Hardware Acceleration of Monte-Carlo Structural Financial Instrument Pricing Using a Gaussian Copula Model

by

Alexander Kaganov

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto

Hardware Acceleration of Monte-Carlo Structural Financial Instrument Pricing Using a Gaussian Copula Model

Alexander Kaganov
Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto
2008

Abstract

In recent years the financial world has seen an increasing demand for faster risk simulations, driven by increasing contract complexity and client portfolio growth. Traditionally many financial models employ Monte-Carlo simulation, which can take excessively long to compute in software. Hence, commonly a hardware accelerator is sought out.

This thesis focuses on accelerating structured financial instruments, namely Collateralized Debt Obligations (CDOs) pricing, which have previously not been targeted for hardware acceleration despite their prominence in the financial market. This thesis presents a hardware implementation for the One-Factor and the Multi-Factor Gaussian Copula models. It also explores the precision requirements and the resulting resource utilization for each such numerical representation.

The results show that the hardware implementation mapped onto a Xilinx XC5VSX50T chip is over 64 and 71 times faster than corresponding software running on a 3.4 GHz Intel Xeon processor, for the One-Factor and the Multi-Factor models, respectively.

Acknowledgements

First I would like to thank my supervisor Professor Paul Chow for his guidance and technical advice which were invaluable throughout the past two years. Thank you for letting me be a part of this project. I would also like to thank Asif Lakhany from the Quantitative Research Department of Algorithmic Inc. for introducing the project and his suggestions throughout the year. I also thank Daniel and Emanuel for their suggestions and conversations, Manuel and Arun for their expert advice. Finally, I would like to thank all the remaining group members Chris Madill for software support, Danny, David, Ali, Andrew, Keith, Daniel Ly, Danyao, Nihad, and Matt for their support.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contributions . . . 2

1.3 Overview . . . 3

2 Background **4**

2.1 Collateralized Debt Obligation Structure . . . 4

2.2 Pricing Equations . . . 6

2.2.1 One-Factor Gaussian Copula Model . . . 8

2.2.2 Multi-Factor Gaussian Copula Model . . . 12

2.3 Related Work . . . 13

3 Random Number Generation **17**

3.1 Uniform Random Number Generation . . . 18

3.2 Gaussian Random Number Generators . . . 22

3.2.1 Central Limit Theory Methods . . . 25

3.2.2 Transformation Method . . . 27

3.2.3 Acceptance-Rejection Method . . . 28

3.2.4 Inverse Transformation Method . . . 31

3.2.5 Wallace . . . 31

3.2.6 Gaussian Number Generator Summary . . . 33

3.3 Wallace Hardware Generator . . . 34

4 Hardware Implementation **43**

4.1 Design Specifications and Limitations . . . 43

4.2 Multi-Core Simulation Architecture . . . 45

4.3 One-Factor Gaussian Copula Hardware Implementation . . . 46

4.4 Multi-Factor Gaussian Copula Hardware Extension . . . 49

5 Test Methodology **51**
5.1 Benchmarks 51
5.2 Test and Verification Methodology 54

6 Results **57**
6.1 OFGC Core Results 57
6.1.1 OFGC: Precision Versus Utilization Tradeoffs 57
6.1.2 OFGC: Performance 60
6.2 MFGC Core Results 66
6.2.1 MFGC Resource Utilization 66
6.2.2 MFGC: Performance 68
6.3 Requirements 70

7 Conclusions **72**
7.1 Summary 72
7.2 Future Work 74

Appendices **76**

A Diehard Statistical Tests **77**

B Multi-Factor Gaussian Copula Performance Results **82**

Bibliography **85**

List of Tables

3.1 Tausworth-88 88-bit LFSR resource Utilization 22
3.2 GRNGs Resource Summary, All Designs are Mapped to XC2V4000-6 34
3.3 Virtex 2 Wallace Hardware Implementation Resource Summary. XC2V4000-6 40
3.4 Virtex 5 Wallace Hardware Implementation Resource Summary. XC5VSX50T-3 40

5.1 Test Benchmarks. 52
5.2 Standard and Poor's Rating Explanation 53

6.1 OFGC Core Resource Utilization 59
6.2 Factor Accumulation Module Resource Utilization 67
6.3 MFGC Core Resource Utilization 67
6.4 MFGC Performance Summary 70

A.1 88-bit LFSR Diehard Test Suite Results 78
A.2 Tausworth-88 Diehard Test Suite Results 79
A.3 Sum of 12 URN Diehard Test Suite Results 80
A.4 32-bit Hardware Wallace generater Diehard Test Suite Results 81

List of Figures

2.1 Collateralized Debt Obligation Structure. 5
2.2 Tranche Losses Step Function Relationship to Pool Losses 7
2.3 Y_i Intersection with $\Phi^{-1}[P(\tau_i < t_l)]$ 11

3.1 8-bit LFSR 19
3.2 Tausworth-88 Uniform Random Number Generator 21
3.3 Zero Mean Unit Variance Gaussian PDF 23
3.4 Zero Mean Unit Variance Gaussian CDF 24
3.5 Standard Normal Distribution vs. Sum of 12 URN 26
3.6 Sum of 12 URNG Hardware Architecture 27
3.7 Gaussian distribution divided into rectangular, wedge, and tail regions in the Ziggurat method 29
3.8 Wallace Hardware Architecture 36
3.9 Address Correction Mechanism 37
3.10 Wallace Transformation Circuitry. Stage 3 of the Overall Architecture 39
3.11 Application Result Differences using a Modified Wallace and a Randn() Over 20 Seeds 42

4.1 Multi-Core Simulation Architecture. 45
4.2 One-Factor Gaussian Copula Hardware Core 47
4.3 Factor Accumulation Module 50

5.1 On-Chip CDO Pricing Engine Testbench 56

6.1 Single-Core OFGC Performance 60
6.2 Multi-Core OFGC Performance 61
6.3 Acceleration vs. Number of Time Steps Across Eight Benchmarks 62
6.4 Acceleration vs. Number of Instruments Across All Nine Benchmarks 63
6.5 % of Execution Time Dedicated to Gaussian Number Generation vs. Number of Simulation Time Steps 63

6.6 Benchmark 9: Acceleration vs. Number of Instruments 65
6.7 Benchmark 9: Acceleration vs. Number of Time Steps 65
6.8 MFGC Benchmark 1: CDX.NA.HY Performance 68

B.1 MFGC Benchmark 2: CDX.NA.IG Performance 82
B.2 MFGC Benchmark 3: CDX.NA.IG.HVOL Performance 82
B.3 MFGC Benchmark 4: CDX.NA.XO Performance 83
B.4 MFGC Benchmark 5: CDX.EM Performance 83
B.5 MFGC Benchmark 6: CDX.DIVERSIFIED Performance 83
B.6 MFGC Benchmark 7: CDX.NA.HY.BB Performance 84
B.7 MFGC Benchmark 8: CDX.NA.HY.B Performance 84
B.8 MFGC Benchmark 9: Semi-Homogenous Performance 84

List of Acronyms

ALU	Arithmetic Logic Unit
BRAM	Block Random Access Memory
CAST	Computer Arithmetic Synthesis Tool
CBO	Collateralized Bond Obligation
CDF	Cumulative Distribution Function
CDO	Collateralized Debt Obligation
CDS	Credit Default Swap
Cell BE	IBM's Cell Broadband Engine
CLO	Collateralized Loan Obligation
CLT	Central Limit Theorem
DDR SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplinx Link
FSM	Finite State Machine
GPU	Graphical Processing Unit
GRNG	Gaussian Random Number Generator
HPC	High Performance Computing

LFSR	Linear Feedback Shift Register
MAC	Multiply-Accumulate
MC	Monte-Carlo
MFGC	Multi-Factor Gaussian Copula
MPI	Message Passing Interface
OFGC	One-Factor Gaussian Copula
PCI	Peripheral Component Interconnect
PDF	Probability Density Function
PLB	Processor Local Bus
RNG	Random Number Generator
RUF	Replication Utilization Factor
SFRF	Systemic Factor Replication Factor
SPV	Special Purpose Vehicle
TRF	Time Replication Factor
URNG	Uniform Random Number Generator
VaR	Value at Risk

Chapter 1

Introduction

1.1 Motivation

In recent years, the financial industry has seen an increasing demand for High Performance Computing (HPC). This demand can typically be attributed to three factors:

1. Increasing contract/model complexity. Financial engineering is a relatively new and fast growing field. As the field keeps evolving, newer and more complex models are developed to better represent real world scenarios [1]. This necessitates the use of more generic Monte-Carlo (MC) methods, which are typically slow in software due to their $O\left(\frac{1}{\sqrt{N}}\right)$ (where N is the number of MC paths) convergence rate [2]. To improve the accuracy of a typical MC simulation by a factor of 10, one has to calculate 100 times more paths, which in turn would run 100 times slower.

2. Increasing portfolio size. As the financial market grows, more contracts are created, and a larger number of instruments are issued. A prominent example of this increase can be seen in the derivative market, which is a subset of financial instruments that derive their value based on the market behavior of a different instrument. The amount of outstanding derivatives has more than doubled from \$260 trillion in December 2004 to \$596 trillion in December 2007 [3]. Hence, even using the same model, now twice as much data has

to be processed.

3. The need to make real-time decisions. In the finance world, market trends can change quickly and nowadays with most instruments traded electronically a small computational delay can result in a significant monetary loss.

While typical financial MC simulations are computationally intensive, they are also highly parallel. This has traditionally made them good candidates for hardware acceleration. However, all previously recorded work has targeted pricing individual instruments within a portfolio [4–11] (see Section 2.3 for details), leaving out a whole class of more complex financial instruments, termed structured instruments. These instruments are typically backed by a whole portfolio of different assets and are created to transfer the risk associated with these assets using a different risk-profile than the portfolio itself. Pricing these instruments requires a more complicated model that evaluates the portfolio as a whole. In collaboration with the research department of a financial risk management software organization, Algorithmics Inc [12], one specific type of structural instrument was examined, Collateralized Debt Obligations (CDOs). CDO has recently been one of the fastest growing instruments, with its total global issuance more than tripling from US\$157 Billion in 2004 to US\$552 Billion in 2006, and despite the recent sub-prime US mortgage crisis, the 2007 issuance still surpassed US\$502 Billion [13], as of August 2008.

1.2 Contributions

This thesis proposes a hardware implementation for pricing a CDO using one of the most widely used Monte-Carlo models proposed by Li in 2000 [14]. Its main contributions are as follows:

1. A multi-core architecture that allows simultaneous off-chip data transfers and computations.

2. Optimized hardware implementations for both single and the multi-factor[1] models, which exploit coarse- as well as fine-grain parallelism inherit in the algorithms.

3. A detailed design space exploration with different precision representations of the data, examining result accuracy versus resource utilization tradeoffs.

4. A performance comparison between a CDO simulation running on a single FPGA against an equivalent representation running in software on a single-processor.

1.3 Overview

The remainder of this thesis is organized as follows. Chapter 2 provides a detailed description of a CDO mechanism, introduces the mathematical pricing model, and provides a summary of previous hardware-accelerated financial applications. Chapter 3 consists of a short summary of software and hardware random number generation techniques and details the process of choosing and modifying a hardware generator to meet the requirments of this work. Chapter 4 describes the hardware implementation of the overall simulator as well as the individual pricing cores. Chapter 5 presents the benchmarks used to test the design and the overall test methodology. Chapter 6 presents the accuracy versus resource utilization tradeoffs, as well as describes and discusses the obtained performance. Finally, Chapter 7 summarizes the results and proposes future research directions.

[1]where the number of factors is the number of external stimuli affecting a given asset

Chapter 2

Background

This chapter provides a brief overview of key concepts on Collateralized Debt Obligation pricing. Section 2.1 presents the overall CDO structure. Section 2.2 introduces key Eqns. essential to both MC and analytical CDO pricing. Finally, Section 2.3 discusses some previous works in hardware acceleration of financial applications.

2.1 Collateralized Debt Obligation Structure

A typical financial organization can own a variety of risky debt obligations as part of its asset portfolio, such as: bonds, loans, credit default swaps (CDS), and even CDOs. To mitigate the risk associated with owning debt obligations the financial company, termed the *sponsor*, creates a separate entity called a Special Purpose Vehicle (SPV), to isolate CDO investors from its own credit risk. The *sponsor* then either sells the actual debt obligations to the SPV, or just the risk associated with them using a CDS, while the actual assets stay with the *sponsor*. The SPV groups all the debt obligations into a collateral pool and issues portions of the pool, termed tranches, to the investors in return for interest payments, as shown in Fig.2.1 [15]. A CDO is typically named based on the constituency of the original pool of assets. For instance, if the initial portfolio consists of bond-type instruments the CDO is referred to as Collateralized Bond Obligation (CBO); conversely if the portfolio consists only of loans the CDO is called

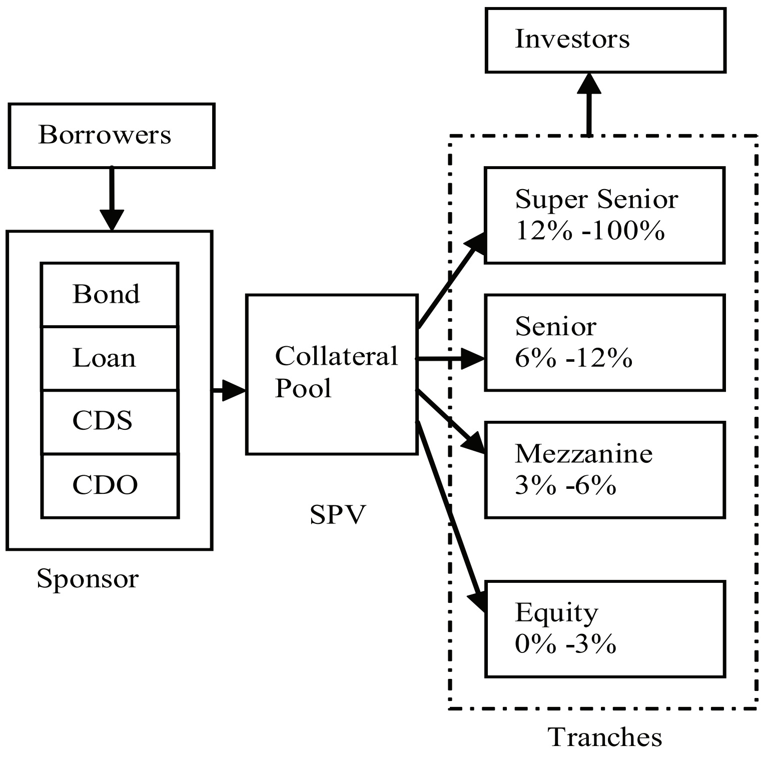

Figure 2.1: Collateralized Debt Obligation Structure.

Collateralized Loan Obligation (CLO). There are typically three motives for a *sponsor* to create a CDO:

- Create market arbitrage: generate money through the difference between tranche sales and premium payment on these tranches.
- Shrink balance sheet: Banks are regulated regarding the number of risky assets it can hold. By selling these assets they can increase returns on risky capital. Finally, the simplest reason is that shrinking the balance sheet simplifies accounting.
- Risk Transfer: Sell the risk of owning a portfolio with debt assets to the investors in return for premium payments. [16]

Each CDO tranche has an attachment and a detachment point. When the cumulative losses in the collateral pool exceed the attachment point of a given tranche, the investors in the tranche start to lose their principal, but continue to receive interest payment on the remaining amount.

For example, an investor purchases the Mezzanine tranche for \$900, which has an attachment of 3% and a detachment of 6%. Suddenly, the pool losses reach 4% of the total pool, inactivating the Equity tranche and affecting the first $1/3$ of the Mezzanine tranche. In turn, the investor will lose \$300, $1/3$ of his original investment, but will receive interest payments on the remaining \$600. When the cumulative losses reach the detachment point the tranche investors lose their entire investment and the tranche is no longer active. Hence, the difference between the attachment and detachment points define the tranche width.

Each tranche has a different risk factor and hence a different return profile. As can be seen from Fig. 2.1, the Equity tranche, with the 0% attachment point, is the riskiest tranche which pays the highest interest, while the Super Senior tranche, with the 12% attachment point, is the safest with the lowest return payments.

2.2 Pricing Equations

In order to issue tranches and establish the amount of interest a tranche investor should receive, an SPV has to determine how much a tranche is really worth. In this section let us take a closer look at how the different CDO tranches are priced. First, let us introduce the variable used in defining pricing problem for a specific tranche:

- A: Attachment point.
- D: Detachment point.
- $S = D - A$: Tranche width.
- s: Premium spread. Investment return rate.
- t_l: Premium dates. Periods at which investors receive premium payments, set relative to the initial CDO tranche issuance date.$0 < t_1 < t_2 < ... < t_k = T$, where T denotes the CDO maturity date

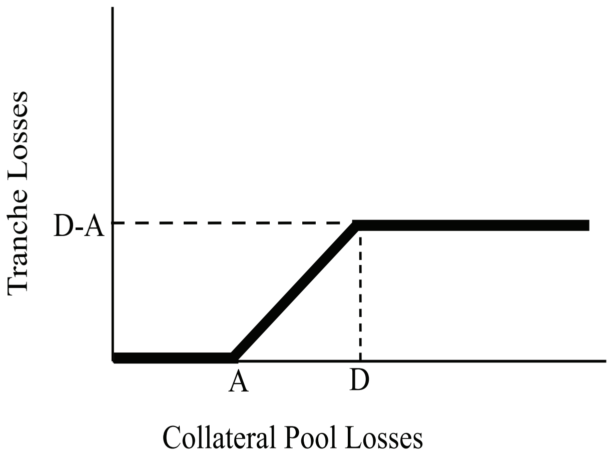

Figure 2.2: Tranche Losses Step Function Relationship to Pool Losses

- d_l: Discount factor for premium date t_l.
- $L(t_l)$:Cumulative collateral pool losses up to premium date t_l.

The tranche losses, $\hat{L}(t_l)$, can be derived from the pool losses using the tranche width and the attachment point:

$$\hat{L}(t_l) = \min\left(S, \max\left(L(t_l) - A, 0\right)\right), \tag{2.1}$$

The tranche losses can be seen as a step function, see Fig. 2.2. If the pool losses are below the attachment point there are no tranche losses; if the pool losses are above the detachment point the tranche losses are constant at the tranche width; between detachment and attachment points tranche losses are linearly proportional to pool losses.

In pricing a CDO tranche there are two important quantities termed *default leg*, Eqn. 2.2, and *premium leg*, Eqn. 2.3 [17]. The default leg measures the expected losses of the tranche over the life of the contract, i.e. the amount a tranche investor will lose from his/her principal investment. The premium leg measures the expected premiums that the tranche investor will receive over the life of the contract.

$$\text{Default Leg} = E\left[\sum_{l}^{k}\left(\hat{L}(t_l) - \hat{L}(t_{l-1})\right) d_l\right], \tag{2.2}$$

$$\text{Premium Leg} = E\left[\sum_{l}^{k} s\left(S - \hat{L}(t_l)\right) d_l\right], \tag{2.3}$$

The value of a tranche to the investor is:

$$\text{Premium Leg} - \text{Default Leg}, \tag{2.4}$$

The tranche width, the discount factors, and the premium spread are fixed hence the valuation of the problem is reduced to computing $E[\hat{L}(t_l)]$, expected tranche loss at a given time instance. Next this work will look into Monte-Carlo evaluation of $E[\hat{L}(t_l)]$ using Li's Gaussian Copula method [14]. Section 2.2.1 introduces the One-Factor Gaussian Copula (OFGC) model. Section 2.2.2 expands the idea to a Multi-Factor Gaussian Copula (MFGC) model.

2.2.1 One-Factor Gaussian Copula Model

In 2000, Li [14] introduced a Gaussian Copula model for estimating collateral pool losses. The flexibility and the simplicity of the model established it as one of the most prominent methods of pricing CDOs. This section will focus on the one factor model. The number of factors refers to the number of global stimuli affecting all the instruments within portfolio. In addition to the previously introduced variables, for a pool of n instruments a few new variables are added:

- N_i: Recovery adjusted notional, which is the monetary amount a financial institution can recover in the case the i^{th} asset defaults.
- α_i: Correlation factor between the state of global market and asset i.
- β_i: $\sqrt{1-\alpha_i^2}$ The influence of local fluctuations on asset i.
- τ_i: The time at which asset i defaults.

