be done using the Modified Booths 2-bit parallel multiplier recoding method. The multiplication took 9 cycles in [40] and hence it is recommended that the multiplication clock be faster than the other block's clock.

The typical division circuit is shown in Figure 2.5. First, unpacks the operands A and B to Exponent A, Exponent B, Mantissa A, Mantissa B, and SignA and SignB. Checks are performed for zero, infinity or NaN. Assuming all operands are normal, then we subtract the dividend Exponent A from the divisor Exponent B in the $A - B$ block, then add the BIAS value in $+$ block, and lastly adjust the exponent result due to the normalization ($\ll$) block or the rounding *RND* block. The we divide the fixed point operands Mantissa A over Mantissa B, align it in $\ll$ block and then round it in the *RND* block .

The hardware implementation of most of the division circuit blocks are now mostly known from the previous figures other than the division / block of the mantissa. The division control algorithm are vendor dependant and are customized for area and speed of results. The division algorithm described in [40] is as follows:
The algorithm is based on a simple non-performing sequential algorithm and the division. First, the remainder of the division is set to the value of the dividend Mantissa A. The divisor Mantissa B is subtracted from the remainder. If the result is positive or zero, the MSB of the quotient is 1 and this result replaces the remainder. Otherwise, the MSB of the result Mantissa is 0 and the remainder is not replaced. The remainder is then shifted left by one place. The divisor is subtracted from the remainder for the calculation of MSB-1 and so on. The mantissa divider calculates one mantissa bit per cycle and its main components are two registers for the remainder and the divisor, a fast ripple-carry adder, and a shift register for mantissa result.

The typical square root circuit is shown in Figure 2.6. First, unpacks the operands A to Ex-

ponent A, Mantissa A, and SignA. Checks are performed for zero, infinity or NaN, or negative number. Assuming operand A is positive and not zero we proceed to calculations of the square root.

The biased exponent ER of the result is first calculated directly from the biased exponent EA :

$ER = (EA + 1022)/2$, if EA is even (and left shift Mantissa A one place)

Or

$ER = (EA + 1022)/2$, if EA is odd

ER is calculated using a fast ripple carry adder, while the division by 2 is just a matter of discarding the LSB of the numerator, which will always be even. The calculation of the square root then is executed after the ER.

The SQRT of the Mantissa A is denoted by $Y1Y2...Yn$ and each Yn is calculated in the following manner:

$$\begin{aligned} Yn &= 1 \quad \text{if} \quad Xn - Tn \geq 1 \\ &= 0 \qquad\quad \text{Otherwise} \\ Yn+1 &= 2(Xn - Yn) \quad \text{if} \quad Yn = 1 \\ &= 2Xn \qquad\quad \text{if} \quad Yn = 0 \end{aligned}$$

Initial inputs $X1 = MantissaA/2$ and $T1 = 0.1$, $n = 1, 2, 3...$

The square root calculation algorithm is quite similar to the division. Based on this algorithm, a mantissa square root circuit calculates one bit of the result mantissa per clock cycle. The main components of this part of the circuit are two flip-flop registers for Xn and Tn, and a fast ripple-carry adder. The contents of register Tn forms mantissa square root calculation result.

Chapter 3

Application of FPGAs in Floating-Point Acceleration

This chapter reviews the recent activities aiming at deploying FPGAs as accelerators for numerical floating- point operations. The first section presents a general overview for the historical evolution of the notion of employing FPGAs as numerical accelerators. section 3.2 describes current computational platforms that were designed to bring this notion into mainstream computational platforms.

3.1 Literature Review

FPGA versus CPU and ASICs

FPGAs have been steadily enhancing since the early 90s to accommodate a wide range of applications. The improvements in the area, speed, logic resources, IP cores, tools, power consumption, and price, made FPGAs a must use for many applications including Floating-Point arithmetic. While ASICs (Application Specific Integrated Circuit) has been always available to any special purpose computation application like arithmetic Floating-Point with TFlops per-

formance, they lack the ability of programmability of the hardware to accommodate every mathematical formulation of every scientific problem [10]. Processors, on the other hand, do not satisfy applications that demand high performance computing with high numerical stability and accuracy like floating point with regards to real-time and performance requirement [13].

Moreover, the power consumption of a processor is drawback in comparison with FPGA. Keith Underwood [17] concludes his FPGAs and CPUs peak performance[1] by observing that while CPU doubled in density and clock rate every 18 months according to Moores law[2] FPGAs quadrupled in performance during the same time period. Furthermore, FPGAs were found to be more suitable for floating-point operations because it can customize its resource allocation for arithmetic operation, while CPU has fixed functional units which limits the performance. FPGAs also outperforms CPU on memory hungry applications such as those involving floating-point, even though they are bounded by the number of flip-flops unlike CPUs [18]. Another advantage of FPGAs over CPUs is the fact that CPU designers choose not add FPU (Floating Point Units) [17].

Floating-Point Accuracy

Other research studied the option of translating floating-point to fixed-point or optimizing the floating-point widths as an alternative for speedups. Reducing the number of bits can significantly reduce area and increase speed of calculations. However most applications that resort to floating-point need the high precision and the wide range numbers available in the IEEE standard even some application would require higher precision than the standard [18]. Other numbering scheme like Logarithmic Number System (LNS) provide similar range and preci-

[1]Peak performance is the number of functional units instantiated in an FPGA times the clock rate i.e. the number of operation that can be done per second

[2]Moores law which states that the number of transistors on a device doubles every two years

sion to floating-point with smaller area implementation in FPGA for the arithmetic operation. A comparison of the floating-point implementation and the LNS implementation in FPGA was done by Michake Haselman [21] who found that while multiplication and addition in LNS is simpler and more efficient addition and subtraction are not. Moreover, the format translation from floating-point to LNS is not accurate and area consuming. It is also worth mentioning that while there is a floating-point standard, LNS doe not have one and is not widely used.

Early attempts

Early experimentation with FPGAs in floating-point acceleration started in the early 90s [6]-[8], with efforts to perform the "add" and "multiply" operations in Single Precision using one FPGA. More specifically Barry Fagin and Cyril Renard [6] implemented the IEEE 754 standard in an FPGA from Actel utilizing the anti-fuse technology. The multiplier was the largest component. Unfortunately, the best performance 24-bit multiplier design did not fit in the FPGA. This fact indicates that early generations of FPGAs did not have the size capacity to handle floating-point arithmetic operations.

Later in 1996, Loucas Louca and his colleagues [7] were able to implement addition and multiplication on an Altera FLEX8000 FPGA but with serializing the multiplier inputs bits to save on area consumption. This, however resulted in greater latency. The peak performance reported in this work was 7MFlops for addition and 2.3MFlops for multiplication. To put things into perspective, these numbers should be contrasted to the performance of modern FPGAs which can easily reach tens of GFlops.

Another attempt came in the late 90s when Walter B. and his colleagues [8] implemented the multiplication and addition on Xilinx 4000 series and were able to achieve 3-40 Mflops at

speeds of 33-40Mhz.Some of the techniques used involved deep pipelining to increase speed, reduced precision to reduce area, serial multiplication or booth recoding(two bits serialize at a cycle) to save on resources.

Hardware Library development

With the increase in size and resources available on FPGAs, serious efforts have been dedicated to encapsulating floating-point operations in libraries that can be utilized by a general user.

For example, the work of Pavle Belanovic and Miriam Leeser [9] delivered a parameterized library of floating-point operation modules written in VHDL. The library includes addition, subtraction, multiply, conversion from floating-point to fixed-point and visa versa, and rounding. The library enabled the user to try a real application for K-means clustering algorithm applied to multispectral satellite images on Xilinx XCV1000 FPGA. In this FPGA a single precision adder amounted to only 2.5 percent of the total size, and the whole application occupied 88 percent of the total FPGA size. Being able to customize the width of the mantissa for precision control and the exponent size for numbers range was a key advantage for the success of this library. The work of Xiaojun Wang and his colleagues [11] has extended this library to include division and square root by using small look-up tables and small multipliers an algorithm developed by M.D. Ercegovac [26] and [27] . Another similar library development was done by Gerhard Lienhart [10] with all Floating-Point operations and parameterizable precision. The library was used in an astrophysical N-Body simulations application where gas-dynamical effects are treated by smoothed particle hydrodynmaics method called SPH. The FPGA used Xilinix Vertex2 (XC2V3000), the performance achieved was 3.9 Gflops.

Tools development

As the FPGA sizes improved also research on algorithms and synthesis tools has generated improvement in the latency and area of the arithmetic units [12]. The tool developed by Jian Liang and Russell Tessier [12] provided better latency and half the area size in hardware implementation than the FPGA vendor tools(Xilinix) using C++ library streamed to the FPGA compiler. The tool allows the user to select his priority for either latency, or throughput and area performance.

Comparison between Xilinx 2004 FPGA and Intel's Pentium 4

Gokul Govindu and his colleagues [13] made an analysis of the 2004 FPGA from Xilinx (Virtex-II Pro XC2V125) and thoroughly reviewed and compared it with Pentium processor performance. The authors used matrix multiplication in order to fill the FPGA with the maximum possible operations (multipliers, additions, and subtractions only). The performance achieved was 19.6 GFlops for 32-bit matrix multiplications at a speed of 240MHz, which is 6x better than Pentium 4 (2.54 GHz) processor and 3x better over 1.25 GHz G4 processor. The design was done using VHDL and synthesized and routed using Xilinx tools and arithmetic units. The author relied totally on the Vendor tools on the design implementation which provided adequate algorithms for trade-offs between throughput, area, and latency.

LU Decomposition on an FPGA

As FPGA built more hardware resources like fixed-point adders, multipliers, shifters, memory, priority encoders, different IP cells which allowed synthesis tools to easily design floating point units with the desired constraints on area, latency and throughput the research has moved

away from building hardware floating point units to effectively use what's available in the FPGAs [15]. In this research the author proposed architecture for LU Decomposition on FPGAs where They were able to achieve 23x improvement in total computation time with GPP. The Systems used Xilinx ISE 5.2i and the FPGA is Virtex-IIPro XC2VO125.

More Complex HPC FP Applications Using FPGA

Another known floating-point arithmetic application is Lennard-Jones Potentials and Forces for molecular dynamics simulations designed in an FPGA accelerator [16]. The author refers to previous work on this project done on both Altera and Xilinx FPGA and found Altera performing better. The application requires division and square root in addition to additions and multiplications, unlike previous work where only multiplications and additions were dominant. The implementation was done on Xilinx Virtex-II XC2VP125-7 Pro and achieved 3.9 GFlops. The acceleration of this system was compared against GPP processors like AMD Athlon XP 1700, Athlon XP 3200, and Intel Itanium-2 900MHZ and 1500 Mhz. The result shows the FPGA based system is faster than both. Another interesting point, is that this application was done previously on an ASIC, called MD-GRAPE chip. The chip uses 1024-piece fourth order polynomial to approximate the calculations of force and potentials and accelerate it. The host processor did the rest of the work. In the FPGA, floating-point equation were used directly and in higher precision than the ASIC. It is important to note here that, in scientific applications, the possibility of changing the hardware implementation at later stages is crucial, since the scientific algorithms are themselves subject to change and modification.

