A Flexible Arithmetic System for Simulation

TSOI Kuen-Hung

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science & Engineering

November 2007

Abstract

Custom hardware accelerators are commonly used in simulation systems requiring high computational power. Such applications often have few data dependencies, allowing implementation using parallel datapaths. For such problems, optimization of the datapath of the circuits leads to significant improvements in overall performance.

The Computer Arithmetic Synthesis Technology (CAST) framework, developed in this work, allows one to quickly explore the design space in three dimensions: the number system, the operator architecture and the configuration of individual operators. It utilizes sophisticated arithmetic algorithms and reconfigurable architectures, captured in the object libraries. The final result is an optimized datapath satisfying user requirements, and the output can be controlled at different levels.

To demonstrate its ability, the CAST framework is used to implement a number of simulation systems including the datapath for the force computation pipeline of N-body simulation and Monte Carlo simulation for interest rate financial derivatives. A novel multiplier generator and an efficient random number generator are also presented as basic building blocks for simulation. Together, these tools provide an easy way to describe simulation system in a number system independent manner, and generate implementation to satisfy different performance, area and accuracy constraints.

Statement of originality

The work presented in this thesis was carried out by the author during his doctoral program in the Department of Computer Science and Engineering, The Chinese University of Hong Kong, between 2003 and 2007, under the the supervision of Prof. Philip H.W. Leong.

The ideas and experiments presented are original with exceptions stated explicitly below.

- The implementation of CAST framework including modeling hardware components as C++ objects, embedded simulation function, circuit generation and performance evaluation are work of the author.
- The floating-point and logarithmic operators are based on the FPLIB packages from Aremnaire project at ENS Lyon [aEL06]. The elementary function approximation is based on the STAM algorithm [SS99a] and developed jointly by the author and Chun Hok HO. Other arithmetic libraries in CAST are the work of the author.
- The three dimension multiplier (TDM) part of the parallel multiplier generator is based on the three-greedy algorithm [SMOR98] from Paul F. Stelling et al.
- The architecture and implementation of the alternating step generator RNG are joint efforts of the author and Ivan Ka Ho LEUNG.

- The interfacing and performance benchmarking of the N-body force pipeline are the work of Jackson Ho Chuen YEUNG. The architecture and implementation of the pipeline core are the work of the author and Chun Hok HO.
- The bit width optimization of the Monte Carlo core is the work of the author and Chun Hok HO. The other parts of the system are done by Guang Lie Zhang and others.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Simulation is a process to imitate real processes in an artificial environment with the aim of extracting characteristics and projecting results of the modeled processes. Simulation can be applied to many areas including science, finance and entertainment. For example, it has been used to study classic physics problems such as heat distribution [Met93, ZHF+07], pressure reactions [LRN+01], wave systems [SSL01], aerodynamics [Cho97], particle dynamics [LKM02], typhoon prediction [DJW03] and quantum phenomena [CLS02]. Other scientific studies that utilize simulation include chemistry [GJS03] and medical care [PDA01]. Besides scientific applications, simulation is also used in financial sectors to address problems in pricing [Fu95], risk measurement [MS01] and market prediction [EM96]. With the rapid growth in the entertainment market, it is also important to apply simulation to create more realistic experiences in both films and games [BHW96].

Since the systems will spend most of their time processing data through a fixed flow in a main loop, optimization via hardware acceleration may greatly improve the overall system performance. In such a situation, a customized datapath is constructed to perform the required computation. Through specialization, it is possible to achieve much higher performance than software due to increased levels of parallelism and higher memory bandwidth.

Table 1.1 Dimensions for Optimization

Number Representation	2's complement, fixed-point, floating-point, logarithm numbers, etc.
Pipeline Stages	Deep pipeline v.s. low latency
Radix	Bit serial, digit serial and bit parallel
Precision	Tradeoffs between precision and hardware resources
Arithmetic	There are many different ways to perform the same arithmetic computation, e.g. ripple carry, carry-save or carry look-ahead for adders.
Synthesis Algorithms	Different optimizations such as common sub-expression removal, resource scheduling, etc. can be applied

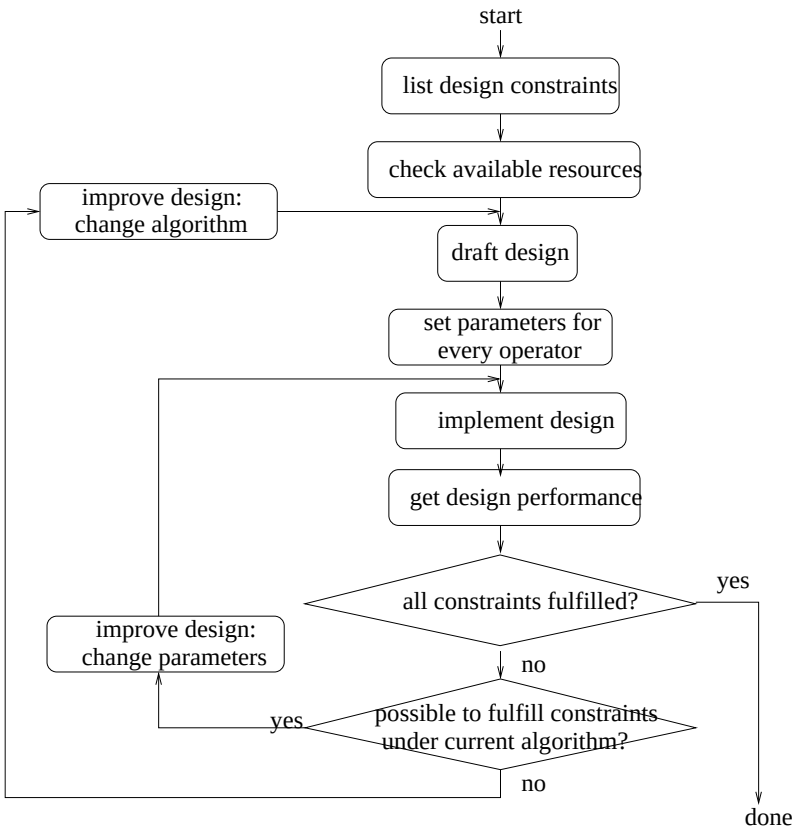
Table 1.2 Constraint Factors

Timing	Many designs require operation above a given minimum frequency
Area	A finite amount of hardware resources are available
Power Consumption	For embedded systems, the power consumption may be an important concern
Special Device requirements	This may include internal memory blocks, internal dedicated multipliers and I/O pins
Bandwidth	Communication channels limited the rate of distributing data to processing elements

In general, the datapath is a directed graph of arithmetic operators. In every portion of the system, their timing and precision requirements may be different and the available degrees of freedom are shown in Table 1.1. At the same time there are issues that constrain the choices of implementation as shown in Table 1.2.

Various systems have been developed to generate designs in this space using high level language to circuit synthesizers, or a set of optimized libraries for circuit construction. The problems associated with these work flows are that developers either sacrifice control over implementation details or spend too much effort at low levels while losing the focus of optimization at the system level. Fig 1.1 shows the current design practice. There is no easy way to optimize a design in various

Figure 1.1 Current design practice.

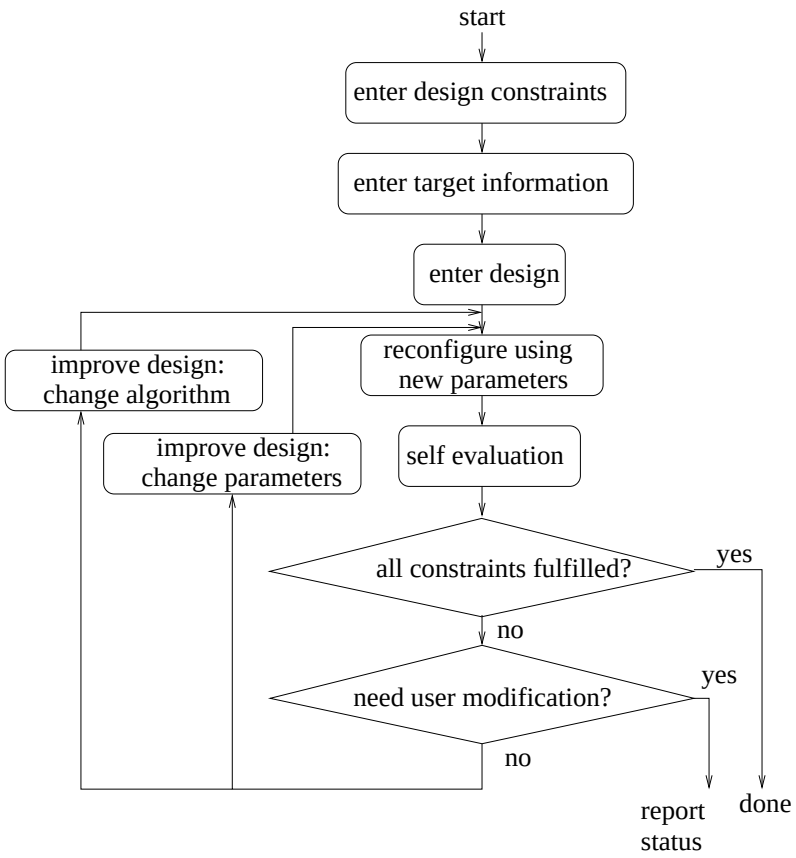


dimensions within a set of given constraints.

The number of iterations required may depend on the experience and skills of the hardware designer. In addition, such skill must be updated frequently to adapt to new technologies as they become available. On the other hand, the arithmetic algorithms and the specifications of the target hardware platforms are well defined. A major design challenge is to link up these two domains and iterate to find the suitable solution.

Productivity can be improved if there is a method to handle such tasks under a single framework. The idea is to store the arithmetic algorithms in a modular and parameterized form. Information of both the arithmetic algorithm and hardware platform can be stored in extensible libraries. Metrics are used to evaluate

Figure 1.2 Improved design practice.



the current configuration and the result is fed back to improve the design. The system can optimize the design by changing parameter values and re-evaluating until pre-defined constraints are met. Figure 1.2 shows the improved design practice.

To maximize reuse and shorten development time, a method of connecting existing blocks to form larger blocks is needed. Such a feature allows complex systems to utilize optimized building blocks. To dissociate the underlaying hardware information from the description, a uniform interface should be used for connection and configuration. Furthermore, the description should be as simple as possible so that development and learning time is reduced. As new arithmetic algorithms become available, it must be able to easily adapt them in the system.

Another important part of the system lies in user configuration. An interface is

required to minimize efforts when switching operators or changing parameters of the operators. Through the common interface, users can specify the constraints to the top-most and/or any sub-blocks. Also, a method is required to propagate the constraints and configuration from top level to submodules of the design. Finally, to enable optimization, the system must have a fast and accurate way to evaluate the current configuration.

1.2 Hardware Accelerated Simulation

A common scenario for simulation is to model real world systems given their initial condition, environment and inputs. The system reports status information while processing input data and changing internal states. This flow usually requires the ability to process large amounts of data within limited time frame. When computer systems are used to perform simulation, the computing power of the system is often a limit to the scale or resolution. Several common characteristics associated with simulations are summarized below:

Parallelism Most real world systems contain objects which act and interact concurrently. In a computer simulation, parallelism should be exploited to reduce execution time. In many systems, the data being processed are independent of one another, increasing the permissible amount of parallelism.

Large and frequent input Many simulation systems require multiple streams of inputs and generate and update a large set of internal states. This requires efficient ways to access large amounts of data with high bandwidth, particularly for parallel systems.

Computationally Intensive The speed and quality of the simulation are dominated by the computing power of the system in a given time frame.

Simulation has been studied and applied since the early days of computers. The advance of computer technology and the increase of available computing power

allow more simulation systems to be built-in computer software. These systems have become increasingly complicated and the timing requirements tighter. Personal computers and work stations often lack the computing power to handle large and real time simulation process. Common techniques to accelerate simulation systems are briefly discussed below.

Microprocessors To address the performance issues of serial/single thread systems, parallel and distributed architectures based on microprocessors or digital signal processor (DSP) have been proposed. The systems are partitioned into modules which can be assigned to different processing elements and processed concurrently. These methods utilize parallelism to improve overall performance. Limiting factors include load balancing, network bandwidth and communications overhead.

ASICs Application Specific Integrated Circuits (ASICs) are capable of the highest performance and even large-scale distributed and DSP solutions cannot compete with fully customized ASIC designs [Mak05]. For fixed datapaths and predictable data access patterns, a custom ASIC design can maximize the utilization of the given silicon resources by providing customization of arithmetic for the required accuracy, fine grain parallelism and point-to-point interconnections of processing elements. The drawbacks of the ASIC solution include high initial cost and long development time. It is also difficult and costly to further modify, extend and improve the design.

Reconfigurable platform This is a middle ground between ASICs and microprocessors. Often a pure software implementation on conventional off-the-shelf processors cannot fulfill the demands of today's sophisticated simulations and an ASIC solution is too expensive. Reconfigurable platforms offer the potential of ASIC-like performance without the high initial cost. In this research, they are proposed for hardware accelerated simulation systems and can achieve a higher level of performance than distributed and DSP solutions, with lower cost than an ASIC design. Some characteristics of a reconfigurable platform in simulation are presented below.

Since the reconfigurable platforms provide a high degree of freedom in designing parallel architectures, fine grain parallelism can be efficiently represented. Most simulation systems are highly customized and domain-specific. The amount of reuse is limited and the design cost of ASIC cannot be amortized over a large number of users. Reconfigurable platforms are suitable for such low volume, high performance application domains.

New algorithms for both scientific and financial simulation can be implemented on the reconfigurable platform without major changes in the physical hardware. This also helps in exploring the behaviors of different models, architectures and simulation algorithms. The parameters and models under which the simulation process is executed are subject to change. In a reconfigurable platform, these changes can be made as easy as in software implementations while maintaining performance.

For reconfigurable platforms, the tools to synthesis the hardware design for a simulation accelerator are relatively easy to master and inexpensive. Small changes in the design, simple modification of the configuration and replacement with new libraries can be performed by experienced users.

With all the above advantages, reconfigurable platforms are becoming popular for accelerating simulation problems [HGG+05, GVH06, BTLM06]. This poses the need for an overall optimization framework. Optimization can be done at several levels including the architecture, arithmetic and the datapath. The complete system must be considered since local optimization of individual levels may lead to a poor global solution.

1.3 Objectives

For accelerating simulation systems on reconfigurable platforms, a framework for designing data paths which considers system design and performance optimization of the final circuit is needed. The main objective of this work is to design and

implement a framework which can be applied to construct advanced simulation systems for real world designs. The framework proposed includes the following functionalities:

- Methods to enter datapath designs at an abstract level. The design entry is based on the conventional object-oriented C++ language and users can control details concerning each individual operator.
- A means to verify the functional correctness of the design at the software level. The verification method uses the same description framework as design entry and thus can be performed by users with little hardware design experience.
- A library of components ranging from simple logic primitives to optimized elementary function generators. Besides arithmetic operators which are the most useful building blocks for constructing simulation systems, auxiliary units such as random number generators are included.
- An interface for specifying configuration and constraints of the operators. This also ensures a correct interface between arithmetic operators in different number systems.

The most significant feature of this work is the ability to capture computer arithmetic and reconfigurable hardware design expertise in a single framework and utilize this information to improve the performance of the design. Several levels of optimization are applied in the system.

The highest level of optimization is performed at the arithmetic level in which the representation and the number systems used the design are considered. Knowledge of computer arithmetic and understanding of the simulation model are required for this task. By providing arithmetic operators in different number systems and a unified interface for using these operators, developers can construct an efficient simulation system with minimal effort.

The second level of optimization is performed when constructing individual operators. The framework provides a simple way to select schemes for implementing an operation after the input and output representation has been fixed at the previous level of optimization.

The third level of optimization is performed after the type and structure of the individual operators have been fixed in the system. Due to different requirements on precision and limitations on physical resources, the bit width of each operator can be fine tuned for further optimization.

There are many dedicated hardware resources available in modern reconfigurable computing platforms. Understanding the availability and usage of these resources can help to further improve the design. The framework will try to optimize designs by automatically utilizing resources according to the target platform specifications.

The framework also allows users to specify requirements and constraints such as available resources and allowed inaccuracies in the results. It can then automatically determine a suitable instance of the rendered design.

In summary, one can use the framework to quickly explore different designs. It provides a means to evaluate designs based on simulation and estimation. Based on this evaluation, users can modify the design by setting different parameters at the design entry description. The unified operator interface significantly simplifies the task of mixing operators from different libraries.

1.4 Contributions

This research work contributes to the knowledge on efficient datapath generation for simulation applications. It is useful for accelerating simulation system on reconfigurable platform. The major contributions are as follows:

- A novel framework, Computer Arithmetic Synthesis Technology (CAST), for designing and optimizing arithmetic datapaths which are an important part of

simulation systems. By combining the knowledge of both computer arithmetic and datapath design, the framework helps to improve the simulation process on reconfigurable platforms.

- A novel and efficient construction for a uniform random number generator (RNG) [TLL07] which combines the unpredictable nature of a real hardware RNG with the efficiency of a pseudo RNG in a compact design. This unit provides the primary inputs to many simulation systems.

- A set of arithmetic operator building blocks [THYL04, TL05] together with auxiliary functions for interfacing and optimization. This important part of the framework allows users to explore and optimize the design in various dimensions.

- The proposed framework was applied to two practical simulations: the N-body problem [THYL04] and Monte Carlo interest rate simulation [ZLH+05]. The achieved results demonstrate the power of the framework and also advance the state of the art in these applications.

1.5 Thesis Organization

The thesis is organized as follows. In Chapter 2, the reconfigurable platform is introduced and current design practices with related research on hardware accelerated simulation systems are reviewed. In Chapter 3, background theory on number systems and datapath optimization are discussed and the details of the proposed framework are presented. In Chapter 4, a parallel multiplier generator is used as an example to demonstrate the usage of the framework. In Chapter 5, a novel uniform random number generator is presented. Then in Chapter 6, the application of the framework to a Monte Carlo interest rate simulator is presented to show the framework's ability to optimize for a given output precision. Chapter 7 presents an N-body force simulator which was optimized at the number system level. The

results of implementing the framework and these examples are given in Chapter 8. Finally, Chapter 9 presents conclusions of this research.

Chapter 2

Background and Review

2.1 Introduction

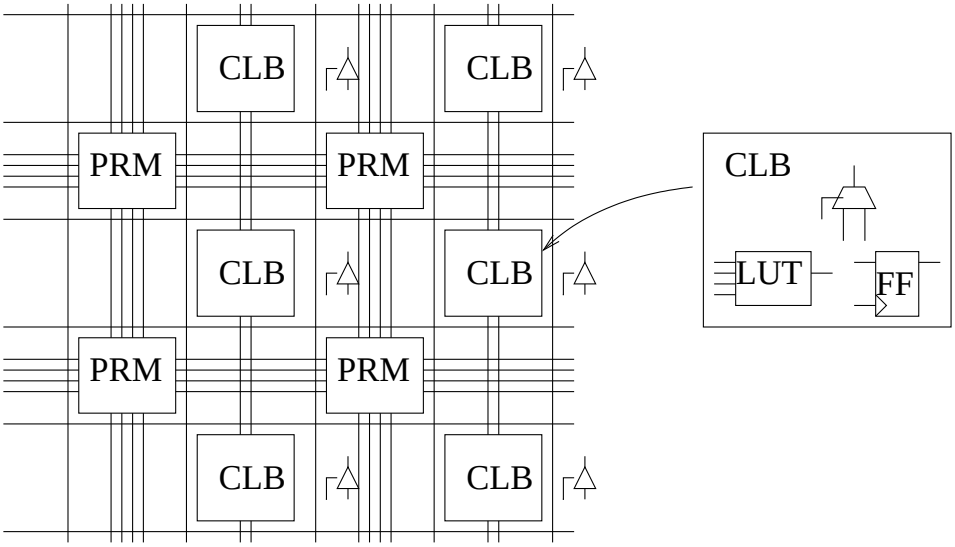
This chapter provides background on Field Programmable Gate Array (FPGA) devices, high level design methodologies and hardware accelerators for simulation systems.

The chapter is organized as follows. In section 2.2, the basic architecture of modern reconfigurable devices is presented. Then in section 2.3, different approaches for improving FPGA design flow are presented. Finally in section 2.4, previous related research on hardware accelerated simulation systems are reviewed.

2.2 Modern Reconfigurable Platforms

The basic structure of an FPGA is a 2-D array of Configurable Logic Blocks (CLB) surrounded by programmable interconnect switches. Inside the CLB, there is a set of primitive function elements including lookup tables (LUT), flip-flops (FF) and other simple logic gates. The process of mapping a design to FPGA platform involves configuring the CLB to perform some logic sub-function and specifying the connections between them through programmable switching nodes. This information is stored in configuration memory which can be programmed by the user using an standard interface such as JTAG. Changing the design is a process of changing

Figure 2.1 FPGA structure.



the content of the configuration memories and no modification of physical elements or rewiring in silicon is required. Inside the CLB, connections between primitives are controlled by multiplexers (MUX). The connections between CLB are relatively flexible and made via Programmable Routing Matrices (PRM) and bus lines; however, they are limited in capacity. A block diagram of a generic FPGAs is shown in Figure 2.1.

Today's advanced FPGA chips also offer special embedded blocks such as large memory blocks (BlockRAM) and fast carry chains between adjacent logic blocks [Xil02]. Dedicated multipliers and other DSP related blocks can also be found. Some features of FPGA designs compared with other VLSI technologies are listed below:

- Fast design to product time. Both the time for circuit development and programming the FPGA device are short compared with the required time for layout and manufacture in ASICs.

- Easy simulation and debugging. Software simulators and debuggers provide efficient methods for finding bugs and estimating performance. Since silicon

level verification and testing have been done by the FPGA manufacture, only simulation and verification on a functional level are required in most FPGA based designs.

- It is possible to use the same FPGA hardware platform for many different applications. This makes designs more flexible, extensible and cost effective. This characteristic of FPGA also makes it suitable for low cost prototyping of early designs which are subjected to change. In fact, FPGAs are commonly used for design verification of ASICs in industry.

- The design can be upgraded after deployment without hardware replacement. It is also possible for skilled and experience end users to improve and optimize the design for specific applications.

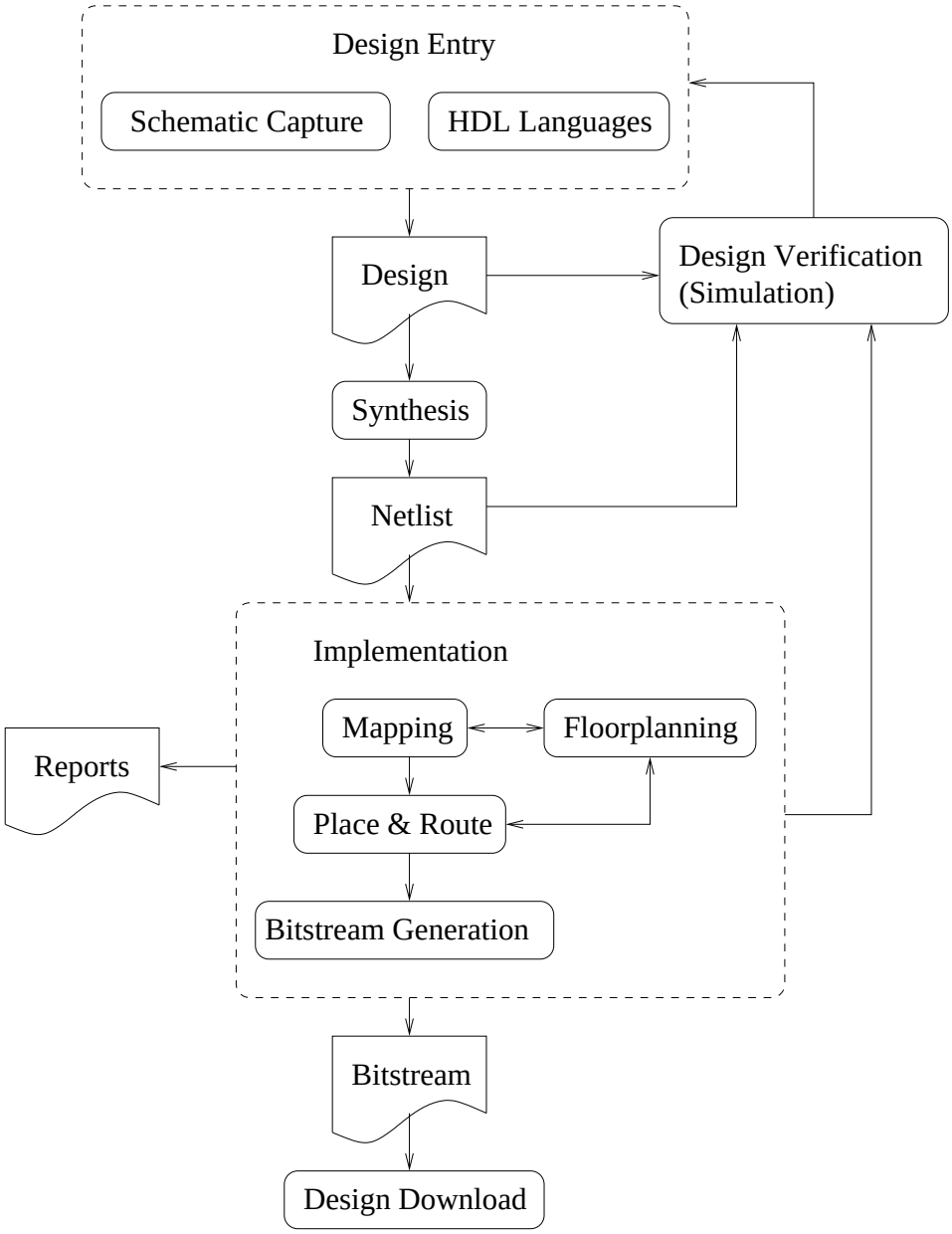
The traditional FPGA design flow is shown in Figure 2.2. Design entry can be either schematic capture or synthesis via a Hardware Description Language (HDL). The schematic flow is more intuitive for small designs while the HDL flow provides an efficient way to implement and manage large and complex designs.

