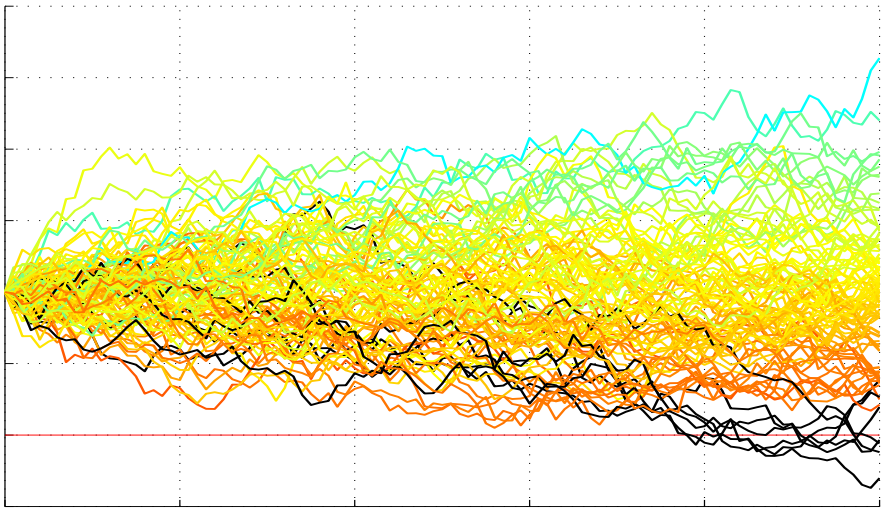
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering in collaboration with the
Department of Management, Technology and Economics

Winter Term 2014

MultiCorePricer:
A Monte-Carlo Pricing Engine for Financial Derivatives



Master Thesis Project

Miguel Angel Benjamin Guerrero
miguelg@ee.ethz.ch

January, 2015

Advisors: D-ITET: Harald Kröll, Marcus Hildmann, Lukas Bruderer
D-MTEC: Donnacha Daly

Professor: Prof. Dr. Qiuting Huang, Prof. Dr. Didier Sornette

Acknowledgements

I would like to thank Prof. Qiuting Huang and Prof. Didier Sornette for the opportunity to do my Master thesis at the Integrated Systems Laboratory and Chair of Entrepreneurial Risks. I also thank Harald Kröll, Donnacha Daly, Marcus Hildmann and Lukas Bruderer for their continued patience and guidance throughout the course of this work. This project would not have been possible without them.

– Miguel Guerrero

Abstract

When the computation power of a high-end workstation cannot match the Monte-Carlo workload of pricing a portfolio of structured products, it is time to consider high-performance solutions. These are often software solutions, achieved by a distribution of computational workload over more or faster processing units. However this project is concerned with hardware-acceleration, whereby computational tasks are conducted with dedicated hardware architectures rather than as a set of software instructions executed on a processing core.

The project's goal was to implement a structured product pricer on a commercially available FPGA and to provide an analysis of the solution in terms of computational acceleration versus a software solution. The main goal was to design a slim and efficient architecture allowing for high parallelization and throughput by allowing a pricing error at a level acceptable for practical applications.

The model chosen for the simulation of the product's underlying assets was the multivariate geometric Brownian motion which represents the well known Black-Scholes model. While this is no longer the most widespread model in use, it allows for easy calibration to analytic prices of European options and can readily be extended by more sophisticated models. The backbone of this model, and practically every other financial model, is the use of normally distributed random numbers. Highly sophisticated models have been proposed to generate such numbers. The approach of this project, however, was to use a more simple generator, based on the central limit theorem, to reduce used area and attain a higher speed.

The results of this thesis show that the chosen components are more than fit for a Monte-Carlo pricing engine. A high speed-up ranging from 550 to 1450 was achieved versus a one core software solution. Using autonomous "Cores" to simulate paths in parallel, the design has a good portability and can be migrated to boards with higher capabilities to increase the pricing speed. A batch pricing program has been implemented onto the processing system to demonstrate a possible way of using this system in a real world application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and problem statement

This chapter gives an insight into the market of structured products and what importance they have within the Swiss market based on up to date figures of their size and volume. An example product is given at the end of the chapter.

The next part introduces the current setups used in banks and financial institutions that are used to evaluate these products and how this project comes into play as part of the recently started adoption of FPGA computing in this field.

The financial product addressed by this project, the worst-of barrier option, will then be presented and its pricing explained. Finally the project description and goals are given.

1.1. Structured Products

A core task of investment banking is the search for possibilities of creating new financial instruments. This task is typically fulfilled by combining already existing components to create new financial instruments. A prominent group of newly introduced financial instruments are termed structured products. These are products which are based on multiple underlying assets and are tailored to meet specific expectations and fit a certain risk profile. As a result they can have arbitrary non-linear payoff profiles depending on the movement in price of their underlyings [2]. They are engineered from traditional assets such as bonds, shares and derivatives. Through the countless possible combinations of different underlyings, practically every imaginable market scenario can be covered. As they enable small investors to speculate on very specific market scenarios they have seen an increasing popularity in recent years.

In October 2014 structured products had a volume of 202 billion Swiss francs in custody accounts at banks in Switzerland representing 3.69% of the total securities volume [3].

Table 1.1 shows the five banks with the highest trading turnover made with structured products in November 2014 [4].

Issuer	Trading Turnover (in Mio. CHF)	Percentage of the Total Turnover	Number of Trades
UBS	486.96	27.64%	11'644
Vontobel	423.39	24.03%	26'725
ZKB	319.67	18.14%	10'431
Julius Bär	164.67	9.35%	4'285
Credit Suisse	71.99	4.09%	1'518

Table 1.1.: Structured products turnover in November 2014 for a selection of banks

1.2. Efficient Monte-Carlo Methods

The pricing of structured products relies on the prediction of price distributions of the underlying assets. Analytical formulas have been developed for simpler products based on various assumptions on the statistics of price movements. The classic example of this is the Black-Scholes formula for the pricing of options. In complex cases however, a closed-form expression for fair value pricing does not exist and can only be achieved via numerical methods such as Monte-Carlo simulations [5, 6, 7, 8].