Li proposed modeling default probabilities of each asset using a Poisson process with a default intensity parameter λ_i. The probability of the i^th asset defaulting prior to time t_l becomes:

$$P\left(\tau_i < t_l\right) = 1 - \exp\left(-\lambda_i t_l\right), \tag{2.5}$$

The curve $P\left(\tau_i < t\right)$ is known as a default boundary curve for asset i. Typically, these curves are grouped based on credit rating. Assets with the same credit rating would have the same default curve. A standard portfolio would usually consist of instruments from different rating groups and hence would have a few different default curves. Furthermore, the model assumes that the default probabilities relate to a random variable Y_i by:

$$P\left(\tau_i < t_l\right) = P\left(Y_i < y_i\right), \tag{2.6}$$

Where Y_i is:

$$Y_i = \alpha_i X + \beta_i Z_i, \tag{2.7}$$

Where X and Z_i are independent zero mean unit variant Gaussian random variables. X is called a systemic factor that represents the global stimuli, which is constant for all assets in the pool for a given MC path. Z_i is the idiosyncratic factor, unique to each asset.

Since, both X and Z_i follow "standard" normal distributions, Y_i is also normally distributed. From that it follows:

$$y_i = \Phi^{-1}\left[P\left(\tau_i < t_l\right)\right], \tag{2.8}$$

Where Φ is the normal cumulative distribution function.

Combining Eqns. (2.8), (2.7), and (2.6), and conditioning on the market condition $X = x$, the default probability becomes:

$$\begin{aligned}\pi_i(t_l, x) &= P(Y_i < \Phi^{-1}[P(\tau_i < t_l)] \| X = x) \\ &= P(\alpha_i X + \beta_i Z_i < \Phi^{-1}[P(\tau_i < t_l)] \| X = x) \\ &= \Phi\left(\frac{\Phi^{-1}[P(\tau_i < t_l)] - \alpha_i x}{\beta_i}\right),\end{aligned} \tag{2.9}$$

Analytical Framework

In analytical methods the full probabilistic distribution of pool losses is computed using Eqn 2.9 as the conditional probability that a given instrument defaults given a certain market condition. There are various approaches to calculating this distribution [15, 17, 18], each imposes a different restriction on the dataset and provides a different tradeoff between accuracy and computation time. The most generic way of calculating the full probabilistic distribution of pool losses for a given market condition, while making no implicit assumption about the data, is by permuting all possible losses multiplied by their corresponding probabilities. To derive the full distribution let us first introduce the probability that the total loss in a pool with n instruments, conditioned on the market state, is equivalent to l. Assume that the probability of loss for $k-1$ instruments is $P_{k-1}[L = l \| X = x]$, using recursion for k instruments this probability becomes:

$$P_k[L = l \| X = x] = P_{k-1}[L = l - N_k | X = x]\pi_k(t, x) + P_{k-1}[L = l \| X = x](1 - \pi_k(t, x)), \tag{2.10}$$

Which is basically the probability that the $k-1$ instrument pool had l losses times the probability that the k^{th} instrument does not default, plus the probability that the $k-1$ instrument pools had $l - N_k$ losses times the probability the k^{th} instrument defaults. From that the expected losses of an n instrument pool with respect to the market state x are:

$$E_x[L(t_l)] = \sum_{m=0}^{\text{Portfolio Size}} mP[L(t_l) = m \| X = x], \tag{2.11}$$

Combining Eqns. 2.11 and 2.1, and integrating over all market scenarios one obtains the

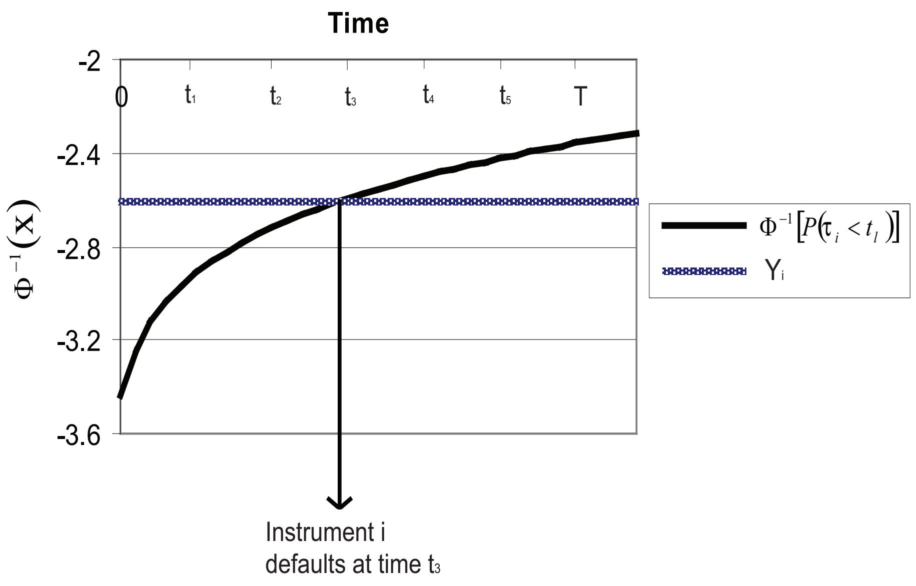

Figure 2.3: Y_i Intersection with $\Phi^{-1}[P(\tau_i < t_l)]$

expected tranche losses:

$$E[\hat{L}(t_l)] = \int_{-\infty}^{\infty} \min(S, \max(E_x[L(t_l)] - A, 0)) d\Phi(x), \tag{2.12}$$

This integral is most commonly estimated using the Gaussian Quadrature Rule [17], which converts it to a weighted sum, homogenous portfolio assumption or alternatively using Vasicek's large portfolio approximation.

Monte-Carlo Framework

In the Monte-Carlo approach instead of calculating the full default distribution at each path, X and Z_i variables are generated and used to sample Eqn. 2.9. This is the equivalent to searching for an intersection between Y_i and $\Phi^{-1}[P(\tau_i < t_l)]$ and recording a default at the time instance that they intersect, see Fig. 2.3.

At each path the overall collateral pool losses for a given time instance t_l and market conditions x are:

$$L(t_l, x) = \sum_{i=1}^{N} N_i I(Y_i(x), t_l), \tag{2.13}$$

Where $I(Y, t_l)$ is the indicator function:

$$I(Y, t_l) = \left\{ \begin{array}{ll} 1 & Y < \Phi^{-1}\left[P(\tau < t_l)\right] \quad 0 \leq t_l \leq T \\ 0 & \text{Otherwise} \end{array} \right\}, \tag{2.14}$$

Combined with Eqn. 2.1 the tranche losses at a given MC path are:

$$\hat{L}(t_l, x) = \min(S, \max(L(t_l, x) - A, 0)), \tag{2.15}$$

The final expected value for the actual tranche loss is the average of all MC paths:

$$E[\hat{L}(t_l, x)] = \frac{1}{\text{\# of Paths}} \sum_{j=1}^{\#ofPaths} \hat{L}(t_l, x), \tag{2.16}$$

2.2.2 Multi-Factor Gaussian Copula Model

Although the OFGC model is prominently used due to its simplicity and ease in which it can be implemented in software, it has limitations in representing real market scenarios [19]. In a real market there is often more than a single stimuli affecting the portfolio, which leads to the expansion of the OFGC model to the multi-factor model, MFGC. In the multi-factor model there could be m different market stimuli affecting a given portfolio. The systemic factor X is replaced by a vector $X_1, X_2, ..., X_M$ of independent zero mean Gaussian variants and Eqn. 2.7 for Y_i becomes:

$$Y_i = \sum_{j=1}^{M} \alpha_{ij} X_j + \sqrt{1 - (\sum_{j=1}^{M} \alpha_{ij}^2) Z_i}, \tag{2.17}$$

Introducing multiple factors also changes the correlation between the portfolio instruments. Previously, when all instruments were only related by a single factor X the correlation between instrument i and j was $\alpha_i\alpha_j$ [17], which is also the covariance between Y_i and Y_j. The multi-factor model introduces more flexibility and the correlation between Y_i and Y_j becomes $\sum_{k=1}^{M} \alpha_{ik}\alpha_{jk}$. This allows having instruments in the portfolio with no correlation between them by setting the corresponding α_{ik} or α_{jk} to zero, which is closer to a real world situation than the OFGC model.

Let **X** represent the vector $X_1, X_2, ..., X_M$, and $\boldsymbol{\alpha_i}$ the vector $\alpha_1, \alpha_2, ..., \alpha_M$, then Eqn. 2.9 becomes:

$$\pi_i(t_l, \mathbf{x}) = \Phi\left(\frac{\Phi^{-1}[P(\tau_i < t_l)] - \boldsymbol{\alpha_i x}}{\sqrt{1 - \boldsymbol{\alpha_i \alpha_i^t}}}\right), \tag{2.18}$$

The remaining calculations in the analytical and Monte-Carlo sections remain the same with new Y_i and $\pi_i(t_k, \mathbf{x})$. The integral in Eqns. 2.12 is extended to M-dimensions, over all systemic factors.

2.3 Related Work

In recent years there have been a significant number of attempts to accelerate financial simulations. Many of these works exploit the idea of heterogeneous computing, where a typical processor is paired with a hardware accelerator device [20], typically a Field Programmable Gate Array (FPGA), a Graphical Processing Unit (GPU), or another accelerator such as IBM's Cell Broadband Engine (Cell BE). In such a heterogeneous system the processor usually just handles data transfer while the computationally intensive calculation is performed on the accelerator. This section will discuss some of these financial accelerators. In examining different implementations it is very difficult to appropriately compare their merit in terms of performance, even if they attempt to accelerate the exact same instrument, due to the use of different reference processor architectures and the lack of a standardized reference code and benchmarks. Hence, this section makes no attempt to compare the implementations but just reports the results.

One of the most targeted classes of applications in the financial world is option pricing. An option gives the holder the right to either buy or sell an asset by a certain date for a set price. There are three main types of options: European, American, and Asian. A European option can only be exercised at the expiration date. An American option can be exercised at anytime before the expiration date. An Asian option differs from both European and American in that its payoff is the average of the asset price over a period of time rather than the current value of the asset. Traditionally, each of these options can be priced using a variation on Black-Scholes model [2]:

$$\frac{dS}{S} = rdt + \sigma dW, \tag{2.19}$$

Where S is the current asset price, r is continuously compounded interest rate, σ is the volatility of the asset, t is continous time, and W is a Weiner process. Even though certain options, like European, have an analytical solution many authors still choose to use MC pricing as a financial engineering benchmark to illustrate the capabilities of their system to perform MC simulations.

Baxter *et al.* [4] present MC Asian option pricing on a 64-FPGA supercomputer, MAXWELL. MAXWELL is a true heterogeneous supercomputer with 32 fully interconnected 2.8Ghz Xeon processors and 64-FPGAs connected in a two dimensional 8X8 torus. The authors claimed that their single FPGA implementation, Virtex 4 FX100, was 322 times faster than their corresponding software implementation. Furthermore, they demonstrated linear scalability of their design by distributing the calculation across 16 FPGA nodes. Unfortunately, the authors do not provide any information regarding the numerical precision used in their design.

As part of a testbench for RapidMind's Development Platform, McCool *et al.* [5] implemented a floating point single precision European option pricing on an NVIDIA 7900 GTX GPU. They demonstrate that using their software flow one can achieve a maximal acceleration of 32-fold compared to an optimized code running on an AMD Opteron 2.6 GHz processor, and over 120-fold compared to a naive software implementation. The paper compares a processor versus a GPU over an increasing number of MC paths, and shows that the GPU performance improves as the number of paths increases while the processor performance remains constant.

Morris *et al.* [6] present a comprehensive comparison for European option pricing between an FPGA, a GPU and a Cell BE. They compare their implementations on each platform in terms of both acceleration and precision. For FPGA designs the authors created a Hyper-Streaming abstraction built on top of the Handel-C [21] programming language, which according to them significantly shortens design time. Using their tool the authors created five designs: floating point double-precision, floating point single-precision, and 48, 32, 18-bit fixed precisions. Their smallest and fastest FPGA design, 18-bit fixed-point, provided the best performance. On a Virtex 4 LX160 it displayed a 146-fold acceleration over a 2.6GHz AMD Opteron processor, while exhibiting a similar error to the single-precision floating point core. It is interesting to note that the single-precision floating point implementations displayed similar acceleration across the three different platforms: 41-fold on an FPGA, 32-fold on an NVIDIA 7900 GTX GPU, and 29-fold on a Cell BE.

Another interesting paper, which explores different hardware architectures for MC based option pricing and portfolio valuation, is [7]. The paper presents five different MC simulation types: Random Walk, Random Jump, Geometric Walk, Bi-Variant Walk, and GARCH Walk. Using a Virtex-4 SX55 device, implementation results show that the designs run on-average 80 times faster than equivalent software running on a Xeon 2.66GHz processor, with a range between 49 times to 137 times faster, seen for GARCH and Random Walk simulations, respectively.

The value of every financial instrument over time heavily depends on the market's interest rate. This makes interest rate predictions crucial in predicting the future worth of an instrument. Hence, some authors choose to accelerate interest rate simulations. Bower *et al.* [8] propose a generic hardware architecture for multi-chip MC simulations, which they use to design hardware implementations of Ho-Lee's and Vasicek's interest rate models. Using eight replications of a (16,16) fixed-point core on a Spartan 3 5000 the authors obtained an eight-fold acceleration over a vectorized version of the software code running on a 2.66GHz Xeon processor. Five replicas of a single-precision floating point Vasicek core, fixed-point was not implemented, was seven times faster than the corresponding optimized software version. Zhang *et al.* [9] implemented a highly optimized LIBOR market model, also known as the BGM (Brace, Gatarek and Musiela) interest rate model. Using the Computer Arithmetic Synthesis Tool (CAST) the authors found the minimal number of bits needed for fixed-point and floating point representations for various stages within the simulation. Using that number of bits a full design was created on a Virtex 2 Pro 30 that showed a 25x speed-up over a Pentium IV 1.5GHz.

Thomas *et al.* [10] and Fixstars Corporation [11] accelerated Value at Risk (VaR) calculations. In VaR simulation one calculates the worst case scenario losses that can occur $(100 - x)\%$, where x is the confidence level, of the time. For instance, the losses calculated for $x = 95\%$ is the amount of money that would be lost in the five worst trading days out of every 100 days. The authors from Fixstars Corporation implemented VaR on a Cell BE. They demonstrated that by parallelizing the program to run on a Cell BE a 16-fold speedup can be achieved compared to a 3 GHz Xeon processor. Thomas *et al.* [10] introduced an architecture for fast Gaussian vector generation on an FPGA, and as a validation of their design the authors built a VaR engine that was supplied data from the vector generator. As a result, Thomas *et al.* demonstrated that their design was 132 times faster than a single core 2.2 GHz Athlon processor.

All of the cases mention above targeted pricing individual portfolio instruments. This thesis presents an implementation for structured instruments that requires a completely different model.

Chapter 3

Random Number Generation

Random Number Generators (RNGs) are essential components of Monte-Carlo simulations. Each MC path has to have its own set of pseudo random numbers that govern the results obtained in that path. Over the years, Gaussian Random Numbers (GRNs) have been very commonly used in such simulation models. This is mostly due to the extent to which they have been studied, the existent framework for analyzing them, and in many cases the Gaussian distribution does in fact closely mimic real world scenarios.

In creating a hardware MC simulation it was necessary, as seen by Eqn. 2.7, to select an efficient method for generating GRNs. Initially, in selecting an appropriate generator the following criteria was considered:

- Efficient mapping to hardware, i.e. low resource utilization
- Produces atleat one new Gaussian variate every clock cycle. This restriction allowed for higher design throughput and simplified the overall design. However, later modifications allow this restriction to be removed and any GRNG to be substituted into the overall architecture.
- High quality random numbers. The generator had to pass the Diehard test suite [22], which is one of the most comprehensive publicly available test of randomness, and the

overall application with the chosen GRNG had to have statistically similar results to running the same application with a standard Matlab randn() function [23].

To meet these criteria, five different methods of generating GRN were explored. See Section 3.2 for details on each method alongside with a specific algorithm presented for each and the corresponding hardware implementation, which has previously been created in other works. Section 3.3 presents the modifications made to Lee's [24] hardware Wallace generator for use within this thesis alongside with the corresponding statistical analysis. Every single method encountered for generating a GRN requires a high quality Uniform Random Number (URN) as an input. Therefore, Section 3.1 provides a brief discussion of URNs.

3.1 Uniform Random Number Generation

Methods for generating higher order random distributions often depend on the existence of a sequence of random variables $U_1, U_2, U_3, , U_n$ [25]. The properties of this sequence are such that:

1. Each U_i is uniformaly distributed between 0 and 1
2. The U_i's are mutually independent; i.e. U_i cannot be predicted based on knowing $U_{i-1}, U_{i-2}, ..., U_1$ [2]

One of the most widely used methods for generating U_i's is using linear recurrence generators in modulos 2, i.e. binary arithmetic. These generators could be represented by a sequence of states, $x_1, x_2, x_3, ..x_n$, a k-by-k transformation matrix A that takes state x_i to x_{i+1} and a w-by-k matrix B that converts k-bit state x_i to w-bit output u_i [26]

$$x_{i+1} = Ax_i \quad u_i = Bx_i, \tag{3.1}$$

Subsequently u_i can either be transformed to a real number in the range (0,1] by dividing by 2^w or just interpreted as an integer, (0,2^w-1). In such a sequence as soon as a single number

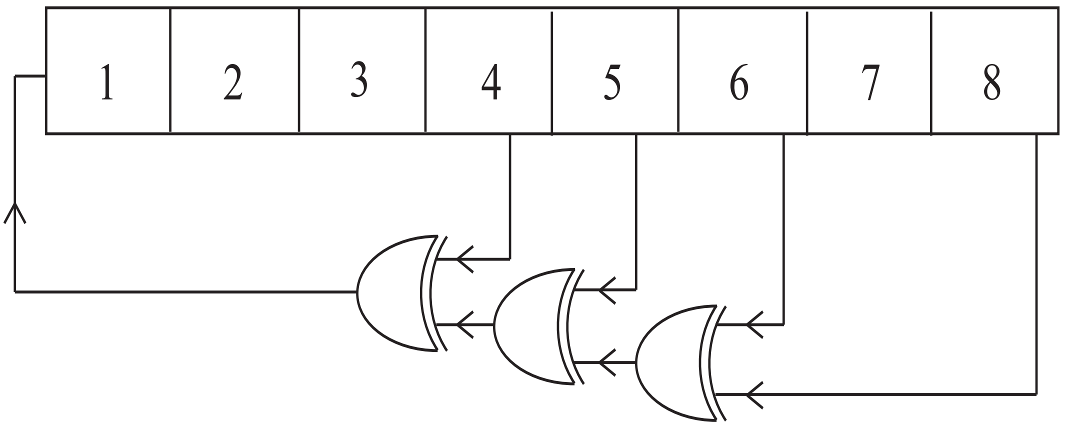

Figure 3.1: 8-bit LFSR

is repeated the whole sequence is repeated. Hence, the largest achievable period is 2^w-1. The state $x_i = 0$ should be unreachable because for any A, $Ax_i = 0$, if $x_i = 0$. The requirement for achieving a maximum period is that the primitive polynomial, $P(z) = \det(A - zI)$, is primitive modulos 2 [27].

One of the most commonly used linear recurrence generators in hardware is a Linear Feedback Shift Register (LFSR). In a LFSR a new state is the shifted version of the previous state with the input bit being a linear function of the bits in the previous state, corresponding to the primitive polynomial. The input bit is given by:

$$b_{i+1} = w_1 b_i + w_2 b_{i-1} + ... + w_k b_{i-k}, \tag{3.2}$$

Where w_i are the primitive polynomial coefficients. Fig. 3.1 provides an example of an 8-bit LFSR with a primitive polynomial $P(z) = z^8 + z^6 + z^5 + z^4 + 1$. LFSRs are popular in hardware because they can often be implemented efficiently [26], for instance in a Xilinx chip the LFSR's shifter component could be implemented as an SRL16. However, every iteration a single LFSR can generate only 1-bit of new data, hence to generate a w-bit word w parallel instances are required.

A popular hardware alternative to an LFSR is a combined Tausworth generator [27]. Unlike the LFSR, a single Tausworth generator produces w output bits every iteration/cycle. A new w-bit sequence is generated from state x_i by taking a w-bit block every s bits. In a Tausworth generator u_i is given by [26]:

$$u_i = \langle b_{is+1}, b_{is+2}, ..., b_{is+w} \rangle \,, \tag{3.3}$$

A Tausworth primitive equation is usually a trinomial, $P(z) = z^k + z^q + 1$. As long as $2^k - 1$ and s are co-prime the period of a Tausworth generator is $2^k - 1$ [27].