In the synthesis flow, a netlist is generated describing the logic functions and their interconnections. The functions are then mapped to the logic primitives of the target FPGA platform. The placement of logic primitives and routing of connections are altered to find an optimized solution which will meet the constraints stated by the designer. The implementation process generates a bitstream representing the configuration of the FPGA, which can be downloaded to the chip.

2.3 FPGA Design Methodology

In this subsection, current FPGA design methodologies are presented. These include library generators and high level language synthesis. The advantages and disadvantages of each method are also discussed.

Figure 2.2 FPGA design flow.



2.3.1 Xilinx Core Generator

Module generators are able to generate customized designs from their input parameters. For example, the Xilinx Core Generator library for FPGAs [Inc02] provides highly optimized libraries for the fixed-point multiplication, multiply-accumulate, division and Coordinate Rotation Digital Computer (CORDIC) operations. Besides the arithmetic operations, peripheral interfaces and common designs such as FIFO are also provided. In the Core Generator system, users select the module to be generated and configure the parameters through a graphical user interface (GUI). It provides a wide range of parameters for each module including the size, functions, I/O, resource constraints and even placement information. The generated modules are usually in format of Xilinx proprietary format used in the Xilinx back end tool chain.

The advantages of this kind of vendor-provided module generator is the achievable optimization through detailed low level customization. This requires knowledge of dedicated hardware available and physical layout on the devices. To achieve the same performance using conventional hardware description languages (HDL) such as Verilog and VHDL is time consuming. Also, it is difficult to maintain and improve the resulting circuits.

Unfortunately, the number of modules provided by this kind of vendor provided system are limited. For arithmetic system, operators such as constant coefficient multipliers, square root and logarithmic number system computation are not included. In such libraries, there is usually little flexibility in the numerical representation which is usually fixed or floating-point. The configuration parameters for the modules are mostly concerned with target platform optimization rather than numerical algorithms. The most important point is that these systems only optimize and generate individual modules and lack the ability to consider the complete design. Developers are required to interface these discrete modules to form a complete design, and optimization on the system level may require several iterations of

reconfiguration and regeneration of these modules.

2.3.2 Floating-Point Module Generator using ASC

Flexible floating-point module generators have been developed [LTM03] using a different approach. This system relies on the A Stream Compiler (ASC) [Osk06] for low level circuit generation. Users can use the arithmetic operators, such as the '$+$' and '$\times$' symbols, in C++ language directly for circuit description. The ASC will then be used to compile the program into corresponding FPGA netlist. Number representation and format of floating-point data are created as custom types and classes in C++. Parameters can be configured to control sign mode, normalization, rounding and bit width of the generated modules. These parameters are automatically determined based on the required optimization option, which is selected by users from among throughput, latency and area.

2.3.3 FPLIB

A similar idea of providing a parameterized module generator had been done at the HDL level. FPLIB [aEL06], developed in the Aremnaire project at ENS Lyon, is a library for hardware operators for floating-point and logarithm number systems (LNS). The library utilizes the *generic* parameter in VHDL for configuring the bit width of the generated modules. Besides the common operators such as add, subtract, multiply, divide and square root, the library also provides a set of conversion operators for bridging different number systems or operators with different precisions.

The above examples of module generator systems can improve FPGA design productive and performance. They configure the module based on parameters either specified explicitly by the user or implicitly from abstract level optimization requirements. The major disadvantages of these systems is the lack of global optimization on the datapath. Also, users have limited control over the optimization

process. The ability for users to extend and improve the modules may depend on different back ends and involve modification of the tools themselves.

2.3.4 PAM-Blox I/II

PAM-Blox [MMF98] was one of the first module generators which allowed programmed generation of circuits from C++. A recent extension, PAM-Blox II [Men02], has been reported in 2002. In the system, complicated circuits such as constant coefficient multipliers, Booth multipliers, CORDIC units are implemented in PAM objects. To implement a circuit, the user should first enter the design in a structural connection of different objects provided in the PAM. The system then generates a circuit in Xilinx netlist format representing the datapath of the C++ program. The final configuration stream will then be produced by Xilinx place and route tools. Higher level design is performed by combining such descriptions in a structural manner.

The PAM system improved productivity by freeing developers from traditional HDL. But the system only generates individual modules and other system level concerns such as interfacing between modules remain the duty of developers. Also, the modules in the PAM are rigid in both size and function. Similar designs with little variations require creation of different objects. Finally, simulation and verification require external tools that operate on the generated netlist. This makes it difficult to debug the circuit by relating the netlist and C++ descriptions.

2.3.5 JHDL

In 1997, the JHDL [HBH+99] project was initiated in the Configurable Computing Laboratory at Brigham Young University. It uses the Java language for design entry in which functional blocks are treated as objects. The lower level libraries are tightly coupled with the primitives in Xilinx FPGAs. The abstraction level increases

as higher level libraries are device independent. Users can design the complete datapath in JHDL using a structural description. The APIs for circuit construction are very similar to the HDL description with input/output port interfaces and a *connect* method to connect the objects' ports. Auxiliary tools such as a simulator, schematic extractor and state machine generator are also provided.

2.3.6 Handel-C

High level synthesis algorithms are also used to speed up the development on FPGA platform. The Handel-C [Pag96] language from Celoxica can accept a C-syntax like description and generate the corresponding circuits. Unlike the JHDL system, Handel-C is a behavior language for circuit generation based on the custom compiler analyzing the extended syntax in the user program. Simulation and debugging tools are also provided in the integrated developing environment (IDE). This helps to realize and verify an existing algorithm on hardware without experience in hardware development. This can also benefit experienced developers for rapid prototyping and evaluation of system architecture. The Handel-C syntax is based on the ANSIC-C standard with extensions. The extensions help to optimize circuit performance by allowing users to specify hardware related information such as variable bit width and parallel constructs.

2.4 Hardware Acceleration on Simulation Systems

Examples of simulation systems accelerated using dedicated hardware are reviewed in this section.

2.4.1 Floating-Point N-Body Simulation

An N-body simulation system using optimized floating-point units was implemented by Lienhart *et al.* in 2002 [GAM02]. Using a 16-bit significand floating-point representation, the system can achieve about 3.9Gflops at 65MHz on a Xilinx XC2V3000 FPGA. The system built is to perform the computation of smoothed particle hydrodynamics method (SPH). The authors prefered floating-point over logarithmic arithmetic as the latter required a large ROM to implement adders. In the design, the resource utilization is dominated by the adder, divider and square root operators. With the help of Xilinx Block Multipliers, a single floating-point multiplier consumes about 0.5% of the total FPGA resources. The complete design fits in 49% of the logic resources. Results also show that a 22-bit implementation will be two to three times larger than the implemented 16-bit version.

2.4.2 Space Plasma Simulator

In 2003, Popoola and Gough developed a system to simulate the space plasma using FPGAs for acceleration [PG03]. The design included a soft core CPU and some co-processing units such as FFT and floating-point sine/cosine computation. The results show that the accelerated system is several orders of magnitude faster than a software implementation. The system uses the 1D electrostatic code to investigate various phenomenon in space plasmas. In a single iteration, the system needs to assign positions and velocities of the particles to the nodes; compute the electric and magnetic fields at the nodes; compute the force field; and interpolate to obtain new particle positions and velocities. For a simulation including hundreds of thousands of particles, the system utilized three parallel co-processing units to off-load the computationally intensive tasks of the controlling processor. The improvement of this parallelized design increases as the number of particles simulated is increased.

2.4.3 ReCSiP System

Reconfigurable platforms have also been used in biochemical research. In 2004, Masato *et al.* [YOFA04a] constructed a stochastic biochemical simulator based on Gillespie's First Reaction Method (FRM) using a high throughput floating-point design. Equipped with a Xilinx XC2V6000 FPGA chip, the ReCSiP system can run 105 times faster than an AMD AthlonXP2800+ processor. The set of fine grain process with heavy intra-process communication in the simulation make it inefficient in distribute or cluster systems. The core of the platform is a set of four parallel reactor modules, each including 6 floating-point multipliers, 5 floating-point adders and a random number generator. Using over 77% of the FPGA area and operating at 76MHz, the system outperforms both AlthlonXP2800+ and Xeon 2.8G Dual processors by utilizing a 27 stage pipeline and multiple parallel simulators in a single chip.

2.4.4 GRAPE Project

The GRAPE project in Japan [Pro05] in 2005 performs double-precision calculations of gravitational N-body simulations in high speed. The modified SIMD architecture is suitable for integration of over 1,000 processing elements on a single ASIC and targets execution speeds exceeding 1 Petaflops. The GRAPE project has a long history of developing platforms for N-body simulation. The previous GRAPE systems are based on the idea of deep pipelines and massively parallel architectures. Examples are GRAPE-2 [TTJD91], GRAPE-4 [MTES97] and GRAPE-6 [MFK00]. For example, the GRAPE-6 project in 2002 delivered up to 64Tflops of computing power. The 1.8M gate GRAPE-6 processor was fabricated using $0.25\mu m$ technology. With 6 parallel pipelines, a single processor can provide 31Gflops using a 22.5MHz system clock. The bottleneck of the GRAPE-6 system is the intra-processor communication. Various approaches were developed to reduce the impact of bandwidth limitations in the system. The final GRAPE-6 system includes

dedicated network board to connect 4 host computers and 16 processor boards as a computer module and 4 modules are connected through a Gigabit Ethernet Switch.

The new GRAPE-DR [Mak05] system proposed another approach for higher computing power. Over 1,000 processors, which are extremely simple yet fully programmable, will be connected through hierarchical broadcast/reduction networks in a single chip.

2.5 Summary

As shown in this chapter, the reconfigurable device is an excellent platform for accelerating simulation systems when high computational power can be achieved by hardware parallelism, deep pipelines and efficient intra-processor communication. The hardware accelerators range from single chip FPGA implementation to large scale multi-core, multi-chip designs. It is shown that these hardware accelerators can achieve better results than pure software designs running on high-end processors. Various tools and development environments have been proposed to improve productivity by providing optimized libraries or high level synthesis. While having achieved their objectives, these tools have limitations on the degree and level of achievable optimization.

Chapter 3

CAST - A Framework for Flexible Datapath Exploration

3.1 Introduction

To efficiently explore the design space for simulation problems, a unified framework is needed to express the various design and implementation details of the hardware. In a traditional design flow, hardware description languages (HDL) such as VHDL and Verilog are used as the primary design entry method. These languages are well defined and wildly adopted in industry. A developer can use HDL to describe and control every detail of a design and achieve highly optimized results. The major disadvantage of the HDL flow is that it is too low level and too much effort is spent on design details.

To address the above problems, new methodologies have been proposed. Most of them can be classified into two classes: high level language synthesis tools [Pag96, BH98, HLT+02] and library generators [Xil00b, MMF98]. The first method is to translate a conventional high level computer language, such as C and Java into an equivalent hardware circuit. The second method provides an interface for user to configure the parameters of a predefined library set and generate the desired circuit. Some examples of these methodologies have been presented in Chapter 2.

Both methods provide an abstraction of the hardware details with an efficient

interface for developers. Each have different emphasis and drawbacks.

- The main advantage of the first method is the familiarly of programming tools. Most developers know how to program in high level languages and one can easily port an algorithm from existing software to hardware with minimum modification. However, such simplicity does not generate high performance results in general. Developers either need insert additional information such as parallel constructs or they have to depend on the embedded optimization algorithms of the tools.

- The library generator approach is usually specialized for certain applications. For example, there are generators for floating-point arithmetic, CORDIC computation, lookup tables for approximation, etc. The performance of the generated circuits depends on the configuration entered by users. While achieving high quality for individual modules, the overall performance of the complete design may not be optimal due to the lack of a high level understanding of the problem. Also, the flexibility is usually limited in the generator and new or customized algorithms are difficult to add to the system.

To address the above problems, the Computer Arithmetic Synthesis Tool (CAST) was developed. It is a framework for datapath design and optimization using an object oriented approach.

One of the major features of CAST framework is to allow users to switch and mix different number representations. Fixed-point, floating-point and logarithmic number systems are supported and these are explained later in this chapter. A lookup table based algorithm is presented for approximating elementary functions. These are examples of computer arithmetic knowledge captured in the CAST framework.

The features and implementation of the CAST framework is also presented here. We will explain the usage and internal architecture of CAST by examples. One of most valuable parts of CAST is the arithmetic operator library. The implementation of each operator and a unified interface for using them is presented.

This chapter is organized as follows. In section 3.2, several number systems with their attributes and applications are presented. Section 3.3 will give an overview of the implementation details of CAST. Section 3.4 presents a set of operators in different number systems implemented in the CAST library. A unified configuration and creation interface for these operators is presented in Section 3.5 Finally, Section 3.6 concludes this chapter on the CAST framework.

3.2 Computer Arithmetic

In this section, a brief review of the fixed, floating and logarithmic number representations is presented. More detailed descriptions can be found in computer arithmetic textbooks such as Koren [Kor93], Flynn [WF82, Fly01], Parhami [Pat00] or Ercegovac [EL04].

3.2.1 Fixed Point

Unsigned integers are used to represent the nonnegative integers. An N-bit unsigned integer has a range $[0, 2^N - 1]$ and can be described in binary form, with u_i being the i'th binary digit:

$$U = (u_{N-1}u_{N-2}\ldots 0),\ u_i \in \{0, 1\}.$$

This represents the number

$$U = \sum_{i=0}^{N-1} u_i 2^i.$$

The two's complement representation is the most widely used scheme for integers. The representation is similar to the unsigned integers except that the most significant bit has a weighting of -2^{N-1}. A two's complement integer X of different N can be represented in binary form, with x_i the i'th binary digit as

$$X = (x_{N-1}x_{N-2}\ldots 0),\ x_i \in \{0, 1\}.$$

X has a range of $[-2^{N-1}, 2^{N-1}-1]$ and represents

$$X = -x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i$$

The two's complement integer representation can be generalized to represent fractional numbers by scaling. A two's complement fraction is represented as a pair $(N, F)_{\mathcal{I}}$, where N is the word length in bits, F is the fractional word length and the subscript $\mathcal{I}$ shows that it is an integer representation. The most significant $N - F$ bits of the number represent the integer part and the remaining F bits are the fractional part of the number

$$Y = (\overbrace{a_{N-1} \ldots a_F}^{\text{integer}} \overbrace{a_{F-1} \ldots a_0}^{\text{fraction}}).$$

This corresponds to a scaling of the two's complement integer representation by the factor $S = 2^{-F}$ and the two's complement fraction number Y represents

$$Y = 2^{-F} \times (-x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i)$$

Note that the two's complement fraction $(N, 0)_{\mathcal{I}}$ corresponds to the two's complement integer case and $(N, N)_{\mathcal{I}}$ has a range of $[-1, 1)$.

3.2.2 Floating-Point

Floating-point numbers are an approximation to the real numbers and offer wider dynamic range than fixed-point numbers, at the expense of reduced precision and larger implementation complexity and area. In the IEEE 754 standard [IEE85] format, three fields are used to represent a floating-point number and it can be represented as the pair $(N, F)_{\mathcal{F}}$ where N is the total word length, F is the word length of the significand (also known as the mantissa) and the subscript $\mathcal{F}$ shows that the pair represents a floating-point number. The most significant bit is a sign bit A, the following $J(= N - F - 1)$ bits, b_i encode the exponent field B and the remaining

F bits c_i encode the mantissa field C

$$Z = (\overbrace{a_0}^{\text{A}}\overbrace{b_{J-1}\ldots b_0}^{\text{B}}\overbrace{c_{F-1}\ldots b_0}^{\text{C}}).$$

A represents the sign S where

$$S = \begin{cases} +1 & \text{if } a_0 = 0 \\ -1 & \text{if } a_0 = 1 \end{cases}$$

The unsigned integers B and C are encoded representations of the exponent and mantissa respectively. The exponent E, is stored in a biased representation with $E = B - (2^{J-1} - 1)$. For normalized numbers, $B \neq 0$ and the significand is represented by $M = 1 + C \times 2^{-F}$. This is a two's complement fraction $(F+1, F)_{\mathcal{I}}$ with the most significant bit being implicitly set to 1. If $B = 0$, it is called a denormalized number, and there is no implicit 1 in the $(F, F)_{\mathcal{I}}$ fraction.

The number represented is given by

$$Z = \begin{cases} S \times 2^E \times M & \text{if } (0 < B < 2^J - 1) \\ S \times 2^E \times (M-1) & \text{if } (B = 0) \\ S \times \infty & \text{if } (B = 2^J - 1 \text{ and } C = 0) \\ NaN & \text{if } B = 2^J - 1 \text{ and } C \neq 0). \end{cases}$$

3.2.3 Logarithmic Number System

The logarithmic number system (LNS) is a special case of floating-point in which the mantissa is always 1 (i.e. only the sign and exponent fields are used). It has the advantages of simplified implementation at the expense of reduced precision. For an N bit LNS number, $(N, F)_{\mathcal{L}}$, the most significant bit is a zero flag, Z. Z is zero if the number is zero (since there is no log of zero), otherwise set. The next most significant bit is used for a sign bit and the rest of the number is the base 2 logarithm of the magnitude of the number to be represented in $(N-2, F)_{\mathcal{I}}$ two's complement fraction format. If E is the value of this two's complement fraction and S is defined

as for floating-point, then

$$L = \begin{cases} 0 & \text{if } Z = 0 \\ L = S \times 2^E & \text{if } Z = 1 \end{cases}$$

The LNS is good for applications where large dynamic range is need. The implementation of multiplication and division in LNS is very efficient compared to floating-point. Unfortunately, computing addition and subtraction requires large lookup tables.

3.2.4 Elementary Functions

In many systems, there is a need to compute elementary functions such as *sin*, *log* and *exp*. To achieve the high throughput requirements in hardware accelerators, table lookup methods are used instead of iterative algorithms as in software implementations. The main idea behind the table lookup approximation algorithms is using the Taylor Expansion. If a function $f(x)$ has continuous derivatives up to $(n+1)^{th}$ order, then

$$\begin{aligned} f(x) &= f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \cdots \\ &\quad + \frac{f^{(n)}(a)(x-q)^n}{n!} + R_n \\ &= \sum_{i=0}^{n} \frac{f^{(i)}(a)(x-a)^i}{i!} + R_n \end{aligned} \tag{3.1}$$

where

$$\begin{aligned} R_n &= \int_a^x f^{(n+1)}(u)\frac{(x-u)^n}{n!}\mathrm{d}u \\ &= \frac{f^{(n+a)}(\xi)(x-a)^{n+1}}{(n+1)!} \quad \text{for} \quad a < \xi < x \end{aligned}$$

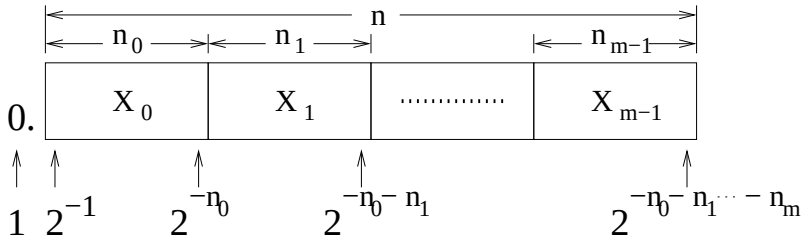
To reduce the required hardware resources and/or computation power, only the first few terms in the Taylor series are used to approximate the function. The selection of a will affect the error introduced and a carefully selected a can be used

to introduce symmetry in the lookup table as explained later. If $a = 0$, the series is call a MacLaurin Series.

The Symmetric Table Addition Method (STAM) [SS99a] method was developed for this approximation. The STAM uses the first two terms of the Taylor series to approximate a function $f(x)$ as $\widetilde{f(x)}$ [SS97]. In the STAM, a set of lookup tables are constructed and the precision of the output is maximized.

Assume that the n-bit input, x, of the function to be approximated ranges in $[0, 1)$. It is first partitioned in m segments as shown in Fig 3.1 where $x = \sum_{i=0}^{m-1} x_i$. In the description below, we follow the original STAM notation in which x_i is an n-bit number with all bits in other segments are masked to zero. It is different from the i^{th} digit notation in fixed-point representation in Section 3.2.1.

Figure 3.1 Input partition of STAM.



The ranges of x_i are shown here:

$$\begin{aligned} 0 &\leq x_0 \leq 1 - 2^{-n_0} \\ 0 &\leq x_i \leq 2^{-p_{i-1}} - 2^{-p_i} \end{aligned} \tag{3.2}$$

where $p_i = \sum_{k=0}^{i} n_k$.

We then select mid points in the ranges of x_i:

$$\delta_i = (2^{-p_{i-1}} - 2^{-p_i})/2 \tag{3.3}$$

Let $a = x_0 + x_1 + \sum_2^m \delta_i$ and use the first two terms of Taylor Expansion:

$$\begin{aligned}\widetilde{f(x)} &= f(x_0 + x_1 + \sum_2^m \delta_i) + \\ &\quad f'(x_0 + \delta_1 + \sum_2^m \delta_i)(\sum_2^m x_i - \sum_2^m \delta_i) \\ &= a_0(x_0, x_1) + \sum_2^m a_{i-1}(x_0, x_i) \end{aligned} \tag{3.4}$$

where

$$a_{i-1}(x_0, x_i) = f'(x_0 + \delta_1 + \sum_2^m \delta_k)(x_i - \delta_i) \quad 2 \le i \le m$$

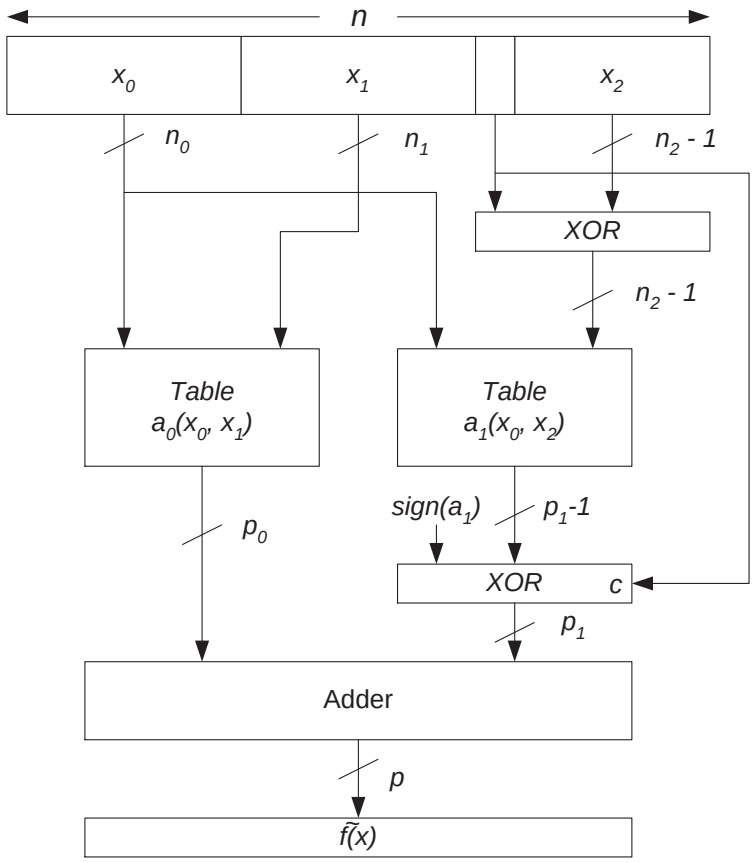
Functions a_i are evaluated by table lookup method with much less entries in the table compared with direct lookup for function $f(x)$. The final step is to sum the outputs of all the small tables as the approximated result.

The number of entries in table a_i is $2^{n_0+n_i}$ in a direct implementation. This size can be reduced by half using the symmetric nature of the table. First, we notice that $2\delta_i - x_i$ equals the bitwise inversion of x_i. This is obvious by listing all the binary patterns of x_i. Then we notice that $a_i(x_0, 2\delta_i - x_i)$ equals the bitwise inversion of $a_1(x_0, x_i)$. This can be shown by replacing x_i by $2\delta_i - x_i$ in $a_i = f'(x_0 + \delta_1 + \sum_2^m \delta_k)(x_i - \delta_i)$.

From the above two properties, the table can be reduced to half its original size. Only the bits from the $p_i + 1^{th}$ to the $p_{i+1} - 1^{th}$ position are used to index the lookup table. The ${p_i}^{th}$ bit is used the transform the index and result. The transform is simply the bitwise XOR of the index and the result with the ${p_i}^{th}$ bit.

Full details including table size reduction achieved and error analysis can be found in [SS99a]. A simplified STAM design using only three segments is shown in Figure 3.2.

Figure 3.2 Structure of simplified STAM with three segments.



3.3 Overview of CAST

The CAST system is a framework for building optimized arithmetic and logic circuits in hardware in which circuits are treated as objects interconnected by wires (which are also objects). Object-oriented features of the C++ programming language are used to allow module generators to interrogate objects for information such as their state and delay. Simulation and generation of synthesizable VHDL code can be performed by direct execution of the program. On top of this environment, a module library which provides a computer arithmetic scheme that is independent of numerical representation, number format and operators is available. The underlying circuit description is a structural one built from primitive elements.