Due to the complexity of structured products, the development of fair value pricing requires a considerable amount of work. An example for this are certain barrier products which only impose losses when a lower price barrier is crossed by an underlying. For certain cases this is very unlikely and a sophisticated algorithm is required to capture fluctuations from these events.

Two approaches exist to overcome these challenges of accurate Monte-Carlo pricing. The first is sophisticated signal processing, the second brute force using high performance computing. Sophistication uses advanced techniques from statistical signal processing such as importance sampling which enable a probabilistically consistent pricing under reduced complexity Monte-Carlo simulations. This family of efficiency improving estimation methods is called variance-reduction [9, 10]. Their industrial application is limited however, as every new structured product may require its own Monte-Carlo framework which would clearly be undesirable when institutions may have thousands of such products on their books.

Under the assumption of good stochastic models for the underlying price processes, it can be assured that Monte-Carlo simulations generate accurate prices by increasing the number of simulated future prices and by reducing the simulation step size. This approach

quickly runs into the computational limits of conventional CPUs, which is where high performance computing becomes attractive. This trend can also be seen in banks and other financial institutions investing heavily in grid computing for Monte-Carlo pricing.

Many models have been proposed to capture the essence of market dynamics. The simplest is the (multivariate) Geometric Brownian Motion (GBM). GARCH models capture the fact that volatility changes over time. Local- and stochastic volatility models calibrate the process to the volatility surface implied from options market. The focus of this project was on the GBM model. While this is clearly not the best model for pricing, given the non-normality of observed market dynamics, it allows the easy calibration to analytic prices of European options. The work developed here can be readily extended to more sophisticated models

1.3. Hardware Acceleration

Today high performance computing has become ubiquitous in banks and other financial institutions burdened with processing data at higher frequencies and asset-valuation/risk-assessment of ever more complex and divers portfolios [11, 12]. Typical applications include dynamic multi-period portfolio optimization, portfolio risk measurement via modeling and large scale simulation, data-mining, low-latency trading and, concerning this project, batch jobs for option pricing.

Figure 1.1 shows alternative solutions for high performance computing. Parallelization using Multi-core processors and/or multi-threading is the most straightforward route to faster computation [13, 14]. Distributed solutions using PC- or virtual machine clusters are very popular in financial institutions, as they can leverage existing IT infrastructure. More recent advances in graphic cards have lead to the development of numerous applications in finance exploiting the ability of these specialized processors (GPUs) to do certain matrix-based computations, for example in option pricing [15], at high speeds.

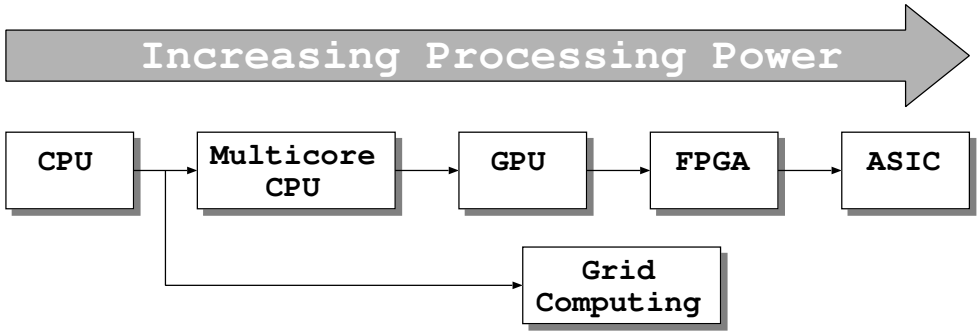


Figure 1.1.: High Performance Computing (HPC) alternatives ranked by increasing throughput capability

The mentioned solutions can be considered software solutions as the pricing algorithm instructions are executed on many fast processing units. By contrast it is possible to move from software to hardware and implement what is known as hardware acceleration by performing the computations at the logical bit level in silicon. Implementing algorithms in silicon in the form of an application specific integrated circuit (ASIC), however, is costly and most often only proves worthwhile for high volume applications such as ICs used in consumer products. The use of a field programmable gate array (FPGA), which enables flexible silicon reconfiguration at low costs, represents an intermediate solution.

FPGAs are increasingly applied in financial settings for low latency trading [16], credit derivatives and -risk [17, 18], option pricing [19, 20], and other Monte-Carlo applications [21, 22, 23, 24]. There are, however, only a few publicly available citations, commercial or academic, on the use of FPGA technology for batch pricing of structured products ([25] is one example). Given the large and growing market for these products, the huge processing power required for accurate pricing using Monte-Carlo methods and the computational acceleration available through the appropriate use of FPGA hardware, a more thorough examination of this area is called for and is given by this project.

1.4. Financial Instruments

This section introduces first one of the most basic financial instrument, the call option, goes to the barrier call option and finally to the worst-of barrier call option with which this project is concerned. The valuation of these instruments is explained for each one and observations concerning their Monte-Carlo simulation are made.

1.4.1. Call Option

A call option is a financial contract which gives the buyer the right but not the obligation to buy an agreed quantity of an underlying asset at a specified time, called maturity, for a specified price, called strike price. Depending on the style of the option (European, American, Asian) it can be exercised during or on maturity or depends on the underlyings development. For this project the only style considered was the European option which can only be exercised on the maturity date. To compensate for the risk the issuer takes, the buyer has to pay a premium, also known as the price of the option. An investor who buys a call option expects the price of the underlying to go above the options strike price, to be able to buy it for a lower price than its current one and sell it for a profit.