Let us take a look into the Tausworth algorithm. The algorithm requires a proper initial seed, A, and a mask with k ones followed by $(w - k)$ zeros, C, then:

1. $B = A << q$;
2. $B = A \oplus B$; where $\oplus$ is a logical XOR
3. $B = B >> (k - s)$;
4. $A = A \& C$; where $\&$ is a logical AND
5. $A = A << s$;
6. $A = A \oplus B$;

The variable A generated at step 6 is used as both the output and the new state for the next iteration. For proper use of the algorithm, A has to be appropriately initialized using the following few steps. Set D to an arbitrary k-bit value followed by $w - k$ zeros, then:

1. $B = D << q$;
2. $B = D \oplus B$;
3. $B = B >> k$;
4. $A = B \oplus D$;

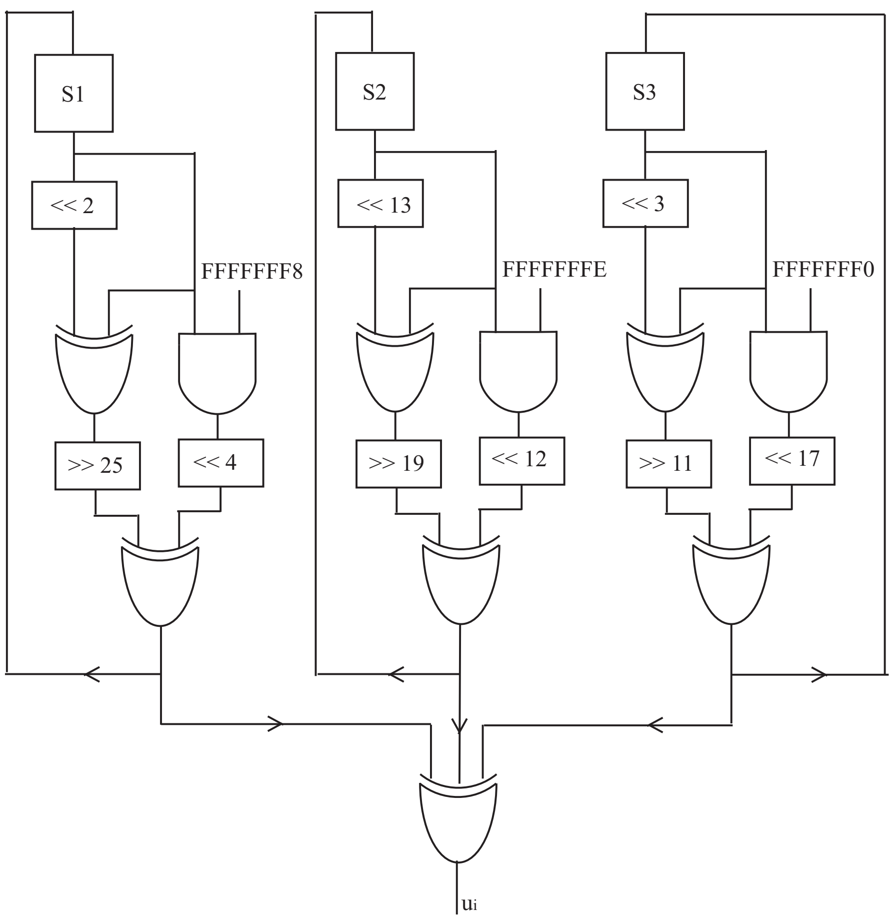

Figure 3.2: Tausworth-88 Uniform Random Number Generator

Tausworth generators are rarely used independently, usually multiple generators are combined together using an XOR. This is mainly due to two reasons. First, recurrences that are based on primitive polynomials with very few non-zero coefficients, like a trinomial, often produce numbers with poor statistical qualities. The second reason is more due to the limitations of a traditional processor, which typically consists of a 32-bit architecture limiting the maximum period of such a generator to $2^{32}-1$. A combined generator has better statistical qualities and a period length that is the product of the lengths of the comprising generators, as long as their periods are relatively prime. A commonly used, both in software and hardware, combined Tausworth generator is Tausworth-88, which consists of three independent generators combined to create a period of approximately 2^{88}. For a hardware schematic see Fig. 3.2.

Table 3.1: Tausworth-88 88-bit LFSR resource Utilization

	Tausworth-88	88-bit LFSR
Flip-Flops	88	160
6-Input LUTs	120	160

To select a hardware URNG for this project the resource utilization of a Tausworth-88 and an 88-bit LFSR were compared. The 88-bit LFSR was chosen to match the Tausworth-88 period of approximately 2^{88}. The primitive equation for the LFSR was $P(z) = z^{88} + z^{87} + z^{17} + z^{16} + 1$. Three million variates from each generator were sent through the Diehard test suite [22]. Both generators pass all the statistical tests, refer to Appendix A for results. Both designs, alongside with the mechanisms required to initialize the generator seeds, were mapped onto a Virtex 5 SX50T. The resource utilization can be seen in Table 3.1. The Tausworth-88 uses fewer resources, both Flip-Flops and LUTs. Due to the smaller resource utilization and its popularity in hardware designs [24, 28, 29] the Tausworth-88 generator was chosen.

3.2 Gaussian Random Number Generators

The generic Gaussian distribution is given by the Probability Density Function (PDF):

$$\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{3.4}$$

Where μ is the sample mean and σ^2 is the variance. Fig. 3.3 shows an example of a "standard" normal curve, which is a zero mean unit variance Gaussian curve.

By integrating the PDF we can obtain the probability function: $P[X < x]$:

$$P[X < x] = \Phi(x) = \int_{-\infty}^{x} \phi(x)dx, \tag{3.5}$$

Eqn. 3.5 is also known as the Gaussian Cumulative Distribution Function (CDF), a CDF for a "standard" normal curve can be seen in Fig. 3.4 [30].

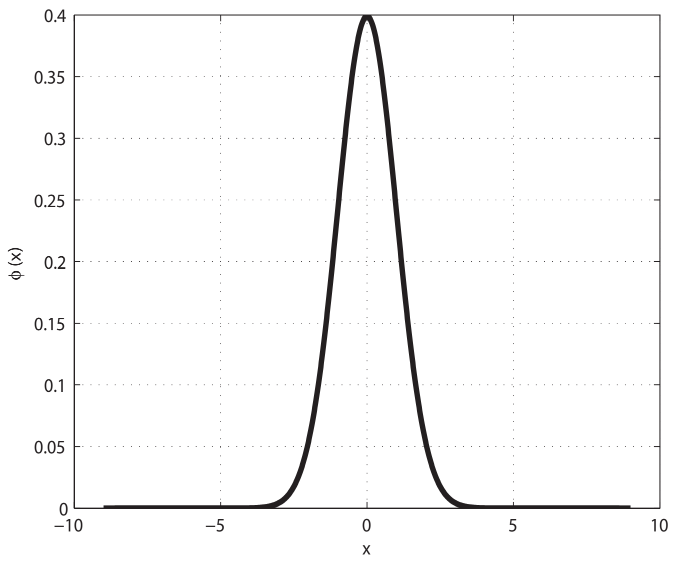

Figure 3.3: Zero Mean Unit Variance Gaussian PDF

There are a variety of ways of generating a Gaussian distribution. This section will take a look at some of these methods.

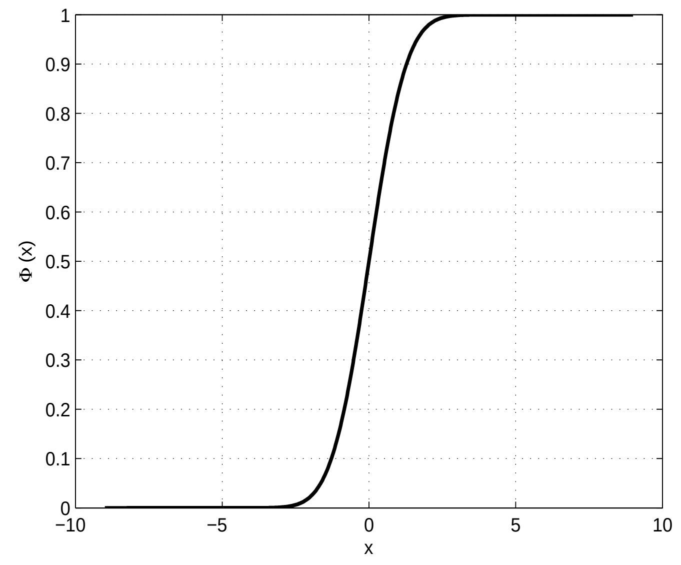

Figure 3.4: Zero Mean Unit Variance Gaussian CDF

3.2.1 Central Limit Theory Methods

The simplest method of generating a Normal distribution is by using the Central Limit Theorem (CLT), which states that the sum of a large number of independent and identically-distributed random variables will be approximately normally distributed. This means that if one sums a sufficiently large number of independent random variates of any distribution the resulting distribution will be Gaussian. Using the CLT the larger the number of summands the closer one is to a true normal distribution. For a rough estimate of a Gaussian distribution it's common to use the sum of 12 URN [23]:

$$\phi(x) \approx \sum_{i=1}^{12} U_i - 6, \tag{3.6}$$

Using 12 URNs allows one to have an efficient estimate of a "standard" normal without additional arithmetic manipulations. The reason for that is that each URN has a variance of $1/12$, combining 12 together creates a unit variance. However, there is a trade-off between the simplicity of the generator and the accuracy with which it estimates a normal distribution. First, if one just visually examines the distribution, Fig. 3.5 (on the left is a Gaussian distribution generated using an established Matlab generator [23], on the right is the distribution generated by adding 12 uniform variates), one can see that it appears very similar to a real normal distribution. Unfortunately, this simple generator does not holdup to statistical tests. First, the Diehard statistical test suite was applied. To use diehard tests, the output of the generator had to be converted to a uniform distribution. This was accomplished using an extension of the Box-Muller algorithm (Section 3.2.2), assuming x and y are normally distributed:

$$u = \exp\left(-\frac{(x^2+y^2)}{2}\right), \tag{3.7}$$

u is uniformaly distributed in (0,1). Three million newly formed u's failed one of the Diehard tests, the Sqeeze test (for full Diehard results refer to Appendix A). This indicates that the original assumption was wrong and the distribution was not Gaussian. The second test was

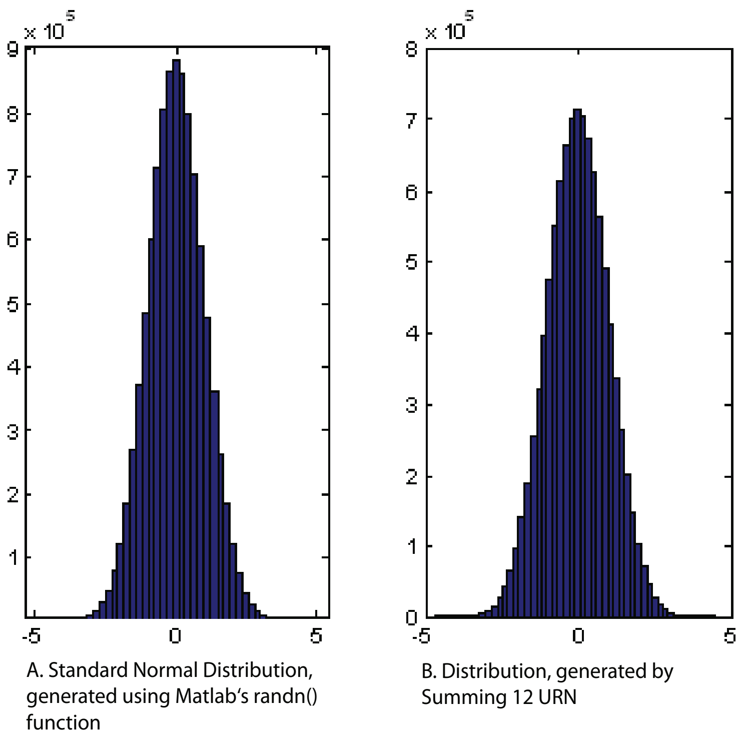

Figure 3.5: Standard Normal Distribution vs. Sum of 12 URN

the Chi-Squared test. One billion samples were divided into 100 equally spaced bins over the range [-7, 7] and the chi-squared test was applied using Eqn. 3.8:

$$\xi^2 = \sum_{i=1}^{\#bins} \frac{(O_i - E_i)^2}{E_i}, \tag{3.8}$$

Where O_i and E_i are the observed and the expected number of samples in bin i, respectively. The resultant ξ^2 of 1013 and p-value of 0 correspond to a failed test. P-value measures the chances of seeing a sample at a given distance from the ideal sample mean. A 0 p-value alongside with one failed Diehard test indicates that although the distribution created by summing up 12 URNs looks very similar to Gaussian distribution it is not a true Gaussian.

To properly compare this method to others, a hardware design was created for it, even though the statistical tests failed. The design consisted of 12 Tausworth-88 generators with a cascaded adder tree, see Fig. 3.6. For comparison purposes the design was mapped to a Virtex 2 XC2V4000-6 chip. The final design occupied 1016 Slices and could run at 247 Mhz, see

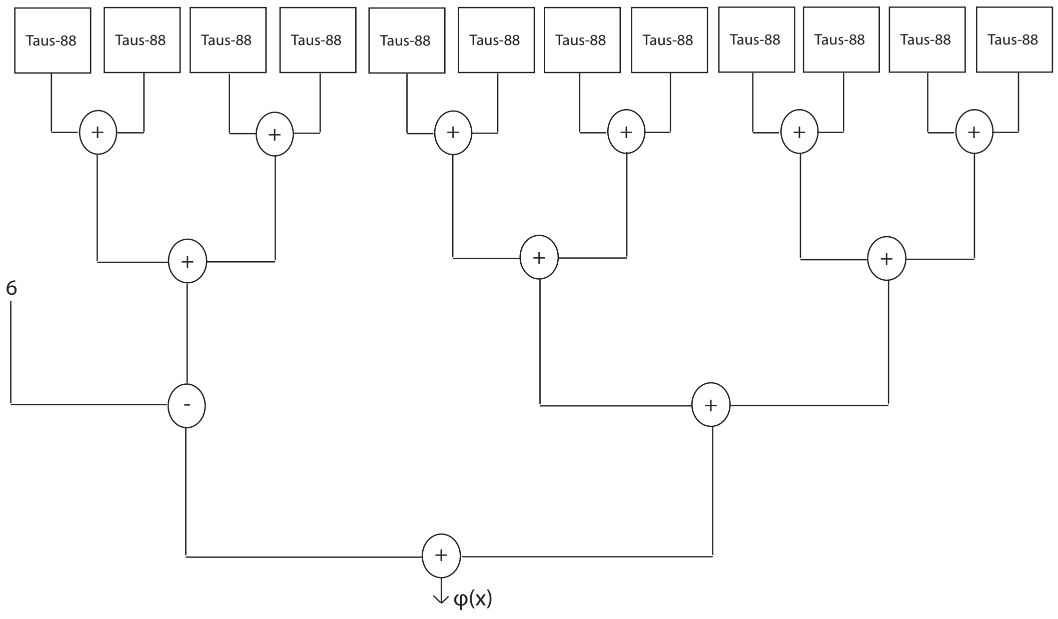

Figure 3.6: Sum of 12 URNG Hardware Architecture

Table 3.2.

3.2.2 Transformation Method

In the most general form the transformation method takes in a vector of URN $\boldsymbol{U_i}$ and produces a vector of GRNs $\boldsymbol{X_i}$ using a transformation $\boldsymbol{F}$.

$$\boldsymbol{X_i} = \boldsymbol{F}(\boldsymbol{U_i}), \tag{3.9}$$

One of the most commonly used transformations is the Box-Muller transform [2]. It takes two URNs and in turn generates two GRNs. The method is based on two properties of a normal distribution. Given two "standard" normal variates X_1 and X_2:

1. $R = X_1^2 + X_2^2$ is an exponential distribution with a mean 2 and has a CDF:

$$P[R < r] = 1 - \exp\left(-\frac{r}{2}\right), \tag{3.10}$$

2. (X_1, X_2) is uniformly distributed in a circle of radius $\sqrt{R}$.

Hence, to generate (X_1, X_2), first $\sqrt{R}$ is generated from a URN U_1, taking the square root of the inverse of Eqn. 3.10:

$$\sqrt{R} = \sqrt{-2\log(U_1)}, \tag{3.11}$$

Then a random angle is selected on a circle with a radius $\sqrt{R}$ using a second URN U_2 as $2\pi U_2$:

$$(X_1, X_2) = (\sqrt{R}\cos(2\pi U_2), \sqrt{R}\sin(2\pi U_2)), \tag{3.12}$$

Lee *et al.* [31] implemented a hardware Box-Muller generator. In their design the complex functions in Eqns. 3.11 and 3.12 are evaluated using linear interpolation with lookup tables that exploit the behavior of the functions over specific ranges. The authors divide the intervals into uniform and non-uniform segments, smaller segments for non-linear portions and larger segments for linear sections. In addition, to avoid using a large number of multipliers, multiplication is performed with 18-bits, to fit the 18X18 multiplier, and the numbers are later scaled to appropriate magnitudes using multiples of 2, hence sacrificing some of the precision. In the final stages the authors added an accumulator that sums two successive outputs to reduce the effect of the approximation error from the lookup tables. However, since the Box-Muller algorithm produces two Gaussian variates every iteration, the overall design still produces one Gaussian variable every cycle. The overall resource utilization on a Virtex 2 XC2V4000-6 is summarized in Table 3.2. The design's final output is a 32-bit fixed-point value with a maximal achievable output of 6.7σ.

3.2.3 Acceptance-Rejection Method

In a standard acceptance-rejection method, samples from an arbitrary distribution are used to approximate the desired distribution. The general description is as follows. Let $f(x)$ be the desired distribution and $g(x)$ an arbitrary distribution, that's usually easier to generate, such that

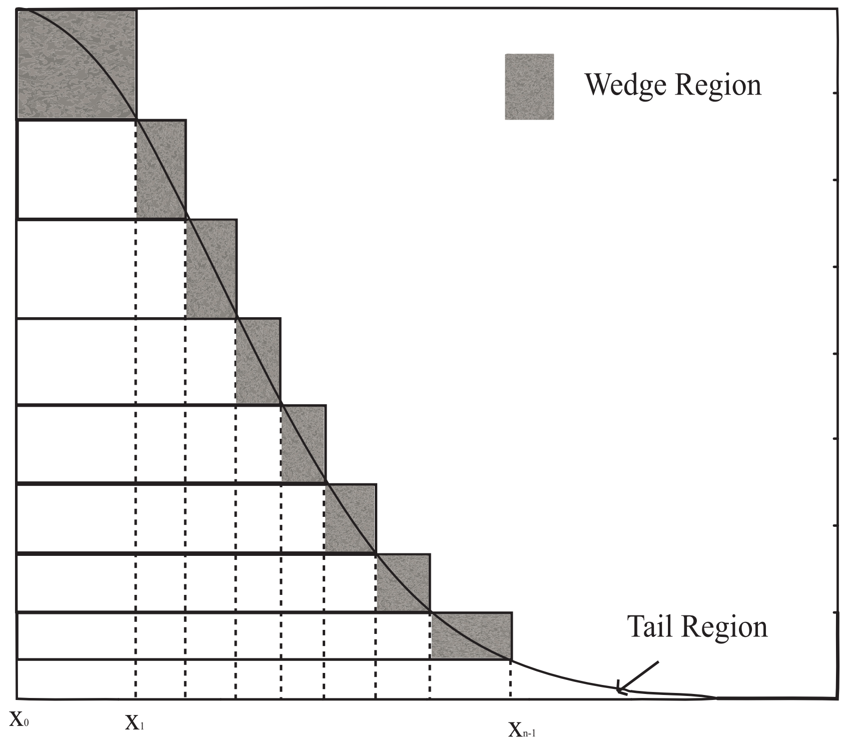

Figure 3.7: Gaussian distribution divided into rectangular, wedge, and tail regions in the Ziggurat method

$f(x) < Cg(x)$ for some constant, $C > 0$. Select a sample x_i from $g(x)$ and a u_i from URN. If $u_i < \dfrac{f(x_i)}{Cg(x_i)}$ the sample x_i is accepted, else it's rejected and the sampling process is repeated. One of the most widely used acceptance-rejection methods is the Ziggurat algorithm [32], which is by some considered the fastest software GRNG.

In the Ziggurat algorithm $Cg(x)$ is selected as n-1 rectangular regions plus a tail region, as shown in Fig. 3.7. The regions are selected in such a way that they have an equivalent area. Each of the rectangular regions could be divided into two parts a sub-rectangular region bounded by x_{i-1}, which is completely underneath a Gaussian PDF, and a wedge region that contains points both above and below a Gaussian PDF. The more regions, the finer is each region and the closer $Cg(x)$ is to a real Gaussian PDF, resulting in less rejected points. However, increasing the number of regions also increases the memory requirements of the application.