FPGA COTS System

Due to the increase in densities and capabilities of FPGAs many commercial companies started building compute-intensive hardware acceleration systems based on FPGAs. Ling and Viktor [14] completed analysis on matrix multiplication design tradeoffs using commercial acceleration systems XD1 from Cray [28] . They are able to achieve 2.06 GFlops using XD1. Other commercial systems were suggested like SRC MAPstation from [27] and Starbridge Hypercomputer [29]. An acceleration system consists of multiple FPGAs and a GPP that shares a memory system. The design trade-offs that were explored in this double precision (64-bit) work related to the FPGA resources like memory size, logic cells, slices and the impact on area and latency, and different implementation algorithms like [30] and [31].

Recent research and development

Table 3.1: Some of the Tool Entry for FPGA Designs

JHDL	Java based circuit design environment
Matlab environment	An environment where Matlab code is translated to C code and VHDL
Handel -C	An extension to C
System - C	An extension to C
Xilinx System generator for Mathwork simulink	Xilinx development tool
Perl generator	Developed by [19]

An improvement of the design of matrix multiplication implementation of [14] was done on [19] on Xilinx Virtex-II Pro. The authors used an algorithm proposed by the FPGA vendor for implementation. It is clear that FPGA vendors tackled the problematic issues of floating-

point by providing better primitive units like 18-bits multipliers to accommodate better performance of area and speed.

Another important item that increased the efficiency and ease of use FPGA is tool development. Besides the known hardware languages like VHDL and Verilog which require knowledge and experience and consume a lot of design time, G. Leinhart [20] summarized either Gui interface like tools or software based tools available (see Table 3.1). The author also developed a tool based on Perl software which he used to demonstrate the known molecular dynamics simulation application. Of course, nowadays, even easier and more flexible tools are available from the FPGA vendors even after 3 years from this paper.

Algorithmic Research

Another vibrant research activites explore new paradigma to optomize the hardware implemnatation of floating-point operations. For example, Nachiket Kapre [22] suggested a new algorithm for parallelizing double precision floating-point accumulations and was able to achieve 6x speedup on Viretx-4 LX160 over traditional algorithm of summation. The algorithm demonstrates how to convert the non-associatively of the IEEE floating-point which cause deep pipelining to parallel accumulation while maintaining the sequential IEEE accumulation semantics. Sandeep K Venishetti [23] provided a comparison of a multiplication algorithm that beats Xilinx own LogicCore IP speed performance but consumes more area. He demonstrated his algorithm on Xilinx Viretx-4 as well. The algorithm looks at the architecture of the logic resources in the FPGA and demonstrates the optimal method of multiplication. The author is considering research on other arithmetic operation improvements on Xilinx platform. In [24] Ronald Scrofano tried to solve the issue of increase of the area as floating-point cores are deeply pipelined as a result of increase of frequency. He presented a new architecture to efficiently

reduce the number of floating-point arithmetic cores to save on area by utilizing more control logic and serialize rather than parallelize the computations. In [25] Michael J. Beauchamp suggests to have new primitives like multiply-add untis, variable length shifters and 4:1 multiplexes. These additional primitives suggestions follow the same line of thoughts of other contributors who suggested primitives like fast carry-chains, cascade chains, embedded multipliers, etc. The author even suggested having a new FPGA architecture to aim for floating point, following the Xilinx model of FPGAs targeted specifically for signal processing application.

In summary the collaboration between FPGA vendors and academia research to make floating-point arithmetic operation available to accelerate large number of scientific computational intensive is on-going. While earlier papers written to entrain the idea of acceleration in FPGA, current papers are still working on demonstrating area, latency, and speedup improvements. As FPGA vendors provide a rich library of different primitives that provide fine-grain floating-point designs with performance based on synthesis constraints (Area, Speed), many industrial companies including the vendors themselves, offer IP cores for coarse-grain design that can be instantiated in FPGAs, which provide superior performances.

3.2 Acceleration Systems

An application written in C/C++ code for example can run at different speeds using different CPUs, what makes an application run faster on processor versus the others is either the frequency of the processor or the dedicated architecture of the processor to solve a specific computation problem. CPU frequencies have already reached a limit of 3.5 GHZ from which moving higher is not feasible in the near future. Hence the architecture is the main differentiator of the speed of an application. Having a dedicated hardware for floating-point arithmetic

in CPUs is not a viable option due to mainly power and area requirement as we saw in chapter 3. The two main solutions is a Hybrid Computing System (HCS) [32] see Fig. 3.1 or a Configurable System-on-a-Chip CSoC [33] [39].

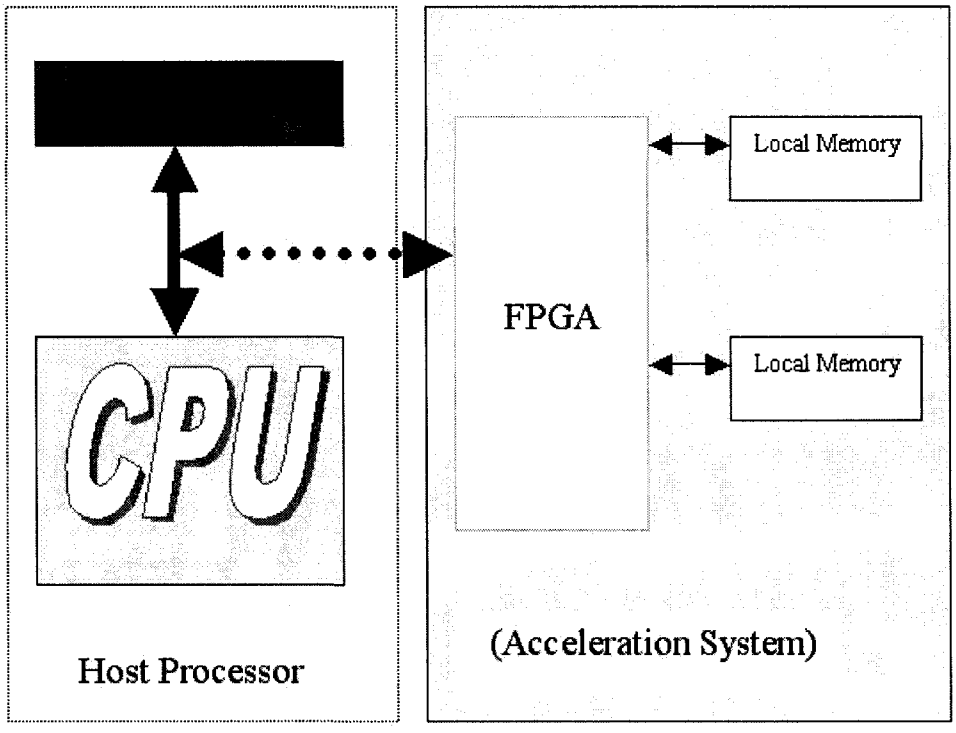

Figure 3.1: Hybrid Computing System

In HCS a host CPU shares a common memory with another application specific hardware system that executes the time consuming application off loading the main host processor.

A typical HCS Fig. 3.1 is built using a host processor connected to FPGA or FPGA modules depending on the size and the architecture of the application and the server used. The bus connecting the processor to the system has a great important on the speed on communi-

cation between the FPGA and the host processor. AMD [38] and Intel [79] has found great interest in acceleration systems and has been contentiously exploring and developing new bus architectures to interconnect with other co-processing systems [37]. In the late 1990, data was clocked at twice the bus clock (Double-pumped), todays Intel's Xeon processors FSBs are Quad-Pumped bring the data at 4 times the bus clock. The top theoretical data rate on FSBs is 1.6GT/s. AMD has developed the HT -Hyper Transport DDR (Double Data Rate) bus versions 1.0, 2.0, 3.0 and 3.1 which run on speeds from 200MHz to 3.2 GHz. The bus width is auto negotiated bus width up to 32-bit LVDS links. The full width, full speed has bandwidth of 25GB/S (3.2 GHz/Link * 2 bits/Hz * 32 links * 1 Byte/8 bits). COTS acceleration systems are available from many vendors [32] [37] [28] [29] based on Intel or AMD processors. Here are examples of these systems Figures 3.2 3.3 from [32] .

Obviously the transfer times and communication between CPU and the FPGA acceleration hardware system can greatly affect the acceleration performance. That is why most systems vendors would quote the maximum floating-point operations that can be done on the FPGA system per second versus the number of operation that can be achieved on the CPU for comparison purposes to show acceleration ratios, but not the actual acceleration in time.

A CSoC has one or more processors integrated with the application specific hardware system and memory inside an FPGA. The dramatic increase of FPGA logic resources in the last decade and the new tools developed by EDA (Electronic Design Automation) for C code synthesis to hardware implementation made these systems widely available. FPGA vendors provide tools to build system on FPGA and provide C-to-Hardware (VHDL or Verilog) code. This option is mostly used by hardware experts who would know how to build a system on an FPGA unlike the COTS HCS systems which target software people. Also a combination between CSoC and HCS can be custom designed for certain application where the acceleration

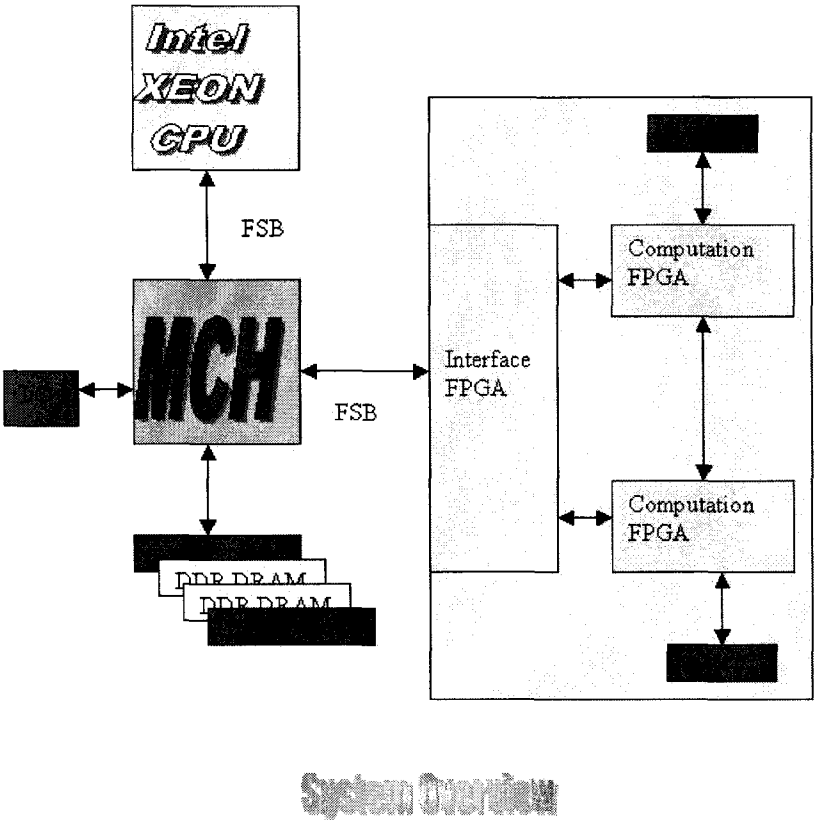

Figure 3.2: Intel Processor Based Acceleration System

FPGA system has a processor that executes some of the computations and communicate via interrupts and a shared memory with the host processor [74] [78].