The payoff of this financial product then solely depends on the underlyings price at maturity whereas the profit additionally depends on the premium which was paid for the option. The corresponding profiles can be seen in figure 1.2. At maturity the payoff is either zero, if the underlying's price lies on or below the strike price (option will not be exercised), or the underlying's price minus the strike price, if it lies above the strike price. To determine the expected payoff, a high number of simulations of the underlying's price

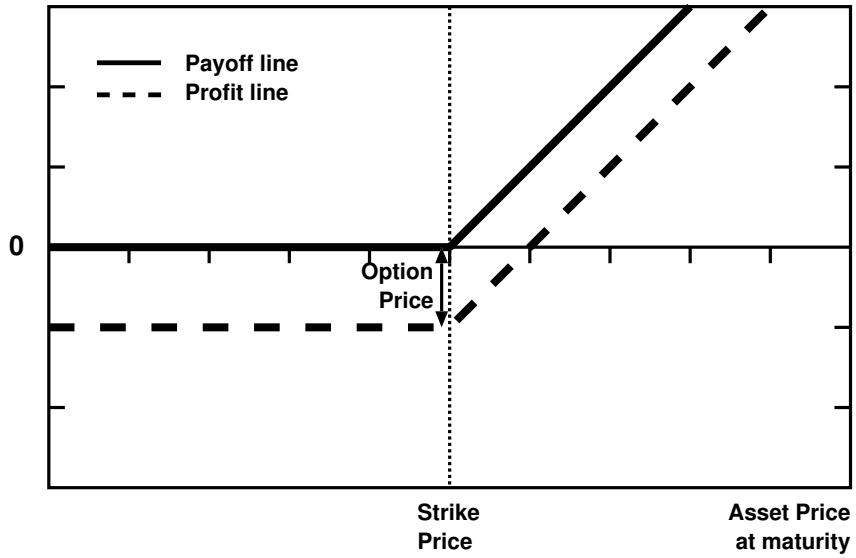


Figure 1.2.: Call option payoff/profit profile

have to be performed and the mean of all their payoffs has to be taken. This expected payoff of the option is then used by financial institutions to set the price of the product, in this project it is assumed to be the same.

Statement 1: The payoff of a European call option only depends on the simulated underlying's price at maturity.

From the first statement it can be concluded that it is not necessary to consider any price other than the maturity price of each simulated path. As it is possible to accumulate the weighted payoffs to determine their mean, it isn't even necessary to store them during simulation.

1.4.2. Barrier Call Option

The barrier call option is a more exotic offshoot of the normal call option which meets more specific expectations in an investor's portfolio. It has a barrier level, predefined by the issuer, which, when reached by the underlying's price, either springs the option into existence or extinguishes an already existing option. Because of these more restrictive conditions, a barrier option is cheaper than a similar option without barrier and therefore provides the insurance value of an option without charging as much premium.

This project is mainly concerned with down-and-out barrier call options which are call options which get extinguished if their underlying's price falls on or the below their barrier level (barrier event). Their payoff and profit profile can be seen in figure 1.3. The only difference to the previously presented call option profile is the barrier, defined

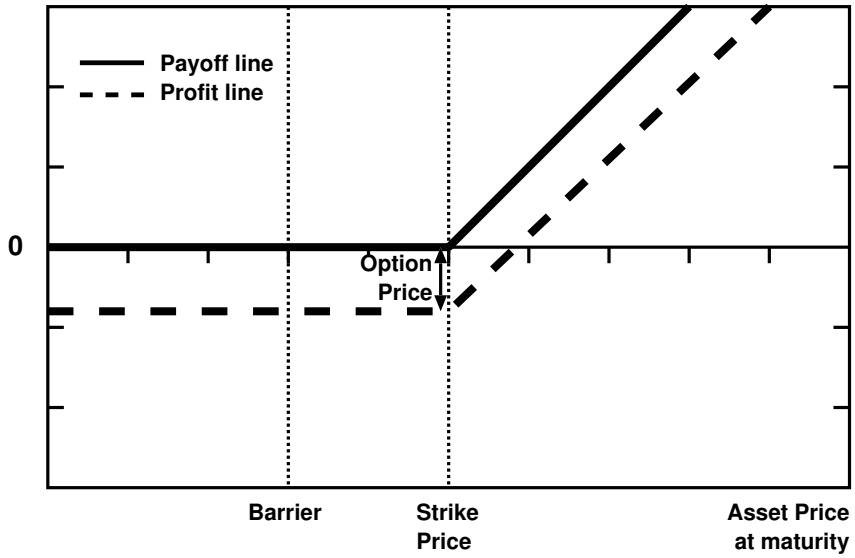


Figure 1.3.: Down-and-out barrier call option payoff/profit profile

as a percentage relative to the underlying's initial price, which acts as a knock-out that extinguishes the option if the underlying's price falls on or below the barrier at any point during the product's lifetime. If such a barrier event occurs the payoff immediately becomes zero and the initially paid premium is lost as the option can no longer be exercised at maturity.

Statement 2: A down-and-out barrier option's payoff becomes zero if a barrier event occurs.

From the second statement it can be concluded that a running price simulation can be aborted if the price falls at any point on or below the barrier.

1.4.3. Worst-of Barrier Call Option

The worst-of barrier call option is a multi-asset version of the barrier call option and belongs to the category of structured products. It is fairly simple to derive its pricing process from the previously presented ones. To determine its payoff the same principles as with the barrier call option are used with the addition that the payoff is determined by the underlying with the lowest price at maturity, relative to its initial price. Therefore an additional minimum check has to be performed over all underlyings at the end of a price simulation. Even though there are multiple underlyings the barrier level is the same for all underlyings and is taken relative to the corresponding initial price. A barrier event occurs if any underlying's price touches or falls on a barrier and causes the optionality of the product to become void.

The architecture presented in this thesis was designed for the pricing of products using three underlyings, as the great majority of these products are defined for this number of underlyings.

1.4.4. Put Options, Best-of, Up/Down-and-In/Out

There are a great variety of products with reversed or altered behaviour to the presented worst-of barrier call option which serve very different market expectations like the use of put options as underlyings which have the reversed payoff profile of the call option, the best-of style whose payoff is determined by the best performing underlying, the up style barrier option with the barrier above the initial price, the in style option which instead of extinguishing on a barrier event is inactive until an event and can only be exercised if the event occurs. These product variants, however, only need minor adjustments in the pricing engine and can be easily derived from the presented design. This coincides with this projects fundamental concept of using a very specific architecture for a single type of product with a specific style to achieve the highest possible throughput. In a real world application this would simply lead to having different FPGAs assigned to the task of pricing of different products.

1.4.5. Contract Example: Multi Barrier Reverse Convertible

Figures 1.4 and 1.5 show an example contract of a multi barrier reverse convertible [1]. It's payoff is explained in the contract. This kind of products are built out of zero coupon bonds and a worst-of option. The buyer of the product effectively sells the issuer a worst-of down-and-in put option. Therefore to price this product it is necessary to price the corresponding down-and-in put option.