The overall algorithm is as follows:

1. Select a random region $1 \leq i \leq n$.

```
2. Randomly select an $x$ within the chosen region $x = U_1 x_i$

3. If $x < x_{i-1}$ return x   /*rectangular region: x is within the sub-rectangular
                                 region bounded by $x_i + 1$, completely underneath the
                                 Gaussian PDF*/

4. If $i = n$ /* tail region */

    (a) Generate $U_2$, $U_3$ URNs
    (b) Compute $x = \frac{-\ln(U_2)}{x_{n-1}}$ and $y = -\ln(U_3)$
    (c) If $2y > x^2$ return $x + x_{n-1}$, Else repeat 4 (a)

5. Compute $y = f(x_i) + U_4(f(x_{i+1}) - f(x_i))$   /*Wedge region: $f(x_i)$ and $f(x_{i+1})$
                                                     are precomputed values*/

6. Compute $f(x)$
   if $y < f(x)$ return x, Else go back to step 1   /* the selected (x,y) point lies above
                                                     the Gaussian PDF and a new point
                                                     has to be selected*/
```

Zhang *et al.* [29] proposed a hardware implementation of the Ziggurat method presented above. In their design they select n = 256, which gave a 0.8% probability of a point being either in the wedge or the tail regions, and 0.7% of a point being rejected. Hence, they highly parallelize steps 1, 2 and 3, which occur 98.5% of the time. The complex $\phi(x)$ and $\ln(U)$ evaluations, steps 4 and 6, are implemented using polynomial approximations, which take a few iterations to compute. However, due to the infrequency of tail or wedge calculations the authors did not see a significant performance impact for having the reiterative polynomial evaluation, 50 stall cycles out of 10^9 execution cycles. At intermediate stages of the design, 18-bit values are used for efficient multiplication, using the 18x18 MULTs units. The final output of the core is a 35-bit fixed-point number. The overall resource utilization on a Virtex 2 XC2V4000-6 is summarized in Table 3.2.

3.2.4 Inverse Transformation Method

The inverse transformation method is a special subclass of the transformation method, described in Section 3.2.2, where the transformation is the inverse Gaussian CDF, $\boldsymbol{F}(\boldsymbol{U}) = \Phi^{-1}(U)$. However, it is mentioned in its own class due to its popularity [30]. Unfortunately, there is no closed-form inversion of $\Phi(x)$, hence $\Phi^{-1}(U)$ is often approximated as a polynomial of varying degree [30]. In such algorithms there is always a tradeoff between accuracy and computation time. Higher degree polynomials provide better accuracy but increase execution time.

There have been a few hardware implementations of the inverse Gaussian method. McCollum *et al.* [33] proposed a linear approximation by dividing $\Phi^{-1}(U)$ into uniform segments, using a total of 2^{16} segments. The total memory usage was 262KBytes. A more recent implementation has been proposed by Lee *et al* [28], which use hierarchical lookup tables along with varying segment sizes. Using their hierarchical scheme the authors implemented a 2-degree spline, polynomial, approximation. Their implementation occupies the least resources out of all other hardware GRNG proposed in this section, shown in Table 3.2. However, the authors only use 16-bit fixed-point notation (with 11-bits for the fraction component) to represent the Gaussian variates. This limits the precision of the results, with error starting to appear as early as the fourth decimal place. To improve precision, increasing the number of bits would correspond to a proportional increase in the resource utilization. The authors present evidence that for 16-bits, a 2-degree spline offers sufficient accuracy. However, if the number of bits were to be increased a higher degree polynomial would have been needed.

3.2.5 Wallace

The Wallace random number generator [34] relies on a basic property of normal variates: linear combinations of normal variates are normal. The algorithm requires an initial pool of $N = KL$ independent normal values, normalized such that the sum-of-squares is N. A pass in the algorithm is defined as a set of transformations that map all the values in the old pool to a new pool. In one pass L vectors of K numbers each, denoted by $\boldsymbol{x}$, are chosen one at a time and transformed to KL new values through Eqn. 3.13.

$$\acute{\boldsymbol{x}} = A\boldsymbol{x}, \tag{3.13}$$

Where A is a k-by-k orthonormal matrix. Each vector of K numbers is taken from the pool using a newly generated address:

$$(StartAddress + i \times Stride) \mod L, \tag{3.14}$$

for $i = 0$ to $L - 1$. To reduce correlation between output variates, the pool values have to be well mixed. To achieve the mixing each pass starts from a different random address and uses a different random stride. For further mixing, one of four orthonormal matrices are chosen at each pass; for the case of $K = 4$, the four matrices presented in Wallace's paper [34] are shown below:

$$\mathrm{A}_1 = \frac{1}{2}\begin{bmatrix} 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \quad \mathrm{A}_2 = \frac{1}{2}\begin{bmatrix} 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

$$\mathrm{A}_3 = \frac{1}{2}\begin{bmatrix} 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix} \quad \mathrm{A}_4 = \frac{1}{2}\begin{bmatrix} -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix}$$

During initialization the pool is filed with GRN normalized to have a sum-of-squares of N. Subsequent passes do not alter the sum of squares due the matrix A being orthogonal $AA^T = I$, and:

$$\sum_{i=1}^{L} \acute{x}^T \acute{x} = \sum_{i=1}^{L} (Ax)^T (Ax) = \sum_{i=1}^{L} x^T (A^T A) x = \sum_{i=1}^{L} x^T I x = \sum_{i=1}^{L} x^T x, \tag{3.15}$$

This can be viewed as a major defect if every block of N values returned to the user had the same sum-of-squares. To alleviate this problem during each pass one of the values from the pool is used to approximate a variate ν from the chi-squared distribution ξ_N^2. Subsequent values are multiplied by $\sqrt{\frac{\nu}{N}}$ to achieve appropriate sum-of-squares behavior. The chi-squared distribution is approximated using Eqns. 3.16:

$$\begin{aligned} \nu &= \frac{1}{2}(C + AX)^2 \\ A &= 1 + \frac{1}{8N} \\ C &= \sqrt{2N - A^2} \end{aligned} \tag{3.16}$$

Where X is a Gaussian variate.

This algorithm was originally presented as one of the fastest software methods: it requires no transcendental function evaluation and uses only 1 URN to generate N GRN. Lee *et al.* [24] demonstrated that this algorithm can also efficiently be mapped to hardware. They have implemented the Wallace algorithm for a pool size of 1024 variates and a vector size of 4, $K = 4$. Throughout their design they found that using 24-bit fixed-point was sufficiently accurate for their application. Their design resource utilization is summarized in Table 3.2.

3.2.6 Gaussian Number Generator Summary

This section provides a summary for the hardware generators presented in Sections 3.2.1 to 3.2.5. Table 3.2 summarizes the hardware utilization of each design on a Virtex 2 XC2V4000-6. All of the designs except for the sum of 12 URNs pass significant statistical tests. The details regarding the tests are explained in the papers in which the designs are introduced. The table includes a row that shows the number of bits the authors used for their final output, which is a way to measure precision in each design. However, many of the designs use a different number of bits for internal function evaluations. Hence taking the number of output bits cannot be used as an absolute measure of precision but just as a representative number.

Table 3.2: GRNGs Resource Summary, All Designs are Mapped to XC2V4000-6

	CLT Sum-of-12	Box-Muller	Ziggurat	Inverse Gaussian	Wallace
Slices	1016	2514	891	584	770
BRAMs	0	2	4	1	6
MULT 18x18	0	8	2	4	4
Frequency (MHz)	247	133	168	231	155
Pass Statistical Tests	No	Yes	Yes	Yes	Yes
Number of Output Bits	32	32	35	16	24

The generators were accessed based on the criteria presented at the beginning of this chapter.

- The CLT: Sum-of-12 generator does not pass statistical test.
- The Box-Muller implementation requires over two times the slices of any other method.
- The Ziggurat method is an acceptance-rejection algorithm and hence cannot deterministically provide a random variate each cycle.
- Both the Wallace and the Inverse Gaussian implementations take up the least slices and meet all other criteria. However, we were not aware of the Lee's [28] inverse Gaussian implementation at the time the generator was selected hence the Wallace design was chosen. Also, increasing the number of bits, to increase precision, for the inverse Gaussian design might have resulted in larger design than the Wallace implementation.

3.3 Wallace Hardware Generator

In Section 3.2.5, the Wallace algorithm was presented alongside with it's hardware implementation proposed by Lee *et al.* [24]. In using that design for this thesis a few modifications were made to the overall architecture and the algorithm itself to try to make the best use of hardware resources. A few of the modifications suggested by R$\ddot{u}$b [35] that improve the quality of the generated numbers were also applied. The following major modifications were implemented:

1. A new URN was generated for every transformation as opposed to every pass. The software method attempts to decrease execution time by generating only 1 URN every pass, i.e. 1 URN for N, 1024, normal variates. However, in hardware there is no additional cost to generating a URN more often considering the circuitry is already there. Using a completely new address and transformation matrix for every transformation helps increase the randomness and the quality of the output variates [35].

2. Lee's design was extended to incorporate four transformation matrices, from the original two, with accordance to Wallace's original paper [34], which further improves the randomness of the generator.

3. For convenience the number of bits in the design was extended to 32 from 24. This made the software and the hardware generators have the same output, which allowed a direct comparison between the software and the hardware CDO simulations. The 32-bit design uses fixed-point Q5.27 representation, 5 bits for integer component and 27 bits for the fraction. Using 2's complement arithmetic the range that could be represented is approximately from [-16, 16]. The choice of the representation was made to allow representing the extreme rare, yet very important (due to their effect on the outcome, i.e. stock market crash) for a financial simulation, cases of variates below -8 or above 8.

4. The sum-of-squares correction mechanism is no longer required due to the change in point 1. Previously, in a given pass all numbers from the pool were taken in a sequential order (although the initial start address was randomly generated), creating blocks of N variates with sum-of-squares of N. By randomly selecting the address for every transformation, four values used per transformation, no longer creates blocks on N with the

same sum-of-squares. While the sum-of-squares of the whole pool is still N, the output received by the user has a random sum-of-squares value.

Fig 3.8 shows the overall design. Stage 4, the sum-of-squares adjustment stage, can be removed and the output can be directly taken from stage 3, it was kept in the diagram to keep consistency with Lee's design.

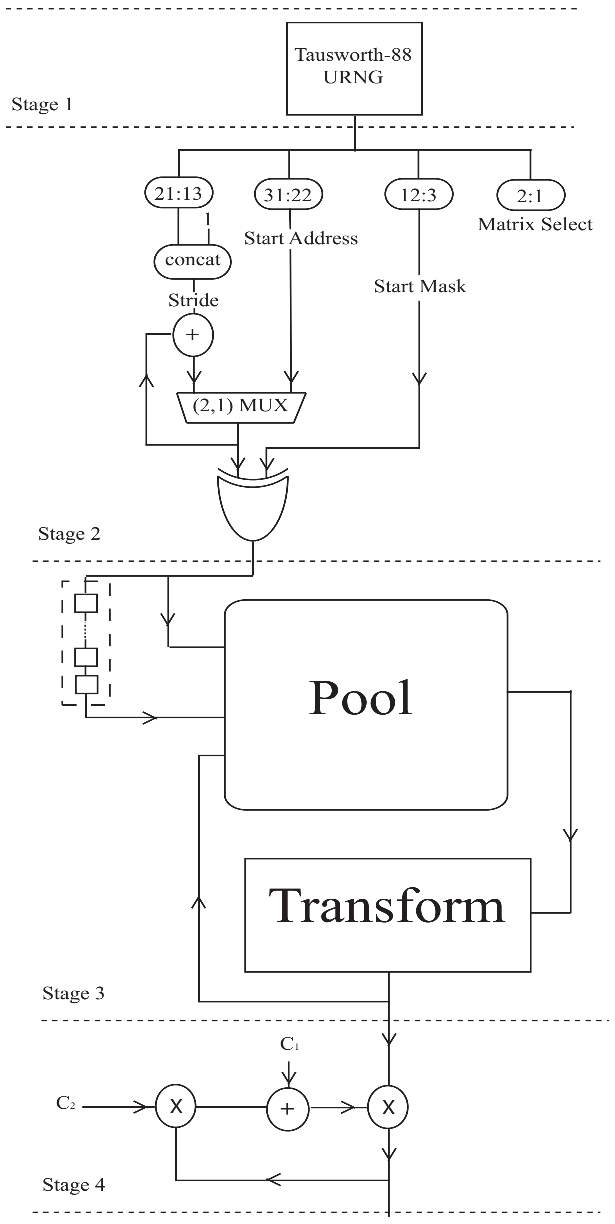

Figure 3.8: Wallace Hardware Architecture

Stage 1 of the design consists of a 32-bit Tausworth-88 URNG that generates a 32-bit random sequence in the range [0, $2^{32} - 1$] and has a period of 2^{88}. For more details refer to Section 3.1.

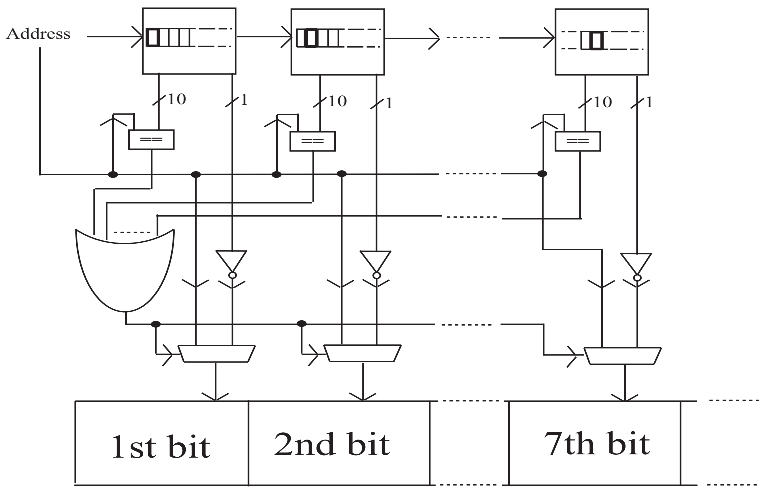

Figure 3.9: Address Correction Mechanism

Stage 2 is a modified version of the address generator presented by Lee. The last 31-bits from the URNG are used: bits 31-22 are used for the starting address, bits 21-13 are concatenated with 1 to generate an odd stride, bits 12-3 are used for a mask (which according to Lee improves the quality of the results), and bits 2-1 are used to select one of the four transformation matrices, only bit 0 remains unused. The address for a vector is given by:

$$Address = (StartAddress + i \times Stride) \oplus StartMask, \tag{3.17}$$

for $i = 0$ to 3. Unlike Lee's design all four addresses are not generated simultaneously but rather in a sequential matter conserving resources by reusing hardware while keeping the dual-ported BRAM fully utilized (one write and one read address is provided every-cycle). Every 4-cycles a new uniform number is generated and the Start Address, Stride and Mask registers are updated. A long hardware pipeline, alongside with a random Start Address, creates a situation in which one can be trying to access an address before it has been updated with a new value. If the old value is used, the pool will no longer consist of independent variates, which will break one of the primary assumption the algorithm is based upon. To avoid this situation all the in-flight addresses are kept in a shift-register (the addresses will also be needed for writing values back into the BRAM). If the newly generated address matches one of the in-flight ones, it has to be changed to an address that doesn't match any of the in-flight addresses. In order to achieve that, in case of a match, the new address is generated as follows: the first bit is the inverse of the first bit of the first in-flight address, the second bit is the inverse of the second bit of the second in-flight address, and so forth (see Fig. 3.9). The final three bits are taken directly from the generated address.

Stage 3 is the transformation stage of the design, refer to Fig 3.10 for a detailed diagram. The transformation implementation is based on the four matrices presented in Section 3.2.5. The four matrices have an interesting quality that any transformed value could be expressed as either one of the old values (denoted p, q, r, and s) minus one-half of their sum ($t = \frac{1}{2}(p + q + r + s)$) or one-half of their sum minus one of the values. This quality allows these four 4-by-4 matrix transformations to be expressed as eight 4-input MUXes and four subtraction units. To keep proper accuracy in calculating t, a 33-bit accumulator was used: such that $\frac{1}{2}(p+q+r+s) = \frac{1}{2}p + \frac{1}{2}q + \frac{1}{2}r + \frac{1}{2}s$.

Stage 4 consists of the sum-of-squares correction module based on Lee's [24] design. The constants C_1 and C_2 are equivalent to $\frac{C}{\sqrt{2N}}$ and $\frac{A}{\sqrt{2N}}$ from Eqn. 3.16, respectively.

The resource utilization of the design is summarized in Table 3.3 for a Virtex 2 XC2V4000. All values within the table are post place-and-route results using Xilinx ISE9.2 [36]. Four different designs are compared to Lee's implementation: 32-bit Wallace with and without sum-of-squares correction and 24-bit Wallace with and without sum-of-squares correction. The 24 bit design and the sum-of-squares module were incorporated for comparison purposes with the original Wallace hardware design, which was 24-bits and had a sum-of-squares correction mechanism. All four designs are smaller than the original Wallace implementation, while

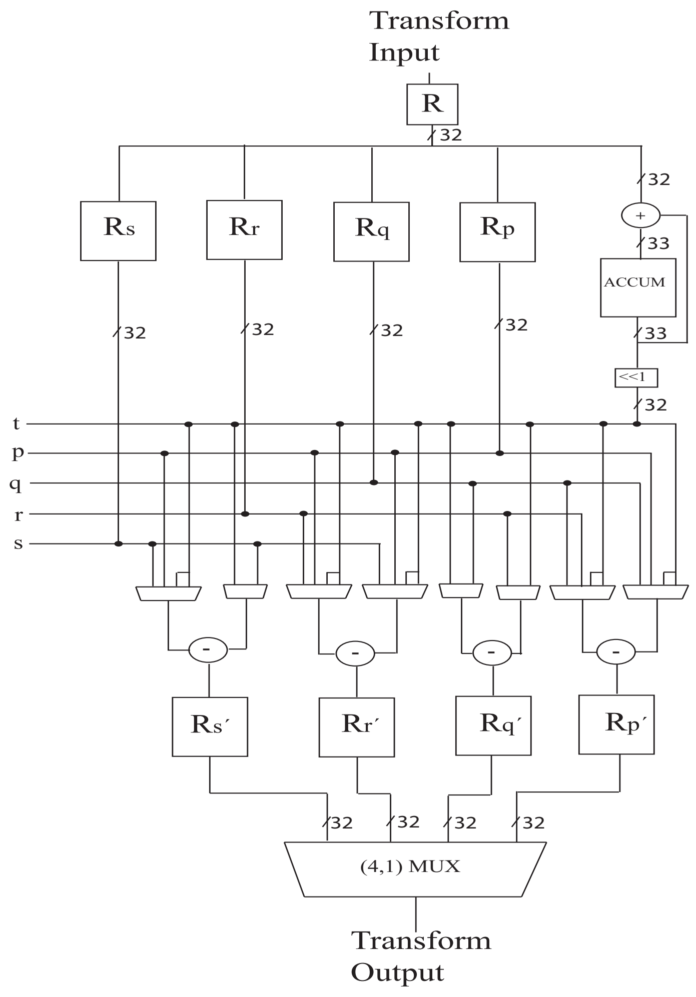

Figure 3.10: Wallace Transformation Circuitry. Stage 3 of the Overall Architecture

the frequency is approximately the same for design with sum-of-square correction module. As can be seen from Table 3.2, removing the sum-of-squares correction significantly reduces resource utilization: for 32-bits 33% of slices are conserved, 3 BRAMs, and 4 MULT18x18; for 24-bits 29% of slices are conserved, 2 BRAMS, and 4 MULT18x18. In general, there is no multiplication in the algorithm outside of the sum-of-square correction, hence by removing that module no multipliers are needed. Frequency has been noticed to be higher in designs without sum-of-square correction, 1.21 times faster for 32-bit and 1.25 times faster for 24-bits.

The 32-bit design was mapped to a Virtex 5 SX50T, which was the target chip for the CDO pricing application. To see the difference in resource utilization on a Virtex 5 both 32-bit Wallace with and without sum-of-squares correction designs were mapped. Unlike in the

Table 3.3: Virtex 2 Wallace Hardware Implementation Resource Summary. XC2V4000-6

	Original Wallace 24-bits	32-bit Wallace with Sum-of-Squares Correction	32-bit Wallace without Sum-of-Squares Correction	24-bit Wallace with Sum-of-Squares Correction	32-bit Wallace without Sum-of-Squares Correction
Slices	770	731	493	571	407
Flip-Flops	-	977	563	786	482
4-Input LUTs	-	890	729	704	600
BRAMs	6	5	2	4	2
MULT 18x18s	4	4	0	4	0
Frequency (MHz)	155	156	188	162	203

Virtex 2 design the stage 4 constant multiplier is implemented using DSP units rather than BRAMs due to smaller resource utilization. On a Virtex 5 a BRAM based Multiplier uses 3 BRAMs (out of 132 available on the SX50T chip [37]), while the DSP48E [38] based multiplier uses 2 DSP48Es (out of 288 available). The design resource utilization is summarized in Table 3.4. Once again the design without sum-of-squares correction module has a smaller utilization. However, the frequency on a Virtex 5 chip for the two designs is approximately the same.