Purely sequential software can't benefit from such HCS or CSoC unless the tasks are distributed between the CPU and the application specific hardware system in parallel fashion and in an effort to maximize performance. Also the software should be written in such away that hardware implementation could easily be implemented [33] [34] [35].

Most application developers are software people who would rather choose a system where they don't have to deal with the hardware implementation [33]. A system that reads C and

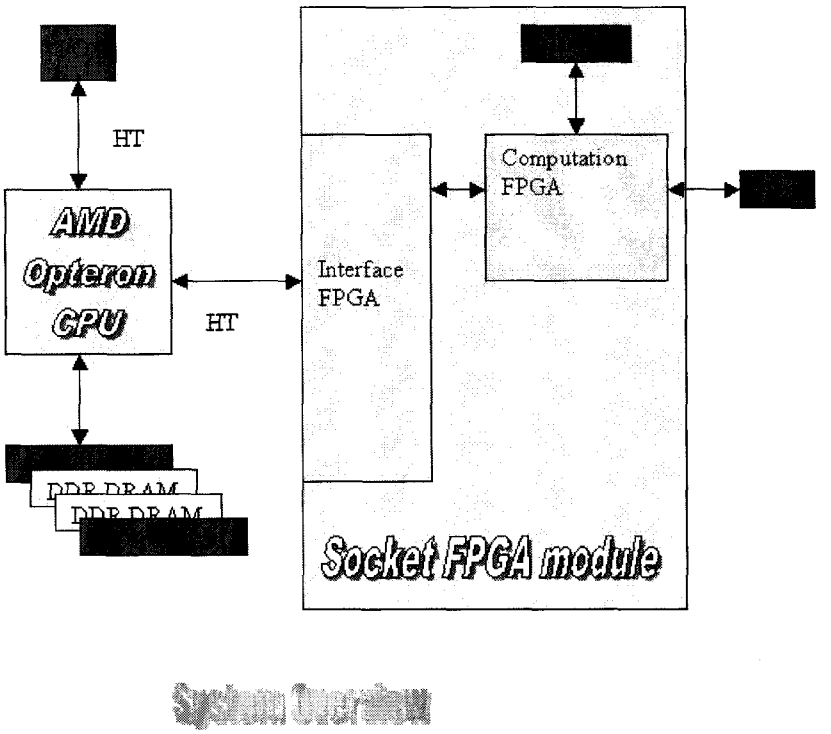

Figure 3.3: AMD Processor Based Acceleration System

produces the hardware acceleration is required on its own. The advantages of such approach that software people do not need to know hardware to build a system. The disadvantage is that one needs to learn the commercial company tool and instructions on how to refine one's code to get the desired acceleration for the specific code and one might loose on the opportunity of tweaking the hardware for better performance.

Chapter 4

Proposed Application

This chapter explains the particular domain in which FPGA acceleration can be very efficient in handling computation-intensive task. The chapter reviews briefly the implementation steps for the method proposed in [2] to demonstrate the man computational blocks and detect which of these blocks can be implemented efficiently on an FPGA-based computational platform.

4.1 Introduction

The problem of numerical solution to ordinary differential equations (ODE) arises in various domains and disciplines. For example, in the transient analysis simulation of electronics circuits, a Computer-Aided Design (CAD) tool formulates automatically a system of differential equations in the time-domain. The task of simulating the transient response of the circuit is basically carried out through applying a numerical algorithm to approximate the solution of the differential equations at discrete time points.

There are several ways to characterize a given ODE solver. First, there is the order of the solver which describes the accuracy which a solution of the ODE is being approximated. Another, important factor is the stability of the solver which describes how small errors can be amplified.

A good solver is said to be of high order with absolute stability (or A-stability).

The two factors have always been in conflict, where A-stability mandates lower order, and high-order necessitates sacrificing the A-stability. The reason for that conflict was shown to be the result of some theoretical barriers inherent in the methods used in the numerical solution of ODE [3].

Nevertheless, a recent class of methods has been published recently [3] and was shown to be free from those barriers. Implementation of this method to circuit simulation has resulted in a significant speedup to transient analysis. This thesis considered the Implementation details for these methods with the objective of detecting the bottlenecks in computations. The goal is to investigate whether, if any, some of these bottlenecks can be addressed via utilizing an FPGA-based computation platform as opposed to the conventional platforms .

As will be shown in the remainder of this chapter, the new proposed methods rely on the computation of high-order derivatives of the nonlinear terms inside the ODE. Upon a closer examination of the hierarchical formula used in computing high-order-derivative, it will be demonstrated that such a computation is better suited to an FPGA-based computation platform as opposed to conventional computational platform built upon a general processor architecture. This chapter will cover the theoretical fundamentals of the ODE solver, while the next chapter will present the simulation results for a sample of the algorithm run on an FPGA platform.

4.2 Formulation of Differential Equations [2] [3]

In the framework of transient simulation of electronic circuits, a general purpose nonlinear circuit can be represented in the time-domain by a set or a system of Differential Algebraic Equations (DAE) that takes the following form:

$$Cdx(t)/dt + Gx(t) + f(x(t)) = b(t)$$

- C and G are matrices in $R^{N\times N}$ that represent the memory (e.g. capacitors) and memoryless (e.g. resistors) elements in the circuit

- $f(x(t))$ is a vector in R^N whose components are nonlinear functions which describe the nonlinear elements in the circuit

- $b(t)$ is a vector in R^N whose components represent the contribution of the independent sources of stimuli, e.g. current or voltage sources.

- $x(t)$ is a vector of unknowns in R^N, whose components represent the unknown waveforms, the computation of which represents the utlimate goal of a given DAE solver, and.

- N is the number of these variables .

The formulation of the above equations is a standard step common to all modern circuit simulators. This step is carried out automatically using one of several well known approaches. Typically, the Modified Nodal Approach(MNA) is the most commonly used in modern circuit simulators [2].

4.3 Numerical Solution of DAE

A numerical solver of DAE seeks to approximate the continuous solution of the DAE through the process of discrete approximation. More precisely, the numerical solver generates a sequence of vectors $x_R \in R^N$ that approximates $x(t)$ at discrete time points $t = t_n, n = 0, 1, 2, ...$ Hence,

$$x_n = x(t_n) + \delta_n$$

where δ_n represents the error incurred by the approximation process. The actual value of the error depends on the size of the time step between two successive time points, $h = t_n - t_{n-1}$ as well as different other factors. Typically, this error is estimated using several complicated techniques, and step size is controlled to maintain that error within well-defined boundaries.

Another important criteria, besides accuracy, that characterizes a given method aiming at the numerical solution of DAE system is the stability of the method. Stability refers to the ability of a certain method to "dampen" the error from one step over successive steps.

For a method to be practical, it must be stable whenever the underlying circuit is stable. That concept of stability is known in the math literature as the A-stability.

The accuracy of the method is described by the order, where a method of order p has an error proportional to h^{p+1}. It is obvious that a method with high order is a method in which the error vanishes quickly with the reduction of the step size.

Thus a method for solving DAE numerically is most useful if it enjoys a high-order, and is A-stable. Nevertheless, the two characteristics have always been is conflict, where methods suitable for circuit simulation that are A-stable, can not have an order higher than 2, i.e. ($p \geq 2$) Recently, however, a new class of methods for circuit simulation was proposed in [2]. It was proved theoretically that this class of methods does not have the problem of the conflict between stability and high-order. The next section presents a brief review of this method to illustrate the main computational blocks and help identify those blocks that are suitable for implementation on an FPGA computational platform.

4.4 Review of the Numerical Solution of DAE method

In the method presented in [3] for the numerical solution of DAE, the approximation for $x(t)$ at $t = t_{n+1}$, or x_{n+1} is obtained via an approximation formula known Obreshkov method.

$$\sum_{i=0}^{k} \alpha_{i,k}(-1)^i h^i x_{n+1}^{(i)} = \sum_{i=0}^{k} \alpha_{i,k} h^i x_n^{(i)}$$

where

$$\alpha_{i,k} = (2k-i)!\frac{k!}{i!(k-i)!}$$

and $x_k^{(i)}$ represents the approximation to $\frac{d^i}{dt^i}x(t)$ at $t = t_k$ with the fact $x_k^{(0)}$ is the approximation to $x(t)$ at $t = t_k$.

Now to obtain x_{n+1}, one first assumes that $x_n^{(i)}$, the approximations obtained at the previous time instant t_n, are available. Next, it can be shown [2]that upon substituting from (4.1) into (4.2), we will have the following system of nonlinear equations:

$$\widetilde{C}\xi_{n+1} + \widetilde{G}\xi_{n+1} + \widetilde{\rho} = \widetilde{b}_{n+1} \tag{4.1}$$

where

$$\widetilde{C} = \frac{1}{h} \times \begin{bmatrix} 0 & C & \cdots & & 0 \\ 0 & 0 & C & \cdots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & \cdots & 0 & C \\ 0 & 0 & & \cdots & 0 \end{bmatrix} \tag{4.2}$$

$$\widetilde{G} = \begin{bmatrix} G & 0 & \cdots & & 0 \\ 0 & G & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & \cdots & G & 0 \\ \alpha_{0,k}I & -\alpha_{1,k}I & 0 & \cdots & (-1)^k\alpha_{k,k}I \end{bmatrix} \tag{4.3}$$

$$\widetilde{b}_{n+1} = \begin{bmatrix} b_{n+1}^{(0)\,T} & hb_{n+1}^{(1)\,T} & \cdots & h^{k-1}b_{n+1}^{(k-1)^T}\sum_{j=0}^{k-p}\beta_{j,k-p}h^j x_n^{(j)^T} \end{bmatrix}^T \tag{4.4}$$

$$\widetilde{\rho}_{n+1} = \begin{bmatrix} \rho_{n+1}^{(0)\,T} & \rho_{n+1}^{(1)\,T} & \cdots & \rho_{n+1}^{(k-1)^T} \end{bmatrix}^T \tag{4.5}$$

$$\rho_n^{(i)} = h^i \frac{d^i}{dt^i} f(x(t))\Big|_{t=t_{n+1}}. \tag{4.6}$$

and h is the size of the time step, i.e.

$$h = t_{n+1} - t_n. \tag{4.7}$$

while the set of unknowns in the system, ξ_{n+1} are given by

$$\widetilde{\xi}_{n+1} = \begin{bmatrix} x_{n+1}^{(0)\ T} & x_{n+1}^{(1)\ T} & \cdots & h^k x_{n+1}^{(k)\ T} \end{bmatrix}^T \tag{4.8}$$

The task of computing x_{n+1}, i.e. the approximation to $x(t)$ at $t = t_{n+1}$, is then reduced to the task of solving the system (4.1) for ξ_{n+1}. Typically, this system is nonlinear system of equations whose solution requires using some iterative techniques such as the Newton-Raphson approach. In the Newton-Raphson method, one usually starts with an initial guess, say $\xi_{n+1}^{(0)}$ and iterates through the following procedure

$$\xi_{n+1}^{(i)} = J^{-1}(\xi_{n+1}^{(i-1)})\phi(\xi_{n+1}^{(i-1)})$$

where

$$\phi(\xi_{n+1}^{(i-1)}) = \widetilde{G}\xi_{n+1}^{(i-1)} + \widetilde{C}\xi_{n+1}^{(i-1)} + \widetilde{\rho}\Big|_{\xi_{n+1}=\xi_{n+1}^{(i-1)}} - b_{n+1}$$

$$J(\xi_{n+1}^{(i-1)}) = \widetilde{G} + \widetilde{C} + \frac{\partial \widetilde{\rho}}{\partial \xi_{n+1}}\Big|_{\xi_{n+1}=\xi_{n+1}^{(i-1)}} i = 1, 2, 3..$$