1.5. Project Details

1.5.1. Project Description

For this project, a Zynq-7020 FPGA was used to accelerate the pricing of a portfolio of multi-asset barrier call options in a Monte-Carlo framework. The concepts used for the implementation on the FPGA are explained in chapter 3. As a first step, the Monte-Carlo framework was implemented in Matlab. It was then analyzed which parts of the framework were suitable for hardware implementation. For the hardware blocks a dynamic range analysis and fixed-point simulation was performed in order to obtain the groundwork for the golden model. The golden model was then used as a reference during the implementation of the pricing engine in hardware description language (VHDL). The core part of this project involved the digital design of the pricing engine and an

empirical assessment of the computational acceleration of a hardware implementation over a conventional software approach.

1.5.2. Project Goals

The following goals were targeted for this project:

- Theory and simulation goals
  - Understanding the basics of stochastic models for asset price time-series in financial markets, particularly correlated geometric Brownian motion.
  - Understanding the nature of multi-basket structured products, and how they are priced in a Monte-Carlo framework.
  - Monte-Carlo pricing framework in Matlab
  - Pricing of call options and multi-asset barrier products in Matlab and determining the dynamic range of parameters.
  - Hardware/software partitioning of the pricing framework for the identification of suitable blocks for hardware acceleration.
  - Exploit parallelism and define proper interfaces.
  - Extraction of a hardware model in form of a detailed block diagram.
  - Implementation of a fixed-point version of the pricing engine in Matlab which serves as a golden model for the subsequent VHDL implementation.

- Hardware goals
  - Pricing engine in VHDL with a relative error of 1% or less. Implementation of a functionally verified synthesizable VHDL design.
  - An FPGA portation of the VHDL code on the Zedboard, a development board for the Xilinx Zynq.
  - Empirical assessment of the achievable hardware speed-up over software implementation of structured product pricing engine (comparison with Matlab implementation, as well as any available published figures).

EFG Financial Products

Brandschenkestrasse 90, P.O. Box 1686, CH-8027 Zurich
+41 58 800 1111 termsheet@efgfp.com www.efgfp.com

COSI® Investor Protection engineered by SIX Group

Termsheet as of 12/03/2012
COSI (Collateral Secured Instruments)

Yield-Enhancement Products
SSPA Product Type: 1230
Collateralised Derivatives

5.125% p.a. Multi Barrier Reverse Convertible on Nestle, Novartis, Roche

Continuous Multi Barrier Observation

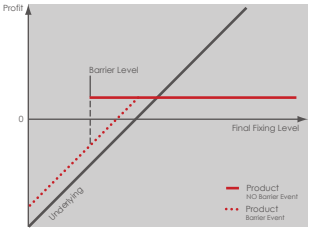
Final Fixing Date 12/03/2015; issued in CHF; listed on SIX Swiss Exchange

This Product is collateralised in accordance with the terms and conditions of the SIX Swiss Exchange Ltd Framework Agreement for Collateral Secured Instruments. More detailed information regarding the collateralisation can be found in the section: "Information about Collateralisation"
This document contains a summary of information of the Product and is for information purposes only. Only the Final Termsheet in English language together with the Programme containing all further relevant terms and conditions, as amended from time to time, shall form the entire documentation for this Product ("Product Documentation").
This Product is a derivative instrument. It does not qualify as unit of a collective investment scheme pursuant to art. 7 et seqq. of the Swiss Federal Act on Collective Investment Schemes (CISA) and is therefore neither registered nor supervised by the Swiss Financial Market Supervisory Authority FINMA. Investors do not benefit from the specific investor protection provided under the CISA.

I. PRODUCT DESCRIPTION

Market expectation
Underlyings trade sideways to slightly higher.
The Barrier Event will not occur.

Product description
This Product offers the Investor a Coupon Rate regardless of the performance of the Underlyings during lifetime whilst combined with a conditional downside protection. If no Barrier Event has occurred, the Investor will receive the Denomination at the Redemption Date. If a Barrier Event has occurred but all Underlyings close above their Initial Fixing Level at the Final Fixing Date, the Investor will still receive at the Redemption Date a Cash Settlement which equals the Denomination. Otherwise the Investor will receive at the Redemption Date either a round number (i.e. Conversion Ratio) of the Underlying with the Worst Performance or, as the case may be, a Cash Settlement in the Settlement Currency, as further described under Redemption.



Underlying

Underlying	Related Exchange	Bloomberg Ticker	Initial Fixing Level (100%)*	Barrier Level (55.00%)*	Conversion Ratio
NESTLE SA-REG	SIX Swiss Exchange	NESN VX	CHF 56.60	CHF 31.13	17.6678
NOVARTIS AG-REG	SIX Swiss Exchange	NOVN VX	CHF 50.00	CHF 27.50	20.0000
ROCHE HOLDING AG-GENUSSCHEIN	SIX Swiss Exchange	ROG VX	CHF 158.40	CHF 87.12	6.3131

Product Details

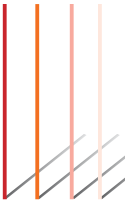
Swiss Security Number	**14911903**
ISIN	**CH0149119031**
SIX Symbol	**EFNBT**
Issue Price	100.00%
Issue Size	CHF 10'000'000 (can be increased at any time)
Denomination	CHF 1'000
Settlement Currency	CHF
Coupon Rate	5.125% p.a.
	The Coupon Rate is split in two components for Swiss taxation purposes:
	Interest Component 0.220% p.a.
	Option Premium Component 4.905% p.a.