To use the CAST system, developer instantiates objects from the CAST library and assign values to these objects. The C++ language is used in the CAST system.

In the CAST system, hardware components are modeled as C++ objects which have configuration attributes. By controlling these attributes, users can modify the datapath without concern as to the internal structure and interface of the circuits. The CAST system also provides an interface to simulate the constructed circuit in software level. This can verify the functional correctness of the design in its early stages. The resulting circuit is generated in structural VHDL codes which can be passed to the hardware vendor's tool chain directly.

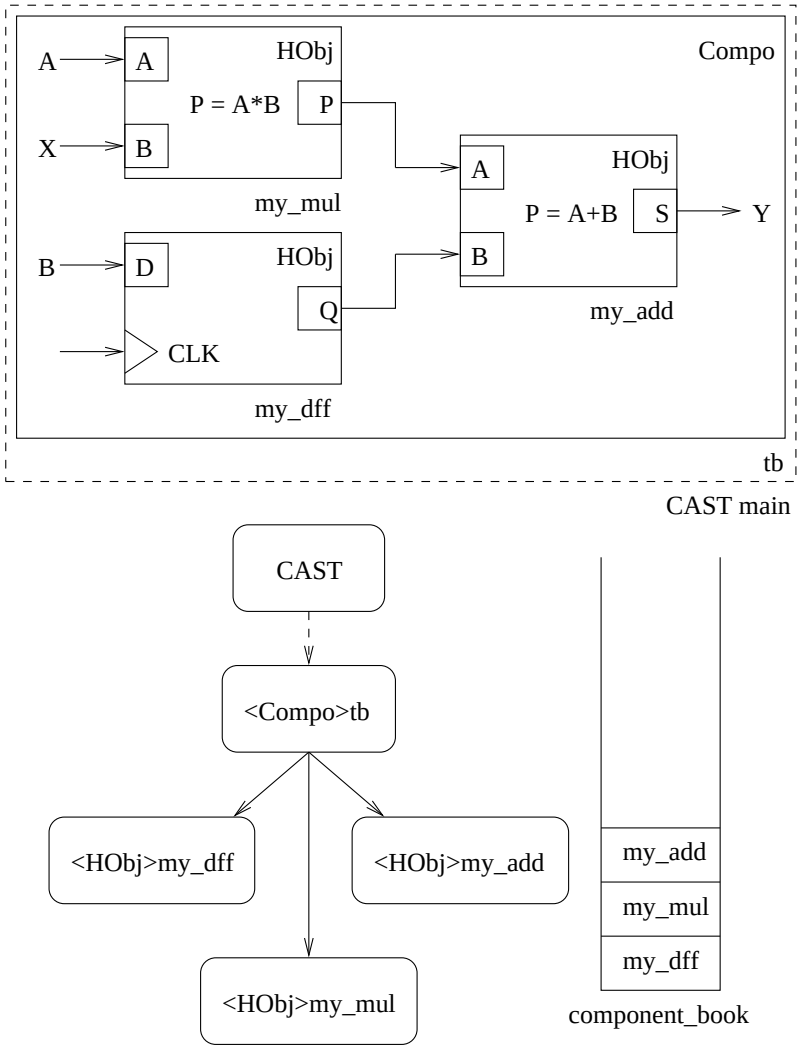
The CAST system also has a built-in simulation feature. In addition, search algorithms are embedded in the system to obtain optimized results for both individual modules and the overall design.

3.3.1 Implementation

Two libraries are used in CAST. One is utility library which is responsible for simulation and rendering of the circuits. The other is a primitive module library which consists of logic gates, adders, multiplexers, registers, etc. They can be connected together to form arbitrary designs and a circuit is modeled as a graph of interconnected objects. An example of a design to compute $y = ax+b$ is given in Figure 3.3. In this example, the testbench module *tb* includes three primitive modules: *my_mul*, *my_dff* and *my_add*. A component booker, also shown in the figure, is responsible for logging the creation of all primitives. In the object hierarchy, the composite module *tb* is called by the top level CAST system and is the parent of all three submodules. When the parent is to be simulated/rendered, all underlying children are simulated/rendered automatically.

Two methods are used to simulate a circuit: `sim_clk()` for registering values at clock edge; and `sim_eva()` for the combinational parts of the circuit. For primitive modules, the `sim_eva()` method is a set of expressions relating the outputs to

Figure 3.3 Example circuit and the object hierarchy.



the inputs. The `sim_eva()` method in a composite module calls the `sim_eva()` methods of the submodules iteratively according to dependencies derived from the interconnection graph. When the `sim_eva()` method of the composite object returns, the circuit is in a stable state and the value of any intermediate signal can be examined.

The simulation function of the CAST system helps designers to debug logic at a software level in the early stages of development. For primitive modules, it is the library designer's duty to ensure the simulation behave the same as that of the generated VHDL circuit. CAST will ensure the consistency between the simulation and implementation for designs formed from an interconnection of primitives. Writing a testbench is also easier since the stimuli can be created using standard C++ functions.

The following example creates the adder object of Figure 3.3, performs a simulation and generates a VHDL description and testbench. "*my_add*" will be the instance name of the adder and the ports *A*, *B* and *S* will be generated automatically.

```
// create adder
my_add=new Add_n("my_add",2*n);
// connect I/O
connect(my_mul->P, my_add->A);
connect(my_dff->Q, my_add->B);
...
// simulate 1 clock cycle
tb.sim_clk();
// print out result
sim_result(add->S);
// generate VHDL (including testbench)
tb.gen();
```

When a module is created, the constructor first saves a local copy of the configuration, e.g. the adder width $2n$. Then the `circuit()` method is called to construct the circuit. Finally, the current object is registered to its parent.

To generate the VHDL code for a circuit, the `gen()` method is used. In this method, the I/O ports are first created, and then the components, their instances and interconnections are generated in a manner which avoids forward references.

3.4 Arithmetic Operator Library

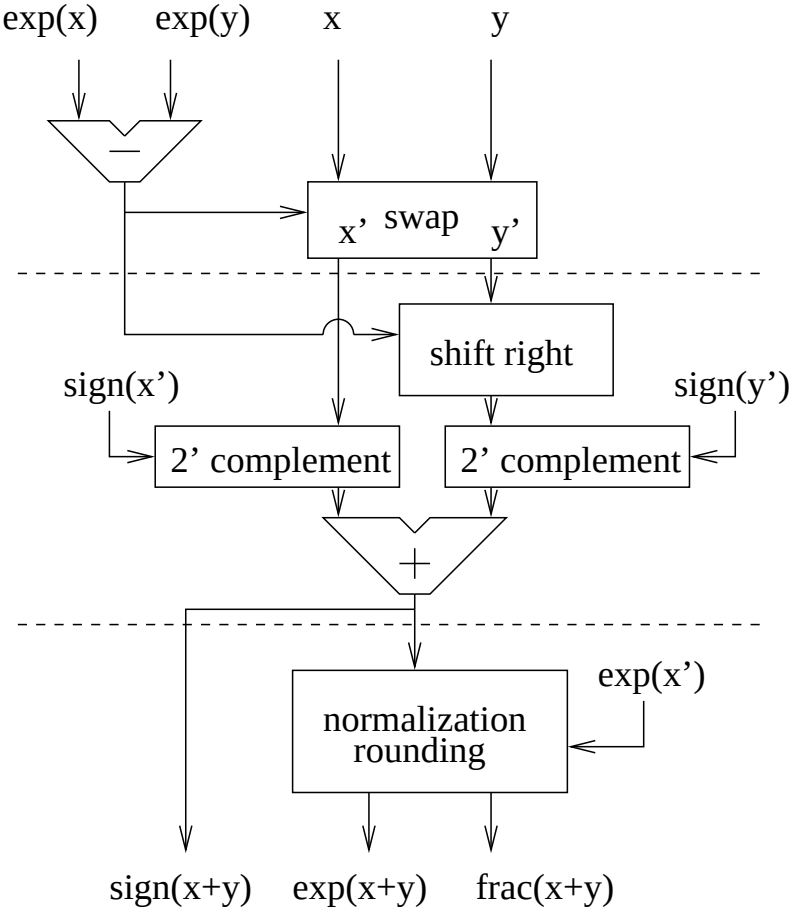
CAST was designed to be extensible with a view that it can be used to support many different number systems, arithmetic operators and implementation schemes. In the current prototype, the fixed-point, floating-point and LNS number systems can be used and the operators supported are addition, subtraction, multiplication and $x^{-3/2}$, those being required for the N-body problem.

The implementation of the $+$, $-$ and $\times$ operators for the fixed-point system follows the standard two's complement integer methods. A common ripple carry adder/subtracter using the fast carry chain was used for addition. Different addition schemes such as carry select and carry lookahead for long wordlengths can be integrated into the CAST system by overriding the `gen()` function of this operator.

The input/output format and precision of the addition/subtraction fixed-point operators are the same and no pre/post-processing is required. In the case of multiplication of two $(N, F)_{\mathcal{I}}$ two's complement fractions, an $(2N, 2F)_{\mathcal{I}}$ result is obtained. In CAST, the operators default to using the same format for inputs and outputs and so in order to convert the result back to $(N, F)_{\mathcal{I}}$ format, it must be scaled by 2^{-F} and the least significant N bits used.

The floating-point operators are implemented in a manner similar to the IEEE 754 standard [IEE85] except that Not-a-Number (NaN) and denormalized numbers are not implemented. The round-to-nearest mode is used for all operations and the size of exponent and fraction is parameterized.

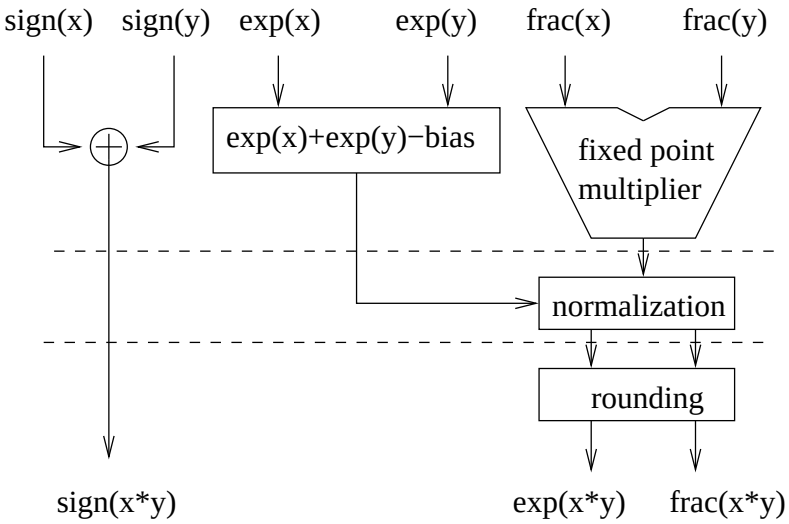
Figure 3.4 Datapath of the floating-point adder.



The floating-point adder accepts two inputs $f1$ and $f2$ and returns the sum in the same format. The implementation is pipelined with a latency of 3 cycles. In the first cycle, $f1$ and $f2$ are swapped if the exponent of $f1$ is smaller than that of $f2$, and the difference between the exponents of $f1$ and $f2$ are calculated. In the second cycle, the significands are aligned. the intermediate sum is computed and the position of the leading one is determined using a priority encoder. In the final cycle, the result is normalized and rounded and the exponent corrected to produce the output.

The floating-point multiplier accepts operands $f1$ and $f2$ and returns the product in the same format as the inputs. In the first cycle, the sign bit is calculated and the intermediate exponent and product are also computed. In the second cycle, the

Figure 3.5 Datapath of the floating-point multiplier.



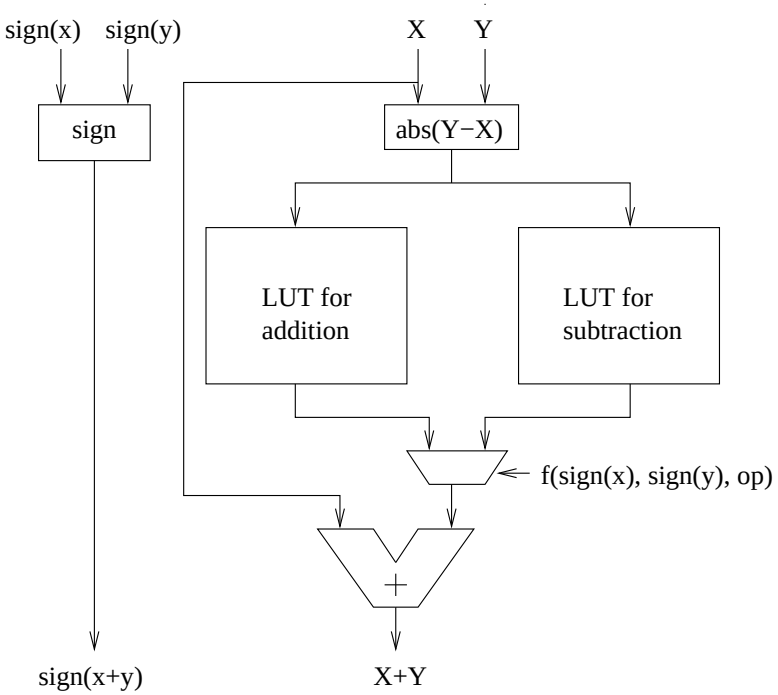
intermediate result is normalized. In the third cycle, the result is rounded to produce the output.

The LNS implementation used in CAST is based on the open source code of the Aremaire project [aEL06]. The LNS operations accept and produce numbers in the format described in Section 3.2. The multiplication in LNS is performed by summing the two exponents and setting the zero flag appropriately. The sign bit is computed as the XOR of the sign bits of the two inputs as in the floating-point case. The LNS addition of $X = \log_2(x)$ and $Y = \log_2(y)$, `ADD_l`, is computed by making use of the following identity [Kor93]:

$$\begin{aligned} Z &= \log_2(x \pm y) = \log_2\left(x(1 \pm y/x)\right) \\ &= \log_2(x) + \log_2(1 \pm 2^{\log_2(y/x)}) \\ &= X + \log_2(1 \pm 2^{Y-X}) \end{aligned}$$

The implementation uses $Y - X$ to index a lookup table which generates $\log_2(1 \pm 2^{Y-X})$, and this table is constructed in Xilinx devices using distributed 16×1 LUT RAM rather than BlockRAM. When the y input is negative, a subtraction must be performed and thus the `ADD_l` module must include tables for both $1 + 2^{\log_2(Y-X)}$ and $1 - 2^{\log_2(Y-X)}$. Figure 3.6 shows a block diagram of the datapath for the LNS

Figure 3.6 Simplified datapath of the LNS addition operation, `ADD_l`.

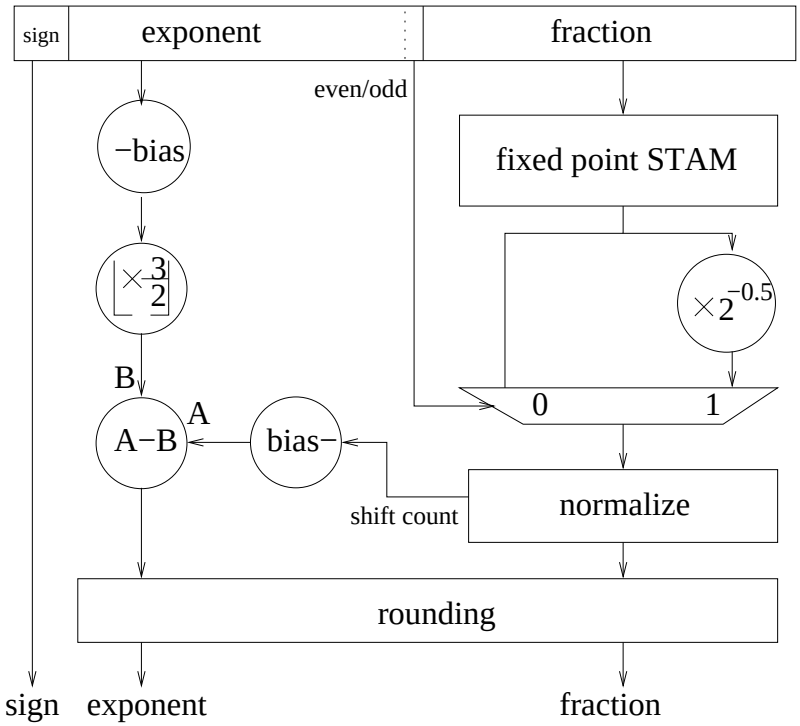


addition operation. In the actual implementation, extra swapping logic is included for the case that $Y - X$ is negative. To perform a subtraction, the sign bit of the second input is inverted prior to being passed to the addition module.

A class implementing the Symmetric Table Addition Method (STAM) [SS99b], which can approximate any twice differentiable function is available to construct operators such as $x^{-3/2}$ [HTY+03] which is useful in the N-body simulation. STAM offers very good flexibility but the tables can become large if high accuracy is required.

In the N-body force pipeline application example in Chapter 7, function $f(x) = x^{-3/2}$ is needed to be evaluated. Computing the function $x^{-3/2}$ in LNS is done by using shift and add operations to multiply the LNS number by -1.5. The fixed-point implementation is computed directly using STAM. For floating-point, STAM can only be directly applied to the significand part of the number. If the number is represented by $x = (1.f) \times 2^E$ where f is the fraction and E is the exponent, the

Figure 3.7 Floating-point STAM datapath.



floating-point case can be handled using [HTY+03]:

$$f(x) = x^{-3/2} = ((1.f) \times 2^E)^{-3/2} = (1.f)^{-3/2} \times 2^{-3E/2}$$

A fixed-point STAM module for $x^{-3/2}$ is used to calculate $(1.f)^{-3/2}$. If the exponent E is even, multiplication by $2^{-3E/2}$ can be achieved by simply multiplying the input's exponent by $-3E/2$. If E is odd, $x^{-3/2}$ can be rewritten as: $(1.f)^{-3/2} \times 2^{-(\lfloor -3E/2 \rfloor + 1)} \times 2^{-1/2}$. In [HTY+03], a floating-point multiplication was used to handle the exponent of the odd exponent case. In the current design, a fixed-point multiplier, as shown in Figure 3.7, was used to optionally multiply by $2^{-1/2}$ and the $2^{-(\lfloor -3E/2 \rfloor + E_0}$ term (where E_0 is the least significant bit of E) is added to the exponent. The new scheme results in a more compact circuit and eliminates the need for a normalization step before floating-point multiplication. To improve throughput, pipeline registers were inserted and a 3 clock cycle latency introduced.

A set of modules for converting between number systems was also developed.

When converting from floating to fixed-point number systems, a shift amount is computed from the exponent. The fractional part (and the implicit '1' of the significand) will be shifted according to the shift amount. The final result should be two's complemented if the sign bit is set. When converting from fixed to floating-point, the absolute value of the number is passed to a priority encoder to find the position of the most significant set bit. Then the number is shifted to form the significand and the exponent calculated. For conversion from LNS to the floating-point system, the significand, $2^{frac(LNS)}$, where $frac(LNS)$ is the fractional part of the LNS number, is computed using a lookup table. The integer part of the LNS goes to the exponent after addition of the bias. In conversion from floating-point to LNS, the integer part of the LNS is formed by subtracting the bias from the exponent. The fractional part of the LNS is computed by a lookup table of the $\log_2()$ function.

For all three number systems, operators may cause overflow/underflow. In the current hardware implementation, these special cases are not handled.

3.5 Unified Arithmetic Operator Class

A class of general arithmetic operators was developed. After the description of a circuit is constructed, the library provides an easy way to change the configuration of arithmetic operators in the circuit. Configuration of an operator includes the number system, the number format and latency allowed. This information is supplied as parameters when the object is created. For example, to use an 8-bit exponent and 23-bit fraction floating-point adder with 3 clock cycle latency, the module is created as:

```
ADD_f("my_add", this, 8, 23, 3);
```

The operator interface for different number systems is unified in a single class: `CAST_ADD`, `CAST_MUL`, etc. The class includes operators from the parameterized fixed-point, floating-point and LNS libraries. As an example, the following code

Table 3.1 Summary of arithmetic operators available in the current CAST system.

	ADD	SUB	MULT	$x^{-3/2}$
Fixed Pt.	ADD_n() width	SUB_n() width	MUL_n() width	POWM15_n() width, segments, guard bits
Float Pt.	ADD_f() exp, frac	SUB_f() exp, frac	MUL_f() exp, frac	POWM15_f() exp, frac, segments, guard bits
LNS	ADD_l() int, frac	SUB_l() int, frac	MUL_l() int, frac	POWM15_l() int, frac
Unified	CAST_ADD_n() a, b, ns	CAST_SUB_n() a, b, ns	CAST_MULT_n() a, b, ns	CAST_POWM15_n() a, b, ns

Key - segments, guard bits: from configuration file according to width. **exp, frac**: width of exponent and fraction. **int, frac**: width of integer part and fractional part of exponent. **ns**: number system selection. **a**: width of fixed-point, exp of floating-point, int of LNS. **b**: frac of floating-point, frac of LNS.

segment creates an LNS adder:

```
CAST_ADD("my_add", this, 8, 23, LNS);
```

Table 3.1 is a summary of the available arithmetic operators for the different number systems as well as the attributes bounded to these operators.

A latency parameter may be used to select different implementations. User can query the latency of any object using the `delay()` method. When different operators for different number systems and/or precision are used, their latency may change e.g. fixed and floating multipliers may have different latencies. When assembling a datapath, the user is responsible for matching the latency of the operators by inserting delay elements.

The unified operator class thus provides a consistent interface to the arithmetic library and encapsulates the internal details of their semantics and implementation in a manner that one can use the library with minimal knowledge about its implementation.

3.6 Summary

In this chapter, we presented background on number systems and the STAM algorithm for approximating elementary functions. An understanding of these arithmetic systems is necessary for building hardware accelerators for simulation systems which involve large numbers of arithmetic operations.

We also introduced the CAST framework and showed its the features and internal structure. The operators for different number systems and approximation algorithms are implemented within CAST. This process captures computer arithmetic knowledge. Users can use the operators through a unified interface. They can be used without detailed knowledge of their implementation. Furthermore, as an generic interface is presented, users do not need to specify the arithmetic system to use. This can be inferred by optimization, the computer deciding which system is best suited to a given application.

Chapter 4

Mullet - A Multiplier Generator

4.1 Introduction

Multipliers are one of the most important operators in simulation applications. Although a wealth of knowledge exists about parallel multiplier design, the best architecture is dependent on the desired multiplier size and the technology which is used. For example, for a small multiplier, the partial products (PPs) might be best generated using a simple AND structure and ripple carry adders used to accumulate them. For larger sizes, a Wallace tree might be faster. Furthermore, the crossover point where the Wallace tree is faster depends on the VLSI technology used as well as whether the design is on an application specific integrated circuit (ASIC) or a field programmable gate array (FPGA).

Many different parallel multiplier architectures have been proposed in the literature (e.g. [Kor02, EL04]). High speed multipliers typically reduce the number of PPs in the partial product generator (PPG) stage via Booth's encoding and reduce the number of logic levels in the partial product summer (PPS) using tree structures. Different kinds of adders can also be used in the PPS stage. Some FPGA devices have hardwired dedicated multiplier units and practical multiplier module generators should use them when appropriate. Given the bewildering number of choices, it is difficult even for an expert to find an optimal multiplier without investing a large amount of time to the task.

In this chapter, we describe an automatic multiplier generator called *Mullet* (MULtpLiEr Tool) that can generate multipliers which are combinations of simpler primitive elements. Mullet is built on the CAST framework and becomes part of the CAST system as a library generator. A search through the different combinations can easily explore tradeoffs. Furthermore, by synthesizing a number of designs and recording their performance, Mullet can determine its own timing, area and power model parameters and calibrate itself. To the best of our knowledge, no other multiplier module generator is able to consider all of these issues in a unified manner. This will demonstrate how the CAST system can be used to balance the performance and resource cost and utilize special features in the target platform. We apply this system to the generation of parallel multipliers for Xilinx Virtex FPGAs [Xil04b] and show, as in the result chapter, that the multipliers generated by our tool are better than those of the Xilinx CoreGenerator and XST tools for large multiplier sizes.

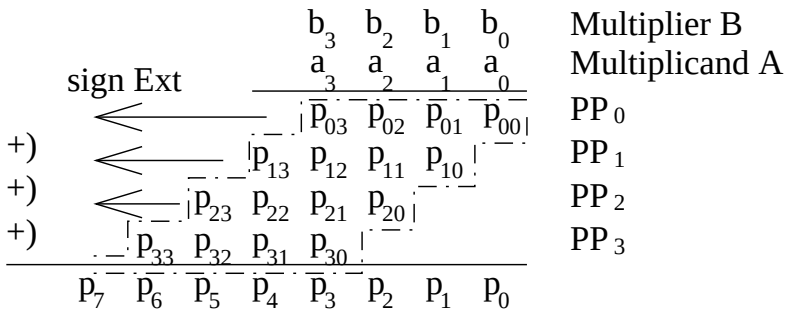
The rest of this chapter is organized as follows: In Section 4.2, we present parallel multiplier architectures which are used as primitive elements in our tool. In Section 4.3 we describe the features and architecture of Mullet. The implementation details and performance results are show in Section 4.4. Finally we draw conclusions about this work in Section 4.5.

4.2 Parallel Multiplier Structure

In this section we introduce the generation and optimization of parallel multiplier constructs in the CAST system.