The solution of the system (4.1) is reached when the above iterative procedure converges, that is when

$$\xi_{n+1}^{(i)} \approx \xi_{n+1}^{(i-1)}$$

4.4.1 Analysis of Computations

The above brief description of the algorithm has demonstrated that the main computational blocks can be summarized as follows

1. Computation of the vector $\phi(\xi_{n+1}^{(i-1)})$ given a certain guess vector $\xi_{n+1}^{(i-1)}$
2. Computation of the Jacobian matrix $J(\xi_{n+1}^{(i-1)})$ given a trial solution vector
3. Inversion of the Jacobian matrix J through applying some suitable factorization techniques

The above three computational blocks have been considered for application on an FPGA platform. Typically, the implementation of matrix factorization on an FPGA platform is far less efficient than implementation on a conventional platform, because of the growth of memory requirement due to the fillings that occur in the factorization process.
We therefore, focus on examining the other two blocks to investigate whether their application on an FPGA can lead to faster execution times than its implementation on the conventional platform. The following section illustrates the process of computing $\widetilde{\rho}_{n+1}$

4.5 Computing $\widetilde{\rho}_{n+1}$

The basic approach used here relies on representing each nonlinear expression entry in $f(x(t))$ as rooted tree with sub-expression as children tree nodes, where a sub-expression represents any linear or nonlinear functions such as the exponential or the logarithmic function. The leaf nodes in such tree are nodes "without descendants or children" which represent sub-expression arising from MNA variables in the nonlinear function (denoted by the wave terms) or simple constant terms. For example, consider that the q^{th} entry in $f(x(t))$ is due to a diode current, $f_q(x(t)) = I_0(\exp((x_q(t) - x_p(t))/V_T - 1)$, then such expression is represented by the

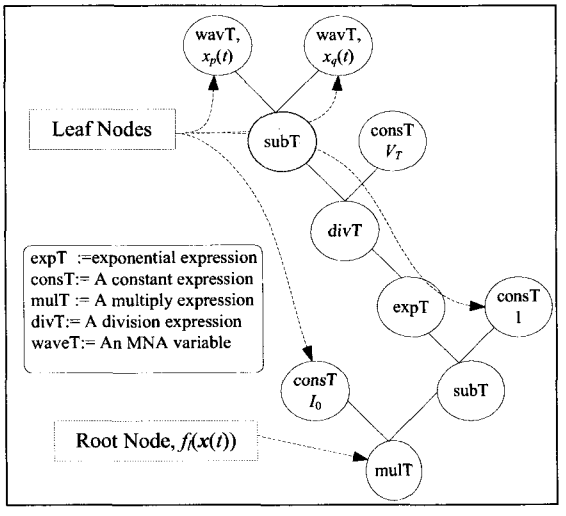

Figure 4.1: A rooted tree representing the diode current [3]

rooted tree shown in Fig 4.1. Computing the entries in $\widetilde{\rho}_{n+1}$ corresponding to $f_q(x(t))$ requires computing the i-th order derivative $(i = 1, ..., k-1)$ of $f_q(x(t))$, which represents the root of the tree. The key idea used to carry out this computation is based on the fact that, the i-th order that derivative for any sub-expression can be obtained provided that knowledge of its lower order derivatives and the j -th order derivatives $(j \leq i)$ of its children sub-expression are readily available. This fact can demonstrated by assuming a simple exponential expression, $f_1(x(t)) = \exp(x(t))$, at the l -th entry in the $f(x(t))$. Expanding both $x(t)$ and $f_1(x(t)) = \sum_{i=0} v_i \tau^i$, with $u_i = 1/i!h^i d^i x(t)/dt^i$, $v_i = 1/i!h^i d^i f_1(x(t))/dt^i$ and $\tau = (t - t_{n+1})/h$, it can be shown that:

$$v_i = \frac{1}{i} \sum_{i=0}^{i-1} (i-m) \underbrace{v_m}_{\text{parent derivate}} \underbrace{u_{i-m}}_{\text{child derivate}}$$

similar recursive formulas can be derived for other types of sub-expressions as shown in Table 4.1

Table 4.1: Some Formulation for the Derivatives of Simple Functions [2]

Equation	$y = \sum_{i=0} c_i \alpha^i, f = \sum_{i=0} d_i \alpha^i, z = \sum_{i=0} e_i \alpha^i$
$f = \exp(y)$	$d_0 = \exp(c_0)$ $d_n = \frac{1}{n} \sum_{i=0}^{n-1} d_i c_{n-i}(n-i)$
$f = \log(y)$	$d_0 = \log(c_0)$ $d_n = (\frac{1}{c_0})(c_m - \sum_{i=1}^{n-1} d_{n-i} c_i((n-i)/n))$
$f = y^p$	$d_n = (p d_0 c_n n + \sum_{i=1}^{n-1} c_i d_{n-i}(i(p+1) - n))/n c_0$
$f = y + z$	$d_n = c_n + e_n$
$f = y - z$	$d_n = c_n - e_n$
$f = yz$	$d_n = \sum_{i=0}^{n} c_i e_{n-1}$
$f = y/z$	$d_n = (c_n - \sum_{i=0}^{n-1} d_i e_{n-i}/e_0)$

4.6 Computing $\frac{\partial \widetilde{\rho}_{n+1}}{\partial \xi_{n+1}}$

Computing the Jacobian matrix involves computing the partial derivative of $f^i(x(t))$ w.r.t. $x^i(t)|_{t=t_{n+1}}$ and for $i = 1, ..., k-1$ and $j = 1, ..., k$. Such computation can also be demonstrated by reconsidering the simple exponential function used in the past subsection and presumed at the l -th entry in the $f(x(t))$. It can be seen that elements in the Jacobian matrix $\partial \widetilde{\rho}_{n+1}/\partial \xi_{n+1}$ corresponding to $f_1(x(t))$ can be obtained by using the derived recursive relation for the exponential function

$$\frac{\partial v_i}{\partial u_j} = \frac{1}{i}(\sum_{m=0}^{i-1}(i-m)\frac{\partial v_m}{\partial u_j} u_{i-m} + v_{i-j})$$

4.7 Discussion

The above analysis clearly demonstrates the major bulk of computations involved in obtaining $\widetilde{\rho_{n+1}}$ requires addition, multiplication, and accumulation. In certain cases such as expressions

involving the power function, or the division operation, a division by a scalar integer or a floating point is needed.

This thesis argues that configuring a hardware and dedicating it to compute those derivatives through implementing the recursive formulaes of Table 4.1 can be more efficiently carried out compared to implementation on conventional platform with a central processing unit.

To illustrate this idea farther, we consider the exponential function as it is the most commonly used in modeling nonlinear circuits. Thus, we could assume that $f(x(t)) = \exp(x(t))$. The goal here is to compute b_i, $i = 1, 2, ...$, which are the Taylor series terms of $f(x(t))$ when expanded around t. The formula used to obtain b_i requires the knowledge of a_i, which are the Taylor series terms for $f(x(t))$, b_j $(j < i)$. More specifically, The formula for b_i are given as shown next.

$b_1 = b_0 * a_1;$

$b_2 = (b_0 * a_2 * 2 + b_1 * a_1)/2;$

$b_3 = (b_0 * a_3 * 3 + b_1 * a_2 * 2 + b_0 * a_1)/3;$

$b_4 = (b_0 * a_4 * 4 + b_1 * a_3 * 3 + b_2 * a_2 * 2 + b_3 * a_1)/4;$

$b_5 = (b_0 * a_5 * 5 + b_1 * a_4 * 4 + b_2 * a_3 * 3 + b_3 * a_2 * 2 + b_4 * a_1)/5;$

$b_6 = (b_0 * a_6 * 6 + b_1 * a_5 * 5 + b_2 * a_4 * 4 + b_3 * a_3 * 3 + b_4 * a_2 * 2 + b_5 * a_1)/6;$

The vales for a_i are available as components in ξ_{n+1}, which represents a trial vector at the beginning of the iteration process presented earlier, or obtained as the solution of the matrix inversion performed at each step. Computation of b_i proceeds incrementally starting with $i = 1$ and up to the desired order required by the solver or the user.

Chapter 5

Results

5.1 Introduction

For the simulation environment, we used Altera Quartus II development software version 8.0, NIOSII EDS version 8.0, and JTAG download cable (USB-BlasterTM) to download the bit-stream. For the development of the hardware, we used Altera evaluation board with StartixII FPGA version 2S60ES. The development board provides two methods to configure the FPGA (a) Using Quartus II software running on a host computer, a designer configures the device directly through an Altera download cable connected to the FPGA JTAG header. (b) When the power is applied to the board a configuration device configures the FPGA from a flash memory. In our case it was easier to use the first method.

In the evaluation board we built a system of a processor and a floating-point hardware unit (FPU). The FPU can be used by the processor to execute the floating point operations. This system will allow us to compare the simulation results obtained from running the application on the processor without the use of the FPU, with the results obtained from running the same application on the same processor but with the FPU.

The FPU hardware implementation is based on NIOSII custom instructions which were written in C-code. The NIOSII IDE compiler uses the custom instructions and the ANSI C math library for any floating-point operation when compiling the target hardware. NIOSII employs RISC architecture, which can be expanded to include custom instruction. The NOISII has a standard interface, see Figures 5.1 5.3 for adding custom instruction in parallel with the arithmetic logic unit [4] .

These custom instructions implement single precision floating-point arithmetic operations in C/C++ application program. The instructions include addition, subtraction, multiplication, and division. However, the division operation consumes a large area size. So, we made a setup to allow the inclusion or exclusion of the custom division hardware to be able compare trade-off between area and acceleration for this specific operation.

In the next sections we describe the precision of the application, a description of the simulation platform , and the simulation results.

5.2 Choice of Precision

As we have shown in the previous chapters, floating-point operation can be done with different word lengths, for example single and double precision. The choice, of course, depends on the application precision requirement and number range. In our implementation, however, we faced a logistical impediment since Altera NIOSII processor is a 32-bit architecture and it only supports single precision application. In order to see if single precision is adequate enough to our application accuracy, we wrote a short C program that runs once with single precision and once with double precision on AMD Athlon 64X2 Dual Core Processor followed by compari-

son of the results. Table 5.1 shows the output of this program for the first six derivatives. The results show that there is no significant loss of accuracy using single precision. Therefore, we decided to proceed and use NIOSII with Altera FPGA with single precision .