** levels are expressed in percentage of the Initial Fixing Level*

Subscription End Date 09/03/2012 OVER	First Exchange Trading Date 20/03/2012	Barrier Observation 09/03/2012 - 12/03/2015 ACTIVE	Barrier Level Nestle (55.00%)	Barrier Level Novartis (55.00%)	Barrier Level Roche (55.00%)	Coupon Amount CHF 51.25 20/03/2013
Coupon Amount CHF 51.25 20/03/2014	Coupon Amount CHF 51.25 20/03/2015	Final Fixing Date 12/03/2015				

CK78: 3b304c25-b929-49ae-bfb5-5d11981db41d

Figure 1.4.: Example contract of a multi barrier reverse convertible [1]



Coupon Amount(s) and Coupon Payment Date(s)	The Coupon Amount(s) per Product will be paid in the Settlement Currency on the respective Coupon Payment Date(s). Following Business Day Convention applies. CHF 51.25 paid on 20/03/2013 CHF 51.25 paid on 20/03/2014 CHF 51.25 paid on 20/03/2015

Dates

Subscription End Date	09/03/2012 14.00 CET
Initial Fixing Date	09/03/2012
Issue Date	20/03/2012
First Exchange Trading Date	20/03/2012 (anticipated)
Last Trading Day/Time	12/03/2015 / Exchange market close
Final Fixing Date	12/03/2015 (subject to Market Disruption Event provisions)
Redemption Date	20/03/2015 (subject to Settlement Disruption Event provisions)

Redemption

The Coupon Amount(s) per Product will be paid in any case at the respective Coupon Payment Date(s). In addition the Investor is entitled to receive from the Issuer on the Redemption Date per Product:

Scenario 1	If a Barrier Event has NOT occurred, the Investor will receive a Cash Settlement in the Settlement Currency equal to: Denomination
Scenario 2	If a Barrier Event HAS occurred and a. If the Final Fixing Level of the Underlying with the Worst Performance is at or below the respective Initial Fixing Level, the Investor will receive a round number (i.e. Conversion Ratio) of the Underlying with the Worst Performance per Product. Any potential fractional Conversion Ratio entitlements (Fraction of Underlyings) will be paid in cash, based on the Final Fixing Level. b. If the Final Fixing Level of the Underlying with the Worst Performance is above the respective Initial Fixing Level, the Investor will receive a Cash Settlement in the Settlement Currency equal to: Denomination
Initial Fixing Level	Official close of the respective Underlying on the Initial Fixing Date on the Related Exchange, as determined by the Calculation Agent.
Final Fixing Level	Official close of the respective Underlying on the Final Fixing Date on the Related Exchange, as determined by the Calculation Agent.
Worst Performance	The lowest performance of the respective Underlyings whereby each performance is calculated by dividing the respective Final Fixing Level by the respective Initial Fixing Level, as determined by the Calculation Agent.
Barrier Event	A Barrier Event shall be deemed to occur if at any time on any Exchange Business Day during the Barrier Observation Period the level of at least one of the Underlyings' prices has been traded at or below the respective Barrier Level, as reasonably determined by the Calculation Agent.
Barrier Observation Period	09/03/2012 - 12/03/2015

General Information

Issuer	EFG Financial Products (Guernsey) Ltd., St Peter Port, Guernsey
Guarantor	EFG International AG, Zurich, Switzerland (Rating: Fitch A with negative outlook, Moody's A3 with stable outlook)
Collateral Provider	EFG Financial Products AG, Zurich, Switzerland
Lead Manager	EFG Financial Products AG, Zurich, Switzerland
Calculation Agent	EFG Financial Products AG, Zurich, Switzerland
Paying Agent	EFG Financial Products AG, Zurich, Switzerland
Distribution Fees	Relevant Fees (as defined in article 26 of the General Terms and Conditions which are a part of the Programme)
Listing/Exchange	SIX Swiss Exchange; traded on Scoach Schweiz AG Listing will be applied for.
Secondary Market	Daily price indications will be available from 09:15 - 17:15 CET on www.efgfp.com, Thomson Reuters [ISIN] and Bloomberg [ISIN] Corp or on EFGZ.
Quoting Type	Secondary market prices are quoted clean; accrued Coupon Amount is NOT included in the prices.

Figure 1.5.: Example contract of a multi barrier reverse convertible [1]

Chapter 2

Related Work

To the best of the author's knowledge there has been so far no publication of an FPGA architecture for option pricing using a random number generator solely based on the use of the central limit theorem, which will be further explained in the next chapter.

An FPGA implementation for the pricing of multi-asset barrier options has been presented by Sridharan et al. [25] in 2012. The main difference to this work is their use of the more sophisticated Heston model for the simulation of the evolution of an underlying asset. They further realized designs for 4, 8, 16 and 32 underlying assets which require more correlation operations compared to this projects.

Another conceptual difference is the arrangement of having multiple threads or simulated product paths in a single core with each core having a scheduler in contrast to this project where each core simulates a single product path and all cores have a single scheduler. The amount of schedulers needed depends mainly on the products maturity time and Δt as these two parameters determine how many iteration steps have to be performed for each for a price simulation.

To produce normally distributed random numbers two inversion-based random number generators, as described by Cheung et al. [26], were used. The design was implemented on a Stratix IV E530 FPGA on Novo-G. The speed-up they achieved from one FPGA compared to one CPU for a product using four underlyings was 350. It is difficult, however, to use their results for a comparison as no exact product parameters are given, as well as no absolute computation times.

Schryver et al. [27] also presented a pricing engine using the Heston model but focused on developing an architecure with a low energy consumption. It prices a double barrier knock-out option, which becomes void if either an upper barrier or lower barrier is hit. They implemented their design on a Xilinx Virtex-5 device. Another difference to this project is the split-up of the pricing process between the FPGA and PC where the FPGA only simulates price pathes and sends the price of each path to the PC using USB. The

Gaussian random number generator used was an inversion based non-uniform random number generator [28].

Compared to a fully loaded 8-core Intel Xeon server running at 3.07GHz it saves 89% of energy and is twice as fast. Compared to a Nvidia Tesla C2050 graphics card it has a 35% speed up and save 60% energy. They additionally list the absolute computation times of a single FPGA and a dual core CPU (Intel Core 2 Duo T7250 at 2.0GHz with 2GB RAM) pricing using 10 million simulated paths. The speed-up achieved in this comparison was 25. As only one product is presented, however, it is difficult to say if the FPGA and/or CPU pricing engines benefit from knock-outs caused by narrow barrier levels. Finally they also list a table showing the hardware utilization of two variants of their design, which is beneficial to comparisons with other projects.