We assume that inputs are in two's complement format and we perform parallel signed multiplication of an n-bit *multiplicand* A with an m-bit *multiplier* B. The resulting product P is $n + m$ bits in size. Figure 4.1 shows the basic architecture of a 4-bit parallel multiplier. The multiplier can be broken down into two independent units, the PPG and PPS.

Figure 4.1 A 4-bit parallel multiplier showing the partial product generator and summer.



4.2.1 Partial Product Generators (PPGs)

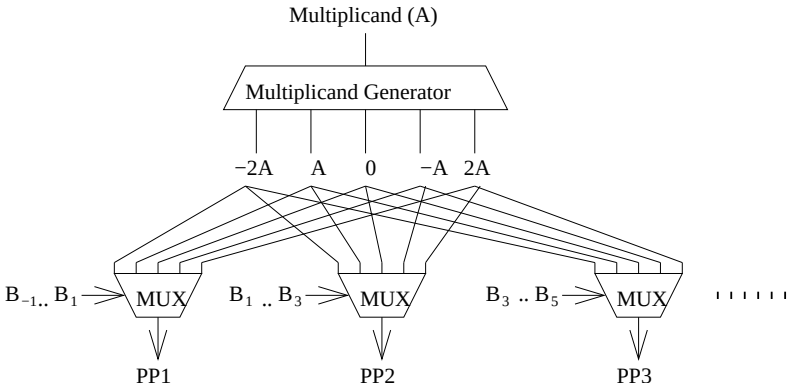
AND scheme

In Figure 4.1, the partial products $PP_0 - PP_3$ are computed by forming the bitwise AND of b_i with A, i.e. $PP_i = b_i A$. Using this method, the number of PPs generated is m and the length of each PP is n. We call this method for generating the partial products the *AND* scheme. For signed multiplication, the PPs should be sign extended as shown in the figure.

Modified Booth Encoding (MBE)

The modified Booth's algorithm [Boo51] considers multiple bits of B. If two bits are considered (radix-4), the partial products are generated according to a coding table. Figure 4.2 shows the circuit for the modified Booth encoding (MBE) PPG, with a lookup table being used to produce the appropriate multiplexor selection according to three bits of multiplier B. PP_i is formed from bits B_{2i+1}, B_{2i} and B_{2i-1} ($B_{-1} = 0$) so only $\lceil m/2 \rceil$ partial products are generated, half as many as for the AND scheme. The scheme can be generalized to higher radixes, a radix-8 MBE scheme requiring only $\lceil m/3 \rceil$ partial products. This is, of course, at the expense of a more complex partial product generation scheme. Variants of Booth's algorithm can further improve performance by introducing more complicated encoders [Mac61] and conditional-sum adders [YJ00].

Figure 4.2 Radix-4 MBE circuit.



4.2.2 Partial Product Summers (PPSs)

Weighted Sum (WS)

The PPs produced by a PPG must be summed in order to form the final result. A straightforward way to do this is to use an array of adders to form the weighted sum of the PPs as show in Figure 4.1.

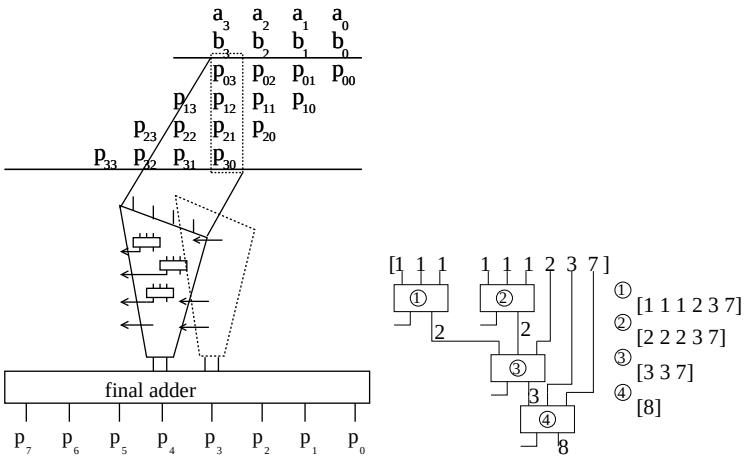
The array can be constructed using simple carry ripple adders (CRAs) or faster schemes such as carry look-ahead or carry select adders. For ripple adders, the critical path is the Manhattan distance from the LSB of the first PP to the carry out from the MSB of the last PP. This delay can be modeled as a carry chain of length $n + m$ and is shown as the dotted line in Figure 4.1.

Three Dimensional Method (TDM)

The three dimension method (TDM) proposed in [OVL96] and [SMOR98] uses compressor trees to sum the partial products and a delay balancing scheme so that signal delays are minimized in a globally optimal manner. For each weight, trees are used to produce two equal weight bits of output, shown as vertical lines connected to the final adder of Figure 4.3(a).

In general, the delays from different inputs to the outputs of a compressor may be different. An optimal method for interconnecting the compressors to reduce the global delay for the TDM has been reported by Stelling [SMOR98]. Unfortunately,

Figure 4.3 TDM model and 3-greedy scheme.



(a) TDM model for PPS.

(b) Three-greedy scheme for 9 inputs example.

the computational requirements are extremely high, making this method unsuitable for schemes in which a search over many different multipliers is applied. Instead, we employ the three-greedy algorithm [OVL96, SMOR98] which produces multipliers of similar quality but is several orders of magnitude faster. The implementation of the algorithm can be described by the simplified pseudocode in Algorithm 1.

Figure 4.3(b) shows an example of using the three-greedy algorithm to compress 9 inputs. Circled numbers represent the order of compressor generation and numbers beside the signals represent the delay of the line. The delays of the inputs to the compressors are (1,1,1,1,1,1,2,3,7). The updated available input delay list after each compressor was generated are also shown on the right. It can be seen that inputs which have large delay are placed in positions with minimum delay to the output. The technique just described uses 3:1 compressors but this can be generalized to deal with arbitrary compression ratios.

Algorithm 1 Simplified pseudocode for the TDM method.

```
create blist[2n][]; // store bits with same weight
initial blist with bits in partial products with same weight;
for (i := 0; i < 2n; i++) do
   sort(blist[i]);
end for
for (i := 0; i < 2n; i++) do
   while size_of(blist) > 2 do
      new compressor X;
      connect first k bits from blist[i] to input of X;
      remove first k bits form blist[i];
      evaluate output delays of X;
      put X_Sum into blist[i];
      put X_Cout into blist[i + 1];
      sort(blist[i]);
   end while
end for
```

4.3 Mullet Architecture

Mullet combines the primitive elements described in the previous section to create multipliers of arbitrary size. In this section, the architecture of Mullet is described in detail.

To isolate the PPG and PPS parts of a multiplier circuit, we create a generalized *PP* object in CAST. A *PP* object represents a partial product which has no logic or circuitry associated with it. i.e. it simply contains the signals associated with a particular partial product. The attributes associated with the *PP* object include the weight of the LSB and the maximum delay from the primary input of the circuit, which is used in the TDM design. The weight information is used to ensure correct alignment in the PPS and the delay information is used for optimization of the circuit such as required in the TDM approach. *PP* objects are stored in a list when the PPG component is generated. This object-oriented implementation scheme provides a clean and uniform interface between the PPG and PPS and allows new algorithms and/or architectures be easily included.

Hardware Multipliers (HWMs)

Modern FPGA devices such as the Xilinx Virtex-II have dedicated hardware signed multipliers of fixed input size [Xil04b]. These do not use the logic resources of the FPGA and are usually faster than a similar multiplier built from logic resources. The HWM element is represented as a primitive object in CAST. For the Xilinx Virtex II devices considered in this work, the multiplier is 18×18-bit signed multiplier which can be used as a 17×17-bit unsigned multiplier. Larger multipliers can be constructed from HWMs.

In order to break a large multiplier into smaller ones the system first partitions the multiplier and multiplicand into several smaller bit segments. If the input segment includes the MSB, it is signed extended to 18-bits. Otherwise, a 17-bit (or smaller) unsigned HWM is used. For maximum speed and minimum logic utilization, a HWM should be used wherever possible. Unfortunately, the number of HWM resources on an FPGA device is limited and there are often situations in which the user may want to save some of the HWMs for other parts of the design. Figure 4.4 shows examples of the assignment when only six HWM units are available. Example $a)$ is a random assignment with longer delay compared with $b)$ which follows the assignment method described. In Mullet, the user can specify how many HWMs to use. The system will assign the HWMs to the least significant segments first and thus reducing the critical path delay of the circuit.

If the size of a sub-multiplier is small, a simple AND/WS multiplier is smaller and faster than a HWM. This sub-multiplier occurs frequently since the inputs are not always a multiple of the width of HWM unit. A calibration procedure was created to find the size of multiplier under which the AND/WS scheme is preferred.

Modified Booth Encoding

Mullet currently supports radix-4 and radix-8 MBE primitives which are called MBE3 and MBE4 respectively since they scan 3 and 4 bits at a time. In the MBE3 example, the $2A$ output is generated by shifting the input A and has no logic delay.

Figure 4.4 Assignments of 6 HWM units to a partitioned design. $a)$ Bad assignment with longer delay. $b)$ Good assignment with shorter delay.

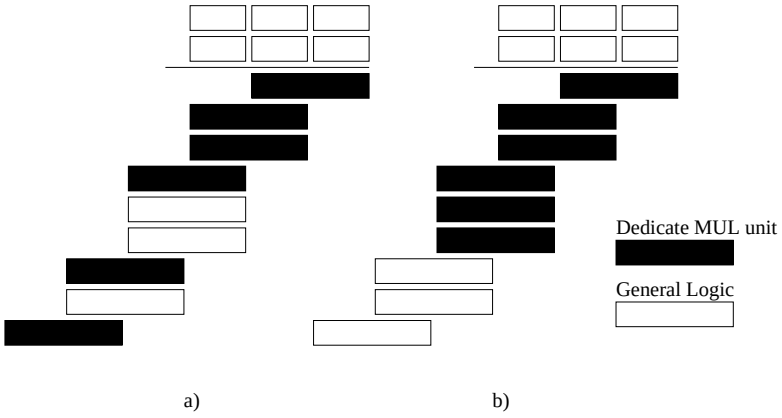
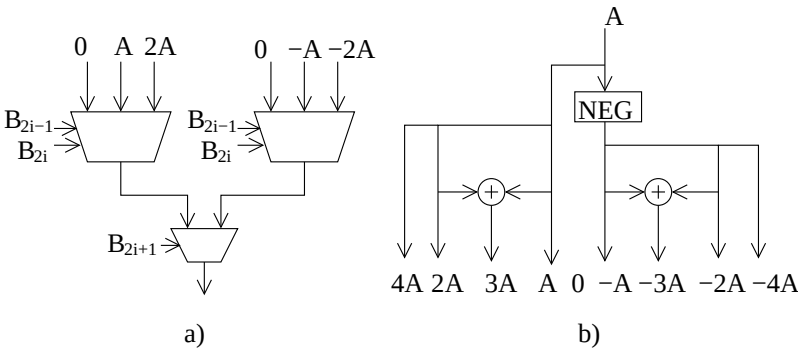


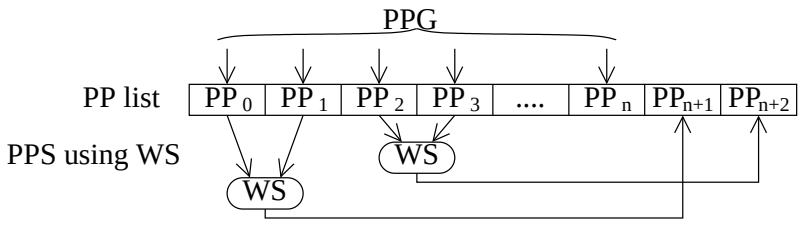
Figure 4.5 MBE components. $a)$ MBE3 MUX; $b)$ MBE4 multiplicand generator



Output $-A$ is generated by 2's complementing A and requires an n-bit adder. The $-2A$ output is generated by shifting the $-A$ value. The total cost of multiplicand generator is an n-bit adder in MBE3 and a 5-to-1 MUX. The PP generation for MBE4 is also shown in the figure.

For MBE4, the multiplicand generator needs to produce $\pm 3A$ and $\pm 4A$. The $\pm 4A$ is computed with a simple shift operation from $\pm 2A$. $\pm 3A$ requires one more n-bit adder level so the total delay introduced is two levels of n-bit adder. The MUX for MBE4 includes two MBE3 MUXes in parallel and an extra 2-to-1 MUX.

Figure 4.6 WS scheme of PPS.



Mullet will first generate the $\pm$ multiplies from the multiplicand. It then segments the multiplier B according the number of bits to be scanned (currently 3 or 4). The final step is to make connections to the MUXs.

Weighted Sum (WS)

The *weight_sum* object in Mullet will accept two PP objects and output a PP object. The circuit for *weight_sum* is dynamically generated in CAST according to the width and weight of the two inputs. The inputs will be appropriately sign extended and aligned before they are summed.

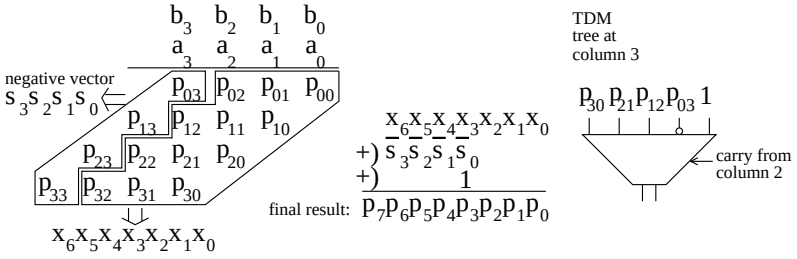
After the PPG circuit is created in the module generator, all PPs are available in a list *pp_list*. Mullet will first sort the list in ascending order of weight. The first two PPs are removed from the list and added to form a new PP which is appended to the list. This process is continued until there is only one PP left in the list which is the final product P of the multiplier. This process is illustrated in Figure 4.6.

Compression Tree

The most simple compressor is a $3:1$ compressor implemented as a full adder. There are different ways to implement the full adder which lead to different area and delay models. In [SMOR98], the full adder delays are modeled as an XOR gate count where the carry out delay is 1 XOR gate delay and the sum output is 2 XOR gate delays. In most FPGA architectures, this is not true due to their implementation using a 4-input LUT and fast carry logic.

We can build larger compressors by interconnecting standard $2:1$ and $3:1$

Figure 4.7 Signed multiplication for TDM.



compressors. CAST will make use of LUT4 and F5 primitives in the FPGA to optimize area and speed when implementing the high ratio compressors. The delay model for these compressors is determined by the number of levels of LUT required.

The original TDM algorithm was proposed for unsigned multiplication. We modified the algorithm to accept signed numbers. The final product for signed multiplication can be constructed by subtracting the negative vector formed from the MSBs of all *PP*s. We embedded this subtraction in the *PPA* by inserting inverters and adding one extra bit to the LSB column. The design is further optimized by starting from the first signed bit instead of the LSB as shown in Figure 4.7. By feeding the resulting vector as the TDM input, we can compute signed multiplication using the original TDM components with a maximum overhead of two extra input bit per each column. If there are two signed bits in a column, such as in the HWM *PPG* result, we use NXOR and NAND operators to produce a negated sum and carry for them. The negated sum will be fed into the current column while the negated carry will be used in the next column. Due to the nature of the *PPG*, the maximum carry length in forming the negative vector only across two columns.

Multiplier Generator

The multiplier generator accepts a set of configuration parameters as input and generate a multiplier. The PPG can be one of AND, MBE and HWM. The PPA can be either WS or TDM. The choices of PPG and PPA are independent.

To implement the TDM algorithm, the system is able to obtain delay and other

information from the circuit objects in the CAST system. Every object has its own delay model which is used to compute the maximum delay at each output. These delays are then propagated through the connections.

New primitive elements for PPG and PPS schemes can be easily added. After supplying a CAST module and the necessary timing and area model information, the new multiplier architecture can be registered in Mullet and will be available to the user to instantiate. The object-oriented nature and clean interfaces within CAST serve to hide unnecessary information.

4.4 Results

Multiplier performance for different input size using different schemes are shown Figure 4.8. All results are collected with the tools set to the highest optimization effort. The results were compared with the Xilinx CoreGen system as well as a multiplier directly generated using the "*" operator in XST on a Xilinx XC2V6000-6 FPGA. The correctness of a multiplier can be verified both by simulation in CAST by compiling the program with a C++ compiler and/or VHDL simulation. In the verification process, we exhaustively test all the possible inputs for a 8×8 multiplier for all possible configurations by comparing the results against software multiplications. Random input vectors were used to verify larger multipliers. In this section we present experimental results based on Xilinx FPGA devices. The VHDL codes generated by Mullet were first synthesized using the Xilinx Synthesis Tools (XST) and then implemented using the ISE 6.2i tools.

The delays are measured between input and output registers of the multipliers. The configurations shown in Figure 4.8 are optimized for speed. As shown in the table, the performance of the generated circuit is better then those from XST and CoreGen when the input width is large. In our experiments, circuits using TDM3 performed better for multipliers larger than 40 bits because of the reduced number of logic levels. Xilinx CoreGen can only accept input up to 64 bits, and so no

Table 4.1 Performance of 52x52 multiplier for all possible schemes. The speed is the minimum clock period in ns unit and the area is the LUT count.

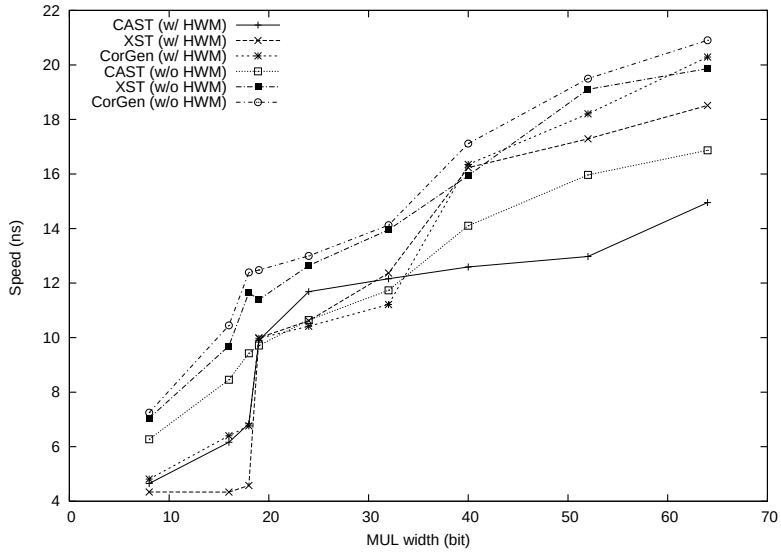
Configuration	speed	area	Configuration	speed	area
AND+WS	21.540	2935	MBE4+WS	25.035	6919
AND+TDM3+CRA	15.563	7869	MBE4+TDM3+CRA	18.963	9086
AND+TDM3+CSA	15.597	8060	MBE4+TDM3+CSA	18.761	8263
AND+TDM4+CRA	41.872	10977	MBE4+TDM4+CRA	58.868	8467
AND+TDM4+CSA	40.163	11111	MBE4+TDM4+CSA	57.606	8606
AND+TDM5+CRA	69.372	10672	MBE4+TDM5+CRA	63.443	8778
AND+TDM5+CSA	68.121	10768	MBE4+TDM5+CSA	63.617	8903
AND+TDM6+CRA	35.480	9903	MBE4+TDM6+CRA	58.848	8125
AND+TDM6+CSA	37.041	11535	MBE4+TDM6+CSA	58.519	8268
MBE3+WS	29.586	7033	HWM+WS	19.711	362
MBE3+TDM3+CRA	19.427	8384	HWM+TDM3+CRA	13.891	469
MBE3+TDM3+CSA	18.855	7390	HWM+TDM3+CSA	12.977	522
MBE3+TDM4+CRA	45.687	8406	HWM+TDM4+CRA	35.486	587
MBE3+TDM4+CSA	40.827	8625	HWM+TDM4+CSA	36.702	697
MBE3+TDM5+CRA	61.837	7641	HWM+TDM5+CRA	40.434	619
MBE3+TDM5+CSA	61.459	7614	HWM+TDM5+CSA	39.890	708
MBE3+TDM6+CRA	40.676	7184	HWM+TDM6+CRA	40.612	965
MBE3+TDM6+CSA	38.298	7312	HWM+TDM6+CSA	39.831	1238

comparison was made for multiplier larger then 64 bits. For the 19 bit multiplier, our tool uses 1 MULT18X18 HWM while the other two use 4 HWMs. The resulting speed is faster at the expense of requiring more LUTs.

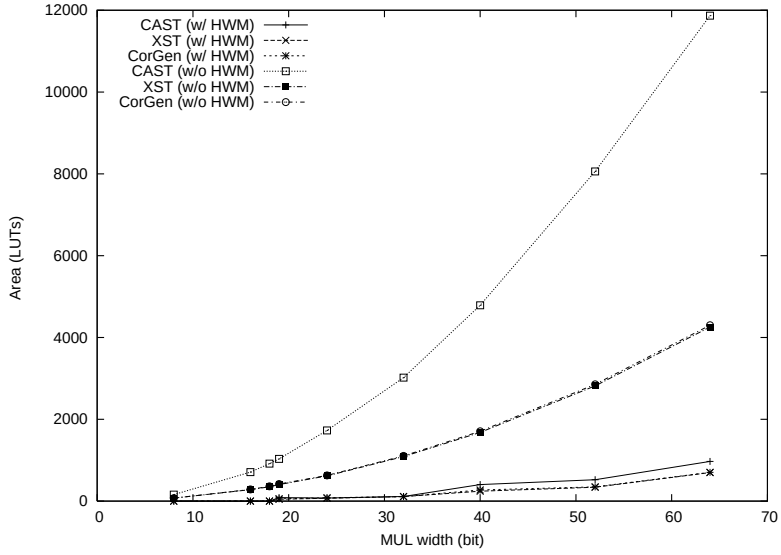
In practice, we often need to find out what is the best implementation scheme for a given sized multiplier. The user may wish to optimize for speed, area or both. Using Mullet a user can easily explore tradeoffs associated with different schemes. A 52x52 bit multiplier is used as an example and the results agree with our expectation for different configurations.

A $w \times w$ table, where w is the width of the HWM, records the feedback and helps making decision of the choice of implementation. When a decision between an AND/WS and HWM is made, the Mullet will use the current input size, $n \times m$, of the sub-multiplier to address an $w \times w$ table where w is the width of the HWM.

Figure 4.8 Performance of different multiplier schemes for different input sizes.



(a) MUL Speed.



(b) MUL Area.

Table 4.2 The calibration table of Virtex II FPGA

	1	2	3	...	18
1	A/W	A/W	A/W	...	A/W
2	A/W	A/W 2.847ns	A/W	...	A/W 3.822ns
3	A/W	A/W	HWM 4.352ns	.	HWM
. .	. .	. .	. .	. .	. .
18	A/W	A/W 3.822ns	HWM	...	HWM

The entries in the table can be one of $\{$AND/WS, HWM, empty$\}$. If the entry is empty, Mullet will call external programs to implement the $n \times m$ multiplier and compare the speed with the HWM, selecting the better of the two and recording the choice back in the table. Furthermore, multipliers of size smaller than $n \times m$ will be marked to be AND/WS. Similarly, if the HWM is better, all multipliers of size larger than $n \times m$ will be marked to be HWM. The method of caching decisions obtains the most accurate information when needed and saves computation time. Table 4.2 shows an fully marked calibration table with delay information of AND/WS scheme for the XC2V6000-6 device.

4.5 Summary

In this chapter, we presented a system that can be used to generate different parallel multiplier structures based on the CAST framework. The multiplier generator utilize the built in area and speed estimation functions in CAST object to evaluate the generated circuits. These features allow different search methods to optimize multiplier circuits automatically. Even without the searching algorithms, it can be used to explore the complete design space in an efficient way.

By isolating the PPG and PPS part, it shows that different implementation schemes

of a operator can work together within the CAST framework smoothly. Both arithmetic knowledge and FPGA specific features are considered when selecting a suitable scheme. It is shown that CAST can be used to provide multiple levels of optimization control while hiding the hardware details with a unified interface.

Chapter 5

A Novel Random Number Generator

5.1 Introduction

The Random Number Generator (RNG) is an important primitive widely used in simulation as an input source. A physical random number generator (PRNG) derives its output from a physical noise source and its output is nondeterministic in nature. Given the importance of random number generation, surprisingly few hardware implementations of PRNGs have been reported. There are three commonly used techniques in the literature, namely oscillator sampling, direct amplification and discrete time chaos. In the oscillator sampling approach, period variation (i.e. oscillator jitter) in a low frequency clock of low quality factor (Q) is exploited by using it to sample a high frequency clock. The direct amplification technique digitizes thermal or shot noise, using an amplifier and comparator. Finally, chaotic systems are also used to produce PRNGs.

In this chapter, a high performance PRNG which uses a physical random source to control two linear feedback shift registers in a manner similar to that of an alternating step generator (ASG) stream cipher is proposed. This approach combines some of the benefits of both approaches and achieves high throughput, small area and good randomness properties. The same approach could be applied to combine other weak physical random number generators with a stream or block cipher.

In 1984, Fairfield, Mortenson and Coulthart [FMC84] developed the first integrated RNG based on oscillator phase noise. In the design, a high frequency oscillator was sampled using a low frequency oscillator. After removing duty cycle biases via a parity filter, the flip flop output was fed into a linear feedback shift register (LFSR) based scrambler. The design generated 27 bps using a 1000 Hz low frequency clock. The Intel RNG is part of the Intel 8xx chipset starting with the Intel 810 and is implemented in the Intel 82802 Firmware Hub Device (FWH). It uses amplified thermal noise to drive a voltage controlled oscillator (VCO), and oscillator sampling is used to detect the phase noise of the VCO to produce a digital random source [JK99].