Table 3.4: Virtex 5 Wallace Hardware Implementation Resource Summary. XC5VSX50T-3

	32-bit Wallace with Sum-of-Squares Correction	32-bit Wallace without Sum-of-Squares Correction
Flip-Flops	672	560
6-Input LUTs	501	461
BRAMs	1	1
DSP48E	6	0
Frequency (MHz)	371.5	378.2

To test the quality of the numbers produced by the 32-bit Wallace without sum-of-squares correction design three different statistical tests were performed.

1. Diehard test: Six million normal variates were generated and converted to a uniform distribution using an extension of the Box-Muller formula, introduced in Section 3.2.2. The resultant three million uniform values, the maximum number that can be tested using the Diehard suite, passed all 15 Diehard tests (see Appendix A for results). This indicates that the transformed numbers are uniformly distributed and the original variates are distributed normally.

2. Chi-Squared test: 1 billion numbers were generated and divided into 100 equally spaced bins over the range [-7,7] . The chi-squared test was applied using Eqn. 3.8, resulting in a ξ^2 value of 113 and a p-value of 0.1590, which is above the critical 0.05 p-value (5% value) also confirming that the numbers are indeed samples from a Gaussian distribution.

3. Application test: The final test compared the results obtained running a double precision single-factor CDO pricing software model with normal samples generated using the modified Wallace generator and a Matlab randn() [23] generator, which uses the Ziggurat method. Each generator is run with 100 different seeds. In the randn() generator runs for every seed, the maximal difference between the new results and the results obtained in the previous run was recorded across nine benchmarks (see Section 5.1 for benchmark information). In the Wallace generator runs for every seed, the maximal difference between its results and the corresponding randn() results were recorded, also across the same nine benchmarks. The results for 20 seeds are plotted in Fig. 3.11. The 100 seed plot has the same characteristics but is too messy to display. From the plot one can see that there is no significant difference in using the two generators, the average maximal distance between results for Wallace-to-randn() and randn()-to-randn-() was 1.2% and 1.5%, respectively. Applying the student-t test on the means produces a p-value of 0.6332, way above 0.05, indicating that the means are statistically equivalent.

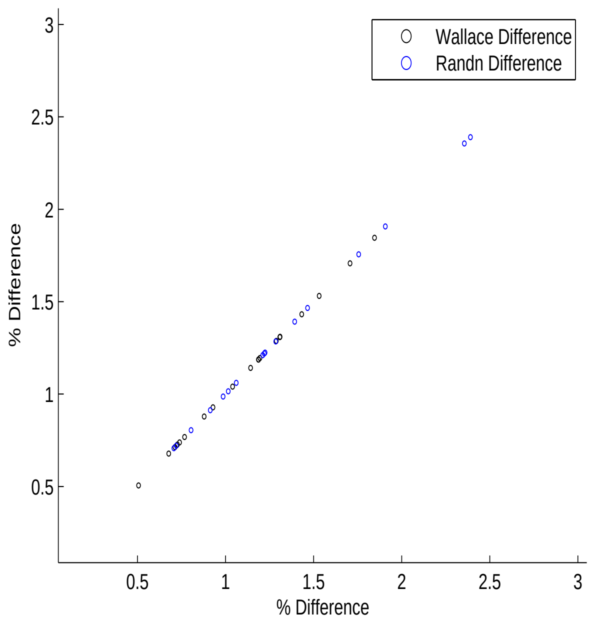

Figure 3.11: Application Result Differences using a Modified Wallace and a Randn() Over 20 Seeds

Considering the 32-bit Wallace without sum-of-squares correction was the smallest design that met the bit width requirement and passed all the statistical tests it was used for the rest of the design as the GRNG.

Chapter 4

Hardware Implementation

Chapter 2 provided an introduction to Collateralized Debt Obligation pricing and presented a mathematical framework for Monte-Carlo based single and multi-factor Gaussian Copula models. Chapter 3 presented an overview of Gaussian random number generators, which are at the heart of every Monte-Carlo simulation, and introduced the criteria for creating a generator for this work. This chapter describes the architecture for the hardware engine that was created based on the Monte-Carlo model presented in Chapter 2. First the design specifications and limitations are discussed in Section 4.1. Section 4.2 presents the overall multi-core simulation architecture. Section 4.3 goes into an in-depth description of a single one-factor Gaussian Copula core. The extension to a multi-factor Gaussian Copula is introduced in Section 4.4.

4.1 Design Specifications and Limitations

While one of the main objectives of this thesis is to keep the design as flexible as possible, certain limitations/assumptions had to be imposed/made to allow efficient hardware mapping. The general design was made to accommodate publicly available published financial data (see Section 5.1). Actual financial data is usually classified. Some design parameters were made to be greater than the datasets required and were based on the extent to which a parameter could have been increased without significantly increasing the resource utilization.

High throughput was the primary design requirement. This requirement made the choice of local Block RAMs (BRAMs) more appropriate than external memory, such as a Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM), considering either 12 or 16 different values were needed per clock cycle per core for one-factor and the multi-factor models, respectively. Similarly, examining the benchmarks, described in Section 5.1, it was found that the largest amount of incoming data that needed to be stored per simulation was 13Kbytes, which can easily be accommodated through BRAMs. However, the use of local BRAMs imposes restrictions on data size. Many of the design limitations can be increased by incorporating more BRAMs into the design with minor alterations to the design itself.

Assumption:

CDO pricing is done in batches. This assumption enables one to ignore data transfer latency as long as computation time is larger or equivalent to transfer time. However, this requires sufficient memory and a mechanism to store incoming data as well as data which is currently being used in computation.

Limitations:

All the restrictions mentioned below could be increased if more BRAMs are added to the design

1. The maximum number of instruments was set at 512.
2. The maximum number of time steps was set at 128. An average simulation contains approximately 20 time steps [17].
3. The maximum number of default curves was set at 32. The largest number of curves seen in the Dow Jones CDX index [39], publicly traded CDO-like instrument, was 5.
4. The maximum number of factors in the multi-factor model was restricted at either 128 or $\frac{2048}{\text{Number of Instruments}}$, which ever was smaller.

4.2 Multi-Core Simulation Architecture

The top-level architecture manages simulation Input/Output, distributing data to individual CDO pricing cores, and final data collection. All data transfer is done through a generic Xilinx Fast Simplex Links (FSLs) [40], which have a standard serial First-In First-Out (FIFO) behavior. In a standard mode of operation, the FSLs would be connected to an Input/Output device, such as PCIe or Ethernet, which in turn connects the FPGA to a host processor running the overall financial simulation suite. The architecture is designed to perform multiple tasks in parallel. The top-level design can be broken into three separate stages as shown in Fig. 4.1: a distributor, independent CDO pricing cores, and a collector module. The three stages are completely independent of each other and are designed to allow the pricing cores to be kept active at all times.

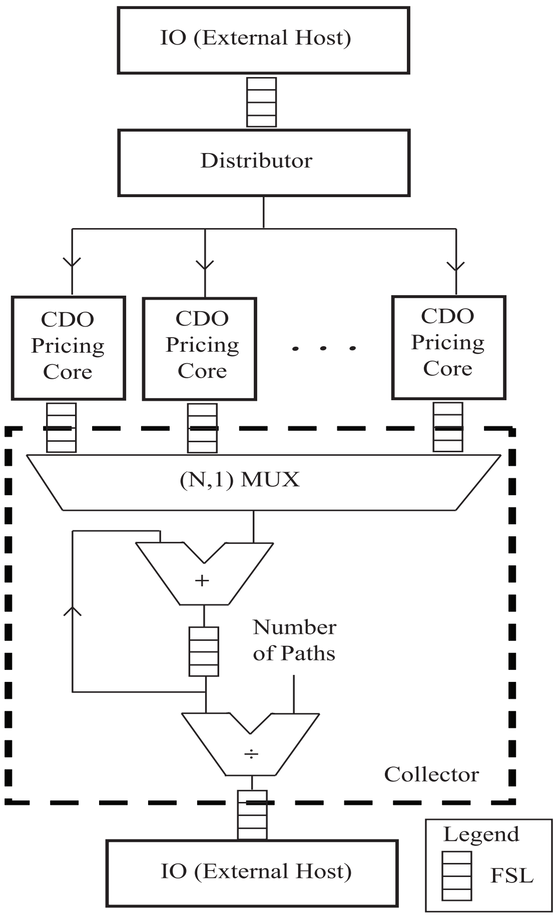

Figure 4.1: Multi-Core Simulation Architecture.

The distributor module, implemented as a Finite State Machine (FSM), is directly connected to the input FSL. The incoming data is distributed among the CDO pricing cores using double buffered dual ported BRAMs, which allows hiding the latency of loading data onto the FPGA. The memory is split into two halves, such that one is used for writing and the other for reading. New incoming data is written to one-half, and the other half is used to read current simulation data. Once both reading and writing are complete the halves are flipped and new data is written to the half that was previously used for reading, and simulation data is read from the half that was previously written.

Top-level parallelization is performed over the MC paths, since all MC paths are independent of each other [2]. The paths are equally divided amongst the CDO pricing cores. Similarly, path independency makes it easier to distribute input data. All CDO pricing cores are loaded simultaneously with the same data. The difference in the outputs stems from the different random numbers generated. The number of pricing cores that can be instantiated depends only on chip resources, the overall architecture allows for an arbitrary number of cores with the design resource utilization being linearly proportional to the number instantiated.

The collector module, shown in Fig. 4.1, is the final stage of the design. It accumulates all the results from the individual CDO pricing cores, divides by the total number of MC paths, and sends the results to the host processor. The collector is decoupled from the individual pricing cores through FSLs. This allows the cores to start a new simulation, while the collector is still processing data.

4.3 One-Factor Gaussian Copula Hardware Implementation

In this section a fully pipelined design of the One Factor Gaussian Copula model is presented, which parallelizes the algorithm presented in Section 2.2.1 over the number of time steps, as shown in Fig. 4.2.

In Phase 1, two Wallace GRNG are used to generate X and Z_i. Their 32-bit fixed-point out-

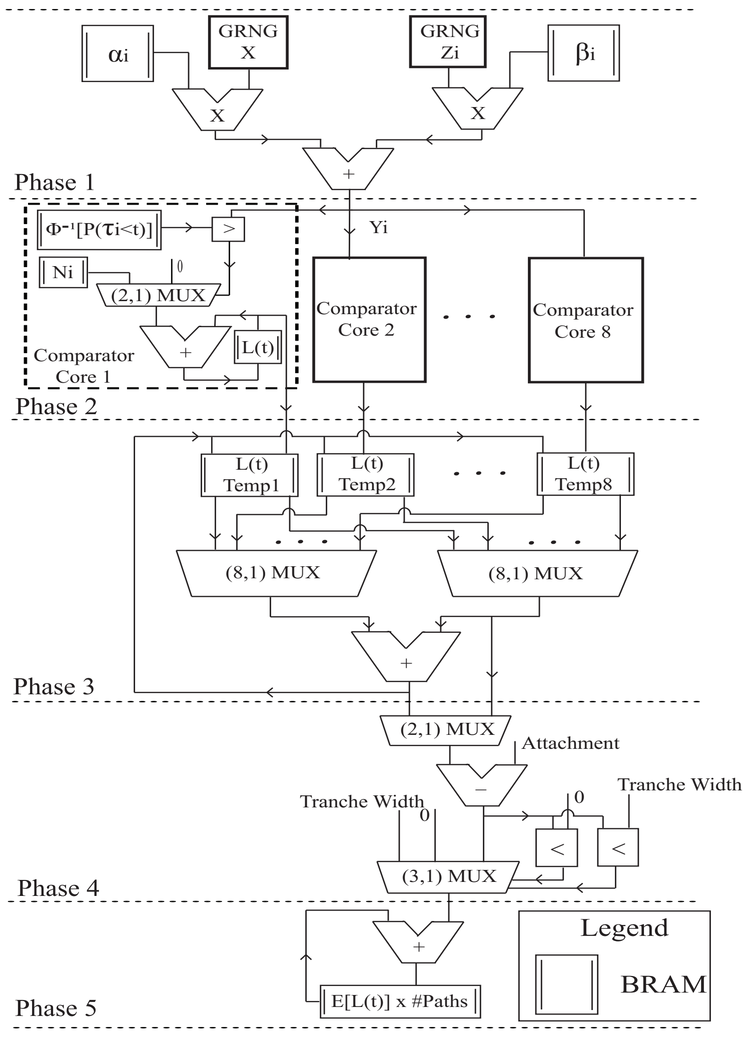

Figure 4.2: One-Factor Gaussian Copula Hardware Core

put is converted to IEEE single-precision format (8-bit exponent and 23 bit numbers mantissa) in order to use the same format as the α_i, β_i, and default curve values from the host processor. From there Y_i is created based on Eqn. 2.7.

Phase 2 consists of eight replicas of a comparator core that implement Eqns. (2.9), (2.13) and (2.14). Each replica performs the $Y_i < \Phi^{-1}[P(\tau_i < t_l)]$ comparison for a subset of t_l's, assigned in a sequential $\mod$ eight manner (i.e. Comparator 1 performs the comparisons for times steps 1, 9, 17, etc.; while Comparator 2 performs the comparison for time steps 2, 10, 18, etc.), where $\Phi[P(\tau_i < t)]$ is the default curve obtained from the host processor. These comparisons themselves are independent and hence can be performed in parallel. The decision to select eight replicas is based on convenience and resource conservation. Let us define a Replication Utilization Factor, RUF:

$$\text{RUF} = T \mod (\text{\# of replicas}), \tag{4.1}$$

Where T is the total number of time steps in a simulation. More replicas potentially provide a greater speedup when RUF is approximately equal to the number of replicas; however, the overall design grows large and many of the comparator units become underutilized when RUF is about 0, which can be seen in Chapter 6. Eight is chosen as a convenient power of 2, which makes partitioning as well as arithmetic operations in the control path more efficient, and provides a good speedup and low utilization cost for a T that is normally distributed with a mean of 20, the theoretically ideal value [17]. While, increasing to the next power of 2, 16, would speedup the larger benchmarks (benchmarks with more time steps and number of instruments), in addition to increasing resource utilization it would actually slow down the smaller benchmarks by increasing accumulation overhead in Phase 3.

In Phase 2, BRAMs are used to store multiple partial sums of each $L(t_l)$. This is done to avoid stalling the pipeline. The adders at Phase 2 have a pipeline latency that often creates a situation where the value of $L(t_l)$ is needed at the input to the adder while it is still being computed. Hence, one of the partial sums is used instead. The greater the adder latency the more partial sums are in-flight. The downside of this approach is that at the end of each MC path these partial sums have to be combined to form the total number of collateral pool losses at a given time step.

In Phase 3, the partial sums are combined. Once all partial sums are available in the comparator cores their values are transferred to the temporary memory storage. This allows all the previous phases to start calculating a new MC path. If the number of assets in the pool or the number of time steps is sufficiently large, combining partial sums and creating new ones can be done in parallel. However, if the new partial sums are ready before the previous ones have been combined, the pipeline has to stall. This was only seen once with the smallest benchmark, Benchmark 5, which contains a pool of only 14 assets and is simulated for only six time steps requiring eight partial sums for each time step, which is the maximum possible value in the design.

Phase 4 is the hardware representation of Eqn. 2.15. It takes the total pool losses for a given time step and calculates the losses within the currently simulated tranche.

Phase 5 is the final accumulator, which combines the tranche losses over all MC paths assigned to the particular OFGC engine.

4.4 Multi-Factor Gaussian Copula Hardware Extension

The algorithm for MFGC is an extension of the OFGC algorithm with Eqn. 2.7 replaced with Eqn. 2.17. In a similar manner the design for the MFGC core uses the same general architecture as shown in Fig. 4.2 with an additional factor accumulation module. One of the main differences between the OFGC and the MFGC architecture is that in Phase 1 the OFGC uses single-precision floating point for the random number representation, while MFGC uses Q5.27 fixed-point. The use of fixed-point allows for single cycle addition, which in turn allows quick systemic factor accumulation with an arbitrary number of factors. The factor accumulation module can be seen in Fig. 4.3. It consists of eight separate memory blocks, four hold the correlation factors and the other four hold the systemic factors for the current MC path. The architecture allows four factors to be processed in parallel; a decision which was made based on a typical MFGC simulation that usually has three to four factors [12]. To minimize stalls, the GRNG generates systemic factors for the next path while the old ones are still being used in the simulation. The new systemic factors are written to the BRAM using the second port, which in this application is only used for writing.

Previously, in the OFGC design a new Y_i was available as input to the Phase 2 of the design every-cycle, which was no longer possible in the MFGC design where it might take a few

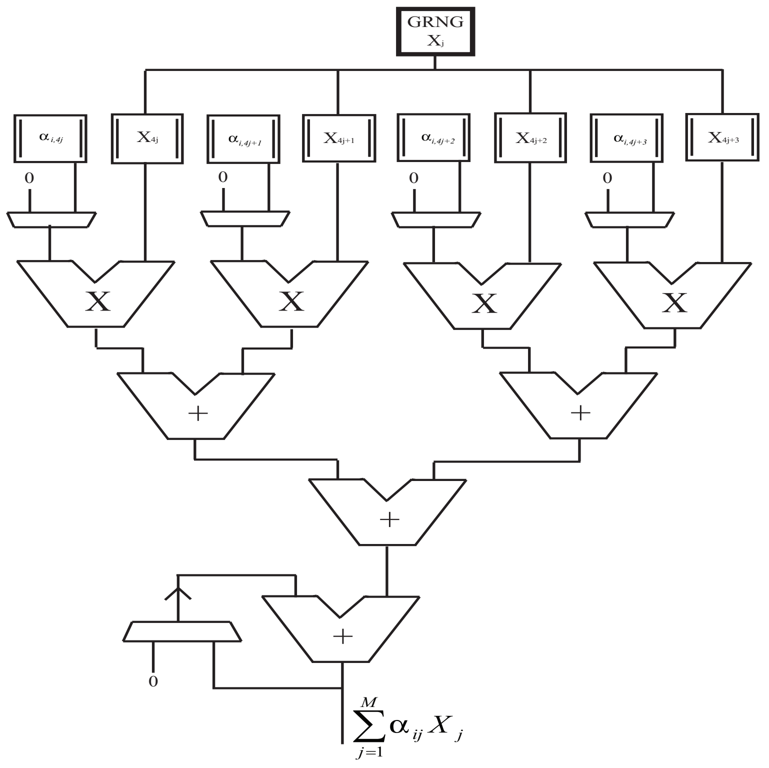

Figure 4.3: Factor Accumulation Module

cycles to produce a Y_i. This can potentially cause a delay. However, for a given simulation a new Y_i is only needed every $\left\lceil \frac{\text{\# of Time Steps}}{8} \right\rceil$ cycles. While in the MFGC design a new Y_i is generated every $\left\lceil \frac{\text{\# of Systemic Factors}}{4} \right\rceil$ cycles. Hence, in certain cases there is no delay due to the multi-factor accumulation, otherwise Phase 2 is forced to stall for $\left\lceil \frac{\text{\# of Time Steps}}{8} \right\rceil - \left\lceil \frac{\text{\# of Systemic Factors}}{4} \right\rceil$ cycles.

Chapter 5

Test Methodology

The previous chapter presented the hardware architecture for a Monte-Carlo Credit Collateral Debt Obligation pricing model. This chapter describes how hardware testing and verifications were performed. Details regarding benchmark creation and the methodology for obtaining valid financial data are presented in Section 5.1. Section 5.2 describes the overall test methodology including the hardware testbench.

5.1 Benchmarks

Unfortunately, there are no widely accepted benchmarks for CDO pricing and all financial transactions are confidential. Hence, for the purpose of this thesis, nine different benchmarks were developed, which attempt to mimic a realistic dataset. The data for these benchmarks were taken from previously published CDO papers [15, 17], Dow Jones CDX indices [39] and publicly available Moody's rating information [41].

The nine test benchmarks are shown in Table 5.1. The first eight are based on Dow Jones CDX indices, which are commercially traded CDO-like instruments that are based on collateral pools consisting of companies and government organizations in North America and emerging markets. Benchmarks 1 through 8 are created using the same number of assets and the credit rating as the original CDX indices, as of March 24th 2008. As can be seen from the Credit Risk

Table 5.1: Test Benchmarks.