Tian et al. [29] presented an implementation of a single option pricing model using the GARCH model, which they implemented on a 64 FPGA node supercomputer called Maxwell [22]. Furthermore they explored log-normal price movements and correlated asset Value-at-Risk calculations. The normal random number generator they used was a Box-Muller generator using Tausworthe uniform random number generators. As Schryver et al. they split-up the simulation of paths to the FPGA and averaging or other operations to the host. Their target precision was 0.01%.

Separation of path simulation and processing of the generated data between the FPGA and host, as done by Schryver and Tian is feasible if the FPGA, allows for high speed communication, e.g. using a PCI connection. If this is not possible a loss of computation time would occur, as the FPGA would have to wait until all data is read out, next to the necessity of RAM to store the generated data. It is reasonable to do this if the pricing process consists of more complex operations than simply taking the average of all prices and contains costly operations like multiplications or divisions.

As also mentioned by other authors, comparison between different publications is very difficult as both FPGA and PC hardware differ and practically no publication gives absolute computation times, only compuatation times of the FPGA relative to software. Additionally the great number of existing mathematical models for financial markets and the even greater number of existing algorithms for generating normally distributed random numbers, make it very unlikely that any two publications will have implementations of these functions with similar complexity. Because of this it is also important to give a comprehensive list of used logical slices, DSPs etc. of the components, even though these metrics may differ between FPGA producers. The comparison of absolute computation time with board utilization would then allow for better comparison between publications.

This thesis tries to achieve this by presenting its results in a table containing utilization information for all component blocks, a list of absolute computation times and speed-ups relative to the compared software implementation.

Chapter 3

Theoretical Background

This chapter introduces different methods and theories used in the pricing of structured products and emphasizes on those which were chosen.

3.1. Simulating an Asset

The first step to simulate any financial derivative is to choose a model which realistically represents its performance. First a list of the most common models with a short description is given, then the reasons for choosing the model used in this project are given and finally the iterative algorithm is derived from the model equation.

3.1.1. Black-Scholes Model

The Black-Scholes model, introduced in 1973, is a mathematical model of a financial market with certain derivative investment instruments. The Black-Scholes formula can be deduced from the model and gives an estimate of the theoretical price of an European option, ignoring any dividends paid during the option's lifetime. The model led to a boom in the option market, but has the flaw that it assumes a constant volatility which cannot be observed in real markets. Next to the constant volatility and zero dividend, the following assumptions about assets and the market are made:

- Constant risk-free interest rate: the rate of return of the riskless asset (e.g. government bond) is constant.
- Efficient markets (asset movements cannot be predicted): the instantaneous log returns of the asset price is an infinitesimal random walk with drift, therefore a geometric Brownian motion.

- No arbitrage opportunity: there is no way of making a riskless profit.
- The possibility of borrowing and lending any amount of the asset is given.
- The possibility of buying and selling any amount of the asset is given.
- Frictionless market: transactions in the market do not have any fees.

3.1.2. Heston Model

The Heston model, introduced in 1993, generalizes the Black-Scholes model and has a stochastic volatility (instantaneous variance), modeled by a CIR process, which is based on a geometric Brownian motion. The two stochastic processes have a correlation which is usually taken to be negative: an increase/decrease of the asset price leads to a decrease/increase of the volatility. The set of parameters for the Heston model have to be determined using market observed prices of European options for various strike prices and maturities, it uses therefore implied volatilities. When this set of parameters have been determined, the model can be used to price European, American or more exotic options.

Because of the additional effort necessary to precisely calibrate a pricing engine using the Heston model to have realistic parameters and the resulting increase in computation time consumed to generate a pool of comparison prices for the FPGA engine, the simpler Black-Scholes model was chosen for this project. This also allows for an evaluation of the used Gaussian random number generator (GRNG) on a model using only one stochastic process before tackling a more complex model using two stochastic processes.

3.1.3. Geometric Brownian Motion (GBM)

The geometric Brownian motion, modeling the price movement of an asset in the Black-Scholes model, is a stochastic process in which the logarithm of the randomly varying quantity follows a Brownian motion with drift. The stochastic differential equation describing the GBM is:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t \, , \tag{3.1}$$

where S_t is the asset price at time t, μ is the percentage drift, σ is the percentage volatility and W_t is a Wiener process or Brownian motion.

Solving this equation results in a log-normally distributed random variable, with initial value S_0 from which a sample of the asset price at any time t can be generated:

$$\ln \frac{S_t}{S_0} = \left(\mu - \frac{1}{2} \sigma^2 \right) t + \sigma W_t \tag{3.2}$$

From this, the formula for an iteration step follows:

$$S_t = S_{t-\Delta t} \cdot \exp(\underbrace{(\mu - \frac{\sigma^2}{2})\Delta t}_{drift} + \underbrace{\sigma W_{\Delta t}(t))}_{diffusion} \tag{3.3}$$

where $W_{\Delta t}(t)$ is the increment of the Wiener process at time t and is therefore a normally distributed random variable. Further the exponential function's input parameters can be subdivided into a drift and diffusion part, with the drift part being constant.

To price multi-asset products it is necessary to simulate correlated assets. This can be done using a multivariate GBM, with each price process observing:

$$dS_t^i = \mu S_t^i + \sigma S_t^i \, dW_t^i \ , \tag{3.4}$$

This extension requires the use of correlated Wiener processes of the following form:

$$\mathrm{E}(dW_t^i dW_t^j) = \rho_{i,j} dt \ , \tag{3.5}$$

where $\rho_{i,j}$ is the correlation coefficient between asset i and j.