We have previously reported an FPGA design which employs oscillator sampling [TLL03]. In this design, a low frequency RC oscillator was used to sample an internal high frequency clock. The design requires only three external passive components to control the time constant of the RC oscillator. Phase noise in the RC oscillator produced randomized output which was filtered through a parity filter. A disadvantage of this approach is that the output rate is limited by the speed of the RC oscillator and in order to pass the NIST and Diehard tests, the maximum rate was limited to 4.7 kbps. The only other FPGA based implementation was one by Fischer and Drutarovsky [FD02] which used a variation of oscillator sampling. Their design was based on the randomness of jitter in an analogue phase locked loop (PLL) and a decimator was used to ensure that at least one sample affecting jitter was included in every output data. The design was implemented on an Altera APEX EP20K200-2X FPGA with a 33.3 MHz external clock. With an 88.245 MHz internal clock, it can generate 69 kbps. For FPGAs such as the Altera APEX E and APEX II devices which have internal PLLs, this approach requires no external components. The disadvantage of this approach is that few FPGAs have this feature.

Physical random number generators based on chaotic systems can lead to very compact CMOS implementations. In 2001, Stojanovski *et al.* [SPK01] implemented an analog chaos-based RNG in a 0.8 μm CMOS process utilizing switched

current techniques. The estimated output bit rate of this design was 1 Mbps. Andrea Gerosa *et al.* [GBP01] also implemented a RNG based on a chaotic system. Their design with a pipelined ADC (analog-to-digital converter) occupied $2.2\ mm^2$ silicon area and the design can generate 8-bits of data using a 20 MHz clock. Petrie *et al.*, combined oscillator sampling, direct amplification and discrete time chaos to produce an analog VLSI chip which was robust to power supply noise and substrate signal coupling [PC00]. Implemented in $2\ \mu m$ CMOS, the chip produces random numbers at 1.4 Mbps. The design occupied an area of $1.5\ mm^2$ and dissipated 3.9 mW of power.

In comparison to the approaches described above, the design presented in this chapter, an output rate of 400 Mbps was achieved on a Xilinx XCV300–8 devices and the design occupies approximately 130 Xilinx Virtex slices. Furthermore, it was implemented entirely in digital technology with no external components.

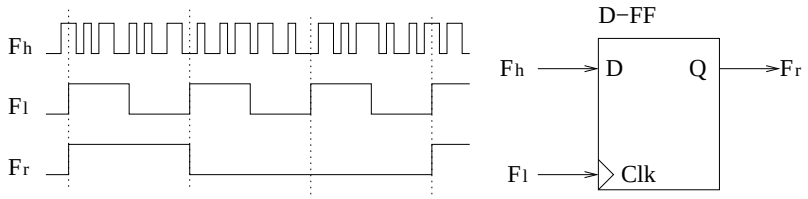
The rest of the chapter is organized as follows: In Section 5.2, background information about physical random source and Alternating Step Generator are presented. The architecture of the PRNG and its FPGA implementation are presented in Section 5.3. In Section 5.4, the experiment results are presented. Conclusions are drawn in Section 5.5.

5.2 Background

5.2.1 Oscillator Sampling based Physical Noise Source

Oscillator sampling based noise sources typically use a low frequency clock (F_l) with large phase noise to sample an accurate high frequency clock (F_h) and resulting a random sequence (F_r) as shown in Figure 5.1. If the phase noise of F_l is of the same order as the period of the high frequency clock, an output which is random is obtained [FMC84]. However, since the output rate of this approach is that of the low frequency clock, the output rate of this PRNG is determined by the frequency

Figure 5.1 Oscillator sampling using D-type flip-flop.



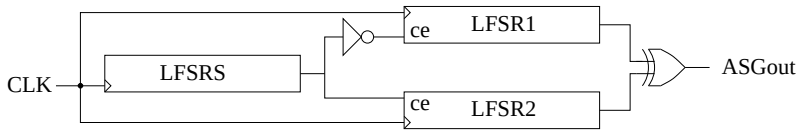
of F_l. If the frequency of F_l is increased to improve the output rate, the phase noise usually decreases, leading to correlations in the output.

There are several factors which affect the randomness of the output [FMC84]. The first is that the duty cycle of F_h may not be 50%. In this situation, F_r will have unequal probability of being zero or one. An N-bit parity filter [ECS94, FMC84] can be used to deskew a non-uniform distribution. If the ratio of ones to zeros in the raw random bitstream is $p : q$, then the probability that the parity will be one or zero is the sum of the odd or even terms of the binomial expansion of $(p+q)^N$. This sum can be evaluated to calculate the probability of a one at the output of the parity filter and is $\frac{1}{2}((p+q)^N + (p-q)^N)$. Since $p+q=1$, this expression reduces to $\frac{1}{2}(1+(p-q)^N)$. As N increases, this expression tends to 0.5.

The second factor is the selection of clock frequency. Period of the generated clock will change from time to time due to circuit internal instability and external noise coupling. If the variation of the period in F_l is not large enough, there will be correlation between bits and so the value of the output can be predicted to some extent from the previous values. Previous research has shown that, from the probability density function of guessing the next bit, the standard deviation of the period variation of F_l should at least be 0.75 times the period of F_h to reduce bit to bit correlation [FMC84]. Thus increasing F_h and reducing F_l leads to more randomness.

A third factor affecting the quality of the RNG is the random source itself. As there are both periodic and aperiodic electromagnetic noise inside a computer system, there may be correlation in the output sequence as the result of coupling of

Figure 5.2 Alternating step generator.



periodic noise from the power supply, clocks, crosstalk, thermal effects etc. This issue is not addressed in this work.

5.2.2 Alternating Step Generator

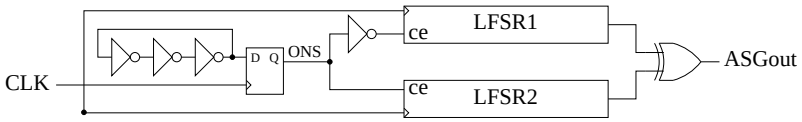
The ASG is constructed from three LFSRs as shown in Figure 5.2 [Gun88, MvOV97]. The binary output of the selection LFSR (LFSRS in the figure), is used to select whether LFSR1 or LFSR2 is clocked. The output of the ASG is the XOR of the output of LFSR1 and LFSR2. The characteristic polynomials of LFSR1 and LFSR2 are irreducible and different. In addition, the greatest common divisor of the periods of LFSR1 and LFSR2 should be equal to 1.

Several attacks on the ASG have been proposed. If the connection polynomials of LFSR1 and LFSR2 are primitive trinomials, the generator can be attacked using the linear syndrome method [ZYR91]. In our design, a high Hamming weight polynomial was chosen to prevent this attack. Golic proposed an attack based on the edit distance [GM98]. This attack requires computing the edit distance for every possible pair of initial states of LFSR1 and LFSR2 and is hence not practical for large shift register lengths (approximately 127 in our case).

5.3 Architecture and Implementation

In the proposed approach, a physical noise source, hereafter called the oscillator noise source (ONS), is produced by oscillator sampling as shown in Figure 5.3.

Figure 5.3 Proposed PRNG circuit.



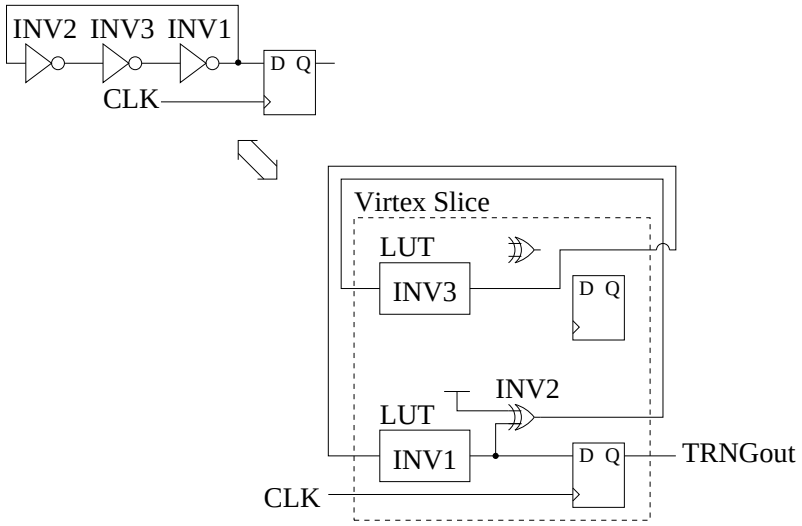
The high frequency clock, F_h, is generated using a 3-inverter ring oscillator implemented in a single Xilinx Virtex slice, while the low frequency oscillator input comes from the system clock (133 MHz) in our tested configuration. These two signals are combined using an edge-triggered D-type flip-flop to produce a non-deterministic but correlated random output. This output is used instead of the selection LFSR of an ASG.

In order to achieve a high output rate, the ONS should produce outputs at the same rate as the system clock. This is normally derived from a crystal controlled oscillator and has low phase noise. Hence the system clock should be connected to the clock input of the D type flip-flop (as shown in Figure 5.3), and a high frequency oscillator connected to the D input. For the FPGA implementation, a high frequency ring oscillator was used. Ring oscillators are commonly used for phase locked loops, clock recovery circuits and frequency synthesizers, but have high phase noise compared with circuits employing passive resonant components [Raz96]. Thus they combine the advantages of being implementable entirely within an FPGA and high phase noise.

It is desirable to make the frequency of the ring oscillator as high as possible in order to reduce the correlation resulting from sampling the ring oscillator with the system clock. A naive implementation would require 3 lookup tables (LUTs) and hence 1.5 Xilinx Virtex slices [Xil00a]. The FPGA implementation used an additional 2-input XOR gate present in the Xilinx Virtex slice to reduce the implementation to 1 Virtex slice as shown in Figure 5.4. This has the advantage of higher speed because wiring is reduced and the XOR gate is faster than a LUT.

The LFSRs were implemented using the SRL16 [Xil00a] feature of the Xilinx

Figure 5.4 Xilinx Virtex ring oscillator implementation.



Virtex chip which enables a 1-16 stage shift register to be implemented in a single LUT.

5.3.1 Clock Doubler

As discussed in Section 5.2, increasing the high frequency clock, F_h, improves the randomness of the ONS output. It is possible to apply a clock doubler to the output of the ring oscillator as shown in Figure 5.5. The poker test in the NIST testsuite [U.S94] was used to observe the effect of different delay values for the clock doubler, and the results are shown in Figure 5.6. The poker test is passed if the result is between 1.03 and 57.4 [MvOV97]. As it can be seen, small and large values of the delay do not result in clock doubling and the poker test results are poor. The poker test results show a significant improvement for delay values, as reported by the Xilinx timing analyzer, of approximately 2.5 ns. Table 5.1 shows a comparison of the best poker test results with and without a clock doubler. Note that although the clock doubler offers an improvement, the ONS output does not pass the poker test.

Figure 5.5 Clock doubler circuit.

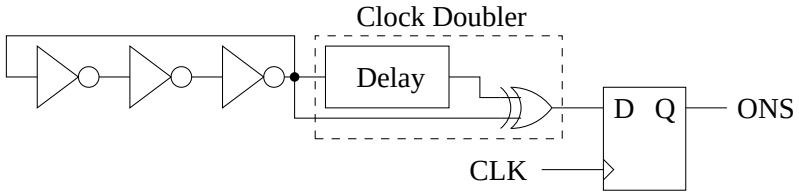


Figure 5.6 Poker test results as a function of the clock doubler delay.

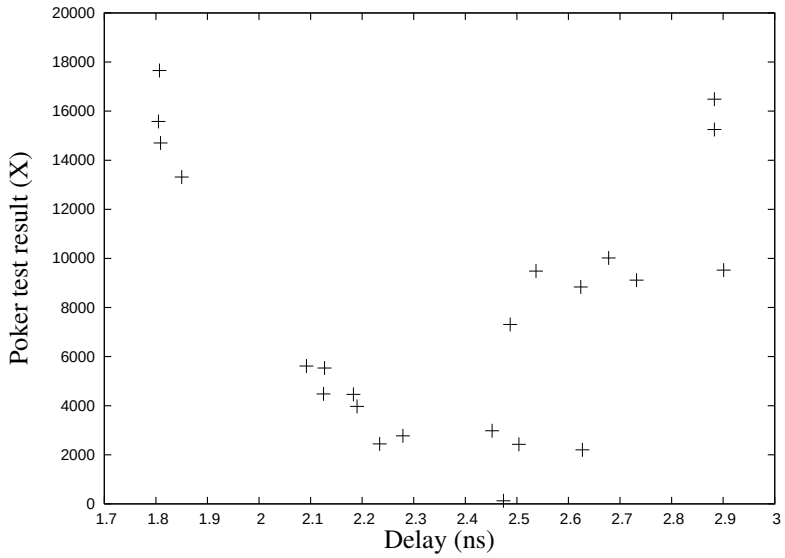


Table 5.1 Comparison of poker test results with and without a clock doubler.

Delay(ns)	Poker test result
0	1579.77
2.474	124.013

Table 5.2 Implementation summary (Xilinx XCV300E-8).

Design Design	Period (ns)	Slices (% XCV300)	BRAM (% XCV300)
PRNG	7.482	129 (4%)	4 (12%)

5.4 Results

An implementation of the PRNG was synthesized and implemented using the Xilinx ISE 8.2i software. The LFSR was inferred as chain of SRL16 components on the device which resulted in very small area cost (only 59 LUTs and 147 FFs as reported by Xilinx tools). The FPGA platform used was a Pilchard FPGA card [LLC+01] which employs the SDRAM bus instead of the PCI bus used in conventional FPGA boards. The FPGA device used was a Xilinx Virtex XCV300E-8 device. The LFSRs were chosen so as to have a random irreducible connection polynomial of degrees 127 and 129 with approximately the same number of 0 and 1 coefficients [Gun88, MvOV97].

The initial states of the LFSRs were random numbers with approximately an equal number of 1's and 0's.

Table 5.2 summarizes the resource utilization and performance of the PRNG including a host interface to read back the data. The high frequency clock of the PRNG can operate at over 400 MHz, but experiments described in this paper used a 133 MHz clock so that the output sequence could be collected via the SDRAM interface of the host computer. As reported by the Xilinx timing analysis tool, the minimum ring oscillator frequency was 800 MHz.

Since the ONS output of the the clock doubler improves randomness, results reported below are without the clock doubler (i.e. the delay was set to 0). It was also verified that the implementation passes the below tests when an appropriate delay for the clock doubler was added. This increases confidence that the design will operate correctly even if the delay of the clock doubler is set to an inappropriate

Table 5.3 NIST RNG test result summary for the PRNG.

Test	P-value	Pass Rate
Frequency	0.145326	0.9900
Block Frequency	0.657933	0.9700
Cusum-Forward	0.383827	1.0000
Cusum-Reverse	0.867692	1.0000
Runs	0.289667	0.9700
Long Run	0.759756	0.9900
Rank	0.514124	0.9900
FFT	0.779188	1.0000
Aperiodic Templates	0.657933	0.9600
Periodic Templates	0.289667	0.9900
Universal	0.162606	1.0000
Approximate Entropy	0.924076	0.9900
Random Excursions	0.637119	0.9565
Serial1	0.534146	1.0000
Serial2	0.262249	1.0000
Lempel Ziv	0.616305	0.9900
Linear Complexity	0.637119	1.0000

value.

5.4.1 NIST Test Suite

For the NIST test suite (version 1.4), all parameters were set according to the recommendations in [Ruk01] and the test sequences were 1 Mbit in size. The sample size, i.e. the number of bit sequences to pass the tests was 100. Table 5.3 summarizes the NIST test results for the PRNG. The significance level α was chosen to be the default of 0.01 (99% confidence) to pass a test if its P-value is larger than this number. The *Pass Rate* is proportion of the 100 binary sequences that passed the test, It can be seen that the PRNG passes all NIST tests.

Table 5.4 Diehard RNG test result summary.

Test	P-value
Birthday Spacings	0.310619
Overlapping 5-Permutation (chisqr 66.743792)	0.994677
Overlapping 5-Permutation (chisqr 107.948832)	0.253086
Binary Rank (31x31)	0.155
Binary Rank (32x32)	0.080
Binary Rank (6x8)	0.051318
Bitstream	0.008018
OPSO	0.996754
OQSO	0.011809
DNA	0.050285
Steam Count-the-1	0.066896
Byte Count-the-1	0.040476
parking Lot	0.921990
Min. Distance	0.496703
3D Spheres	0.016095
Squeeze	0.456598
Overlapping Sums	0.080856
Runs up	0.053444
Runs down	0.738119
Craps	0.985720

5.4.2 Diehard Test Suite

Although the Diehard test suite is one of the most comprehensive publicly available sets of randomness tests, unfortunately there are no well-defined pass criteria. Intel calculated that the entire 250 test suite passes with a 95% confidence interval for P-values between 0.0001 and 0.9999 [Int99], and this method was used for our testing. The Diehard test results are summarized in Table 5.4. If multiple p-values are in the result,the worst case value is presented. The PRNG random sequence passes the Diehard test.

5.4.3 TestU01 Test Suite

TestU01 [LS07] is a set of C libraries for RNG performance evaluation. We developed programs to test our RNG results using this library. The random data was stored in a file and then read in as an external RNG source. The reports shows that our RNG passes the *Rabbit*, the *Alphabit*, then *SmallCrush* and the *Crush* test batteries (The *BigCrush* test was not run due the huge data requirement).

5.5 Summary

In this chapter, a new random number generator (RNG) was introduced. This circuit combines a physical random number source with a high speed stream cipher to produce a physical noise source based random number generator with small area, high output rate and good statistical properties. This RNG would be suitable for simulation and cryptographic applications. This RNG can be instantiated as black box in CAST framework as a reliable and fast input for the simulated process.

Chapter 6

Monte Carlo Simulation

6.1 Introduction

Monte Carlo simulation (MC) is a technique which makes a large number of randomized trial runs (each trial called a *path*) to infer the probability distribution of the outcome. MC simulation is often the only tool for treating otherwise intractable problems such as the pricing of financial derivatives and scientific calculations on stochastic processes. Computation speed is a major barrier for deployment of MC solutions in many large and real-time applications.

Previous work on applying reconfigurable computing to accelerating Monte Carlo simulations has been proposed. McCollum et. al. described a hardware design for generating random numbers from arbitrary distributions and applied it to several MC problems including computation of π, Monte Carlo integration and stochastic simulation for chemical species [MLBP03]. Gokhale et. al. described the application of FPGAs to heat transfer simulation [GFA+04] and Yoshimi et. al. applied FPGAs to the stochastic simulation of biochemical reactions [YOFA04b]. Cowen and Monaghan presented a generic MC architecture targeting mainly physics simulations [CM94], and Postula et. al. reported an MC processor for the simulation of sintering [PAL96]. In each case, considerable speedups over standard software based implementations were observed.

In this chapter, We demonstrate the feasibility of applying reconfigurable computing technology to practical, large scale simulation problems which require floating-point arithmetic.

A major issue faced when developing scientific applications in digital hardware is the choice of number representation and wordlength. We propose that a generalized, number system independent description of the algorithm based on the CAST framework. Thus the most suitable number representation and accuracy for a given application can be found via optimization [THYL04].

A generic architecture for MC simulation in which an on-chip processor is combined with a hardware path generator which combines flexibility and speed is presented. The same design methodology can be applied to a class of MC applications. Moreover, since processor and hardware accelerator are on the same chip, their interconnection does not impose a bottleneck on system performance.

The MC design methodology is applied to two different problems, the first being to compute an approximation to π. The second example is a real-world financial engineering application, the BGM interest rate model [BGM97]. In the BGM example, different paths are calculated simultaneously in order to avoid data dependencies. Using the MC design methodology, with the help of the π and BGM examples, it shows that the performance of single chip machines which can be used to accelerate complex MC simulations.

The chapter is organized as follows. In section 6.2, a general architecture for Monte Carlo Simulations is presented along with its application to an example in which an approximation to π is computed. In section 6.3, the BGM model, hardware architecture and core used in its implementation are presented. Conclusions are given in section 6.4.

6.2 Computation of π via Monte Carlo Simulations

This section describes a simple MC processor for computing the value of π. We use this method to illustrate the MC architecture; there are other methods for computing π that are faster. Imagine a circle of radius r circumscribed by a square with sides of length $2r$. If a large number of darts are thrown uniformly at the square, the proportion of darts which hit inside the circle is given by:

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi r^2}{(2r)^2} \tag{6.1}$$

$$= \pi/4. \tag{6.2}$$

The above proportion is the same if only the top right-hand quarter of a square centered at the origin is considered. Thus, if $r = 1$, π can be approximated by randomly generating two random numbers, x and y, $x, y \in [0, 1)$, calculating whether the coordinate (x, y) is within the top quarter of a circle ($x^2 + y^2 < 1$), calculating the proportion of trials inside and outside the circle, and multiplying this result by 4 to obtain an approximation to π. In pseudocode form, this can be described as:

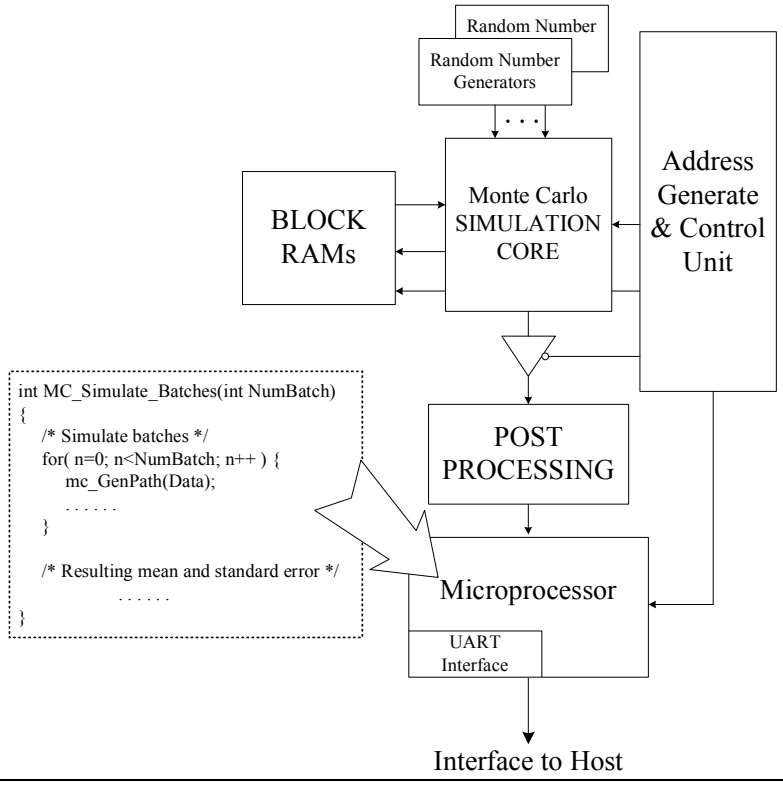
Step 0: $h = 0$
Step 1: for $k = 1$ to $NumBatch$
Step 2: $x = rand(), y = rand()$
Step 3: $if\ ((x^2 + y^2) < 1)$
Step 4: $h = h + 1$
Step 5: $\pi \approx \frac{4*h}{NumBatch}$

and will be referred to as the π-simulation.

FPGA technology is used to implement a Monte Carlo simulation with the goal of reducing the execution time as compared with a traditional software implementation. With a fully pipelined implementation, an iteration can be computed every cycle. The hardware architecture of a generic MC engine is shown as a block diagram in Figure 6.1. As applied to the π-simulation, the random number generator block consists of two parallel uniform number generators, implemented using linear

feedback shift registers. The MC core computes steps 3 and 4 of the pseudocode, no post processing is required, and step 5 is implemented in the on-chip microprocessor.

Figure 6.1 The system architecture block diagram.



6.2.1 MC Arithmetic System and Wordlength Determination

Although experience may tell us that a fixed-point implementation would be the most suitable for the π-simulation, for other MC simulations, perhaps those involving variables with larger dynamic range, floating-point may be a better choice. Moreover, even for a fixed-point implementation, the wordlength requirements of the variables cannot be explicitly determined. In order to address this problem, the CAST framework was used to provide an environment in which tradeoffs between different arithmetic systems of arbitrary wordlength can be compared. It saves design time, facilitates quantitative comparisons between different arithmetic systems

Table 6.1 Latency of arithmetic operators in CAST.

Arithmetic	Adder	Multiplier	Divider
Fixed-Point	1	1	3
Floating-Point	3	3	4

at different precisions and is well suited for designing the fully pipelined datapaths of MC cores. Table 6.1 shows the latency of operators for the different number systems.