Table 5.1: Comparison Results Between Single and Double Precision

Precision mode	Input parameter	Output Parameter Processor Calculated parameter
Double :	$a[0] = -0.668137$	$b[0] = 0.512663$
Single :	$a[0] = -0.668137$	$b[0] = 0.512663$
Double :	$a[1] = -0.004842$	$b[1] = -0.002482$
Single :	$a[1] = -0.004842$	$b[1] = -0.002482$
Double :	$a[2] = 0.001521$	$b[2] = 0.000786$
Single :	$a[2] = 0.001521$	$b[2] = 0.000786$
Double :	$a[4] = 0.000082$	$b[4] = 0.000044$
Single :	$a[4] = 0.000082$	$b[4] = 0.000044$
Double :	$a[5] = 0.000000$	$b[5] = -0.000001$
Single :	$a[5] = 0.000000$	$b[5] = -0.000001$
Double :	$a[6] = 0.000000$	$b[6] = 0.000000$
Single :	$a[6] = 0.000000$	$b[6] = 0.000000$

In order not to underestimate Altera Stratix II double precision capability, it's important to note that Altera had achieved in its bench marking white paper [80] 14.25 GFLOPS for double-precision floating point using Stratix II EP2S180 version. Altera used Impluse C and API software from [36] to generate the hardware, implement the multipliers, and the interface to write/read results to the SRAM. They're able to have 39 adders and 39 multipliers to fit at a speed of 200 MHz. In this work however the processor NIOSII is not part of this benchmark,

hence the acceleration factor is not available.

5.3 Description of the Simulation Platform

5.3.1 The Implementation Platform

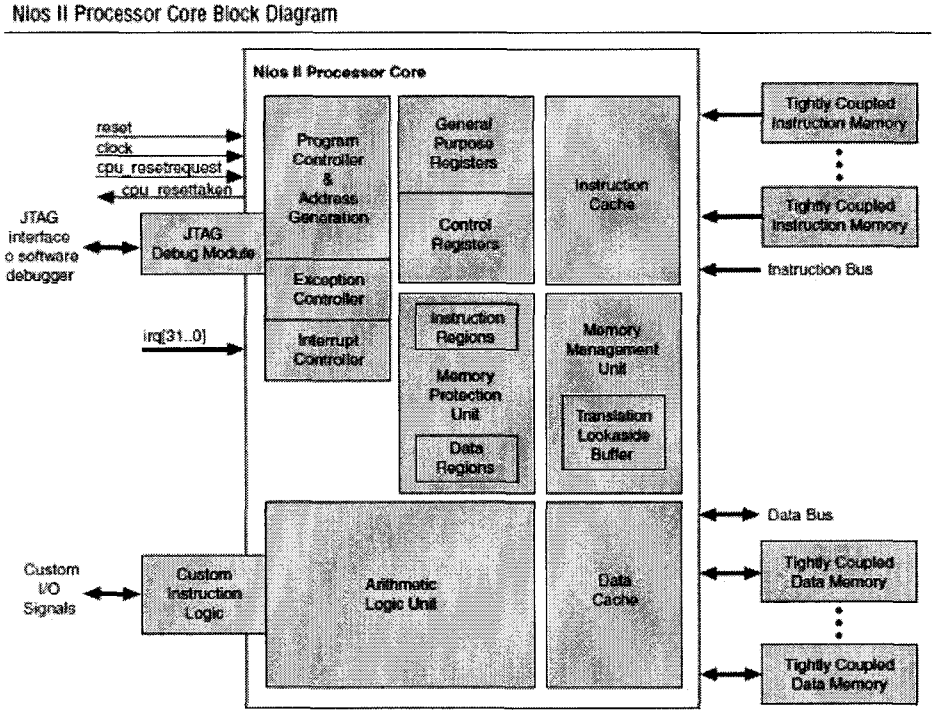

Figure 5.1: NOISII Block Diagram [4]

Stratix II logic structure is built from basic logic units known as adaptive logic modules (ALMs). Each ALM contains a variety of (look-up table) LUT-based resources, two full adders, carry-chain segments, two flipflops, and many additional logic enhancements that can be flex-

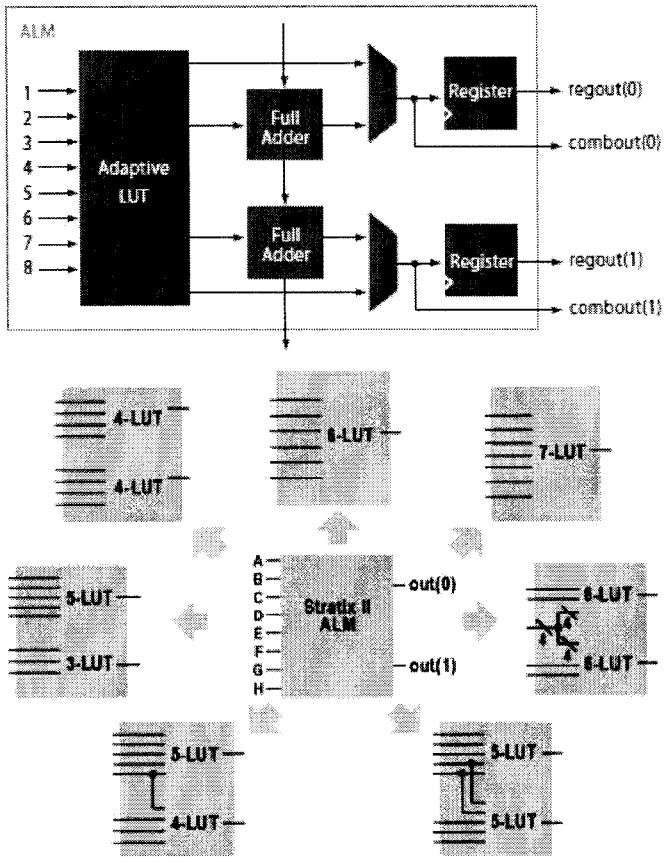

Figure 5.2: ALM architecture [4]

ibly divided into two adaptive LUTs (ALUTs). Logical functions with up to 7 inputs and complex logic-arithmetic functions can be implemented in one ALM. Figure 5.2 shows the ALM basic architecture and the different LUT configurations that a single ALM can support.

StartixII FPGA has 13500 ALM (Adaptive Logic Modules) , 1.3 million on-chip memory, 16MByte of flash memory, 2MByte of synchronous SRAM, 32MByte of SDRAM memory, and several user I/Os and interfaces [4].

Altera has developed a library of hardware functions called Megafunctions. These Megafunctions are intellectual property blocks, based on Altera specific FPGA architecture and are parameterizable; they range from simple arithmetic units such as adders and counters, to advanced PLL modules. The nature of this work mainly required by the floating point arithmetic operations such as addition, subtraction, multiplication, and division.

The Megafunctions are normally used using the MegaWizard, which allows the user to create the design files of the functions he needs and instantiate them in the design. This feature saves design time for some of the most commonly used functions and most likely it will be more efficient because of its awareness of the vendor logic synthesis and the device architecture. In our case, however, we have added these floating-point Megafunctions to the NIOSII CPU as custom instructions in addition to the NIOSII hard core hardware.

The following subsection presents a brief background on the addition, subtraction, multiplication, and division of the Megafunction Library, which were added to the NIOSII as custom instructions in addition to the NIOSII core hardware. The functions are abbreviated as altfp-add, altfp-sub, altfp-mult, altfp-div.

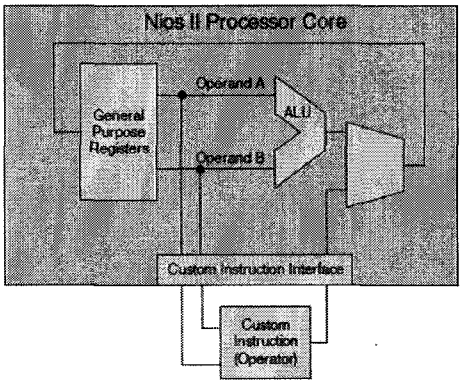

Figure 5.3: Single Custom Instruction hardware block and interface [4]

Addition and Subtraction Function altfp-add-sub

Table 5.2: Addition and Subtraction Function Resource Usage and Performance

Device	Precision	Latency	Logic Usage
StratixII	Single	11	1132 ALUT
	Double	11	1938 ALUT

The multiply function complies with IEEE-754 standard. It does not however support double extended format, and rounds only to the nearest even mode. The IEEE 754-1985 standard has four rounding modes, round to the nearest value(with a number that falls midway between two others being rounded to the nearest value with an even low order digit), round toward zero, round toward plus infinity, round toward minus infinity. altfp-add-sub supports rounding to the nearest even low-order digit only. With the round to nearest even mode, the result is rounded to the nearest floating-point number. If the result is halfway between two floating numbers, it is rounded so the the LSB becomes zero, which is even.

Table 5.2 shows the resource utilization and performance of the altfp-add-sub function. Latency, the number of clocks it takes to complete the operation, is 11 and the logic utilization is based on the number of ALUT, adaptive look-up table primitive.

Multiply function altfp-mult

Table 5.3: Multiply Function Resource Usage and Performance

Device	Precision	Latency	Logic Usage		Fmax(MHz)
			ALUT	DLR	
StratixII	Single	5	126	148	228
	Double	5	383	410	122

The multiply function complies with IEEE-754 standard. It does not however support double extended format, and rounds only to nearest even mode. The number of dedicated hard primitive multipliers in the device largely limits the maximum floating-point operations of an FPGA.

Table 5.3 shows the resource usage and performance of the multiplier.

Note the difference between double and single implementation resources is more than triple and the frequency is less than half. This is of course due to the routing congestions reported in many papers.

Division function altfp-div

Table 5.4: Division Function Resource Utilization and Performance

Device	Precision	Optimization	Latency	Logic Usage		Fmax(MHz)
				ALUT	DLR	
StratixII	Single	Area	33	1442	1854	231
		Speed	33	3379	3131	228
		Low Latency	6	198	270	123
	Double	Area	61	4661	6854	166
		Speed	61	13382	12652	188
		Low Latency	10	657	1027	124

The division function complies with IEEE-754 standard. It does not however support double extended format, and rounds only to nearest even mode. The division function consumes the most area out of all floating-point arithmetic functions. To improve this area of consumption, there are three optimization switches available, optimize for area, speed, or low-latency. Table 5.4 shows the division function resources utilization of each of these constrains. When optimizing for area, the performance of the divider is not the best but it's definitely smaller

in size. Optimization for speed option improves the timing but requires more resources. Optimization for low-latency, provides the output in a shorter time, uses less resources, but the frequency is less and produces less accurate rounding results.

Others floating point function in Megafunction library

Table 5.5: Additional MegaFunctions Dupported with Stratix III (some II) or Higher

Megafunction name	Function	Notes
altfp-sqrt	Square root	Calculates the square root of a number
altfp-exp	Exponential	Calculates the exponent of an input number
altfp-inv	Inverse	Calculates the inverse value of an input
altfp-inv-sqrt	Inverse square root	Calculates the inverse square root value of an input.
altfp-log	Natural Logarithm	Implements natural logarithm
altfp-abs	Absolute	Calculates the absolute value of an input
altfp-compare	Comparator	Compares two numbers and produces seven outputs : $A > B, A < B, A \leq B, A \geq B$, $A = B, A \neq B$ or inputs are NAN
altfp-convert	Converter	integer to floating-point,floating-point to integer,floating-point to floating-point
altfp-matrix-mult	Matrix Multiplier	Performs multiplication of two matrices. Only in single precision mode

Table 5.5 shows other useful functions available in Altera library which we have not used. They are all supported in Single, Double and Single extended, and offer the exceptions handling outputs(NaN, Underflow, Oveflow, Zero).