3.1.4. Correlated Wiener Process

To generate correlated Wiener processes for n assets, n correlated simulation paths must be generated. For a set of GRNs to be correlated, the following equations must hold:

$$\sum_{k=1}^{i} \alpha_{i,k} x_k = \varepsilon_i \qquad 1 \leq i \leq n \tag{3.6}$$

$$\sum_{k=1}^{i} \alpha_{i,k}^2 = 1 \qquad 1 \leq i \leq n \tag{3.7}$$

$$\sum_{k=1}^{i} \alpha_{i,k} \alpha_{j,k} = \rho_{i,j} \qquad \forall \, j < i \tag{3.8}$$

where ε_i are the correlated GRNs, x_i are some uncorrelated GRNs, $\rho_{i,j}$ the correlation coefficients and $\alpha_{i,j}$ the coefficients required for the calculation of ε_i. It should be noted that the GRNs must have unit variance and zero mean, as this will become important in the next section. Writing these conditions in matrix form results in a particular matrix decomposition called a Cholesky factorization. In the case of three assets it has the form:

$$A = LL^* = \begin{bmatrix} 1 & 0 & 0 \\ a_{2,1} & a_{2,2} & 0 \\ a_{3,1} & a_{3,2} & a_{3,2} \end{bmatrix} \begin{bmatrix} 1 & a_{1,2} & a_{1,3} \\ 0 & a_{2,2} & a_{2,3} \\ 0 & 0 & a_{3,3} \end{bmatrix} \tag{3.9}$$

where $\alpha_{i,j} = \alpha_{j,i}$. The correlation of the RNs then breaks down to five multiplications:

$$\begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ a_{2,1} & a_{2,2} & 0 \\ a_{3,1} & a_{3,2} & a_{3,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3.10}$$

3.2. Gaussian Random Number Generator (GRNG)

Often used methods for producing non-uniformly distributed RNs out of uniform RNs are the transformation, rejection and inversion methods presented here as well as the central limit theorem, which is limited to generating GRNs.

3.2.1. Box-Muller Method

A common method used for GRNGs is based on the Box-Muller transformation, which transforms a pair of uniformly distributed RNs into a pair of GRNs through the use of trigonometric functions:

$$Z_0 = R\cos(\Theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{3.11}$$

$$Z_1 = R\sin(\Theta) = \sqrt{-2\ln U_1}\sin(2\pi U_2) \tag{3.12}$$

where U_0 and U_1 are independent random variables with a uniform distribution and Z_0 and Z_1 are the resulting independent random variables with a Gaussian distribution.

As most uniform random number generators (URNG) cannot take a zero state, the bit width of the generated RN gives a limit to how close a number can be to zero, which limits the resulting tail of the Gaussian distribution. For a RN with a bit width of 32, the Box-Muller transform will produce a normal random variable with a standard deviation up to 6.66.

$$\sqrt{-2\ln(2^{-32})}\cos\left(\frac{2\pi}{2^{32}}\right) = 6.66 \tag{3.13}$$

An advantage of this method is, that it only needs one URN per GRN. Disadvantages for the implementation on hardware are the high hardware cost and the dependence of the bit width with the tail region making it impossible to have a small tail distribution with a lot of subdivisions and vice versa.

3.2.2. Cumulative Distribution Function Inversion Method

CDF inversion methods work by applying the respective inverse CDF to an input sample from a uniform distribution, therefore it produces one GRN per URN. For a continuous distribution the integral of the PDF of the desired distribution has to be taken. Since this is analytically impossible for most distributions, including the Gaussian distribution, an approximation has to be computed. One method of approximation is the use of rational polynomials, however, these are not suitable for an area efficient hardware implementation. A more suitable implementation on hardware has been presented by Cheung et al. [26]. It evaluates the ICDF via piecewise polynomial approximation and look-up tables (LUTs) with a hierarchical segmentation scheme.

3.2.3. Ziggurat method

A popular rejection sampling algorithm for software implementations of GRNGs is the Ziggurat method as it provides high quality RNs. It randomly generates a point in a distribution slightly larger than the desired distribution, then tests whether the generated point is inside the desired distribution. If this is not the case, the RN is rejected and the process repeated. A typical value generated by the algorithm requires one random floating-point value and one random table index, followed by one table lookup, one multiply operation and one comparison. In some cases, about 2% of the time for a normal distributions, more computations are required. A drawback of this method is, that not every input RN produces one GRN, as certain samples are rejected, which requires special considerations for its use in a Monte-Carlo simulation.

An implementation optimized for FPGAs has been presented by Edrees et al. [30]. It has a main and tail unit where the latter is used for the before mentioned 2% of cases where the PDF of the distribution must be computed. By using a different clock frequency and/or sharing the unit across multiple generators they compensate for the infrequent use of this component. This would also allow the efficient integration for Monte-Carlo simulations.

3.2.4. Central Limit Theorem

The central limit theorem (CLT) states that the arithmetic mean of a sufficiently large number of independent and identically distributed random variables (i.i.d.), each with a well-defined expected value and well-defined variance, will be approximately normally distributed. This distribution approaches an ideal bell curve as the number n of added RNs goes to infinity. Both figures in 3.1 represent the PDF of n added URNs ranging from zero to one, with the left-hand distributions unchanged and the right-hand distributions shifted and scaled to have a variance of one.

CLT methods are, however, seldom used for applications in need of high precision normal distributions as a large n value is required to achieve a good bell curve approximation. This can be seen in the right-hand figure in 3.1, where the higher error in the tail region of CLT distributions with a small n can be seen. Small n values additionally result in a limited distribution range. It is, however, used in combination with other methods like error correction as proposed by Malik et al. [31].

They derived the following formula for the variance of a CLT distribution:

$$Var(x) = \sigma_x^2 = \frac{nm^2}{3} \,, \tag{3.14}$$

where ±m is the range of the URNs and n is the number of URNs added. They further describe how the use of specific n values give a distribution variance which is a power of two, transforming a costly scaling multiplication into a simple shift operation in hardware.