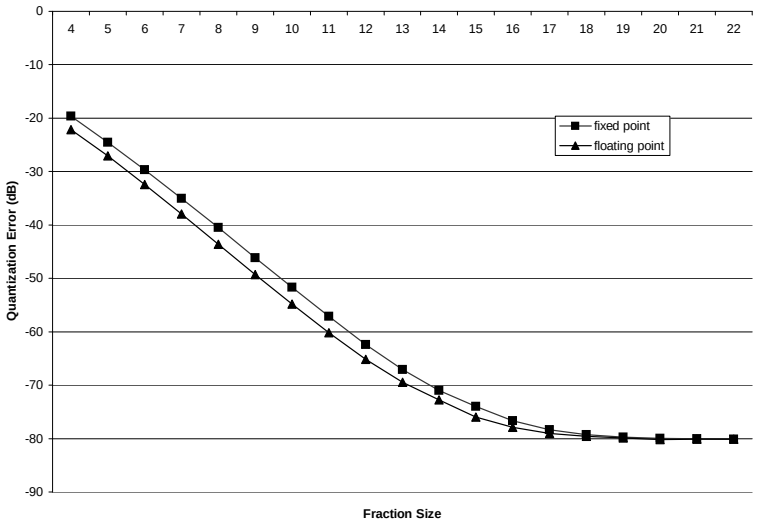
In the CAST system, fixed-point numbers are represented as two's complement fixed-point fractions. Floating-point numbers are similar in format of the IEEE 754 standard except that the size of the exponent and fraction are parameterized; there are no denormalized numbers, and a round-to-nearest scheme is used. Since the resource requirements for high precision LNS adders and subtractors in CAST is very high [THYL04], only fixed and floating-point number systems were considered in this application.

6.2.2 Determining Fraction Size

To evaluate the minimum amount of resources required to produce at least 4 decimal place accuracy (as required in financial applications), the CAST library is used to generate the C++ code for a bit-exact simulation of the different fixed and floating-point operations provided by the arithmetic library, parameterized by the number format (i.e. integer, exponent and fraction sizes).

In this way, only quantization error and number system account for differences between double precision floating-point (used as a reference) and the simulation of the quantized hardware implementation. Figure 6.2 shows the results. According to these results, we find implementation schemes which minimize area subject to accuracy requirements. The actual implementation selected used 17 bits for the multiplier fraction and 21 bits for the adder fraction.

Figure 6.2 Quantization error as a function of fraction size for fixed-point and floating-point implementations of the π-simulation.



6.3 The BGM Model, Interest Rate Cap and Monte Carlo Simulation

Interest rates fluctuate over time and since nearly all economic activity is dependent on this instrument, there is considerable interest in modeling for valuing and hedging purposes. The BGM model [PPvR05] is commonly used because of its theoretical elegance and ease of calibration[1] .

Interest rate caps can be explained by first considering a floating-rate loan where interest rate is updated periodically (e.g. every 3 months) according to the market rates. A cap is an option which gives the holder the right to stick with a specified rate if the market rate goes higher than it. This provides insurance to the borrower against rises in interest rates.

Within the BGM framework, the price of a cap or other interest rate derivative is usually computed using Monte Carlo simulation since it is difficult to apply other approaches under the BGM model. An advantage of Monte Carlo simulation is its

[1]Hull [Hul00] provides a good introduction to financial derivatives. See section 4.5 for forward interest rate, section 20.3 for caps and section 22.3 for an introduction to the BGM model and other interest rate products.

applicability to pricing a large range of derivatives, and straightforward implementation directly from the stochastic model rather than requiring further derivation (as for tree or finite difference methods). However, it has the drawback of being computationally expensive.

Denote $F(t, t_n, t_{n+1})$ as the forward interest rate observed at time t for a period starting at t_n and ending at t_{n+1}. Suppose the time line is segmented by the reset dates $(T_1, T_2, ..., T_N)$ (called the standard reset dates) of actively trading caps on which the BGM model is calibrated. In the BGM framework, the forward rates $\{F(t, T_n, T_{n+1})\}$ are assumed to evolve according to a log-normal distribution. Writing $F_n(t)$ as the shorthand for $F(t, T_n, T_{n+1})$, the evolution follows the stochastic differential equation (SDE) with d stochastic factors:

$$\frac{dF_n(t)}{F_n(t)} = \vec{\mu_n}(t)dt + \vec{\sigma}_n(t) \cdot d\vec{W}(t) \qquad n=1 \ldots N. \tag{6.3}$$

In this equation, dF_n is the change in the forward rate, F_n, in the time interval dt. The drift coefficient, $\vec{\mu_n}$, is given by

$$\vec{\mu_n}(t) = \vec{\sigma}_n(t) \cdot \sum_{i=m(t)}^{n} \frac{\tau_i F_i(t) \vec{\sigma}_i(t)}{1 + \tau_i F_i(t)} \tag{6.4}$$

where $m(t)$ is the index for the next reset date at time t and $t \leq t_{m(t)}$, $\tau_i = T_{i+1} - T_i$ and σ_n is the d-dimensional volatility vector. In the stochastic term (the second term on the right hand side of Equation 6.3), $d\vec{W}$ is the differential of a d-dimensional uncorrelated Brownian motion $\vec{W}$, and each component can be written as

$$dW_k(t) = \epsilon_k \sqrt{dt} \tag{6.5}$$

where ϵ_k is a Gaussian random number drawn from a standardized normal distribution, i.e. $\epsilon \sim \phi(0, 1.0)$. A Gaussian random number generator [Knu81] is required to implement the Brownian motion.

A number of financial derivatives, including caps, knock-out caps, swaps, Bermudan bond options and flexi-caps can be priced under the BGM model [Hul00]. To

simplify the example of pricing a derivative with FPGA-based hardware, we only consider caps in this application[2] .

The cap consists of a series of caplets in each of which the payoff between the floating rate and the cap rate in the standard period is settled. In pricing the cap via Monte Carlo simulation, a large number of interest rate paths are generated using pseudorandom numbers according to Equation 6.3 with a time-discretization step size being 0.01 to 0.05 years. In each path, the forward rate $F_n(t_n)$ is realized in each standard period which enable the caplet payoff at time t_{n+1} to be calculated.

$$\textit{payoff}_n = \textit{principal} \times \tau_n \times max(F_n(t_n) - \text{cap rate}, 0.0) \tag{6.6}$$

The amount *payoff*$_n$ is to be received at t_{n+1}, and its value at time zero (t_0) is the amount that would grow to *payoff*$_n$ with the interest rates from t_0 to t_{n+1}. Solving the value of *payoff*$_n$ at t_0, the *discount factor* for discounting *payoff*$_n$ at t_{n+1} back to t_0 is given by:

$$\textit{discountFactor} = \prod_{i=0}^{n} \frac{1}{(1 + F_i(t_i))} \tag{6.7}$$

The payoff of each caplet is discounted back to time zero and summed to form the value of the cap under the Monte Carlo trial. The average value of the cap in all the Monte Carlo trials is the price of the cap.

In financial applications, the accuracy requirement on derivative prices is generally four decimal places (1 in 10000), e.g., if the principal is $100, the answer of the cap price should be correct to cents.

The entire Monte Carlo simulation is divided into three stages, namely simulation initialization, BGM path generation and post processing. The initialization stage initializes the volatility vector $\vec{\sigma}$, reset the Gaussian random number generators and initializes the Brownian motion generator.

[2]In general, other derivatives may depend on forward rates in "non-standard periods" which do not coincide with the "standard periods" of the instrument for calibration. The non-standard forward rates follows another SDE which is not discussed in this text.

In the second stage, the BGM paths are generated according to equation (6.3). The pseudocode for the main BGM model is described as:

Step 1: for $n = CurrPeriod + 1$ to N

Step 2: $factor = \tau_n F_n/(1.0 + \tau_n F_n)$

Step 3: $\vec{\mu}_n = factor \times \vec{\sigma}_n$

Step 4: $\vec{\mu}_n = \vec{\mu}_n + \vec{\mu}_{n-1}$

Step 5: $\kappa = (\vec{\mu}_n \cdot \vec{\sigma}_n)dt + (\overrightarrow{dW} \cdot \vec{\sigma}_n)$

Step 6: $dF_n = \kappa \times F_n$

Step 7: $F_n = F_n + dF_n$

where $CurrPeriod$ is the index of the current standard period, i.e. $m(t) = CurrPeriod+1$ and N are the number of standard forward rates.

The *for-loop* (step 1) is the main loop of the BGM model. The computation consists of one division (step 2), one vector addition (step 4) and three vector product operations (step 3, step 5) in each iteration of the *for-loop*. We use a Taylor series expansion to implement step 2. In order to maximize parallelism, the vector operations are implemented as parallel scalar operations.

Finally, post-processing involves pricing the cap according to Equations 6.6 and 6.7 and calculate the mean and standard error of the generated BGM paths on the PowerPC processor. We discuss the details in section 6.3.5.

6.3.1 Hardware Architecture

The MC architecture implementing the BGM model of this system is shown in Figure 6.3. There are seven major blocks in the system architecture: Brownian motion generator, Volatility vector unit, Datapath core (BGM core), Address generation and control unit, Block RAMs, Cap Price post processor and the processor core. The Brownian motion generator generates the dW vectors according to equation (6.5) and is driven by three Gaussian random number generators. The Datapath core is responsible for the generation of BGM paths and a detailed description is given in

section 6.3.3. The Address generation & control unit and Block RAMs are used for data storage during the BGM simulation. In order to perform postprocessing (computing the cap price in our example), a module, placed between the BGM core and the processor to accelerate this computation is added. The final block is the processor core which is responsible for coordinating the processing between the various cores as well as postprocessing of the BGM paths for different financial derivatives. We have used both the Xilinx Microblaze soft processor as well as the PowerPC processor in the Xilinx Virtex-II Pro for the π and BGM examples respectively. The CAST framework was used to implement the BGM path core and other blocks were implemented using VHDL.

Figure 6.3 The system architecture block diagram for BGM-simulation.

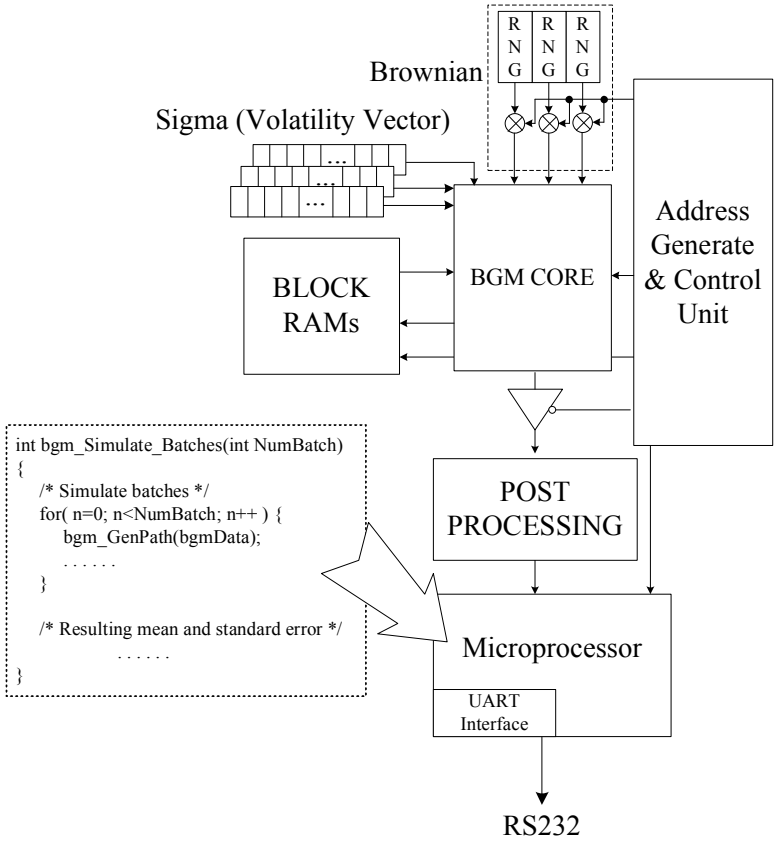
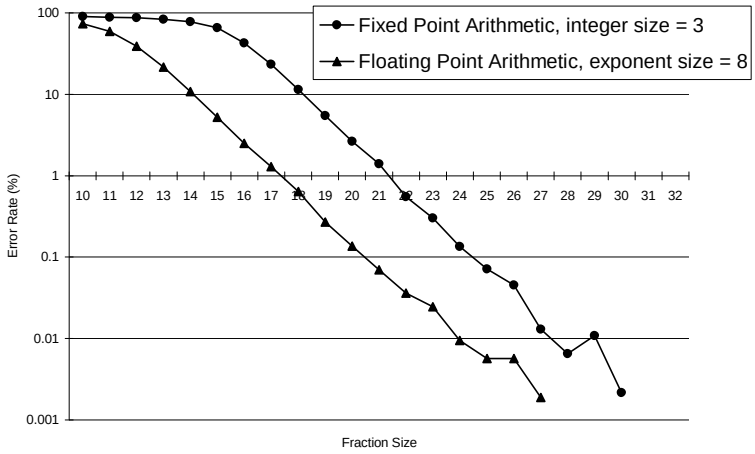


Figure 6.4 Quantization error as a percentage with varying fraction size.



6.3.2 BGM Number System and Wordlength Determination

A software implementation of the BGM model is made using the CAST simulation function. Given representative input data, one can determine the quantization error against a double-precision IEEE software implementation for different fraction sizes. Figure 6.4 shows how the percentage error changes for the π-simulation for several different number formats.

Contrast to the π simulation, it is likely that different variables in the BGM simulation have different precision requirements. Thus in the BGM simulation, each operator is allowed to have a different wordlength, and a multi-dimensional minimization was performed to find a balance between quantization error and circuit size. A cost function is defined as:

$$f_{cost}(c_1, c_2, ...c_n) = a \times error\_rate(c_1, c_2, ..c_n) \quad (6.8)$$

$$+b \times area(c_1, c_2, ...c_n) \quad (6.9)$$

where c_i represents the fraction size of operator i. $error\_rate$ is the quantization error of the result if the answer is not correct to 4 decimal digits. In the equation, $area$ is an estimate of the required logic resources for the given configuration of operators, and a and b are non-negative weighting factors for the error and area terms

Table 6.2 Results obtained from optimizing the wordlengths of the arithmetic operators. The pairs (a,b) refer to (integer wordlength, fractional wordlength) and (exponent wordlength, fractional wordlength) for the fixed and floating-point cases respectively.

Fraction Size Before Optimization				
Arithmetic	mul	add	div	acc
Fixed-Point	(2, 31)	(2, 31)	(2, 31)	(2, 31)
Floating-Point	(8, 28)	(8, 28)	(8, 28)	(8, 28)
Fraction Size After Optimization				
Fixed-Point	(2, 31)	(2, 30)	(2, 15)	(2, 20)
Floating-Point	(3, 22)	(3, 30)	(3, 15)	(3, 15)

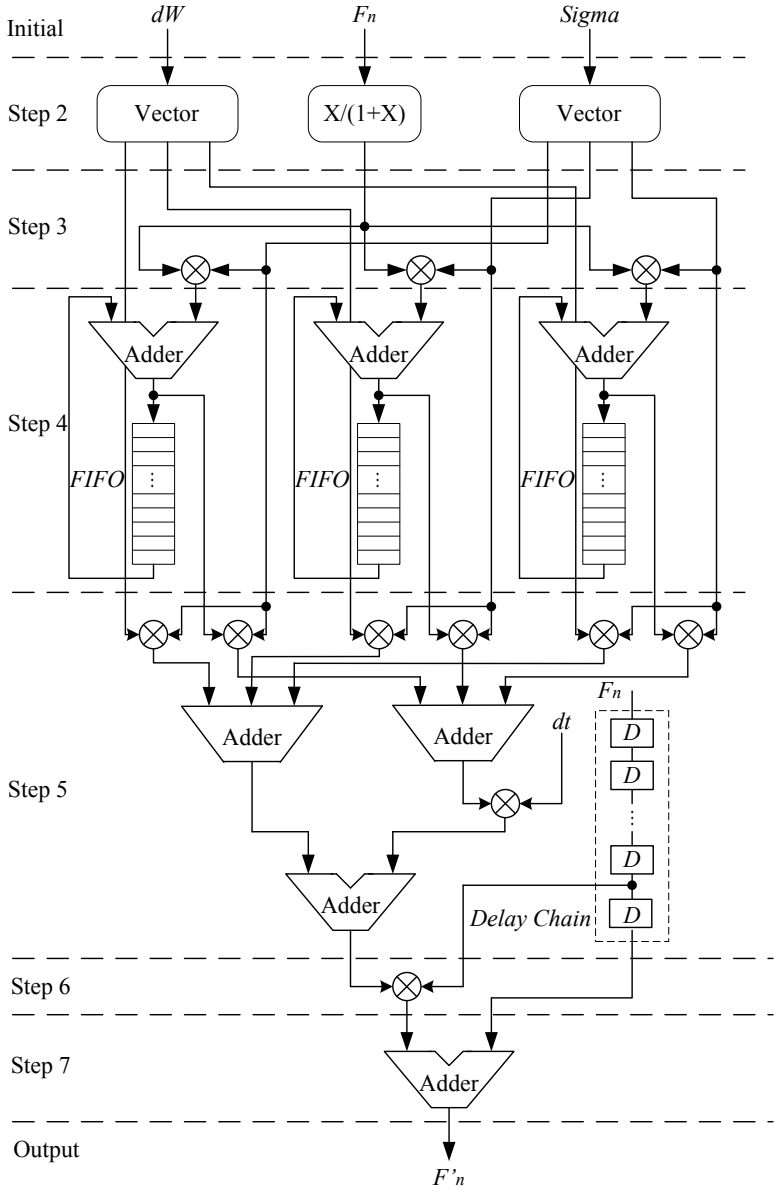
respectively. As the BGM application must maintain 4 decimal place accuracy, the value of a is typically several orders of magnitude larger than b.

The Nelder-Mead optimization method [NM65] was used to minimize the fraction size of the numerical representation. The range for each operator during a BGM simulation is stored in the class and then used to determine an appropriate choice of integer and exponent size in the number representation. Since many operators are used in the BGM core, it is computationally intensive to optimize each of their precisions individually. A faster but perhaps less optimal approach in which some variables are constrained to the same fraction size is adopted. Operators are categorized into 4 groups, namely adders, multipliers, accumulators and dividers. The optimization routine varies the fraction size of adder, multiplier and accumulator to find the configuration which can obtain the desired four decimal place precision using minimal resources. Table 6.2 shows the results obtained.

6.3.3 BGM Core Architecture

The BGM core implements the path generation loop of the BGM model as shown in Figure 6.5. The figure describes the arithmetic operations of the pipelined architecture in detail and corresponds to the pseudocode architecture.

Figure 6.5 The Primitive Processing Loop Architecture for BGM Core.



In the first initial step, Brownian motion parameter $\overrightarrow{dW}$, the volatility vector $\vec{\sigma}$ and the forward rate F are initialized. As $\overrightarrow{dW}$ and $\vec{\sigma}$ are vectors, we use a parallel architecture to implement the vector operations. There are two "Vector" blocks in the second step to convert $\overrightarrow{dW}$ and $\vec{\sigma}$ to scalars. The computation of $F/(1.0+F)$ is also performed in this stage. In step 3 and step 4, vector $\vec{\mu}$ is computed according to Equation 6.4. FIFOs (First-In First-Out) are used to implement the accumulator ($\vec{\mu}_n = \vec{\mu}_n + \vec{\mu}_{n-1}$). The depth of the FIFO is decided by the number of BGM paths being simulated, as described in the following section.

According to Equation 6.3, the change in the forward rate dF_n is computed in step 5 and step 6. As the BGM core architecture is pipelined, we use a delay chain to adjust the timing of F_n. The result is obtained in the output stage.

6.3.4 Pipelined Path Generation

The Monte Carlo simulation generates a set of independent random forward rate paths, and computes their average. As the number of paths are large, this results in a long simulation time.

The architecture of BGM core is organized as a deep pipeline. If only one path is simulated using the BGM core, data dependencies mean that the pipeline must stall until the output is generated since each iteration of the algorithm depends on the previous iteration. This would result in the pipeline being mostly idle. A 2-D data flow arrangement was proposed such that each stage computes a different path and all stages operate in parallel. The operation can be described as follows,

for ($i = StartStep$; $i < StopStep$; $i++$) {
  if ($i == NextResetDateStep$) /* Record forward rates */
    Output forward rate - $F(i)$;
  for ($n = 1$; $n < N$; $n++$) {
    for ($m = 0$; $m < NumPath$; $m++$) {
      /* Evolve one time step */

```
            bgm_evolve_step(i, n, m);
        }
    }
}
```

where, bgm_evolve_step$(\cdot)$ evolves one step of the simulation according to the pseudocode description from step 2 to step 7 in section 6.3.3. The data flow is shown in figure 6.6. After one processing loop, i.e. one BGM simulation step, all the values F_n of the BGM paths will be updated. $F_n^m(i)$ is the forward rate of the model, where i is the iterative step, m is the index of the path and n is the index of the forward rate.

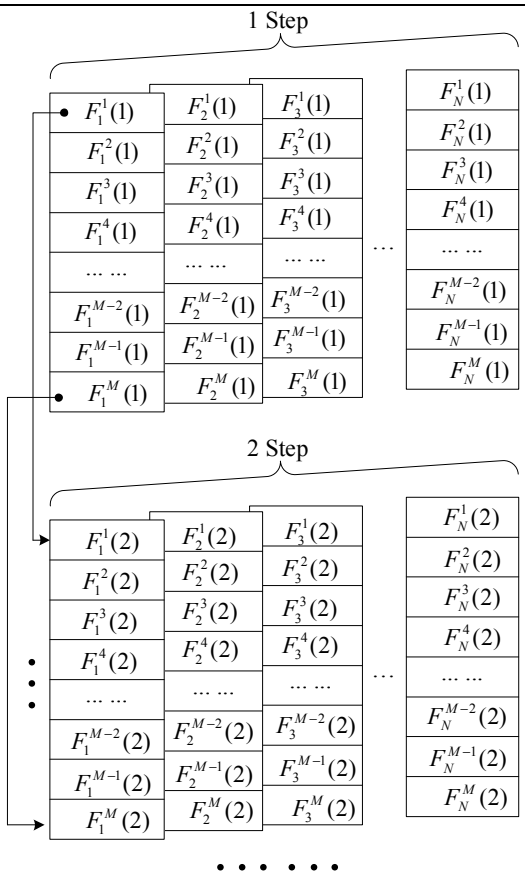
6.3.5 Cap Pricing and Post-Processing Implementation

After generating numbers of interest rate paths, we reach the post-processing step of cap pricing for forward interest rates.

In the Monte Carlo simulation, we use the means and standard errors of the randomized trial runs to describe the simulation results. These operations run with program on the PowerPC. The program is described as follows,

```
/* Simulate batches */
    for (k = 0; k < NumBatch; k++) {
        bgm_GenPath(bgmData);
        SumBatchMean+ = bgmData;
        SumSqBatchMean+ = bgmData * bgmData;
    }
/* Calculate the resulting mean and standard error */
    Mean = SumBatchMean/NumBatch;
    SqMean = sqrt((SumSqBatchMean−
        SumBatchMean * SumBatchMean/
```

Figure 6.6 The 2-D data flow arrangement for the BGM Simulation.



$$NumBatch)/(NumBatch - 1.0)/NumBatch);$$

where $bgm\_GenPath(\cdot)$ is the function to read the path data from the hardware which generates the pricing cap data with BGM core and the post-processing core and $NumBatch$ is the number of the simulation batches. In each simulation batches, we generate numbers of paths in parallel with the hardware core.

6.4 Summary

A novel implementation of a FPGA based system for Monte Carlo simulation was presented. The design used an embedded soft core processor together with a coprocessor core in order to achieve high speed with good flexibility. The π computing and BGM model can be implemented in the same architecture by altering the MC core block which is implemented using the CAST framework.

Using customized low precision floating-point formats, many floating-point operations can be executed in parallel, improving execution speed as compared with a microprocessor which is essentially serial. In order to explore precision and area tradeoffs in the datapath of the coprocessor, different designs could be generated from the same description using CAST. For individual operators in the MC core block, the performance is evaluated and the configurations are modified in an iterative manner using a built-in search method.

Using this approach an order of magnitude improvement in performance for the π and BGM problems was achieved over a purely software based approach, demonstrating the feasibility of applying reconfigurable computing to the problem of accelerating large scale Monte Carlo simulations in floating-point arithmetic.

Chapter 7

N-Body Simulation

7.1 Introduction

The N-body problem is computationally intensive and involves a large number of arithmetic operations on numbers with large dynamic range. This together with the fact that relatively low precision is required makes it a good candidate for hardware acceleration. Using the CAST tool, an FPGA based processor was developed for the gravitational N-body problem similar to GRAPE, with the additional advantages of being flexible in the choice of arithmetic system and precision.

Besides the bitwidth of individual operators, the number systems were also explored in this application. Inputs to the N-body problem have large dynamic range and a $x^{-1.5}$ function is to be evaluated in the datapath. LNS numbers are suitable for this class of simulation systems. On the other hand, there is also a large percentage of computation based on the add and subtract operators, which require a large amount of area in LNS. This inspired the idea of mixing different number systems in a single datapath. The CAST framework allows tradeoffs between the different designs to be quantified much more easily than with previous approaches.

The remainder of the chapter is organized as follows. In Section 7.2, the N-body problem is defined. In Section 7.3, the implementation of an FPGA based coprocessor for this problem is presented. Conclusions are drawn in Section 7.4.

7.2 The N-body Problem

A wide range of physical systems can be studied by modeling them as an N-Body problem. The N-Body problem is extensively used in various fields of science such as astrophysics [MT98] and molecular biology [NSE+99]. In the N-body problem, particles are modeled as points in space and the evolution of the system of N particles is computed by solving a differential equation of the form:

$$\frac{d^2\mathbf{x}}{dt^2} = \sum_{j=1}^{N} \mathbf{F}(\mathbf{x}_i, \mathbf{x}_j) \tag{7.1}$$

where $\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j)$ represents the force between particles i and j and is application dependent. This force is usually the gravity.