5.4 Simulation Results

The program implements HOD example in software without the FPU hardware, and a second run with the FPU acceleration hardware. We used a sample of the results obtained in the course of running the solver for the a parameters:

$$a_0 = 7.985732045844021 \exp -001 - 1.466710507868863 \exp +000;$$
$$a_1 = 2.182737546336707 \exp -001 - 2.231153167809049 \exp -001;$$
$$a_2 = -5.294685060820403 \exp -002 - (-5.446830824309763 \exp -002);$$
$$a_3 = -1.292328525185155 \exp -002 - (-1.264710331973320 \exp -002);$$
$$a_4 = 2.221759026111466 \exp -003 - 2.139903058330181 \exp -003;$$
$$a_5 = 1.1234 \exp -3/6;$$

where our goal is to compute the $b0$-$b6$

$$b_1 = b_0 * a_1;$$
$$b_2 = (b_0 * a_2 * 2 + b_1 * a_1)/2;$$
$$b_3 = (b_0 * a_3 * 3 + b_1 * a_2 * 2 + b_0 * a_1)/3;$$
$$b_4 = (b_0 * a_4 * 4 + b_1 * a_3 * 3 + b_2 * a_2 * 2 + b_3 * a_1)/4;$$
$$b_5 = (b_0 * a_5 * 5 + b_1 * a_4 * 4 + b_2 * a_3 * 3 + b_3 * a_2 * 2 + b_4 * a_1)/5;$$
$$b_6 = (b_0 * a_6 * 6 + b_1 * a_5 * 5 + b_2 * a_4 * 4 + b_3 * a_3 * 3 + b_4 * a_2 * 2 + b_5 * a_1)/6;$$

Tables 5.6- 5.9 report the execution times in clock cycles. These tables show the execution time using a software-based (SW) computation and compares it with the time taken by the Custom Instruction (CI) execution times. Tables 5.8 and 5.9 also demonstrate the acceleration obtained without a division hardware as well as with the division hardware.

The best acceleration for computing the b parameters that we have achieved is between

13.5x to 18.5x. The logic utilization used in this case is 34 percent of the FPGA area.

In the next subsection, we provide the summary results tables and explanations of these findings.

5.4.1 Comparison of Speed

Tables 5.6 and 5.7 summarizes the accelerations results of the NIOSII with the custom FPU instructions. These are the original run from Altera [4] performance bench marking. The program runs all arithmetic operations, add, subtract, multiply, divide with a 1000 random pair of variables. The program executes both the floating-point custom instructions and the equivalent software operation. Using the performance counter component we compare the time it takes to execute each command.

I have summarized each run and the acceleration achieved in Tables 5.8 and 5.9. We noticed that the divide hardware unit has a great acceleration factor. Without it, the b_1-b_5 computation acceleration factor is only 3x, while with the divide unit is between 13.5x to 18.5x . We also noticed that the acceleration of the b_1-b_5 computation is more than the primitive operation acceleration. That could be due to the fact that in our example we have a fixed run while in the primitive operation we have random numbers. Also could be attributed to better performance of hardware for multiple operation in one command.

Table 5.6: The Original Acceleration Results (FPU with Division Hardware)

Operation	*Time(SW) / Time (CI)*	*Acceleration*
ADD	288450/22030	13.09
Subtract	291757/22010	13.25
Multiply	219278/24000	9.13
Divide	589062/47012	12.53

Table 5.7: The Original Acceleration Results (FPU without Division Hardware)

Operation	*Time(SW) / Time (CI)*	*Acceleration*
ADD	289011/22012	13.13
Subtract	305085/22010	13.86
Multiply	237637/24000	9.90
Divide	515103/569943	0.90

5.4.2 Comparison of Area and Power

Area Report of NOISII hardware without FPU

Family Stratix II

Device EP2S60F672C5ES

Logic utilization 20 %

- Combinational ALUTs 7,328 / 48,352 (15 %)
- Dedicated logic registers 4,853 / 48,352 (10 %)

Total registers 5106

Table 5.8: The Acceleration Results (FPU without Division Hardware)

Operation	*Time(SW) / Time (CI)*	*Acceleration*
b_1	455 / 36	12.63
b_2	1859 / 869	2.14
b_3	2541 / 922	2.75
b_4	3177 / 9222	3.44
b_5	3661 / 991	3.69
b_1-b_5	9922 / 3152	3.14

Table 5.9: The Acceleration Results (FPU with the Division Hardware)

Operation	*Time(SW) / Time (CI)*	*Acceleration*
b_1	459 / 34	13.5
b_2	2218 / 121	18.33
b_3	2948 / 166	17.76
b_4	3535 / 213	16.59
b_5	3994 / 282	16.13
b_1-b_5	10544 / 750	14.06

Area Report of NOISII with FPU but excluding the Division hardware unit

Family Stratix II

Device EP2S60F672C5ES

Logic utilization 23 %

- Combinational ALUTs 8,227 / 48,352 (17 %)
- Dedicated logic registers 5,863 / 48,352 (12 %)

Total registers 6116

The area report shows 23% of logic utilization, which is 3% over the NIOSII hardware without FPU hardware. The floating point unit consumes only 3% of the FPGA logic. The total registers, which are the main power consumption factor are 20 percent more than without the FPU unit.

Area Report NOISII with FPU but including the Division hardware unit

Family Stratix II

Device EP2S60F672C5ES

Logic utilization 34 %

- Combinational ALUTs 11,744 / 48,352 (24 %)

- Dedicated logic registers 9,728 / 48,352 (20 %)

Total registers 9981

The area report below shows that the logic utilization has increased to 34% which means that the division unit alone consumed 11% of the FPGA logic resources. The total registers has increased by 50 percent due to the division custom hardware in comparison to the FPU without the division custom instruction.

5.4.3 Proposed New Hardware Architecture Custom Instructions

In this application implementation of the coefficients $b0$-$b6$ we used Altera FP MegaFunction as custom instructions where each custom instruction can take two FP numbers at each processor access and then provide the result back to the processor (see Figure 5.3). This is basically the simple method without the need to design any hardware. The alternative approach is to take all coefficients equations and synthesize them to hardware using C2H or writing the equations in HDL code and then synthesize the code with the processor. Each coefficient hardware code could be added as custom instruction and connected to the processor. This approach can achieve higher acceleration results at the cost of area. For example $b1$ will have three FP multipliers, one adder, and a divider; $b2$ will have five FP multipliers, two adders, and a divider; $b6$ will have eleven FP multipliers, five adders and a divider. This is the same approach used in COTS acceleration systems but without the processor since the COTS systems design their own communication control logic between the FPGA where the equations reside and the host processor. While this approach would produce the best acceleration results, it is not efficient in area. For example, the divider alone consumed 10% of the FPGA. Of course, we can resolve this problem by sharing the resources between the equations since we are not using the equation at the same time. To share the resources, we need to add control logic since each equation hardware is no longer an independent custom instruction hardware. Also, the number of multipliers, adders and divider will be based on the largest equation which could also be large enough and might not even fit in an FPGA.

The approach we used in this implementation capitalized on Altera solution which is not dependant on the size of the equations nor does it require any hardware alteration and provides good acceleration results. Investigating the equations mathematical formulation led me to propose minor changes to Altera MegaFunctions by adding up to four types of multipliers which

would consume less than 12% of the FPGA (since we saw in the area report section that the multiplier, adder and subtractor area is only 3%) that would achieve even higher acceleration results than we got with the current FP custom instruction. Note, for example, the calculation of $b2$ is an addition of two terms, one term is a multiplication of two FP numbers and an integer which is 2, and the other term is a multiplication of two FP numbers. Since the multiplier custom instruction takes only two FP numbers, the FP multiplication of the first term would have to be done in sequence. First, $b0 * a2$ then $TheResult * 2$. From Table 5.6 we can see that the multiplication average execution time in the processor using the custom instruction takes 22 cycles. If we can reduce the computation of this term to one multiplication then can save a lot of time. The integer number 2 representation in FP single format is just an exponent of 1, and a mantissa of 1 (upper bit, and all zeros in the lower bits). If we look at Figure 2.4 we can do this three way multiplier by just adding 1 to the addition of the exponents addition block. I have modified Figure 2.4 to illustrate the changes required in Figure 5.4. In other words, we can have a multiplier custom instruction that has the result of multiplying two FP numbers and 2 i.e. $a * b * 2$ which would require only adding 1 to the exponent addition block. Same would apply for the multiplying two FP numbers and 4 which would require only adding 2 to the exponents addition block. The mantissa multiplication block would not even change in the case of multiplying by 2 or 4, see green notes in Figure 5.4. The integer number 3 representation in FP single format is an exponent of 1 and a mantissa of 11 (1 in upper two bits and all zeros in the lower bits). In order to perform three way multiplication, we need to alter the exponents addition by adding 1 and perform three way multiplication in the mantissa multipliers. Same concept would apply for multiplying by the number 5.

Hence, the new custom instructions that I would add are fp-mult-by2, fp-mult-by3, fp-mult-by4, fp-mult-by5. Let us summarize estimated acceleration results on average in Table 5.10, based on Tables 5.9 and 5.6 results.

Table 5.10: Estimated Average Acceleration Results of New Multipliers

Operation	Original		Estimated	
	Time(SW)/(CI)	*Acceleration*	*Time(SW)/ (CI)*	*Acceleration*
b_2	2218 / 121	18.33	2218 / 99	22.6
b_3	2948 / 166	17.76	2948 / 122	24.16
b_4	3535 / 213	16.59	3535 / 147	24.04
b_5	3994 / 282	16.13	3994 / 194	20.58

We can see from the Table 5.10 that we can achieve $4 - 9$x acceleration improvement with the addition of these 4 special multipliers. The hardware alteration of the MegaFunction to accommodate this kind of change is minimal and should not change the latency of the hardware.