Benchmark #	Based on Data from	# of Assets	# of Time Steps	Credit Risk Distribution of Instruments	# of Default Curves
1	CDX.NA.HY	100	15	4-BBB 42-BB 36-B 17-CCC 1-D	5
2	CDX.NA.IG	125	35	3-AAA 3-AA 52-A 63-BBB 4-BB	5
3	CDX.NA.IG. HVOL	30	19	4-A 21-BBB 4-BB 1-CCC	4
4	CDX.NA.XO	35	22	4-BBB 29-BB 1-B 1-CCC	4
5	CDX.EM	14	6	1-A 3-BBB 9-BB 1-B	4
6	CDX. DIVIRSIFIED	40	23	1-AA 5-A 17-BBB 15-BB 2-B	5
7	CDX.NA.HY.BB	37	13	2-BBB 30-BB 3-B 2-CCC	4
8	CDX.NA.HY.B	46	26	11-BB 30-B 4-CCC 1-D	4
9	Jackson *et al.* [17] Semi-Homogenous	400	24	200-BBB+ 200-BBB-	2

Distribution of Instruments column, which uses Standard and Poor's [42] credit rating system, each benchmark has a widely different composition. Benchmark 2 consists of the safest instruments (least likely to default), while benchmarks 1 and 8 have the riskiest instruments (likeliest to default). For a credit rating explanation refer to Table 5.2. Using the credit rating information, the default boundary curves, $P(\tau_i < t)$, are obtained from Moody's [41]. However, since Moody's uses annual default curves, the values had to be extended using Eqn. (2.5) to attain quarterly time steps. The actual notionals are also obtained from Moody's; the notionals are corporate bond defaults for 1999. There is a wide range for notional values from \$0.6 million to \$6.6 billion.

Table 5.2: Standard and Poor's Rating Explanation

Rating	Grade	Risk
AAA	Investment	Lowest risk, obligator is highly likely to meet commitment
AA	Investment	Low risk, only slightly riskier than AAA
A	Investment	Low risk, however obligator is more susceptible to market changes
BBB	Investment	Medium risk, still investment grade however susceptible to large changes in market
BB	Non-Investment	High risk, faces difficulties in meeting commitment
B	Non-Investment	High risk, still has the potential to meet commitment baring market changes
CCC	Non-Investment	Highest risk, very vulnerable and depends on favorable market conditions to meet commitment
D	Non-Investment	In default

A ninth benchmark is added to represent a very large semi-homogenous collateral pool of 400 assets. The data for it was obtained from [17]. The default curves presented in [17] are between A and BB, and hence were denoted as BBB+ and BBB-. The assets in [17] consists of four groups: 20, 50, 100, and 200 million.

All other input data for every benchmark was randomly generated:

- α_i: uniformly distributed from [0, 1].
- Return rate: normally distributed with a mean of 0.40, ideal return rate [15], and 0.15 variance.
- Number of time steps: Normally distributed with a mean of 20 steps and a variance of 10 steps.
- Each asset in the pool is randomly assigned to one of the default boundary curves.

The tranche attachment points were taken to be the same as in CDX.NA.IG: 0%, 3%, 7%, 10% and 15%.

Multi-Factor Benchmark Extension

The same benchmarks as presented in Table 5.1 were used to test the multi-factor model. For each benchmark 16 sub-benchmarks were generated with the number of factors ranging from 1 to 16 (except for benchmark 9 for which the maximum number of factors was four, due to the current hardware resource constraints). For each new factor sub-benchmark the new α_{ij} is generated from a_i in a recursive manner:

$$\alpha_{ij}^2 = U_j(1 - U_{j-1}) \dots (1 - U_1)\alpha_i^2, \tag{5.1}$$

Where U_1 to U_j are URN.

5.2 Test and Verification Methodology

Software Models

Developing a hardware CDO pricing engine required a software model that could be used to compare/validate results. Two software models were created for this purpose, a Matlab model and a C++ model. The Matlab model was created for debugging purposes while the C++ model for performance measurements.

The Matlab design was created to mimic the real hardware design (as closely as possible) and was used as the "golden" reference. The Matlab model simplified debugging the hardware design, which couldn't have been achieved using a standard hardware simulator, such as ModelSim [43], alone. Numerous arithmetic operations could occur before an error would surface, and to correct such an error, internal simulation values had to be examined and compared to a model that performed the exact same sequence of arithmatic operations and had the correct value. Three different Matlab versions were created to mimic the three different notional representations: floating-point single precision, floating-point double precision, and fixed-point 64 bits (the actual hardware design was 42 bits but Matlab representations are limited to either 32 or 64 bits).

An optimized floating-point double precision C++ model was created for performance measurements. For all comparisons the C++ program was run on an Intel Xeon 3.4 GHz processor with 3GBytes of RAM. To obtained performance and accuracy measurements each benchmark was run 100 times, with different GRNG seeds, for 100,000 MC paths on all hardware designs and software. For accuracy measurements each benchmark error is calculated as the summation of the absolute distance between the design's result and the baseline result divided by the baseline result, and averaged over the number of runs. For the Multi-Factor model the error is also averaged over all the factors. The final error reported is the benchmark errors averaged over the number of benchmarks and the max error is the largest observed benchmark error.

On-Chip Testbenches

All designs were created using the Verilog hardware description language and synthesized, placed and routed using Xilinx 9.2 software suite [36]. All hardware tests were performed on a Xilinx ML506 Evaluation Platform [44], which has a Virtex 5 XC5SX50T -1 speed grade chip. Performance results in chapter 6 assume the same chip with a faster -3 speed grade.

To test the design in hardware two on-chip testbenches were created. The initial benchmark is shown in Fig 5.1. It consists of a MicroBlaze connected to the CDO simulation engine through two FSLs, one for sending test data and the other for receiving simulation results. Test data is initially stored in on-chip memory which is connected to the Micro-Blaze through a Processor Local Bus (PLB). A hardware counter is placed inside the CDO simulation engine that starts counting once all the data is received and stops once the data enters the collector module. The time it takes to load data is ignored based on the assumption that in a real scenario computation and data transfer is done in parallel. This counter value is sent along with the simulation results to the MicroBlaze, which in turn sends all the received values through the UART to a host processor.

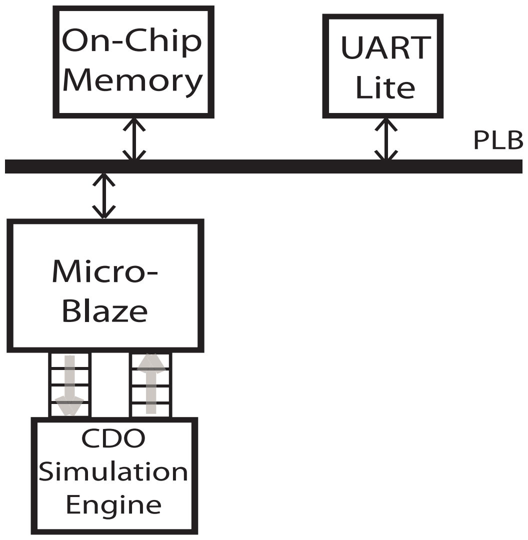

Figure 5.1: On-Chip CDO Pricing Engine Testbench

In the second on-chip testbench the CDO simulation engine was connected in the same manner as in a real world application. The CDO simulation engine was connected through a 1-lane PCIe Endpoint Block Plus [45] to a host processor. The host processor ran a simple C++ program that either kept sending data through the PCIe slot or was waiting for new data on the PCIe slot.

The results obtained from both testbenches were identical and are summarized in Chapter 6.

Chapter 6

Results

This chapter examines the resource utilization and the acceleration obtained for different precision representations for rate adjusted notionals. Section 6.1 examines the precision versus resource utilization tradeoff for the OFGC core and provides an in-depth look into the speedup over the software model. Section 6.2 does the same for the MFGC core. Finally, Section 6.3 presents the minimum requirements needed to keep both designs fully utilized.

6.1 OFGC Core Results

6.1.1 OFGC: Precision Versus Utilization Tradeoffs

The most resource intensive portion of the design is in notional summation, Phases 2 through 5 in Fig. 4.2. In this thesis, three different precision representations for the notionals were explored: floating-point single-precision, double-precision, and integer (fixed-point). The smallest value the integer representation was chosen to represent was one cent. It was assumed that no financial transaction could contain values smaller than one cent and since Li's [14] algorithm just performs additions and subtractions on the notionals amounts, no smaller values could be generated. For the floating-point designs the benefits of using built-in Virtex 5 DSP slices, DSP48E [38], were explored for the pipelined floating-point adders, in Phases 2 and

3. DSP48Es are specialized slices optimized for complex arithmetic and Multiply-Accumulate (MAC) operations.

Post place and route resource utilizations and design frequencies are summarized in Table 6.1, all results are obtained using Xilinx Xplorer [46], which iteratively narrows in on an optimal design frequency. The utilization results are for a single pricing core. The percentages next to each utilization value indicate the portion of the total resource available on the chip that is being used by the design. As seen from Table 6.1 for both single- and double-precision floating-point designs, incorporating DSP48E units reduced the LUT and Flip-Flop utilization. However, the benefits are more evident in the single-precision representation where incorporating DSP units resulted in a larger LUT utilization savings (18.6%), as well as a higher design frequency. Hence, for a chip abundant with DSP slices, such as the Virtex 5 SXT series, it is more economical to use DSPs for floating point operations.

As one might expect, the single-precision floating-point design uses almost half the resources of the double-precision counterpart. However, the single-precision design has an average 0.39% (maximum 1.07%) accuracy error associated with the results. To try to achieve the best of both worlds (i.e. the resource utilization of single-precision and the accuracy of double-precision) single-precision notionals are used in Phases 2, 3 and 4, and a double-precision accumulator is incorporated at Phase 5. This design significantly reduces the average accuracy error to 3.02E-5% (maximum 5.27E-5%) while keeping the resource utilization similar to single-precision.

Further exploring the accuracy to utilization tradeoff, an integer representation model of the notionals was designed. Using the nine benchmarks, data at all phases within the simulation was examined and it was established that it was sufficient to use 42-bits to represent the notionals and 54-bits for the final accumulator to obtain identical results to the double-precision floating-point representation. Furthermore, through ISE [36] it was found that each additional notional bit requires 62 additional Flip-Flops and 74 LUTs, and each additional accumulator bit requires 1 additional Flip-Flop and 2 LUTs (up to 64 bits at which point a new BRAM is

Table 6.1: OFGC Core Resource Utilization

	Single-Precision Floating-Point		Double-Precision Floating-Point		Single-Precision Notionals & Double-Precision Accumulator	Integer
	Without DSP	With DSP	Without DSP	With DSP		
Flip-Flops	7097 (21.7%)	6530 (20.0 %)	10454 (31.2%)	9910 (30.4%)	6721 (20.5%)	4906 (15.0%)
LUTs	8660 (26.5%)	7052 (21.6%)	13548 (41.5%)	13325 (40.8%)	7599 (23.3%)	5224 (16.0%)
BRAMs	15 (11.4%)	15 (11.4%)	31 (23.4%)	31 (23.4%)	15 (11.4%)	15 (11.4%)
DSP48Es	9 (3.1%)	29 (10.1%)	10 (3.4%)	40 (13.9%)	30 (10.4%)	7 (2.4%)
Freq (MHz)	235.2	248.8	187.3	190.9	244.8	268.2
Average Error (%) [Max Error]	0.39 [1.07]	0.39 [1.07]	0	0	3.02E-5 [5.27E-5]	0
# of Cores	4		2		4	5
Replicated Freq (MHz)	208.4		140.8		210.0	218.5

required).

The least resource consuming design from each representation, for floating-point cores it was the DSP based design, was replicated as many times as resources would permit and incorporated into the overall simulation architecture. Since MC paths are completely independent adding additional pricing cores creates a linear speedup. While additional cores increase the overhead of the final collecting phase, the collector latency (which for the used benchmarks was always within 200 cycles) can be hidden by simultaneously starting a new simulation. Using the Virtex 5 SX50T chip, the single precision design could have been replicated four times, the double precision two times, and the integer design five times. The frequency of the replicated design is denoted as Replicated Freq in Table 6.1. For single core and multi-core performance measurements refer to Section 6.1.2.

6.1.2 OFGC: Performance

One design from each numerical representation, for floating-point cores it was the DSP based design, was compared to a C++ software program, running on a Pentium Xeon 3.4 GHz 3 GB RAM. The Single-Precision Notionals Double-Precision Accumulator design is grouped together with the single-precision design, because both occupy almost the same amount of resources, have the same cycle-by-cycle behavior, and operate at almost an identical frequency. Fig. 6.1 summarizes the single-core performance across the nine benchmarks and Fig. 6.2 the speedup achieved by instantiating the maximum number of cores that the resources on the Virtex 5 SX50T chip permit.

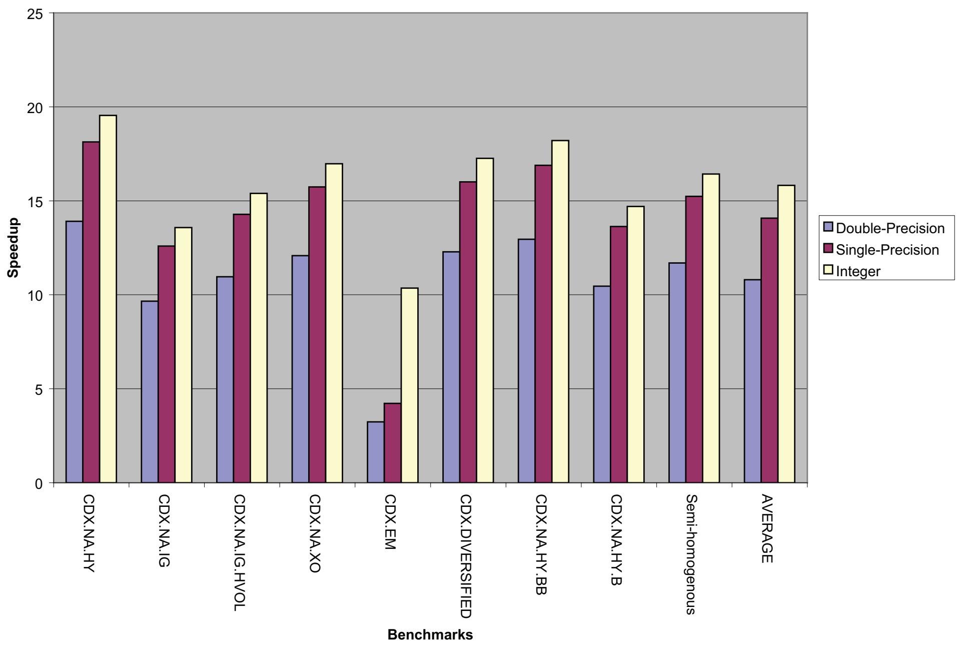

Figure 6.1: Single-Core OFGC Performance

In Fig. 6.1 one can see the acceleration ranges from 19.5X to 3.2X, with the average

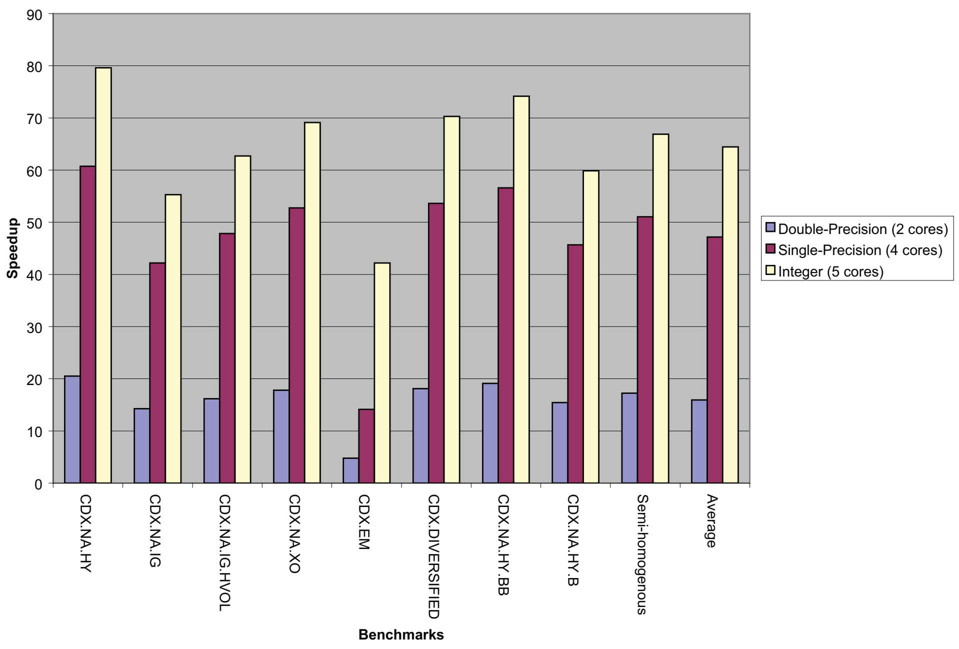

Figure 6.2: Multi-Core OFGC Performance

speedup for double-precision, single-precision and integer designs being 10.8X, 14.0X, and 15.8X, respectively. The difference between the single-precision and double-precision acceleration is mainly due to the frequency difference between the two designs. In addition to the higher frequency the speedup differences between the integer and the rest of the designs is due to a shorter pipeline at Phase 2, Fig. 4.2, and in turn a smaller number of partial sums, which was especially evident for Benchmark 5 (CDX.EM). The CDX.EM is the smallest benchmark, 14 instruments, and causes all of the designs to stall between Phase 2 and 3 as the partial sums, $L(t_l)$'s, are combined, however, the integer design creates the least number of partial sums and hence stalls for the least number of cycles, creating a significantly larger acceleration than the floating-point designs, 10.4X to 4.2X. The difference in acceleration between the benchmarks can mainly be attributed to two factors: number of time steps and number of instruments. Both factors will be explored later on. Fig. 6.2 provides an illustration of the acceleration that was achieved by filling the chip with the OFGC cores. The maximum acceleration achieved was 79.6X. The smallest core, integer, allowed the most replications, five, which results in an average 64.5-fold acceleration. For the other designs, the average acceleration was 47.2X and 15.9X for four single-precision and two double-precision floating-point cores, respectively.

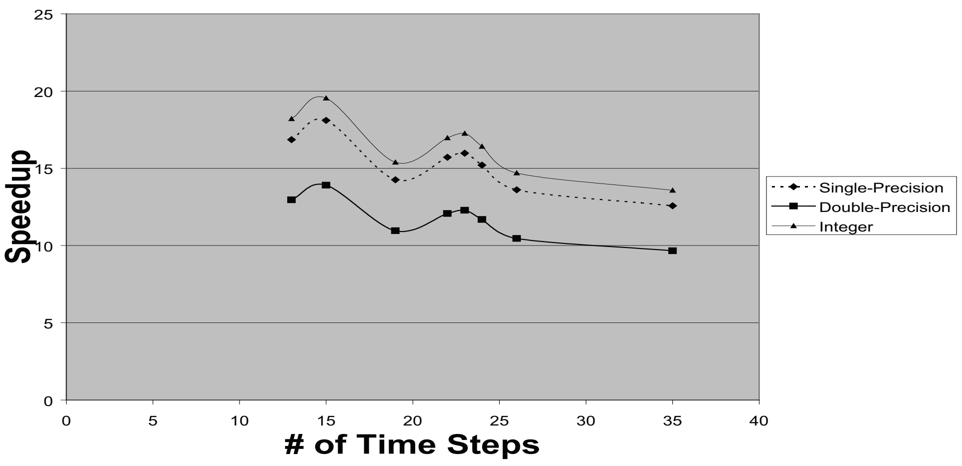

Figure 6.3: Acceleration vs. Number of Time Steps Across Eight Benchmarks

To examine the effects of the number of time steps and the number of instruments on acceleration, two graphs were created, Fig. 6.3 and 6.4. Fig. 6.3 is a plot of a single OFGC core speedup versus number of time steps across eight benchmarks, Benchmark 5 (CDX.EM) was omitted due to it being too small to be representative of what really occurs at six time steps. While the graph is not entirely clear, due to each benchmark consisting of different default curves and hence will behave slightly differently in software, there seems to be a definitive trend. As the number of time steps gets closer to the next multiple of eight, larger RUF factor (Eqn. 4.1), the speedup increases. This trend is expected: the number of time steps dictates the number of comparisons done at Phase 2 of the design, while in software these comparisons are done sequentially, in hardware eight comparisons are done in parallel. Hence, in software it would take more time to price $8n+7$ time steps than $8n+1$, while in hardware it would take the same amount of time. There is also a more global trend in Fig. 6.3: speedup decreases as the number of time steps increase. One plausible explanation for this behavior is that as the number of time steps increase, a smaller portion of the software execution time is dependent on the GRNG and hence less advantage can be taken from having single-cycle hardware GRNG. The dependence of the percentage of execution time dedicated to GRN generation on the number of time steps is depicted in Fig. 6.5. From this figure one can clearly see that as the number of time steps increases the percentage decreases similarly to the speedup curve. Fig. 6.4 is a plot of a single OFGC core speedup versus the number of instruments across the nine benchmarks. No clear pattern could be found in the plot, there seems to be too much interference from the number of time steps and the composition of each benchmark.