The force is computed using the following equation where $\mathbf{x_i}$ and $\mathbf{x_j}$ are the position vectors of particles i and j respectively, $\mathbf{r}_{ij} = |\mathbf{x}_i - \mathbf{x}_j|$ and ϵ is the softening constant.

$$\mathbf{F}(\mathbf{x_i}, \mathbf{x_j}) = \sum_{j=1}^{N} \frac{\mathbf{x}_i - \mathbf{x}_j}{(\mathbf{r}_{ij}^2 + \epsilon^2)^{\frac{3}{2}}} \tag{7.2}$$

N-Body problems are solved using numerical integration in which the majority of the computation time is spent on calculating $\mathbf{F}(\mathbf{x}_i, \mathbf{x}_j)$. The results after applying the force on particles are their new positions.

There are usually millions of particles involved in N-body simulation and the complexity of the force computation is N^2 as shown in Equation 7.2. Since the force calculation part is computational intensive but the algorithm is rather simple, the computation can be accelerated with hardware assist.

7.3 Coprocessor Implementation

An FPGA based coprocessor handling the force calculation part of the algorithm was built. The arithmetic core of the processor was generated from a C++ description using the CAST system. Since the accuracy requirement for different

simulation runs can differ greatly and depends on the source data and the nature of problem being solved, being able to experiment with different wordlength and arithmetic systems facilitates better exploration of the design space.

The processor was designed to work together with a host computer, which runs the NEMO N-body simulation code [NEM]. The host computer handles all computation except the force calculation. Particle positions are sent to the coprocessor board from a host processor through the board's interface. The coprocessor computes the force acting on a particle, i, using Equation 7.2.

The architecture of the implementation is shown in Figure 7.1. The main components are the control, particle memory and the force pipeline. The particle memory stores the predicted position of all particles while the force pipeline calculates the force acting on each particle. In each timestep, the predicted particle positions are written to the particle memory by the host. For each particle i that is to be advanced in that timestep, the corresponding index is sent to the coprocessor. The corresponding particle position is then read from the particle memory and stored in a register. The force pipeline then begins the calculation as the positions of all j particles are retrieved and fed to the pipeline. The host polls the coprocessor to check if the calculation has completed and then reads the result from the coprocessor.

The force pipeline is the most critical part of the design. The speed of the pipeline directly affects the performance of the system. Figure 7.2 shows the datapath of the force pipeline. It is a fairly straightforward implementation of Equation 7.2 and is generated by the CAST system.

Although our implementation is similar in architecture to that of GRAPE-3 [MT98, ABLM98], three features were not implemented in our design. Firstly, all the particles in GRAPE can be of different mass whereas our implementation assumes they are of the same mass. Secondly, GRAPE-3 calculates the gravitational potential as well as the gravitational force. In our integration algorithm, gravitational potential was not used and hence not implemented. Finally, GRAPE-3 has a neighbor function flag which is raised when two particles are closer than a certain amount.

Figure 7.1 Top level block diagram showing the architecture of the coprocessor.

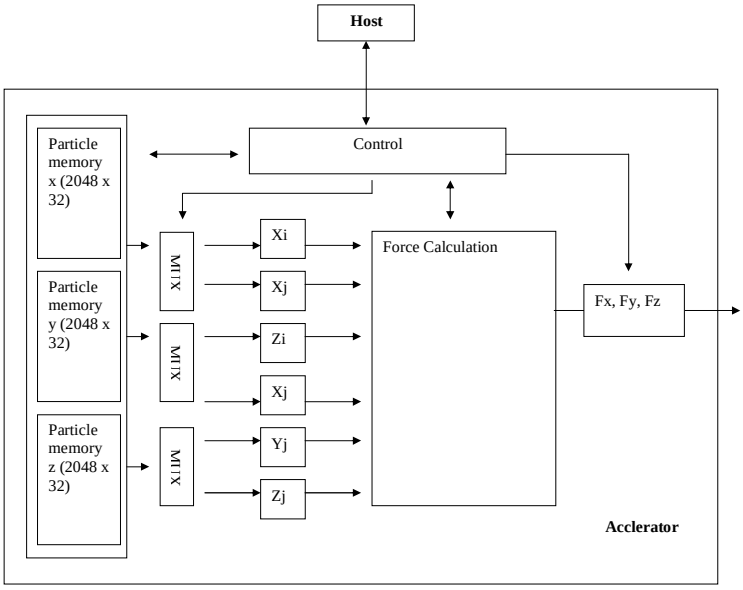
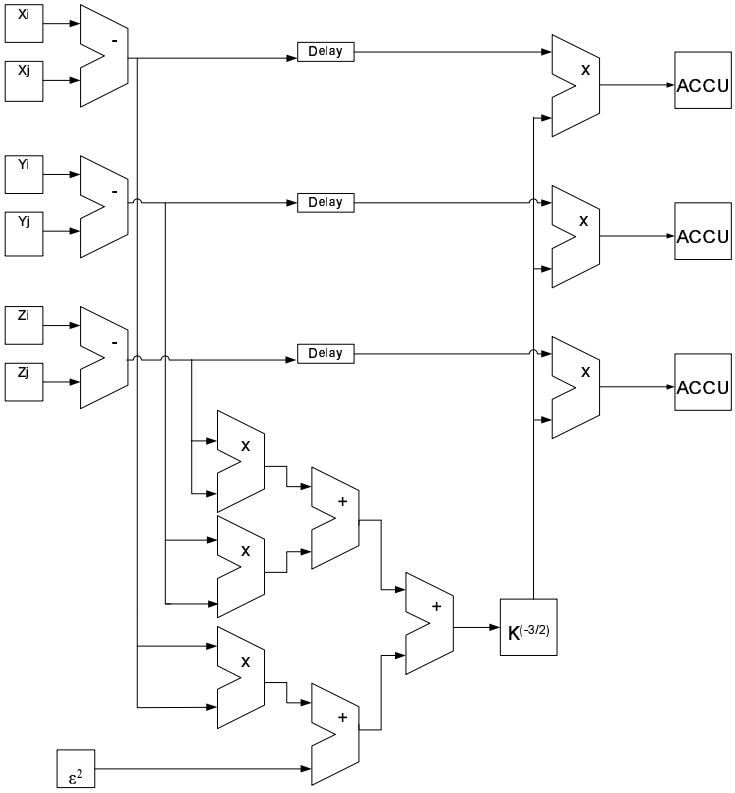


Figure 7.2 Architecture of the force pipeline.



7.4 Conclusion

The CAST system was applied to the design of a coprocessor to compute the solution of the N-body problem. From a structural description of the computation to be performed, a large number of different designs were simulated in C++ and the corresponding VHDL code rendered, each implementation having different tradeoffs in precision, area and speed. By constraining the design to be of a certain precision, it was possible to determine the smallest fractional wordlength which could meet the accuracy criteria for the fixed-point, floating-point, LNS and hybrid implementations.

Chapter 8

Experimental Results

In this chapter, the experimental results of Monte Carlo BGM model simulation and N-body force pipeline are presented.

8.1 Monte Carlo Simulator

The embedded system consists of the PowerPC 405 core, supporting IBM CoreConnect bus architecture (including the Processor Local Bus (PLB) and On-chip Peripherhal Bus (OPB)), the on-chip block RAM, the user logic and other OPB peripherals (such as UART lite and the Debug module etc). The application program is stored in the on-chip block RAM.

To implement the design, we used the Xilinx ML310 FPGA development board [Xil04a] with a Xilinx XC2VP30-6FF896C Virtex-II Pro FPGA [Xil03]. The FPGA has two embedded hard core PowerPC 405 microprocessors and the board provides an environment for the FPGA system. The entire MC simulation is implemented on the FPGA and other features of the board such as the FPGA serial port connection and standard JTAG connectivity are used.

The methodology described in section 6.3.3 was used for both fixed and floating-point implementations for the BGM core. Four BGM cores (corresponding to the before and after optimization designs of Table 6.2) were implemented and their resulting resource utilization and maximum clock frequencies are shown in Table 8.1.

Table 8.1 Optimized Implementation for BGM core

Configuration	Fixed-31	Fixed-Opt	Savings
Frequency (MHz)	57.97	60.07	-
Slices	2,384	2,552	-7.0%
Multiplier	49	49	0%
Block RAM	116	1	99.1%
Configuration	Float-28	Float-Opt	Savings
Frequency (MHz)	61.44	61.56	-
Slices	7,041	5,875	16.6%
Multiplier	48	48	0%
Block RAM	29	1	96.6%

The most significant savings are for the block RAMs used in the construction of the divider in which over 96% of the block RAM can be saved in both arithmetic schemes. After optimization, 16.6% of the slices can be saved for the floating-point implementation since both the size of the exponent and fraction can be reduced. One interesting result is the fixed-point optimized implementation requires more logic resources after optimization. This is because rounding logic is implicitly added to the implementation when conversion between formats are required. It turns out that the rounding logic consumes more slices than the eliminated logic. However, 99% of the BlockRAM is saved because of this optimization. The BlockRAM are used for the lookup tables in the division operator. In addition, even though the fraction size of the multiplier can be reduced in the floating-point implementation, the design tools report the same number of primitive multipliers because a primitive multiplier performs a 17 bit unsigned multiplication and for any fraction size between 20 and 34, the design tool requires 4 multipliers.

According to the analysis of section 6.2.2 and 6.3.2, we synthesize the design of π-simulation with floating point configuration – 8 bits for exponent, 17 bits for multiplier fraction, 21 bits for adder fraction and one sign bit, and BGM-simulation with the optimized fixed point configuration, as shown in Table 6.2 . The synthesis results of π-simulation and BGM-simulation with Virtex-II Pro(XC2VP30-6FF896C)

Table 8.2 Synthesis results for the π-simulation with Virtex-II Pro XC2VP30FF896.

Number of PPC405s	1 out of 2	50%
Number of SLICEs	4,746 out of 13,696	34%
Total Number 4 input LUTs	6,556 out of 27,392	23%
Number of Block RAMs	22 out of 136	16%
Number of MULT18X18s	18 out of 136	13%
Number of DCMs	3 out of 8	62%
Number of JTAGPPCs	1 out of 1	100%
PERIOD analysis for net "CLK"	20ns	50MHz

Table 8.3 Synthesis results for the BGM-simulation using a Virtex-II Pro XC2VP30FF896.

Number of PPC405s	1 out of 2	50%
Number of SLICEs	13,266 out of 13,696	96%
Number of Block RAMs	74 out of 136	54%
Number of MULT18X18s	58 out of 136	42%
Number of DCMs	4 out of 8	50%
Number of JTAGPPCs	1 out of 1	100%
PERIOD analysis for net "CLK"	20ns	50MHz

are shown in Tables 8.2 and 8.3. The details of the device utilization summary are described in Table 8.4.

To compute the speedup of the FPGA π-simulation design over software, we compare execution time for different numbers of paths between the FPGA design operating at 50 MHz and a software implementation on an Intel P4 1.5 GHz machine as shown in Table 8.5. The FPGA-based design achieves a 10+ speedup factor for

Table 8.4 Device utilization summary for BGM-core modules

Number of SLICEs	2,775 (20%)
Number of Block RAMs	16 (11%)
Number of MULT18X18s	40 (29%)
Number of PPC405s	-

Table 8.5 Comparison of Speed-up for π-simulation

Paths Number	50,000	500,000	5,000,000	50,000,000
FPGA (Sec.)	0.0013	0.0103	0.1003	1.0003
PC (Sec.)	0.010	0.130	1.351	12.947
Speedup	7.7	12.6	13.5	12.9

Table 8.6 Comparison of Speed-up for BGM-simulation

Paths Number	50,000	500,000	5,000,000	50,000,000
FPGA (Sec.)	2.63	25.2	242	2400
PC (Sec.)	63	630	6300	63000
Speedup	24.9	25	26	26.2

over 500K paths.

In the BGM-simulation, the hardware BGM core generates fifty paths in one simulation batch using the hardware BGM core in a pipelined fashion. Repeated batches cover the whole simulation. Therefore, number of total paths is,

$$TotalNumPath = NumPathperBatch \times NumBatch \tag{8.1}$$

where $NumPathperBatch$ is equal to 50.

The total simulation time is composed of two parts. One is consumed by the BGM-core simulation of batches and the other is post-processing to calculate the mean and standard error of the generated BGM paths using the processor in software. The total execution time can be calculated as follows,

$$TotalTime \approx t_h \times NumBatch + t_s \tag{8.2}$$

where t_h and t_s are the time consumed by hardware in each batch and software respectively. According to our simulations using a $50MHz$ clock, t_h is $2.42ms$ and t_s is $2.12ms$.

Tables 8.5 and 8.6 show the FPGA-based accelerator's measured execution time

on ML310 board compared with a P4 1.5G Hz machine. The FPGA-based accelerator can generate one BGM path in $63\ \mu s$, and nearly a twenty-fold reduction in execution time was achieved. Parallel cores on larger FPGAs can achieve an even larger speedup. As there are two PowerPC cores in the FPGA used, it is also possible to use one PowerPC core for the Monte Carlo simulation and the other to run embedded Linux. This would enable us to utilize Ethernet connected clusters of FPGA boards, providing virtually unlimited scalability since paths can be generated independently for this type of Monte Carlo simulation.

8.2 N-body Simulator

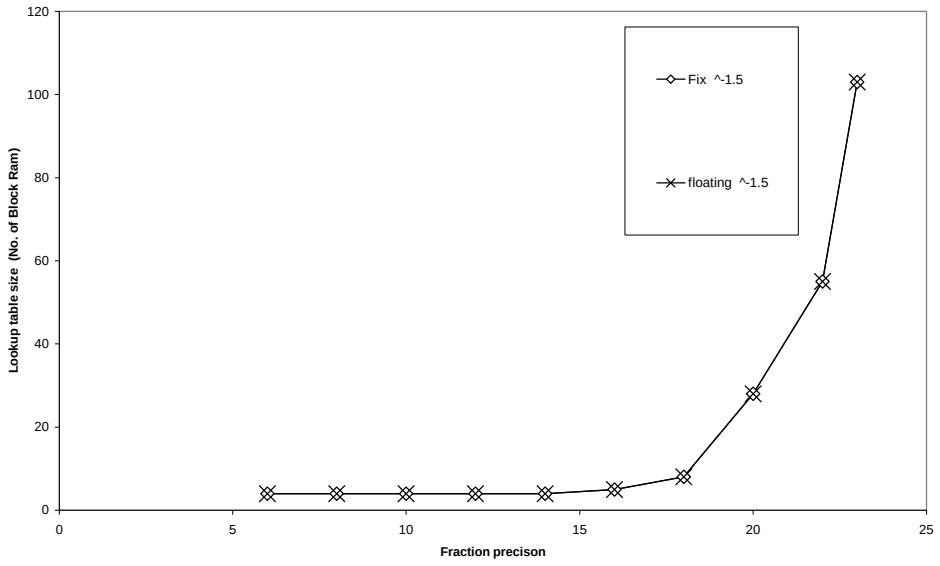
In this section, results showing the resource utilization and performance of the individual operators in the CAST library, along with the precision and performance of the N-body coprocessor are presented. All of the results were simulated using both the CAST system in C++ and Synopsys VSS for verifying the generated VHDL. The target device was a Xilinx Virtex-II XC2V1000FG456-5 for all cases except those which required more than the 40 block RAMs available on that device. For those cases, namely the fixed and floating point implementations with a fraction size greater than or equal to 22, results for an XC2V4000-FF1152-5 are reported. Performance measurements are based on the reports from the Xilinx ISE 5.2i development tools.

8.2.1 Arithmetic Library

Three measurements were used to evaluate the performance of the operators: the maximum frequency as reported by the Xilinx tools, the logic resource utilization and the BlockRAM memory utilization.

The exponent wordlength of the floating point implementation and the integer part of the LNS system were fixed to be 8 bits in width. This configuration is similar

Figure 8.1 Memory usage of ADD, MUL and $x^{-3/2}$ (number of Virtex-II 18-Kbit BlockRAMs).

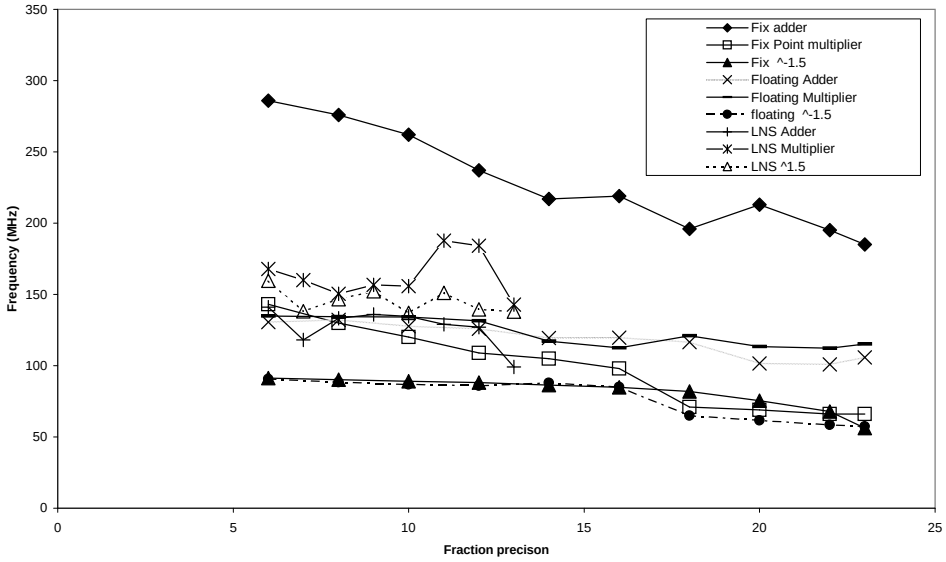


to the IEEE 754 single precision standard and can operate without overflow in our simulations. For all 3 number systems, the SUB operations has similar performance to the ADD operation, and therefore they are not shown in the figure.

The number of BlockRAM memory resources required for the $x^{-3/2}$ operator are plotted in Figure 8.1. This is determined by the memory requirements of the STAM tables for both fixed and floating point implementations. As can be seen in the figure, since the floating point implementation uses the fixed point STAM for its significand, the memory requirements are identical. For the LNS implementations, $x^{-3/2}$ can be computed by multiplying by -1.5 and no memory resources were used.

The operating frequency and logic utilization are plotted against the number of fractional bits for different operators and number systems in Figures 8.2 and 8.3

Figure 8.2 Frequency comparison of the ADD, MUL and $x^{-3/2}$ operators.



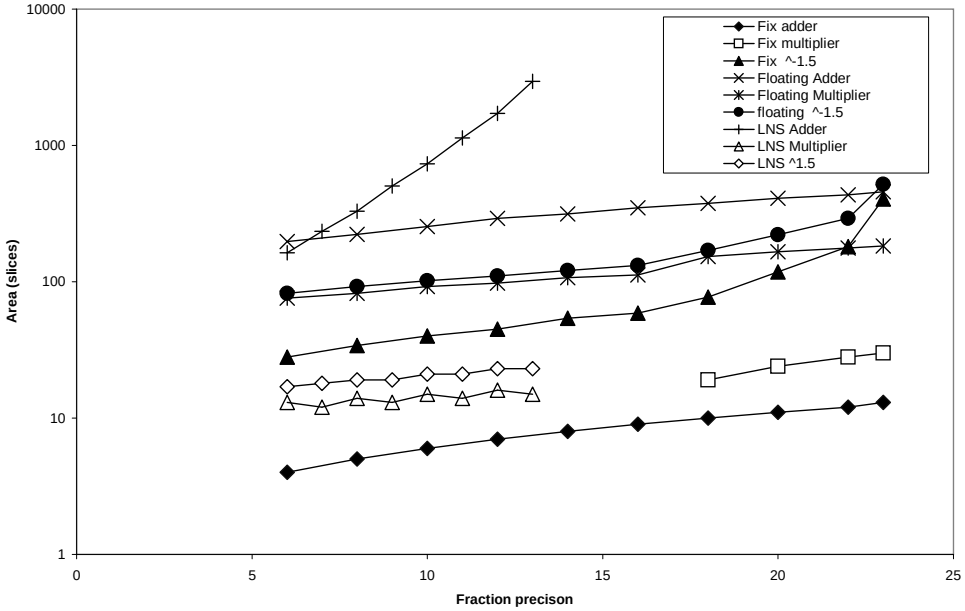
respectively. These tables can be used to compare different implementations, precisions and numbering systems in the CAST arithmetic library, allowing a quantitative assessment of which approach is most suitable for a given application. Note that the LNS library [aEL06] has a maximum LNS fractional wordlength of 13-bits and this limitation is carried over to CAST.

8.2.2 N-body Coprocessor

Using the CAST system, implementations of the N-body coprocessor with different fractional wordlengths using the fixed point, floating point and LNS number systems were made. The exponent wordlength of the floating point implementation and the integer part of the LNS system were fixed to be 8 bits in width.

In order to show the ability of CAST to deal with several number systems, an implementation, similar to GRAPE-3 [OME+93] was built. In this hybrid format, a similar configuration as GRAPE-3 was used and thus a $(20, 10)_{\mathcal{I}}$ fixed point format

Figure 8.3 Area utilization of the ADD, MUL and $x^{-3/2}$ operators.



was used to represent the position vectors of the particles and calculate the difference between the position vectors. The difference was then converted to a $(15, 6)_{\mathcal{L}}$ bit LNS format, which was used for all subsequent operations in calculating the partial force. The partial force was converted to a $(28, 28)_{\mathcal{I}}$ fixed point format which was accumulated to obtain the sum in Equation 7.2.

The implementations were simulated using the CAST system to evaluate their accuracy. To compare the precision of various implementations, the relative mean squared error $S_r(s)$, introduced in [ABLM98], was used. The relative mean square error measures the error in force calculation between a pair of particles and is defined as:

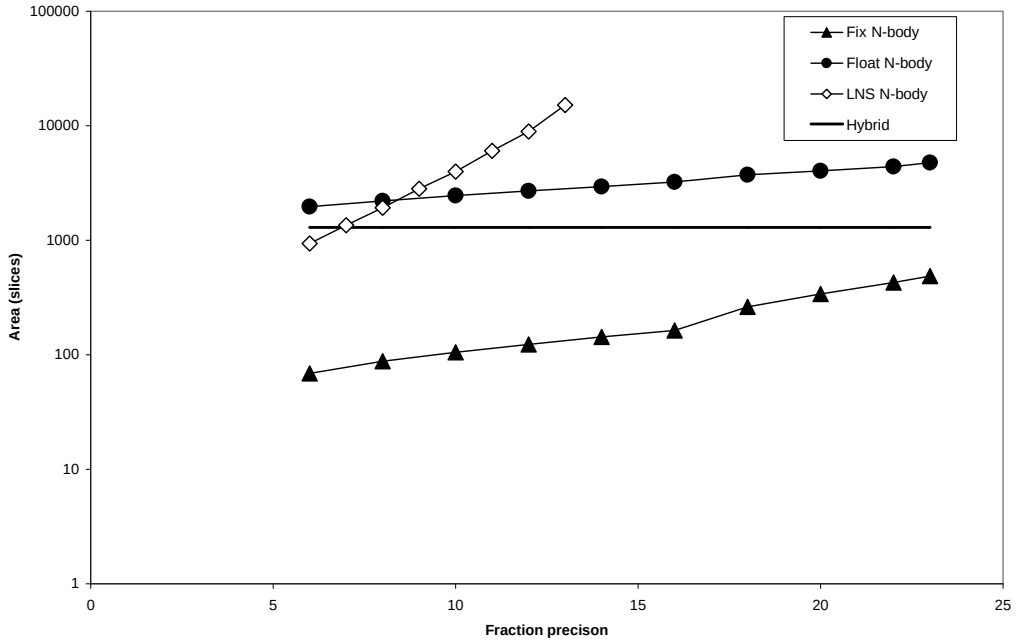
$$S_r(\mathbf{f}) = \frac{|\hat{\mathbf{f}} - \mathbf{f}|^2}{\mathbf{f}^2} \tag{8.3}$$

where $\mathbf{f}$ is the force computed by the hardware coprocessor and $\hat{\mathbf{f}}$ is the reference value computed using IEEE double precision floating-point arithmetic. Since the relative mean square error depends on the distance between the 2 particles, pairs of particles with varying distance r was generated ($\epsilon = 0$) and the errors computed. The resulting error function $\sqrt{S_r(s)}$ was plotted against r to obtain the error curves of Figure 8.6. The average error curve for GRAPE-3 [MT98, ABLM98] is also shown for comparison. The fixed point implementation suffered from overflow for small r and underflow for large r due to insufficient dynamic range for this problem leading to large errors. Thus we do not consider fixed point to be a good representation for this problem.

A comparison of the area utilization for different numerical representations and fractional wordlengths is given in Figure 8.4. As expected, fixed point has the smallest area requirements. The LNS system has smaller area than floating point up to 8 bits, after which floating point is smaller. The main area overhead for the LNS lying in the addition operations which requires a large number of slices for large fraction sizes. The hybrid implementation has area between fixed and floating.

The reported maximum clock frequency for the different schemes is given in Figure 8.5. The fixed point implementation has the highest operating frequency and

Figure 8.4 Area comparison of N-body implementations.