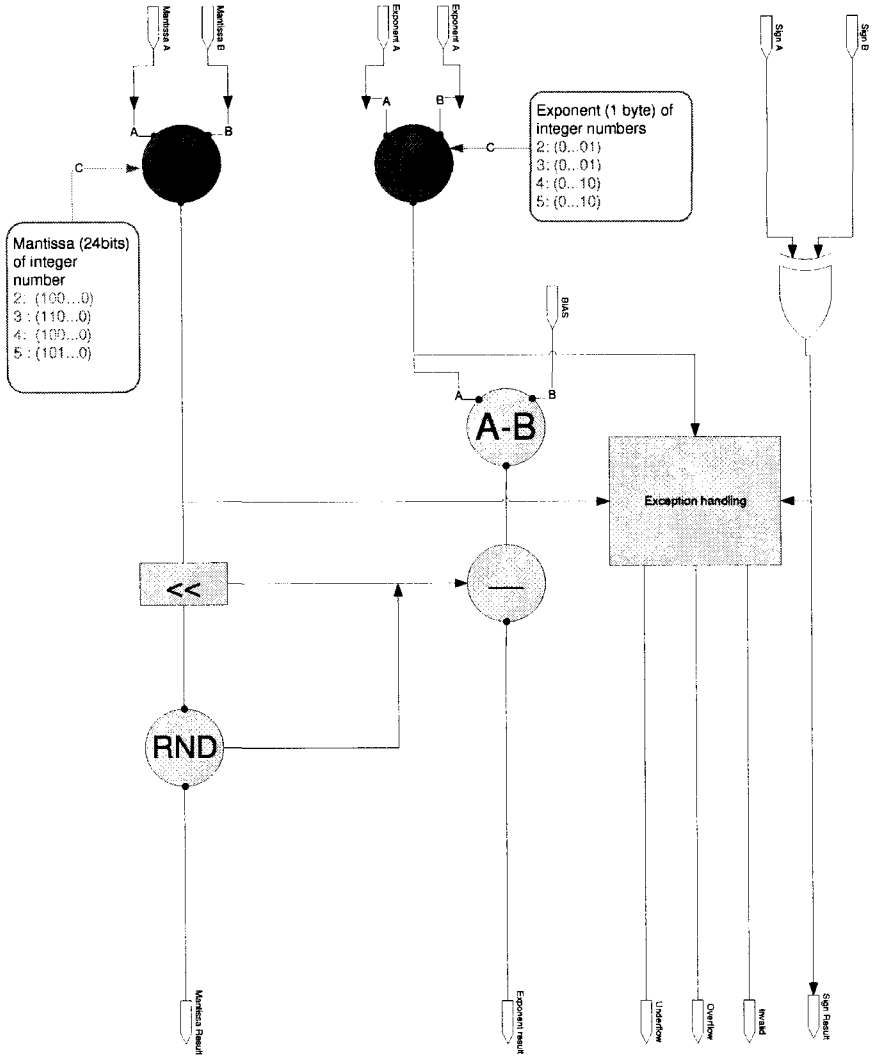

Figure 5.4: Special Multiplier Architecture As Custom Instruction

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

In this research work, we have surveyed many of the acceleration applications using FPGA. The performance advantage of FPGAs over CPUs and multicore CPUs, the FPGA continuous resource growth, the relatively little cost and low power, has attracted many applications to use it in acceleration systems. The trend will even increase as COTS acceleration systems will become more and more software developer friendly and become cost attractive.
The FPGA parallel execution has produced great acceleration results for many applications, and the reconfigurability of FPGA devices has offered easy options for experimenting trade-offs with minimal time.

In our HOD application implementation we were able to achieve 13-18x acceleration over the NIOSII processor. This achievement opens the way to research the computation acceleration of the high order derivatives employed in many application.

The area reports showed that the logic utilization of the division unit alone consumed 11% of the FPGA logic resources. This result means that we must minimize the number of division

units used in any implementation. The addition, subtraction and multiplication units together consumed only 3% of the total area. The utilization of more than one unit of addition and multiplication should increase the acceleration of the equation calculations due to more parallel computing at relatively low area price.

We have noticed that the coefficients $b2$-$b6$ of the high order derivatives have terms summation that include a multiplication of three numbers. In Section 5.4.3 we have proposed a new type of multiplier that multiplies two FP numbers with a built in multiply by an integer. By introducing this kind of multiplier, we can reduce the number of multiplications significantly. Hence speeding up the acceleration time. The proposed custom instruction multipliers are the following: fp-mult-by2, fp-mult-by3, fp-mult-by4, fp-mult-by5.

6.2 Suggestions for Future Work

We have used in this specific implementation Altera Megafunctions FP units without altering or modifying the original design. While Altera has optimized their solution for addition, subtraction, multiplication and division of two floating point numbers, we believe with minor modification of this Altera original Megafunction solution we can achieve even higher acceleration results. In the previous chapter we have proposed a new hardware architecture that should speed up the acceleration of this platform. The simulation of this experiment is left for future work.

Also, other future work can focus on developing a system with a host processor running the application software which interacts with an FPGA or FPGAs that compute the floating-point arithmetic operations offloading the host processor. We would like to experiment different

function sizes, possibly on a COTS acceleration system or custom designed acceleration system. This kind of system will allow us to quickly turn C-language code to hardware without much knowledge of the FPGA implementation details.

The system should be able to handle much larger floating point operations. For example, let us assume we have a circuit that includes a summation of multiple functions like exponent, logarithm, and a multiplication. From Table 4.1, we can build the function derivatives using the formulas created by [2]. Thus, the first step will require a software tool that builds the derivatives equations for any given function based on the formulas in the table. The result should be of similar equations to the exponent function that we did in this implementation application which amounts to floating-point multiplication, addition, and division. The next step is to implement the hardware of these equations. These equations could be large enough to require partitioning of the implementations to multiple FPGAs and could have many division operations which we need to minimize in order to save area. The problem will become how to optimize each FPGA to achieve best acceleration performance based on the area that it requires.

The other important factor when choosing a system is how will the communication between the FPGA and the host processor be achieved. The objective is to be able to run as many equations in parallel in the FPGA while the host processor is not tied waiting for a result from the FPGA.

The other consideration is how involved we would like to be in the hardware implementation. For FPGAs hardware experts, all options are wide open. For example, the developer can build an acceleration system similar to the ones discussed in Chapter 3, Figures 3.1-3.2, or build a custom one that employs an embedded processor with FPU unit. Then, define the handshaking protocol between the host processor and the embedded one, or build a custom

FPGA that has a controller that communicate with the host processor on each computation step and the transfers of data to and from FPGAs to host processor. Once the system is designed, the hardware implementation of these equations will be a minor coding problem.

On the other hand, for software developers , the process of making the hardware implementation seamless is still far fetched. However, the best idea is to utilize a COTS systems like [37] and learn their software tool, or use a software tool like the one from [36] that takes care of the conversion of the c-code to hardware implementation similar to the C2H[1] from Altera.

Another option is to join effort between software and CPU system architect researchers and the FPGA vendors to build a tool that will make the process of parallelizing some of the c-code program in an FPGA while keeping the host processor free from the bottleneck computation as seamless to the software developer as possible. Furthermore, ensuring that the communication channel between the host processor and the FPGA consumes much less time than the computation time that the host processor would have consumed for these computations without the FPGA. This will open many application opportunities.

[1]C-language to Hardware language conversion tool

Bibliography

[1] IEEE std 754-1985, *IEEE standard for Binary Floating-Point Arithmetic.*

[2] E. Gad, R. Khazaka, M. S. Nakhla and R. Griffith, "A circuit Reduction Technique for Finding the Steadu-State Solution of Nonlinear Circuits", *IEEE Transactions on Microwave Theory and Technique, Vol. 48, No. 12, Decemeber 2000, pp. 2389 - 2396.*

[3] E. Gad, M. Nakhla, R. Achar and Y. Zhou, "A-stable and L-stable High Order Integration Methods for Solving Stiff Differential Equations", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 28, No. 9, Jan 2009, pp. 1359 - 1372.*

[4] Altera Corporation, www.Altera.com, Literature: Stratix II Devices, March 2010.

[5] J. P. Deschmaps, G. Jean, A. Bioul and G. D. Sutter, "Synthesis of Arithmetic Circuits FPGA, ASIC and Embedded Systems.", *Wiley Interscience, Hoboken, New Jersey, 2006.*

[6] B. Fagin and C. Renard, "Field Programming Gate Arrays and Floating Point Arithmetic", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 2, No.3, September 1994, pp. 365 - 367.*

[7] L. Louca, T. A. Cook and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines, April 1996, pp. 107 - 116.*

[8] W. B. Ligon III, S. McMillan and G. Monn, "A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs", *Proceedings 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998, pp. 206 - 215.*

[9] P. Belanovic and M. Leeser, "A Library of Parameterized Floating-point Modules and Their Use", *Proceedings 12th International Conference on Field Programmable Logic and Applications, Vol. 2438, September 2002, pp. 657 - 666.*

[10] G. Lienhart, A. Kugel and R. Manner, "Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations", *Proceedings 10th Annual IEEE Symposium in Field-Programmable Custom Computing Machines (FCCM 2002), pp. 182 - 191.*

[11] X. Wang, M. Leeser and H. Yu, "A Parameterized Floating-Point Library Applied to Multispectral Image Clustering", *Proceedings 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), pp. 151 - 162.*

[12] J. Liang, R. Tessier and O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs", *Proceedings 11th Annual IEEE Symposium in Field-Programmable Custom Computing Machines (FCCM 2003), pp. 185 - 194.*

[13] G. Govindu, L. Zhuo, S. Choi and V. Prasanna, "Analysis of High Performance Floating-Point Arithmetic on FPGAs", *Proceedings 18th International Parallel and Distributed processing Symposium (IPDPS 2004), pp. 149.*

[14] L. Zhuo and V. K. Prasanna, "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems", *IEEE Transactions on Parallel and Distributed Systems, April 2007, Vol. 18, No. 4, pp. 433 - 448.*

[15] V. Daga, G. Govindu, V. Prasanna, S. Gangadharpalli and V. Sridhar, "Efficient Floating-point based Block LU Decomposition on FPGAs", *Proceedings 11th Reconfigurable Architectures Workshop, New Mexico, USA, April 2004.*

[16] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware", *Proceedings International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA), June 2004, pp. 284 - 290.*

[17] K. Underwood, "FPGA vs. CPUs: Trends in Peak Floating-Point Performance", *Proceedings ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, 2004, pp. 171 - 180.*

[18] K. Underwood and K. S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance", *Proceedings 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004), pp. 219 - 228.*

[19] Y. Dou, S. Vassiliadis, G. K. Kuzmanov and G. N. Gaydadjiev, "64-bit Floating- Point FPGA Matrix Multiplication", *Proceedings ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA 2005), pp. 86 - 95.*

[20] G. Lienhart, A. Kugel and R. Manner, "Rapid Development of High Performance Floating-Point Pipelines for Scientific Simulations", *Proceedings Symposium Parallel and Distributed Processing (IPDPS 2006), pp. 8.*

[21] M. Haselman, M. Beauchamp, K. Underwood and K. Hemmert, "A Comparison of Floating and Logarithmic Number Systems for FPGAs", *Proceedings 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), pp. 181 - 190.*

[22] N. Kapre and A. Dehon, "Optimistic Parallelization of Floating-Point Accumulation", *Proceedings 18th IEEE Symposium on Computer Arithmetic (ARITH 2007), pp. 205 - 216.*

[23] S. K. Benishetti and A. Akoglu, "A Highly Parallel FPGA Based IEEE-754 Compliant Double-Precision Binary Floating-Point Multiplication Algorithm", *Proceedings of International Conference on Field-Programmable Technology (ICFPT 2007), pp. 145 - 152.*

[24] R. Scrofano, L. Zhuo and V. K. Prasanna, "Area Efficient Arithmetic Expression Evaluation Using Deeply Pielined Floating-Point Cores", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Feb 2008, Vol. 16, No.2, pp. 167 - 176.*

[25] M. J. Beauchamp, S. Hauck, K. D. Underwood and K. S. Hemmert, "Architectural Modifications to Enhance the Floating-Point Performance", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Feb 2008, Vol. 16, No. 2, pp. 177 - 187.*

[26] M. D. Ercegovac, T. Lang, J. M. Muller and A. Tisserand, "Reciprocation, Square Root, Inverse Square Root, and Elementary Functions Using Small Multipliers", *IEEE Transactions on Computers, Vol. 49, No. 7, July 2000, pp. 628 - 637.*

[27] SRC Computers, www.srccomp.com, March 2010.

[28] Cray Inc., www.cray.com, Supercomputing Solutions, March 2010.

[29] Starbridge Hypercomputers, www.starbridgesystems.com, March 2010.