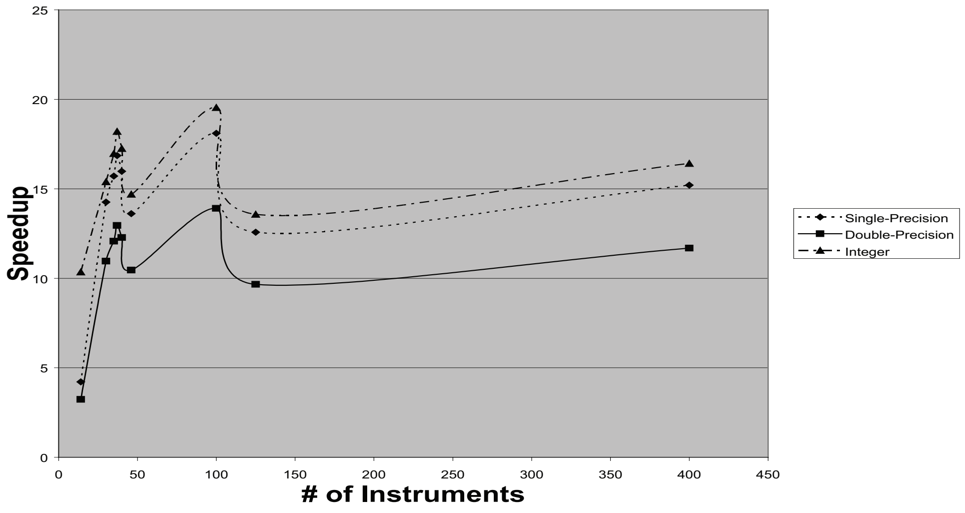

Figure 6.4: Acceleration vs. Number of Instruments Across All Nine Benchmarks

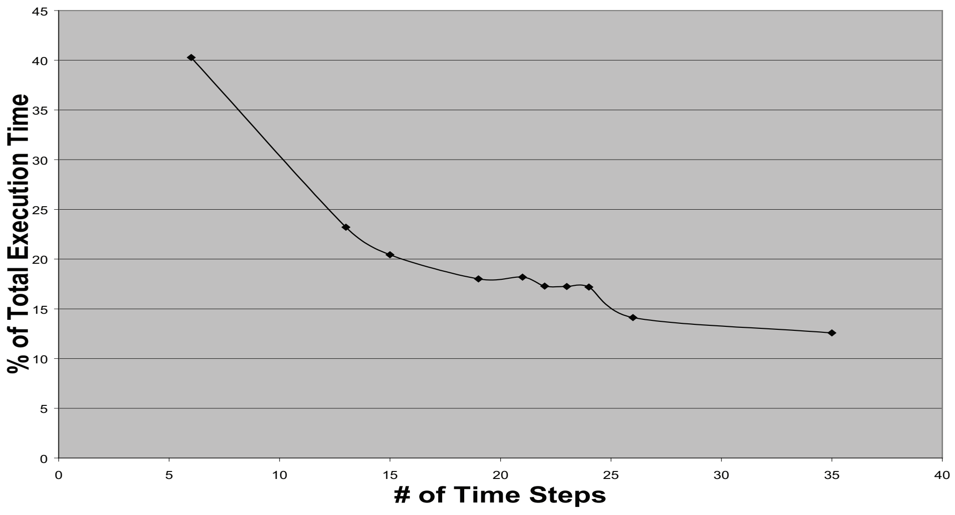

Figure 6.5: % of Execution Time Dedicated to Gaussian Number Generation vs. Number of Simulation Time Steps

To obtain a clearer idea of the influence of the number of time steps or the number of instruments have on speedup one has to hold one of these parameters constant while changing the other. However, changing the number of instruments for benchmarks 1 to 8 would change their default curve composition and hence would introduce another factor. Benchmark 9, in which half of the instruments belong to one default curve and the other half to another, is the only one that can be changed without altering the default curve composition. Hence, benchmark 9 was used to clarify both relationships. In the first study the number of instruments for benchmark 9 was varied from 4 to 512 (512 being the largest allowed due to design restrictions) while the number of time steps was kept constant at 24, refer to Fig. 6.6 for results. One can see that speedup doesn't change as the number of instruments is increased. However, when the number of instruments drops below 20 the floating-point designs start stalling and the performance starts to drop. The same happens to the integer design when the number of instruments is around 10. In a second study, the effect of number of time steps on acceleration was examined: for benchmark 9 the number of instruments was kept constant at 400 and the number of time steps was varied from 2 to 64, see Fig. 6.7. Fig. 6.7 provides a clearer representation of the trends seen in Fig. 6.3:

1. The closer the number of time steps to the next multiple of eight the larger the acceleration.

2. More generally: the more time-steps the smaller the acceleration.

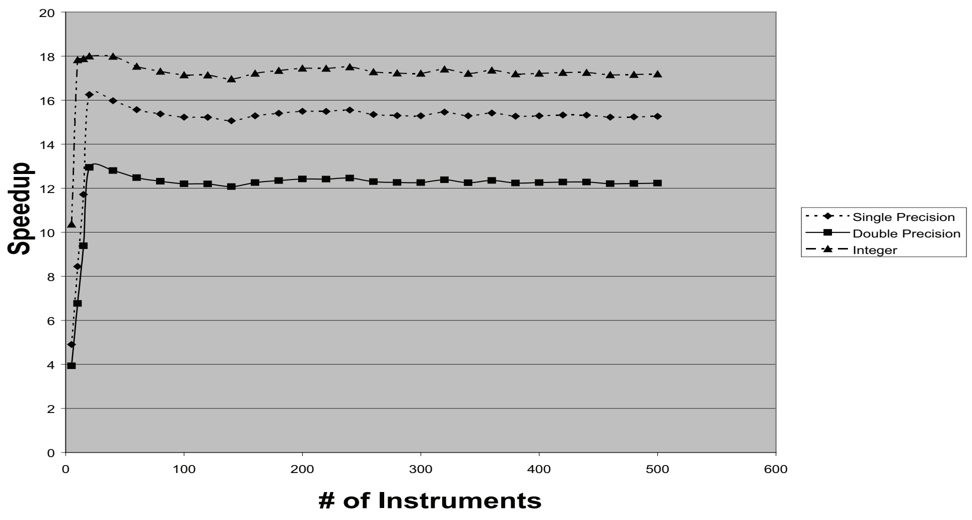

Figure 6.6: Benchmark 9: Acceleration vs. Number of Instruments

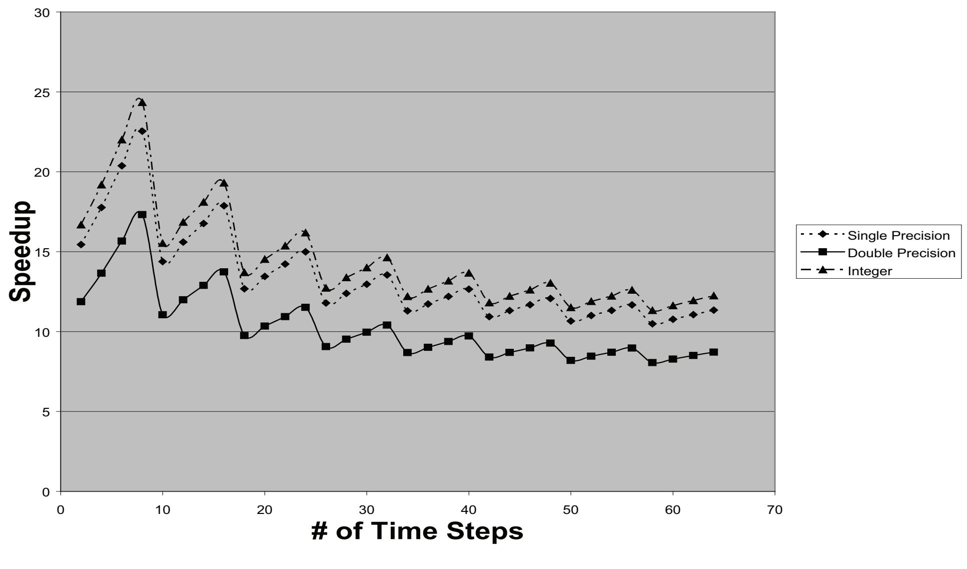

Figure 6.7: Benchmark 9: Acceleration vs. Number of Time Steps

6.2 MFGC Core Results

6.2.1 MFGC Resource Utilization

The post-place and route resource utilization of the factor accumulation module, initially presented in Fig. 4.3, is summarized in Table 6.2. For direct resource comparison with the OFGC core, the same four different numerical precision representations were explored for the MFGC core. For floating-point representations the smaller area DSP based designs were chosen. The results are summarized in Table 6.3, the first percentages next to each utilization value indicating the portion of the total resource available on chip that is being used, the second percentage, in bold, indicates the change over the OFGC design. For the floating-point designs the increase in flip-flops and LUTs over the corresponding OFGC designs was approximately 10% and 5%, respectively. Furthermore, there was a significant increase in BRAMs and DSP slices at 23% and 43%, respectively. For the smaller integer design (it was found that the use of 42-bit notionals and 54-bit accumulator was also sufficient for the multi-factor designs), the flip-flop and LUT increase is almost double the increase seen in floating-point designs, at 19.3% and 10.5%, respectively. In addition, in the integer design the use of DSP units has tripled. The single core frequency changes were very small and can likely be accredited to a slightly different design routing. However, when the number of cores was replicated to fill the chip (four times for single precision, two times for double precisions, and five times for integer) the increase in the overall design size was evident as the overall frequency went down by approximately 7.2%. The accuracy error for each numerical representation remained practically the same as observed in the OFGC designs.

Table 6.2: Factor Accumulation Module Resource Utilization

	Factor Accumulation Module
Flip-Flops	1024
6-Input LUTs	922
BRAMs	7
DSP48Es	16

Table 6.3: MFGC Core Resource Utilization

	Single-Precision Floating-Point	Double-Precision Floating-Point	Single-Precision Notionals & Double-Precision Accumulator	Integer
Flip-Flops	7205 (22.1%) **(+10.3%)**	10811 (33.1%) **(+9.1%)**	7564 (23.1%) **(+12.5%)**	5852 (17.9%) **(+19.3%)**
LUTs	7347 (22.5%) **(+4.2%)**	13779 (42.2%) **(+3.4%)**	8037 (24.6%) **(+5.8%)**	5775 (16.7%) **(+10.5%)**
BRAMs	20 (15.2%) **(+25.0%)**	37 (28.0%) **(+19.4%)**	20 (15.2%) **(+25.0%)**	20 (15.2%) **(+25.0%)**
DSP48Es	43 (14.9%) **(+48.3%)**	54 (18.8%) **(+35.0%)**	44 (15.3%) **(+46.7%)**	21 (7.3%) **(+200.0%)**
Freq (MHz)	246.9 **(-0.8%)**	195.9 **(+2.6%)**	244.5 **(-0.1%)**	262.1 **(-2.2%)**
Average Error (%) [Max Error]	0.38 [1.10]	0	3.19E-5 [4.99E-5]	0
# of Cores	4	2	4	5
Replicated Freq (MHz)	195.1 **(-6.4%)**	138.6 **(-1.6%)**	185.8 **(-11.5%)**	198.1 **(-9.3%)**

6.2.2 MFGC: Performance

For each of the nine benchmarks in Section 5.1, 16 sub-benchmarks were evaluated with the number of factors ranging from 1 to 16 for each, except for benchmark 9 for which due to resource constraints the maximum number of factors was four. Fig. 6.8 is a plot for Benchmark 1 (CDX.NA.HY). It provides a good generic representation of what was seen in all other benchmarks. For plots of other benchmarks refer to Appendix B. For illustrative purposes the plot represents single core acceleration. Due to similar performance, results of the Single-Precision Notionals Double-Precision Accumulator design was once again grouped together with the Single-Precision floating-point design.

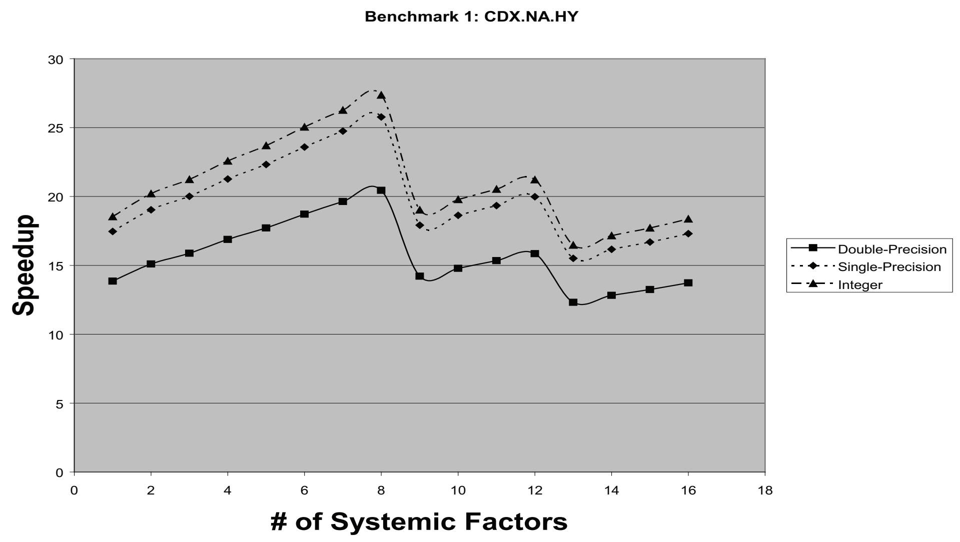

Figure 6.8: MFGC Benchmark 1: CDX.NA.HY Performance

Prior to examining the plot, let's define two factors: Time Replication Factor (TRF) and Systemic Factor Replication Factor (SFRF). Let TRF be:

$$\text{TRF} = \left\lceil \frac{\text{\# of Time Steps}}{8} \right\rceil, \tag{6.1}$$

Where TRF is the number of cycles before Phase 2, refer to Fig. 4.2, requires a new Y_i. Let SFRF be:

$$\text{SFRF} = \left\lceil \frac{\text{\# of Systemic Factors}}{4} \right\rceil, \tag{6.2}$$

Where SFRF is the number of cycles before a new Y_i is generated. Fig. 6.8 and all other figures in Appendix B could be divided into two regions: SFRF smaller or equal to TRF, and SFRF larger than TRF. For the region in which SFRF is smaller or equal to TRF the speedup increases linearly with an increase in the number of systemic factors. In this region increasing the number of systemic factors increases the amount of calculations a software program has to perform hence increasing runtime. While in hardware Y_i generation is done in parallel to Y_i consumption hence while SFRF $<=$ TRF a new Y_i is generated before it's needed resulting in a constant runtime with respect to an increase in the number of systemic factors. In the region in which SFRF is larger than TRF, the curve exhibits the same behavior seen in Section 6.1.2 for speedup versus number of time steps curves. In this region the speedup follows a step-like curve, the closer the number of systemic factors to the next multiple of four, the factor accumulation module processes four factors at a time, the larger the acceleration. The only exception to this trend can be seen in Benchmark 5 (CDX.EM), which as explained by stalls between Phases 2 and 3, and hence doesn't require a new Y_i as often as its TRF would indicate resulting in a linear speedup (although the speedup itself is lower than seen in the other benchmarks).

Table 6.4 summarizes the result across all benchmarks based on three regions SFRF$<$TRF, SFRF$=$TRF, and SFRF$>$TRF. As one would expected the best speedup is obtained in the SFRF$=$TRF region. On average using five integer cores, the maximum that can fit a Virtex 5 SX50T chip, for the SFRF$=$TRF region one can obtain 84.5X acceleration. On average the acceleration for five integer cores is 71.5X.

Table 6.4: MFGC Performance Summary

	SFRF<TRF	SFRF=TRF	SFRF>TRF	Average
Single-Precision				
1 Core	16.7X	20.1X	15.1X	17.0X
4 Cores	52.8X	63.4X	47.7X	53.6X
Double-Precision				
1 Core	13.3X	15.9X	12.0X	13.5X
2 Cores	18.7X	22.5X	16.9X	19.0X
Single-Precision Notionals Double-Precision Accumulator				
1 Core	16.5X	19.9X	14.9X	16.8X
4 Cores	50.3X	60.4X	45.4X	51.1X
Integer				
1 Core	17.7X	22.4X	18.6X	18.9X
5 Cores	67.0X	84.5X	70.1X	71.5X

6.3 Requirements

To maintain the CDO simulation engine fully utilized and really obtain the speedups mentioned in the sections above, once one pricing simulation completes all input data for the next simulation has to be already present in the double buffered memory. This requires two criteria:

1. CDOs have to be priced in batches.

2. The IO device has to provide sufficient transfer rate to fully load data for the next simulation while the previous one is stilled being priced.

In this thesis it is assumed that Point 1 is a given, and CDO pricing does in fact occur in batches [12]. In this section the second criteria is examined: the transfer rate required to keep the accelerator fully active. The worst case scenario, which is the one that requires the fastest data transfer, occurs when a simulation with the shortest computation time is followed by one with the largest input dataset. In such a case to avoid stalls the large input dataset has to be transferred within the computation time of the short simulation.

Examining computational times across all benchmarks and precision representations it was found that the shortest computation times were 2.66 and 2.93 ms, for the OFGC and the MFGC models, respectively. In both cases these times were for Benchmark 5 (CDX.EM), running on five integer cores. The largest number of bits that had to be transfered for a single simulation were 66Kbits and 104Kbits fir OFGC and MFGC models, respectively. In both cases this largest transfer belonged to Benchmark 9 (semi-homogenous). In the MFGC case Benchmark 9 consisted of four systemic factors. Hence, for this dataset a transfer rate over 24.6 Mbits/s (3.1MBytes/s) and 35.4 Mbits/s (4.4MBytes/s) with the host (for OFGC and MFGC designs, respectively) will be sufficient to keep all pricing cores fully utilized.

Let us examine the theoretical maximum input data size that can be transferred to the OFGC and MFGC designs. Based on the internal BRAM sizes, the maximum that can be transfered to the OFGC and MFGC designs is 213Kbits and 262Kbits, respectively. Let us assume the theoretical shortest computational time is the same as was seen in our test data (2.66 ms for OFGC and 2.93 ms for MFGC), it is unlikely a financial institute would desire to price an instrument significantly smaller than Benchmark 5 on a hardware accelerator. Hence, the theoretical data transfer rate required to avoid computational core stalls is 80Mbits/s (10MBytes/s) and 90Mbits/s (11.3MBytes/s) for the OFGC and MFGC designs, respectively.

As a reference typical fast speed communication links have the following bandwidth:

- Gigabit Ethernet -125MBytes/s.
- Peripheral Component Interconnect (PCI) bus - 132 MBytes/s.
- 1-Lane PCI express -250Mbytes/s (one direction) [47].

Given that these link speeds are significantly higher than the design's maximum transfer requirements the pricing cores can be kept busy.

Chapter 7

Conclusions

7.1 Summary

The goal of this research was to develop a hardware accelerator for complex structured financial instrument pricing, namely Collateralized Debt Obligation pricing. The hardware implementation of a CDO pricing engine was designed based on one of the most widely used Monte-Carlo models proposed by Li [14], the Gaussian Copula model.

For the purpose of this thesis, in Chapter 3 one of the smallest reported Gaussian random number generators was created based on Wallace [34] and Lee's *et al.* [24] hardware implementation. Resource wise both the 32- and the 24-bit proposed Wallace generators without sum-of-squares correlation are smaller than the 16-bit inverse Gaussian design [28], the smallest design reported in literature. The proposed generator passed all the applied statistical tests and was shown to produce statistically identical results to an established Matlab Gaussian generator when tested using the CDO pricing application.

Unfortunately, there were no widely accepted benchmarks for CDO pricing and all financial transactions are confidential. Hence, to validate the proposed CDO pricing designs, in Chapter 5 nine different benchmarks were created based on publicly available financial data, such as CDX indices [39] and Moody's [41], in an attempt to closely mimic real world datasets.

The work in Chapter 4 details the hardware pricing engine. The overall multi-core simulation architecture is presented, which uses double-buffering to enable simultaneous calculations and data transfer. In addition single and multi-factor Gaussian Copula cores are presented, each core tries to exploit fine-grain in the algorithm in order to achieve acceleration. The OFGC design parallelizes across time steps by performing eight simultaneous time comparisons. The MFGC design in addition to exploiting the same time parallelism also parallelizes over the factors by computing four factors in parallel.