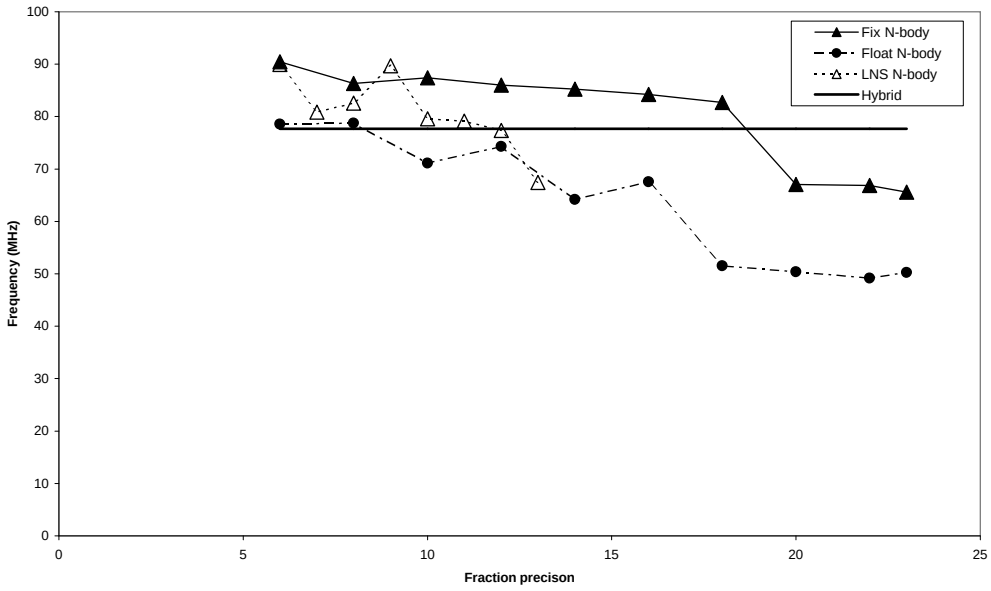


the floating point implementation is the slowest. The LNS and hybrid implementations achieve operating frequencies between the two.

If comparable accuracy to GRAPE-3 for the entire input range is desired, as mentioned earlier, the fixed point implementation is not suitable. This leaves the floating point $(21, 12)_{\mathcal{F}}$, LNS $(21, 11)_{\mathcal{L}}$ and hybrid implementations as candidates. By comparing their area and frequency requirements in Figures 8.4 and 8.5, it can be seen that the hybrid implementation offers a smaller area and higher frequency than the other two candidates. Thus, for the N-body example presented, based on considerations of precision, and circuit area, the hybrid implementation appears to be the most suitable implementation scheme for the Xilinx Virtex-II XC2V1000FG456-5 device chosen. If, for example, a different device such as a Virtex device which does not have dedicated multipliers, were to be used, the tradeoffs may be different, and the same methodology could be used to aid in making the best choice.

A comparison of area and frequency suggested that the hybrid implementation was the best solution. Different constraints on precision, area and speed may lead to different choices, easily identified from the graphs obtained.

Figure 8.5 Performance comparison of N-body implementations.



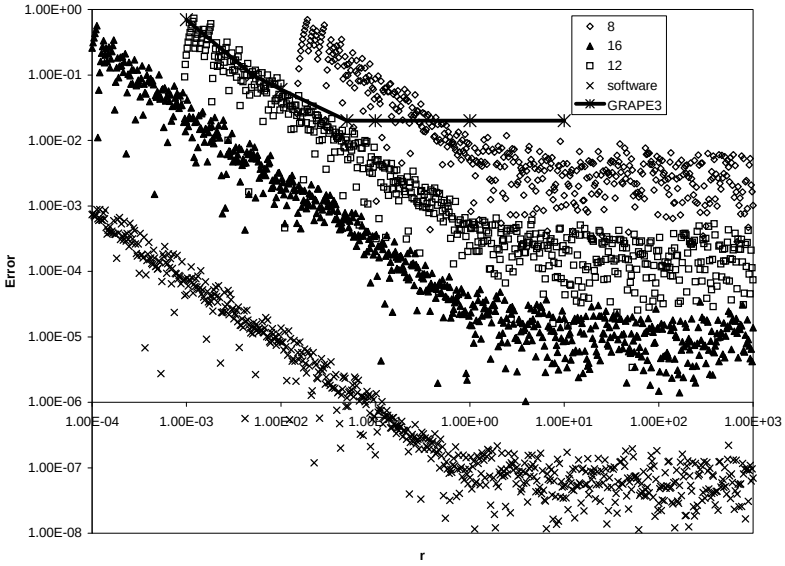
8.3 Summary

From the above results, we see that the datapath generated using CAST framework can improve performance by quickly explore different design tradeoffs.

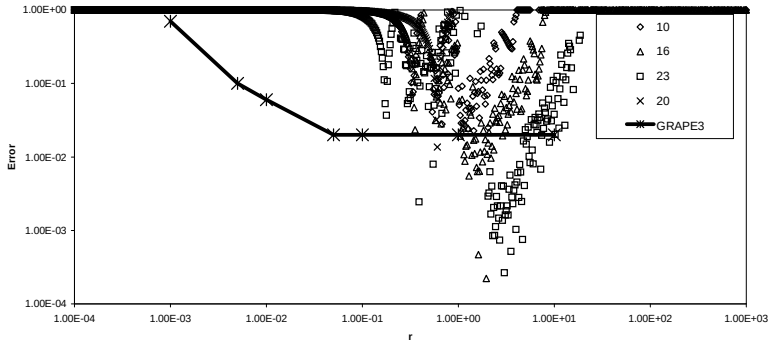
The bit width optimization in the BGM simulation requires generating a large number of datapaths. The built-in search function finds circuits that balance area and performance while fulfilling the precision constraints. The use of the CAST system significantly reduces the development time in this case.

In the N-body force pipeline example, the hybrid scheme is difficult to develop using traditional design techniques. Mixing different number systems in N-body pipeline is done in the CAST framework with the unified interface and built-in number system conversion objects.

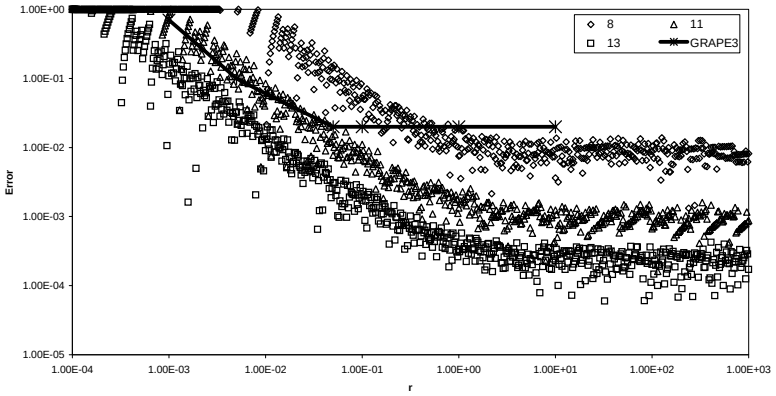
Figure 8.6 Quantization error for force calculation in the N-body problem.



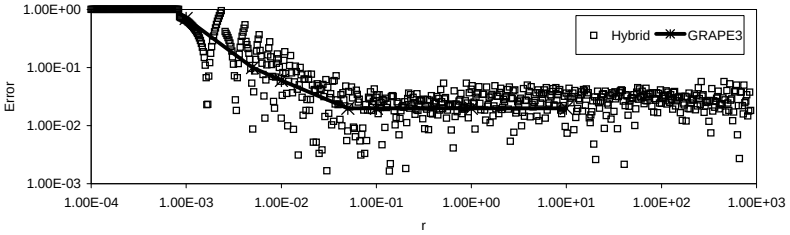
(a) Floating-point quantization error.



(b) Fixed-point quantization error.



(c) LNS quantization error.



(d) Hybrid quantization error.

Chapter 9

Conclusion

The CAST system was developed to provide a framework to design and optimize datapaths which are treated as arithmetic operator networks. The set of unified arithmetic libraries and associated helper functions made this framework an ideal environment of building hardware accelerator for simulation systems on reconfigurable platform. Through several application examples of simulation systems, this thesis demonstrated that the proposed design methodology can be used to optimize datapaths in design in various levels and achieve significant improvements.

In the N-body force pipeline example, it is shown that the system can optimize datapaths by maxing different number systems in a same design. This is the highest level when optimization is performed and the results shows huge improvement of using single number system. The unified configuration interface of CAST library components simplifies this optimization process over previous manual methods.

In the parallel multiplier example, the system's ability to optimize individual operators was demonstrated using different arithmetic algorithms. The example demonstrated that CAST can be used to evaluate performance of arithmetic algorithms on target hardware. This helps users decide on suitable implementation schemes, subject to given constraints.

The Monte Carlo simulation system shows CAST's ability to optimize data paths at the lowest hardware level on a reconfigurable platform. By fine tuning

the bit width of each operator in the design, CAST can automatically generate improved datapaths with smaller footprint while maintaining the required accuracy.

All these achievements are based on the novel idea of capturing both arithmetic and hardware design expertise in a unified framework and considering different abstract levels in optimization.

In the future, we would like to enhance the arithmetic library in CAST by adding more number representations (e.g. redundant and residue number systems), arithmetic schemes (e.g. online arithmetic, division, square root etc), and incorporate existing libraries (e.g. the Xilinx LogiCore library, the UCLA Astra library for online arithmetic [EPM02] and the floating-point module generator in [LTM03]) into the framework.

Bibliography

[ABLM98] E. Athanassoula, A. Bosma, J.-C. Lambert, and J. Makino. Performance and accuracy of a GRAPE-3 system for collisionless N-body simulations. In *Monthly Notices of the Royal Astronomical Society*, pages 369–380, Feb 1998.

[aEL06] Aremnaire Project at ENS Lyon. A vhdl library of parametrisable floating-point and lns operators for fpga. 2006.

[BGM97] A. Brace, D. Gątarek, and M. Musiela. The market model of interest rate dynamics. *Mathematical Finance*, 7(2):127–155, April 1997.

[BH98] P. Bellows and B. Hutchings. JHDL - an HDL for reconfigurable systems. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 175, Washington, DC, USA, 1998. IEEE Computer Society.

[BHW96] Ronen Barzel, John F. Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation*, pages 183 – 197, 1996.

[Boo51] A. D. Booth. A signed binary multiplication technique. *Quart. J. Mechanical and Applied Math.*, 4:235–240, 1951.

[BTLM06] Jacob A. Bower, David B. Thomas, Wayne Luk, and Oskar Mencer. A reconfigurable simulation framework for financial computation. In

IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig), pages 1–9, 2006.

[Cho97] S. Chongwe. Simulation of aerodynamics problem on a distributed shared-memory machine. In *High Performance Computing on the Information Superhighway*, pages 93–98, 1997.

[CLS02] Wanqiang Chen, Register L.F., and Banerjee S.K. Simulation of quantum effects along the channel of ultrascaled Si-based MOSFETs. *IEEE Transactions on Electron Devices*, 49:652 – 657, 2002.

[CM94] C.P. Cowen and S. Monaghan. A reconfigurable Monte-Carlo clustering processor (mccp). In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 59–65, 1994.

[DJW03] Long D.G., Luke J.B., and Plant W. Ultra high resolution wind retrieval for seawinds. In *IEEE International Proceedings of Geoscience and Remote Sensing Symposium (IGARSS)*, pages 1264– 1266, 2003.

[ECS94] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. *Network Working Group*, RFC 1750, 1994.

[EL04] M.D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[EM96] R. Even and B. Mishra. Cafe: a complex adaptive financial environment. In *IEEE/IAFE 1996 Conference on Computational Intelligence for Financial Engineering*, pages 20–25, 1996.

[EPM02] M. Ercegovac, J. Pipan, and R. Mcllhenny. ASTRA: Arithmetic scripting tool for reconfigurable architectures. 2002. http://unagi.cs.ucla.edu/Astra.

[FD02] Viktor Fischer and Milos Drutarovsky. True random number generator embedded in reconfigurable hardware. In *Proceedings of the Cryptographic Hardware and Embedded Systems Workshop (CHES)*, pages 415–430, 2002.

[Fly01] Michael J. Flynn. *Advanced computer arithmetic design*. Wiley, 2001.

[FMC84] R.C. Fairfield, R.L. Mortenson, and K.B. Coulthart. An LSI Random Number Generator (RNG). In *Advances in Cryptography: Proceedings of Crypto 84*, pages 203–230. LNCS 0196, Springer-Verlag, 1984.

[Fu95] Michael C. Fu. Pricing of financial derivatives via simulation. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 126–132, 1995.

[GAM02] Lienhart G., Kugel A., and R. Manner. Using floating-point arithmetic on fpgas to accelerate scientific n-body simulations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 182–191, 2002.

[GBP01] A. Gerosa, R. Bernardini, and S. Pietri. A fully integrated 8-bit, 20MHz, truly random numbers generator, based on a chaotic system. In *SSMSD. 2001 Southwest Symposium on Mixed-Signal Design*, pages 87–92, 2001.

[GFA+04] M. Gokhale, J. Frigo, C. Ahrens, J.L. Tripp, and R. Minnich. Monte Carlo radiative heat transfer simulation. In *Proceedings of the IEEE Conference on Field-Programmable Logic and Applications (FPL)*, pages 95–104, 2004.

[GJS03] Liang Ge, S. Casey Jones, and Fotis Sotiropoulos. Numerical simulation of flow in mechanical heart valves: Grid resolution and the assumption of flow symmetry. *Journal of Biomechanical Engineering*, 125:709 – 718, 2003.

[GM98] J.D. Golic and R. Menicocci. Edit distance correlation attack on the alternating step generator. In *Advances in Cryptology: Crypto '97*, pages 499–512, 1998.

[Gun88] C.G. Gunther. Alternating step generators controlled by de bruijn sequences. In *Advances in Cryptology: Proceedings of Eurocrypt 87*, pages 5–14, 1988.

[GVH06] Yongfeng Gu, Tom VanCourt, and Martin C. Herbordt. Improved interpolation and system integration for fpga-based molecular dynamics simulations. In *FPL '06: Proceedings of Field-Programmable Logic and Applications*, pages 1–8, 2006.

[HBH+99] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting. A CAD suite for high-performance FPGA design. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–24, April 1999.

[HGG+05] H.H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, and H. Klar. Emulation engine for spiking neurons and adaptive synaptic weights. In *IEEE International Joint Conference on Neural Networks*, pages 3261 – 3266, 2005.

[HLT+02] C. H. Ho, Philip Heng Wai Leong, K. H. Tsoi, Ralf Ludewig, Peter Zipf, Alberto Garcia Ortiz, and Manfred Glesner. Fly - a modifiable hardware compiler. In *FPL '02: Proceedings of the Reconfigurable*

Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications, pages 381–390, London, UK, 2002. Springer-Verlag.

[HTY+03] C.H. Ho, K.H. Tsoi, H.C. Yeung, Y.M. Lam, K.H. Lee, P.H.W. Leong, R Ludewig, P. Zipf, A.G. Ortiz, and M. Glesner. Arbitrary function approximation in HDLs with application to the N-body problem. In *2003 IEEE International Conference on Field-Programmable Technology (FPT)*, pages 84–91, Dec 2003.

[Hul00] J.C. Hull. *Option, Futures, and Other Derivatives*. Prentice Hall, 2000.

[IEE85] IEEE. *IEEE standard for binary floating-point arithmetic: ANSI/IEEE std 754-1985*. 1985.

[Inc02] Xilinx Inc. *Xilinx Core Generator*. http://www.xilinx.com/-ipcenter/, 2002.

[Int99] Intel Platform Security Division. The intel random number generator. *Intel technical brief*, 1999. ftp://download.intel.com/design/security/rng/techbrief.pdf.

[JK99] B. Jun and P. Kocher. The Intel random number generator. *White paper by Cryptographic Research Inc.*, 1999. ftp://download.intel.com/design/security/rng/CRIwp.pdf.

[Knu81] D. Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*. Addison-Wesley, 1981.

[Kor93] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[Kor02] I. Koren. *Computer Arithmetic Algorithms*. A.K. Peters, 2nd edition, 2002.

[LKM02] G. Lienhart, A. Kugel, and R. Manner. Using floating-point arithmetic on fpgas to accelerate scientific n-body simulations. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 182–191, 2002.

[LLC+01] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Lee. Pilchard – a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[LRN+01] X. Le, J. Rasty, A. Neuber, J. Dickens, and M. Kristiansen. Calculation of air temperature and pressure history during the operation of a magnetic flux compression generator. In *IEEE Conference of Pulsed Power Plasma Science*, pages 224–, 2001.

[LS07] P. L'Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators. *to appear on ACM Transactions on Mathematical Software*, 33, 2007.

[LTM03] J. Liang, R. Tessier, and O. Mencer. Floating Point Unit Generation and Evaluation for FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 185–194, 2003.

[Mac61] O. L. MacSorley. High speed arithmetic in binary computers. *Proc. IRE*, 49:67–91, 1961.

[Mak05] Junichiro Makino. Modified simd architecture suitable for single-chip implementation, 2005.

[Men02] O. Mencer. PAM-Blox II: Design and evaluation of C++ module generation for computing with FPGAs. In *Proceedings of the*

IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 67–76, 2002.

[Met93] M. Metzger. Modelling and simulation of transient states in the heat distribution network. In *International Conference on Systems, Man and Cybernetics*, pages 136–141, Oct 1993.

[MFK00] Junichiro Makino, Toshiyuki Fukushige, and Masaki Koga. A 1.349 tflops simulation of black holes in a galactic center on GRAPE-6. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 43, Washington, DC, USA, 2000. IEEE Computer Society.

[MLBP03] J.M. McCollum, J.M. Lancaster, D.W. Bouldin, and G.D. Peterson. Hardware acceleration of pseudo-random number generation for simulation applications. In *Proceedings of the 35th Annual Southeastern Symposium on System Theory*, pages 299–303, March, 2003.

[MMF98] Oskar Mencer, Martin Morf, and Michael J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 167–174, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[MS01] J. Moody and M. Saffell. Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, 12:875 – 889, 2001.

[MT98] Junichiro Makino and Makoto Taiji. *Scientific Simulation with Special-Purpose Computers - the GRAPE systems*. John Wiley & Sons Ltd, 1998.

[MTES97] J. Makino, M. Taiji, T. Ebisuzaki, and D. Sugimoto. Grape-4: A massively parallel special-purpose computer for collisional N-body simulations. In *ApJ 480*, pages 432–446, 1997.

[MvOV97] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NEM] NEMO - A Stellar Dynamics Toolbox. In *http://bima.astro.umd.edu/nemo/*.

[NM65] J. Nelder and R. Mead. A simplex method for function minimization. *Computer*, 7:308–313, 1965.

[NSE+99] T. Narumi, R. Susukita, T. Ebisuzaki, G. McNiven, and B. Elmegreen. Molecular dynamics machine: Special-purpose computer for molecular dynamics simulations. In *Molecular Simulation*, pages 401–415, 1999.

[OME+93] S. K. Okumura, J. Makino, T. Ebisuzaki, T. Fukushige, T. Ito, D. Sugimoto, E. Hashimoto, K. Tomida, and N. Miyakawa. Highly parallelized special-purpose computer, GRAPE-3. In *Field Programmable Logic and Applications*, volume 45, pages 329–338, 1993.

[Osk06] Mencer Oskar. ASC: A Stream Compiler for Computing with FPGAs. *IEEE Transactions on Computer-Aided Design*, 2006.

[OVL96] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Trans. Comput.*, 45(3):294–306, 1996.

[Pag96] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[PAL96] A. Postula, D. Abramson, and P. Logothetis. The design of a specialised processor for the simulation of sintering. In *Proceedings of the 22nd EUROMICRO Conference*, pages 501–508, 1996.

[Pat00] C. Patterson. High Performance DES Encryption in Virtex FPGAs using JBits. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 113–121, 2000.

[PC00] C.S. Petrie and J.A. Connelly. A noise-based IC random number generator for applications in cryptography. *IEEE Journal of Solid State Circuits*, 47(5):615–621, 2000.

[PDA01] G. Picinbono, H. Delingette, and N. Ayache. Nonlinear and anisotropic elastic soft tissue models for medical simulation. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1370 – 1375, 2001.

[PG03] Ben Popoola and Paul Gough. Evaluating the performance of space plasma simulations using fpga's. In *High Performance Computing for Computational Science - VECPAR 2002*, pages 169–188, 2003.

[PPvR05] Raoul Pietersz, Antoon Pelsser, and Marcel van Regenmortel. Fast drift approximated pricing in the bgm model. Finance 0502005, EconWPA, February 2005. available at http://ideas.repec.org/p/wpa/wuwpfi/0502005.html.

[Pro05] The Grape Project. http://astrograpе.org, 2005.

[Raz96] B. Razavi. A study of phase noise in cmos oscillators. *IEEE Journal of Solid-State Circuits*, 31(3):331–343, 1996.

[Ruk01] A. Rukhin. *A Statistical Test Suit For Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST Special Publication 800-22, 2001.

[SMOR98] Paul F. Stelling, Charles U. Martel, Vojin G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Trans. Comput.*, 47(3):273–285, 1998.

[SPK01] Toni Stojanovski, Johnny Pil, and Ljupco Kocarev. Chaos-based random number generators. Part II: practical realization. *IEEE Transactions on Circuits and Systems – I: fundamental Theory and Application*, 48(3):382–385, March 2001.

[SS97] Michael J. Schulte and James Stine. Symmetric bipartite tables for accurate function approximation. In Tomas Lang, Jean-Michel Muller, and Naofumi Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 175–183, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[SS99a] James E. Stine and Michael J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[SS99b] J.E. Stine and M.J. Schulte. The symmetric table addition method for accurate function approximation. In *Journal of VLSI Signal Processing*, pages 167–177, 1999.

[SSL01] J. Schulz-Stellenfleth and S. Lehner. Ocean wave imaging using an airborne single pass across-track interferometric sar. *IEEE Transactions on Geoscience and Remote Sensing*, 39:38 – 45, 2001.

[THYL04] K.H. Tsoi, C.H. Ho, H.C. Yeung, and P.H.W. Leong. An arithmetic library and its application to the N-body problem. In *Proceedings*

of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 68–78, 2004.

[TL05] K.H. Tsoi and P.H.W. Leong. Mullet - a parallel multiplier generator. In *Proceedings of the IEEE Conference on Field-Programmable Logic and Applications (FPL)*, pages 691–694, 2005.

[TLL03] K.H. Tsoi, K.H. Leung, and P.H.W. Leong. Compact FPGA-based true and pseudo random number generators. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 51–61, 2003.

[TLL07] K.H. Tsoi, Ka Ho Leung, and Philip H.W. Leong. A high performance physical random number generator. *IEE Proc. Computers & Digital Techniques*, 2007. Accepted for publication, March 2007.

[TTJD91] Ito T., Ebisuzaki T., Makino J., and Sugimoto D. A special-purpose computer for gravitational many-body systems: Grape-2. In *PASJ 43*, pages 547–555, 1991.

[U.S94] U.S. Department of Commerce. *Security Requirements for Cryptographic Modules*. Federal Information Processing Standards Publication FIPS 140-1, 1994.

[WF82] Shlomo Waser and Michael J. Flynn. *Introduction to arithmetic for digital systems designers*. Holt, Rinehart and Winston, 1982.

[Xil00a] Xilinx Inc. *Virtex 2.5V field programmable gate arrays*, 2000.

[Xil00b] Xilinx, Inc. *Xilinx Coregen Reference Guide*, 2000. Version 3.1i.

[Xil02] Xilinx Inc. *Virtex-E Extended Memory: Detailed Functional Description*, 2002.

[Xil03] Xilinx Inc. *Virtex-II Pro^{TM} Platform FPGAs: Complete Data Sheet*, Oct. 2003. Advance Product Specification, DS083.

[Xil04a] Xilinx Inc. *ML310 Development Platform*, 2004.

[Xil04b] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, 2004. Version 3.3.

[YJ00] Wen-Chang Yeh and Chein-Wei Jen. High-speed booth encoded parallel multiplier design. *IEEE Transactions on Computers*, 49:692–701, 2000.

[YOFA04a] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano. Stochastic simulation for biochemical reactions on fpga. In *Proceedings of the IEEE Conference on Field-Programmable Logic and Applications (FPL)*, pages 105–114, 2004.

[YOFA04b] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano. Stochastic simulation for biochemical reactions on fpga. In *Proceedings of the IEEE Conference on Field-Programmable Logic and Applications (FPL)*, pages 105–114, 2004.

[ZHF+07] Ye Zhao, Yiping Han, Zhe Fan, Feng Qiu, Yu-Chuan Kuo, Kaufman A.E., and Mueller K. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13:179–189, 2007.

[ZLH+05] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 215–222, 2005.

[ZYR91] K. Zheng, C.H. Yeng, and T.R.N. Rao. An improved linear syndrome algorithm in cryptanalysis with applications. In *Advances in Cryptology: Crypto '90*, volume LNCS 537, pages 34–47, 1991.

Publications

Journals

- K.H. Tsoi, Ka Ho Leung, and Philip H.W. Leong. A high performance physical random number generator. IEE Proc. Computers & Digital Techniques, 2007. Accepted for publication, March 2007.

Conference Papers

- K.H. Tsoi and P.H.W. Leong. Mullet - a parallel multiplier generator. In Proc. International Workshop on Field Programmable Logic and Applications (FPL), pages 691-694, 2005.
- G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for monte carlo based financial simulation. In Proc. International Conference on Field Programmable Technology (FPT), pages 215-222, 2005.
- K.H. Tsoi, C.H. Ho, H.C. Yeung, and P.H.W. Leong. An arithmetic library and its application to the N-body problem. In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 68-78, 2004.
- K.H. Tsoi, K.H. Leung, and P.H.W. Leong. Compact FPGA-based true and

pseudo random number generators. In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 51-61, 2003.