[30] L. E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm", *PhD Dissertation, Montana State Univ, 1969.*

[31] G. C. Fox and S. W. Otto, "Matrix Algorithms on Hypercube I: Matrix multiplication" , *Proceedings Parallel Computing, Vol. 4, 1987, pp. 17 - 31.*

[32] Mitrionics AB, www.mitrionics.com, "Low Power Hybrid Computing for Efficient Software Acceleration", March 2010.

[33] Z. Guo, W. Najjar and B. Buyukkurt, "Efficient Hardware Code Generation for FPGAs", *ACM Transactions on Architecture and Code Optimization, Vol. 5, No. 1, May 2008, Article 6.*

[34] G. Stitt, F. Vahid and W. Najjar, "A Code Refinement Methodology for Performance Improved Synthesis from C", *Proceedings IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2006), pp. 716 - 723.*

[35] Z. Guo, W. Najjar and B. Buyukkurt, "Optimized Generation of Data-Path from C Codes for FPGAs", *Proceedings IEEE Conference Design, Automation and Test, Europe (DATE 2005), Vol. 1, pp. 112 - 117.*

[36] Impulse Accelerated Technologies, http://www.impulseaccelerated.com, C-to-FPGA tools, March 2010.

[37] XtremeData, Inc., http://www.xtremedata.com, March 2010.

[38] Advanced Micro Devices, www.amd.com, March 2010.

[39] A. Mitra, Z. Guo, A. Banerjee and W. Najjar, "Dynamic Co-Processor Architecture for Software Acceleration on CSoCs", *Proceedings IEEE International Conference on Computer Design (ICCD 2006), pp. 127 - 133.*

[40] S. Paschalakis and P. Lee, "Double Precision Floating-Point Arithmetic on FPGAs", *Proceedings 2nd IEEE International Conference on Field Programmable Technology (FPT 2003), pp. 352 - 358.*

[41] M. Langhammer, "Floating Point DataPath Synthesis for FPGAs", *Proceedings IEEE International Conference on Field Programmable Logic and Applications (FPL 2008), pp. 355 - 360.*

[42] B. Lee and N. Burgess, "Parameterisable Floating-Point Operations on FPGA", *Proceedings Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, 2002, Vol.2, pp. 1064 - 1068.*

[43] H. C. Neto and M. P. Vestias, "Architectural Tradeoffs in the Design of Barrel Shifters for Reconfigurable Computing", *Proceedings 4th Southern Conference on Programmable Logic, 2008, pp. 31 - 36.*

[44] S. Braganza and M. Lesser, "The 1D Discrete Cosine Transform for Large Point Sizes Implemented on Reconfigurable Hardware", *Proceedings IEEE International Conference On Application Specific Systems, Architecture and Processors (ASAP 2007), pp. 101 - 106.*

[45] M. Tichy, J. Schier and D. Gregg, "FPGA Implementation of Adaptive Filters Based on GSFAP Using Log Arithmetic", *Proceedings IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS 2006), pp. 321 - 326.*

[46] A. Walters and P. Athanas, "A Scaleable FIR Filter Using 32-bit Floating-Point Complex Arithmetic on A Configurable Computing Machine", *Proceedings IEEE Symposium, FPGA for Custom Computing Machines, 1998, pp. 333 - 334.*

[47] A. Suardi, R. Abbiati and A. Geraci, "Double Precision FIR Filtering for High Resolution Digital Spectroscopy in Programmable Logic", *Proceedings IEEE Nuclear Science Symposium Conference, 2007, pp. 348 - 352.*

[48] Z. Bo, S. Rong and W. Qun, "Implementation of Haar Transform with PDDA Architecture for Flexible Scales", *Proceedings Second International Conference in Intelligent Computation Technology and Automation, 2009, pp. 617 - 620.*

[49] H. C. Nguyen, B. R. Hayes-Gill, Y. Zhu, and S. P. Morgan, "A High Frame Rate and High Accuracy Implementation Using an FPGA for Calculating Laser Doppler Blood Flow", *Proceedings IEEE Conference International Instrumentation and Measurement Technology (I2MT 2008), pp. 212 - 217.*

[50] J. Living, B. M. Al Hashimi and M. Moniri, "High Performance Distributed Arithmetic FPGA Decimators for Video-Frequency Applications", *Proceedings IEEE International Conference Electronics, Circuits and Systems, 1998, Vol. 3, pp. 487 - 490.*

[51] A. Damak, M. Krid and D. S. Masmoudi, "Neural Network Based Edge Detection with Pulse Mode Operations and Floating-Point Format Precision", *Proceedings Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS 2008), pp. 1 - 5.*

[52] S. Kobayashi, S. Y. Lee, T. Kino, I. Kozuka and T. Tokui, "Audio Application Implementations on a Block-Floating-Point DSP", *Proceedings Workshop on Signal Processing Systems (SIPS 2002), pp. 51 - 56.*

[53] M. Gorgon and J. Przybylo, "FPGA Based Controller for Heterogenous Image Processing Systems", *Proceedings Euromicro Symposium on Digital Systems Design, 2001, pp. 453 - 457.*

[54] A. B. Abche, A. Maalouf, R. Ayoubi, E. Karam and A. M. Alamdedine, "An FPGA Implementation of High Resolution Phase Shift Beamformer", *Proceedings IEEE International Conference on Signal Processing and Communications (ICSPC 2007), pp. 1319 - 1322.*

[55] Y. Gu, T. VanCourt and M. C. Herbordt, "Integrating FPGA Acceleration into the Promotol Molecular Dynamics Code: Preliminary Report", *Proceedings 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006), pp. 315 - 316.*

[56] M. Chin, M. Herbordt and M. Langhammer, "Performance Potential of Molecular Dynamics Simulations on High Performance Reconfigurable Computing Systems", *Proceedings Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA 2008), pp. 1 - 10.*

[57] R. Scrofano and V. Parsanna, "Preliminary Investigation of Advanced Electronics in Molecular Dynamics on Reconfigurable Computers", *Proceedings ACM/IEEE Conference Supercomputing (SC 2006), pp. 45.*

[58] J. S. Kim, L. Deng, P. Mangalagiri, K. Irick, K. Sobti, M. Kandemir, V. Narayanan, C. Chakrabarti, N. Pitsainis and X. Sun, "An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization", *IEEE Transactions on Computers, Dec 2009, pp. 1654 - 1667.*

[59] R. Scrofano, M. B. Gokhale, F. Trouw and V. K. Prasanna, "Accelerating Molecular Dynamics Simulations with Reconfigurable Computers", *IEEE Transactions on Parallel and Distributed Systems, 2008, pp. 764 - 778.*

[60] L. Zhenyu, "Tracking Radar Digital Matched-Filter ASIC Design and its Error Anaylsis", *Proceedings 5th International Conference on ASIC, 2003, Vol.2, pp. 777 - 782.*

[61] Z.-J. Sun and X.-Mei, "The Realization of SAR-Real Time Signal Processor by FPGA", *Proceedings International Conference Computer Science and Software Engineering, 2008, pp. 79 - 82.*

[62] K. S. Hemmert and K. D. Underwood, "An Analysis of the Double-Precision Floating-Point FFT on FPGAs", *Proceedings 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), pp. 171 - 180.*

[63] N. Mahdavi, R. Teymourzadeh and M. B. Othman, "VLSI Implementation of High Speed and High Resolution FFT Algorithm Based on Radix 2 for DSP Applications", *Proceedings 5th Student Conference on Research and Development (SCOReD 2007), pp. 1 - 4.*

[64] S. Sahin, S. Dikmese and A. Kavak, "Which Number Format to Use for Baseband Wimax Modem Implementation on an FPGA", *Proceedings IEEE 16th Conference Signal Processing, Communication and Applications (SIU 2008), pp. 1 - 4.*

[65] J. G. Nash, "A High Performance Scalable FFT", *Proceedings IEEE Conference Wireless Communications and Networking (WCNC 2007), pp. 2367 - 2372.*

[66] J. G. Nash, "A New Class of High Performance FFTs", *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2007), Vol. 2, pp. II-21 - II-24.*

[67] H. Ochi, " RTL Design of Parallel FFT with Block Floating Point Arithmetic", *Proceedings IEEE Conference on Soft Computing in Industrial Applications (SMCia 2008), pp. 273 - 276.*

[68] N. Mahdavi, R. Teymourzadeh and M. B. Othman, "On-Chip Implementation of High Speed and High resolution Pipeline Radix 2 FFT Algorithm", *Proceedings International Conference on In Intelligent and Advanced Systems (ICIAS 2007), pp. 1286 - 1288.*

[69] R. K. Snider, A. J. Lukes, Y. Zhu and J. P. Miller, "A Digital Methodology Integrating Experimental and Theoretical Neuroscience", *Proceedings First International IEEE EMBS Conference on Neural Engineering, 2003, pp. 376 - 379.*

[70] N. Alachiotis, A. Stamatakis, E. Sotiriades and A. Dollas, "A Reconfigurable Architecture for Phylogenetic Likelihood Function", *Proceedings International Conference on Field Programmable Logic and Applications (FPL 2009), pp. 674 - 678.*

[71] N. Alachiotis, E. Sotiriades, A. Dollas and A. Stamatakis, "Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function", *Proceedings IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), pp. 1 - 8.*

[72] A. Yonemoto, T. Hisakado and K. Okumura, "An Implementation of Numerical Inversion of Laplace Transforms on FPGA", *Proceedings International Symposium on Circuits and Systems (ISCAS 2003), Vol.4, pp. IV-492 - IV-495.*

[73] A. Postula, D. Abramson and P. Logothetis, "The Design of A Specialised Processor for the Simulation of Sintering", *Proceedings 22nd EUROMICRO Conference EUROMICRO 96, 'Beyond 2000: Hardware and Software Design Strategies', pp. 501 - 508.*

[74] G. Danese, F. Leporati, M. Bera, M. Goachero, N. Nazziaceri and A. Spelgatti, "An Application Specific Processor for Montecarlo Simulations", *Proceedings 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP 2007), pp. 262 - 269.*

[75] G. W. Morris and M. Aubury, "Design Space Exploration of the European Option Benchmark Using Hyperstreams", *Proceedings International Conference on Field Programmable Logic and Applications (FPL 2007), pp. 5 - 10.*

[76] G. L. Zhang, P. H. W. Leong, C. H. Ho and K. H. Tsoi and C. C. C. Cheung, "Reconfigurable Acceleration for Monte Carlo based Financial Simulations", *Proceedings IEEE International Conference on Field-Programmable Technology, 2005, pp. 215 - 222.*

[77] N. A. Woods and T. Vancourt, "FPGA Acceleration of Quasi-Monte Carlo in Finance", *Proceedings International Conference on Field Programmable Logic and Applications (FPL 2008), pp. 335 - 340.*

[78] G. Danese, F. Leporati, M. Bera, M. Giachero, N. Nassacari and A. Spelgatti, "An Accelerator for Physics Simulations", *Journal of Computing in Science Engineering, Vol. 9, No. 5, 2007, pp. 16 - 25.*

[79] Intel , www.intel.com, "Intel QuickPath Architecture", "QuickPath Technology: Unleashing the Performance", *March 2010.*

[80] Altera COPR., www.Altera.com, White paper: "Designing and using FPGAs for Double-Precision Floating-Point Math", *March 2010.*