In Chapter 6 the precision requirements for notionals and the resulting resource utilization were explored. It was established that a special integer representation can adequately represent the data, while utilizing the least resources. This is due to bounded notionals and a final accumulator that only needs to be large enough to sum a known maximum number of notionals. Furthermore, in the MFGC design the advantage of FPGA parallel execution was demonstrated. In certain cases the time for multi-factor accumulation can be completely hidden and increasing the number of factors would have no effect on the overall computation time.

Overall, using a Virtex 5 SX50T chip the maximally replicated integer design on average exceeded the performance of a single-processor 3.4Ghz Xeon machine by 64.5-fold and 71.5-fold for the OFGC and MFGC models, respectively.

The results of this work demonstrate that a generic highly repetitive Monte-Carlo financial simulation, which contains a high degree of both coarse- and fine-grain parallelism (such as Li's models), can be significantly accelerated using reconfigurable hardware. While, other acceleration methods, such as a GPU or a multi-core processor, can effectively exploit coarse-grain parallelism, the flexibility of an FPGA also allows it to fully exploit fine-grain parallelism. The results also establish that by knowing the approximate bounds of the input dataset, one can create a design that achieves similar accuracy to using double-precision format throughout the design while utilizing significantly fewer resources.

7.2 Future Work

There are three natural extensions to the work presented in this thesis: extending the design to a multi-chip system, examining the performance of Li's CDO pricing algorithm on other architectures, and exploring other Copula functions (i.e. not Gaussian).

The current architecture had been designed at tested on a single Virtex 5 chip. To further increase performance one either needs a larger chip, to instantiate more cores, or use a multiple-chip architecture. In many cases due to a lower price and a higher degree of scalability the multiple-chip architecture is more desirable. In the current design there is no inter-pricing core communication. The distributor and collector modules handle all data dispersion and gathering. Hence, to create a multi-chip system no significant changes are needed on the single-chip design. Just add an additional Message Passing Interface (MPI) [48] wrapper that would allow communication with an external global distributor and accumulator module that would handle input and output for every chip. For such an architecture it would be interesting to investigate how many chips can be added before data transfer becomes a bottleneck.

Another venue of exploration could be to examine the acceleration that can be obtained by mapping the algorithm to a different architecture, such as a GPU. In recent years, GPUs have become more and more prevalent in high performance computing [5,6]. It would be interesting to attempt to parallelize the software version of the CDO pricing algorithm to run on a GPU and see how well it compares to an optimized FPGA version, given that there would be relatively little data transfer between the GPU and the main memory.

In addition to the Gaussian copula model there have been proposals for many other copulas [15], such as the double-t copula or the mixture Gaussian copula. All these models use the same general algorithm as presented in Section 2.2 but X and Z_i are substituted with another class of a random numbers, for instance double-t copula uses the student-t distribution. It would be interesting to explore the existent architecture along with the different hardware mapped generators and compare the results to the corresponding software implementation.

The final step would be to try to integrate the FPGA CDO pricing into a typical financial simulator.

Appendices

Appendix A

Diehard Statistical Tests

Table A.1: 88-bit LFSR Diehard Test Suite Results

Test	p-value	pass/fail
Birthday Spacing	0.716828	pass
Overlapping 5-Permutation	0.563281	pass
Binary Rank (31x31)	0.328498	pass
Binary Rank (32x32)	0.496422	pass
Binary Rank (6x8)	0.760208	pass
Bitstream	0.98139	pass
OPSO	0.4679	pass
OQSO	0.5548	pass
DNA	0.6989	pass
Stream Count-the-1	0.77362	pass
Byte Count-the-1	0.911591	pass
Parking Lot	0.546080	pass
Minimum Distance	0.891665	pass
3-D Sphere	0.162825	pass
Squeeze	0.6958393	pass
Overlapping Sums	0.691439	pass
Runs-up	0.352844	pass
Runs-Down	0.867424	pass
Craps	0.769927	pass

Table A.2: Tausworth-88 Diehard Test Suite Results

Test	p-value	pass/fail
Birthday Spacing	0.501205	pass
Overlapping 5-Permutation	0.957269	pass
Binary Rank (31x31)	0.477277	pass
Binary Rank (32x32)	0.970408	pass
Binary Rank (6x8)	0.644971	pass
Bitstream	0.92318	pass
OPSO	0.8256	pass
OQSO	0.6525	pass
DNA	0.9858	pass
Stream Count-the-1	0.751297	pass
Byte Count-the-1	0.789967	pass
Parking Lot	0.637266	pass
Minimum Distance	0.940924	pass
3-D Sphere	0.341797	pass
Squeeze	0.998024	pass
Overlapping Sums	0.901832	pass
Runs-up	0.605756	pass
Runs-Down	0.768902	pass
Craps	0.842063	pass

Table A.3: Sum of 12 URN Diehard Test Suite Results

Test	p-value	pass/fail
Birthday Spacing	0.763723	pass
Overlapping 5-Permutation	0.999813	pass (suspiciously high)
Binary Rank (31x31)	0.329408	pass
Binary Rank (32x32)	0.755372	pass
Binary Rank (6x8)	0.869723	pass
Bitstream	0.92318	pass
OPSO	0.91031	pass
OQSO	0.9308	pass
DNA	0.8866	pass
Stream Count-the-1	0.861942	pass
Byte Count-the-1	0.794820	pass
Parking Lot	0.435624	pass
Minimum Distance	0.740371	pass
3-D Sphere	0.384417	pass
Squeeze	**0.999999**	**fail**
Overlapping Sums	0.874146	pass
Runs-up	0.781097	pass
Runs-Down	0.960614	pass
Craps	0.265090	pass

Table A.4: 32-bit Hardware Wallace generater Diehard Test Suite Results

Test	p-value	pass/fail
Birthday Spacing	0.262246	pass
Overlapping 5-Permutation	0.523412	pass
Binary Rank (31x31)	0.712721	pass
Binary Rank (32x32)	0.330276	pass
Binary Rank (6x8)	0.935021	pass
Bitstream	0.75148	pass
OPSO	0.8867	pass
OQSO	0.9333	pass
DNA	0.6100	pass
Stream Count-the-1	0.751859	pass
Byte Count-the-1	0.284251	pass
Parking Lot	0.660451	pass
Minimum Distance	0.072508	pass
3-D Sphere	0.126510	pass
Squeeze	0.998500	pass
Overlapping Sums	0.898440	pass
Runs-up	0.567118	pass
Runs-Down	0.249868	pass
Craps	0.206318	pass

Appendix B

Multi-Factor Gaussian Copula Performance Results

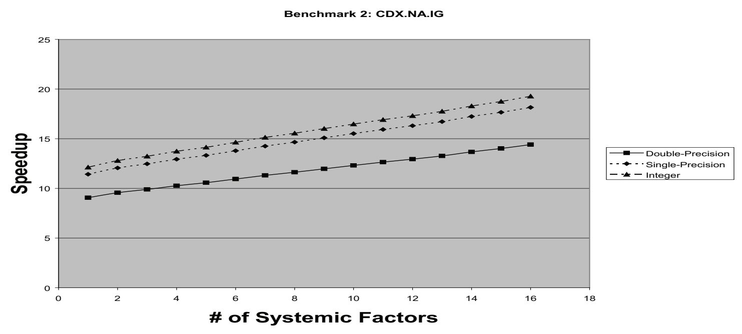

Figure B.1: MFGC Benchmark 2: CDX.NA.IG Performance

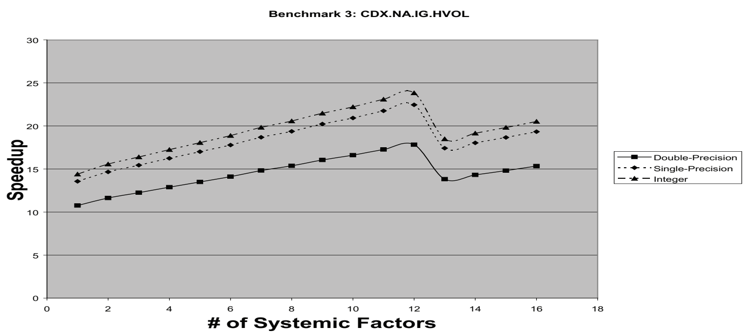

Figure B.2: MFGC Benchmark 3: CDX.NA.IG.HVOL Performance

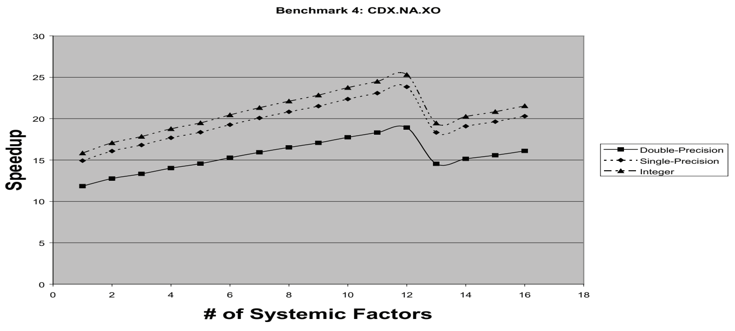

Figure B.3: MFGC Benchmark 4: CDX.NA.XO Performance

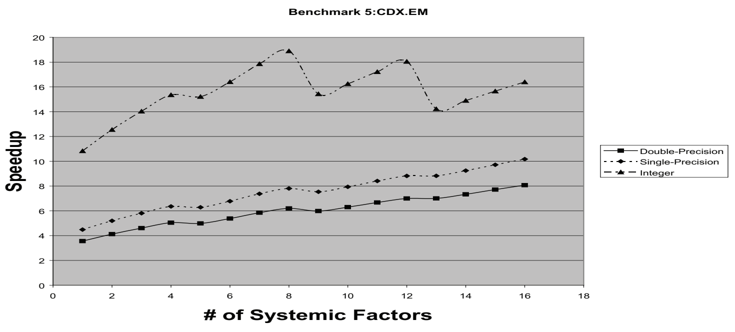

Figure B.4: MFGC Benchmark 5: CDX.EM Performance

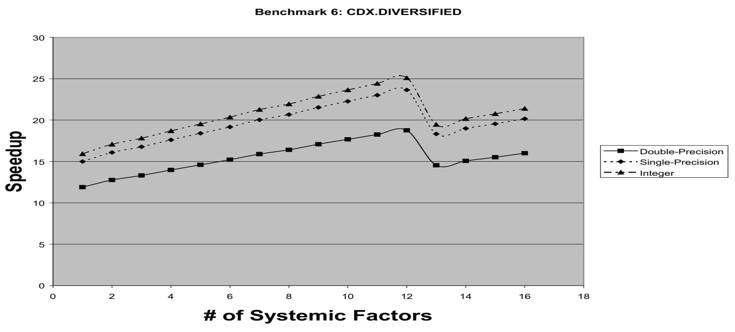

Figure B.5: MFGC Benchmark 6: CDX.DIVERSIFIED Performance

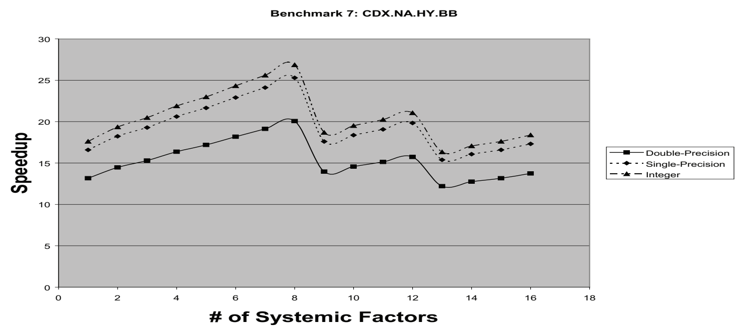

Figure B.6: MFGC Benchmark 7: CDX.NA.HY.BB Performance

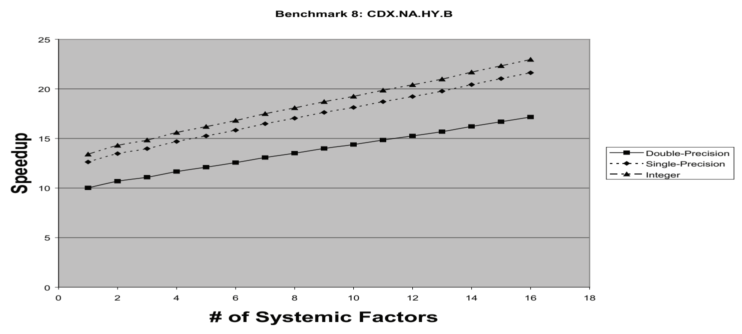

Figure B.7: MFGC Benchmark 8: CDX.NA.HY.B Performance

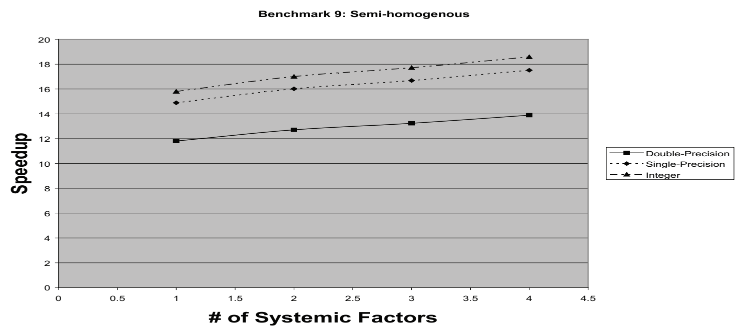

Figure B.8: MFGC Benchmark 9: Semi-Homogenous Performance

Bibliography

[1] M.B. Haugh and A.W. Loe. Computational Challanges in Portfolio Managment. *Computing in Science & Engineering*, 3(8):54–59, may/june 2001.

[2] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[3] Bank of International Settlements (BIS). Amounts Outstanding of Over-The-Counter (OTC) Derivatives. `http://www.bis.org/statistics/otcder/dt1920a.pdf`.

[4] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell - a 64 FPGA Supercomputer. in Adaptive Hardware and Systems. In *AHS 2007. Second NASA/ESA Conference on,*, pages 287–294, 2007.

[5] M. McCool, K. Wadleigh, B. Henderson, and H.Y. Lin. Performance Evaluation of GPUs Using RapidMind Development Platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[6] G.W. Morris and M. Aubury. Design Space Exploration of the European Option Benchmark using Hyperstreams. In *Field Programmable Logic and Applications, 2007. FPL 2007*, pages 5 –10, 2007.

[7] D.B. Thomas, J.A. Bower, and W. Luk. Automatic Generation and Optimisation of Reconfigurable Financial Monte-Carlo Simulations. In *Application -specific Systems, Architectures and Processors, 2007.*, pages 168–173, July 2007.

[8] J.A. Bower, D.B. Thomas, W. Luk, and O. Mencer. A Reconfigurable Simulation Framework for Financial Computation. In *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006.*, pages 1–9, September 2006.

[9] 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 *Field-Programmable Technology, 2005. Proceedings. 2005*, pages 215–222, December 2005.

[10] D.B. Thomas and W. Luk. Sampling from the Multivariate Gaussian Distribution using Reconfigurable Hardware. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007*, pages 3–12, April 2007.

[11] Fixstars Corporation. Accelerating the Monte Carlo Simulation using Cell/B.E.. White paper.

[12] Algorithmics Inc. `http://www.algorithmics.com/EN/`.

[13] SIFMA. Global Market Issuance Data. `http:/www.sifma.org`, 2008. available online.

[14] D.X. Li. On Default Correlation: A Copula Function Approach. *The Journal of Fixed Income*, 9:43–54, 2000.

[15] D. Wang, T. Svetkizarm, and F.J. Fabozzi. Pricing Tranches of a CDO and CDS Index: Recent Advances and Future Research. `http://www.defaultrisk.com/pp_cdo_44.htm`, October 2006.

[16] L.S. Goodman and F.J. Fabozzi. *Collateralized Debt Obligations*. John Wiley & Sons, 2002.

[17] K.Jackson, A. Kreinin, and X. Ma. Loss Distribution Evaluation for Synthetic CDOs. `http://www.defaultrisk.com/pp_cdo_14.htm`, February 2007.

[18] J.Hall and A. White. Valuation of a CDO and an nth to Default CDS Without Monte Carlo Simulation. *Journal of Derivatives*, 12(2):8–23, September 2004.

[19] Annelis L*ü*scher. Synthetic CDO Pricing using the Double Normal Inverse Gaussian Copula with Stochastic Factor Loadings. Master's thesis, University of Zurich, 2005.

[20] A. Patel, C. Madil, M. Salda*ñ*a, C. Comis, R. Pom*è*s, and P. Chow. A Scalable FPGA-based Multiprocessor. In *In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pages 111–120, April 2006.

[21] Agility Design Solutions. `http://www.agilityds.com/default.aspx`.

[22] G. Marsaglio. Diehard: A Battery of Tests of Randomness. `http://stat.fsu.edu/~geo/diehard.html`, 1997.

[23] C. Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, 2004.

[24] D.U. Lee, W. Luk, J.D. Villasenor, G. Zhang, and P.H.W. Leong. A Hardware Gaussian Noise Generator Using the Wallace Method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(8):911–920, August 2005.

[25] P. L'Ecuyer. Uniform Random Number Generation. *Annals of Operation Research*, 53:77–120, 1994.

[26] D.B. Thomas and W. Luk. High Quality Uniform Random Number Generation Using LUT Optimised State-transition Matrices. *Journal of VLSI Signal Processing*, 47:77–92, 2007.

[27] P. L'Ecuyer. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation*, 65(213):203–213, 1996.

[28] D.U.Lee, R.C.C Cheung, J.D. Villasenor, and W.Luk. Inversion-Based Hardware Gaussian Random Number Generator: A Case Study of Function Evaluation Via Hierarchical Segmentation. In *Field Programmable Technology, 2006. FPT 2006*, pages 33–40, December 2006.

[29] G. Zhang, P.H.W. Leong, D.U. Lee, J.D.Villasenor, R.C.C Cheung, and W.Luk. Ziggurat-Based Hardware Gaussian Random Number Generator. In *Field Programmable Logic and Applications, 2005. FPL 2005*, pages 275–280, August 2005.

[30] D.B. Thomas, W.Luk, P.H.W Leong, and J.D. Villasenor. Gaussian Random Number Generators. *ACM Computing Surveys (CSUR)*, 39, 2007.

[31] D.U. Lee, W.Luk, J.D. Villasenor, and P.Y.K. Cheung. A Gaussian noise generator for hardware-based simulations. *IEEE Transactions on Computers*, 53:1523–1532, December 2004.

[32] G. Marsaglia and W.W. Tsang. The Ziggurat Method for Generating Random Variables. *Journal of Statistical Software*, 5, October 2000.

[33] 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 Southeastern Symposium on System Theory*, pages 299–303, March 2003.

[34] C.S. Wallace. Fast Pseudorandom Generators for Normal and Exponential Variates. *ACM Transactions on Mathematical Software (TOMS)*, 22:119–127, March 1996.

[35] C. Rub. On Wallace's method for the generation of normal variates. Technical report, Max-Planck-Institut fr Informatik, Germany, 1998.

[36] Xilinx Inc. *Xilinx ISE 9.2i Software Manuals and Help*, 2007.

[37] Xilinx Inc. Virtex-5 Family Overview. Technical report, Xilinx Inc., June 2008.

[38] B. Przybus. The Virtex-5 SXT Option for High-Performance Digital Signal Processing. On-Line, 2008.

[39] Markit CDX Indecies. `http://www.markit.com`, 2008.

[40] Xilinx. Fast Simplex Link (FSL) Bus (v2.11a). Technical report, Xilinx Inc., June 2007.

[41] Moody's Investors Services. Historical Default Rates of Corporate Bond Issuers, 1920-1999. Technical report, Moody's KMV, 2000.

[42] Standard & Poor's. `http://www.standardandpoors.com`.

[43] ModelSim. `http://www.model.com`.

[44] Xilinx Inc. *ML505/ML506/ML507 Evaluation Platform User Guide*, 2008.

[45] *Endpoint Block Plus v1.5 for PCI Express*, October 2007.

[46] Hitesh Patel. Accelerate Design Performance Using Xplorer. `http://www.xilinx.com/publications/xcellonline/xcell$_$55/xc$_$pdf/xc$_$xplorer55.pdf`, 2005.

[47] M.J.Chong. A PCI Express to PCIX Bridge Optimized for Performance and Area. Master's thesis, Massachusetts Institute of Technology, 2004.

[48] M. Salda$\tilde{n}$a and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL 2006*, pages 1–6, August 2006.