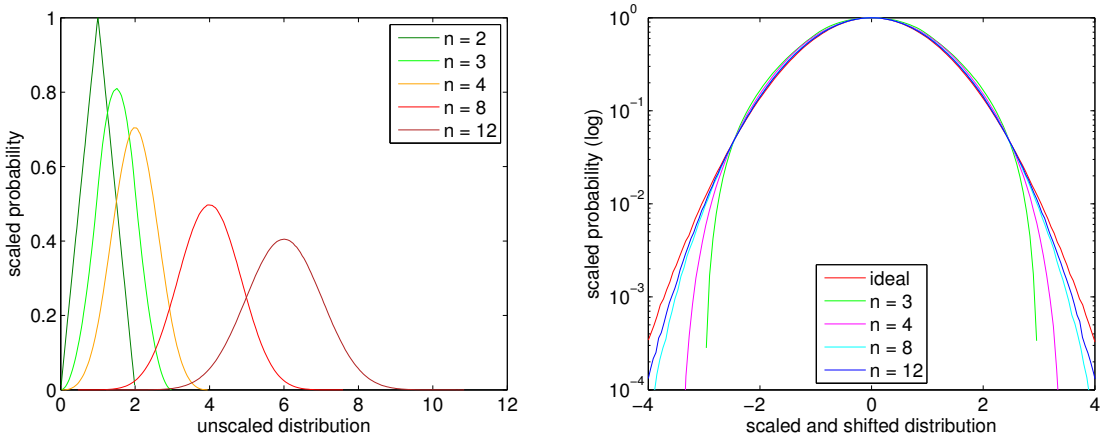


Figure 3.1.: PDF of n added URNs

Starting from the simplest version of a CLT GRNG using n=3 without error correction, multiple Monte-Carlo pricings were performed and their precision analysed. As the Monte-Carlo pricings have no analytical solution, the precision has to be determined by running multiple pricings to find a convergence point for the price. Figure 3.2 shows the average convergence point error and standard deviation of six product pricings using different CLT GRNGs.
Surprisingly the relative convergence point errors were far below the desired one percent and the relative standard deviations are practically the same for all n values. A reasonable explanation for this behavior is the fact that a CLT GRNG has mainly an error in the tail region of its distribution. In the case of the Monte-Carlo pricing of a worst-of barrier option this only amounts to errors in the GBM for very rare large movements during a single iteration. It can therefore be assumed that a small error in the tail region of the GRNG's distribution is negligible for the pricing of worst-of barrier options.

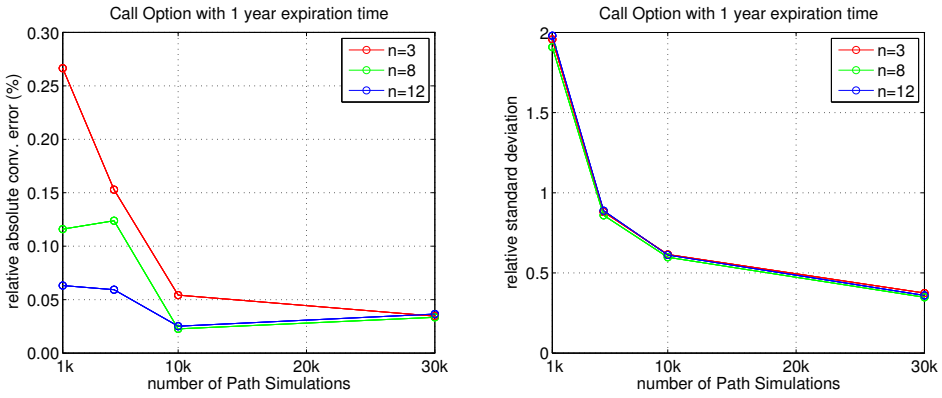


Figure 3.2.: Convergence error and standard deviation of CLT GRNGs with order 3, 8 and 12.

As the simplest CLT GRNG using only three RNs already fulfilled the required precision, it was chosen to be used in the architecture. It has the additional benefit that the GRNs generated have a variance of one and do therefore not require any scaling, which would result in a hardware costly multiplication operation.

3.3. Uniform Random Number Generator (URNG)

This section describes the two URNGs considered for use in the GRNG and the reasons for choosing the latter.

3.3.1. Linear Feedback Shift Registers (LFSR)

The first URNG considered was a LFSR, which is a shift register whose input bit is a linear function, usually an exclusive-or (XOR), of single bits of its previous state. They are often used to generate URNs in hardware, as they provide acceptable randomness at moderate hardware cost. The initial state of the generator is called seed. As the operation on the register is deterministic, the complete sequence generated by an LFSR is determined by the current state, an LFSR used as an URNG therefore generates pseudorandom numbers. As such the sequence repeats after a certain number of steps, called periodicity, which has a maximum possible value of 2^N-1, where N is the bit length of the LFSR. To achieve the maximum periodicity it is necessary to use an irreducible polynomial over a finite field of order 2, which is achieved by using xor operations on single bits of the shift register. An extensive list of irreducible polynomials is given by [32]. Figure 3.3 shows an example of an LFSR with bit width 8, with the irreducible polynomial $x^8 + x^6 + x^5 + x^4 + 1$ resulting in a maximum periodicity of $2^8 - 1$.

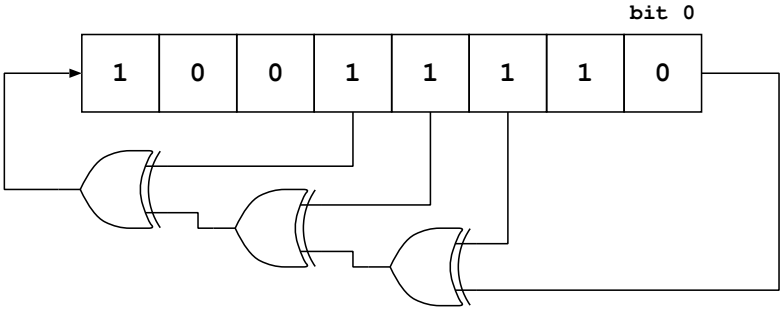


Figure 3.3.: Example of a 8 bit LFSR with maximum periodicity $2^8 - 1$ and irreducible polynomial $x^8 + x^6 + x^5 + x^4 + 1$.

As previous diehard tests with an 32-bit LFSR performed by Greisen [33] have shown to fail, this URNG was dismissed and the more sophisticated URNG, presented in the next section, was considered.

3.3.2. Combined Tausworthe Generator

The generator proposed by Tausworthe [34] generates pseudorandom numbers using a linear recurrence modulo 2 of the following form:

$$x_n = a_1 x_{n-1} + ... + a_k x_{n-k} \; mod \; 2, \quad x_i, a_i \in \{0, 1\} \tag{3.15}$$

which has the characteristic polynomial [35]:

$$P(z) = z^k - a_1 z^{k-1} - ... - a_k \tag{3.16}$$

Fractional numbers from zero to one are then formed by taking blocks of the generated bit sequence:

$$u_n = \sum_{i=1}^{L} x_{ns+i-1} 2^{-i} \tag{3.17}$$

where L is the word-length and s the shift value. An example of this principle can be seen in figure 3.4. This generator has a maximum possible periodicity of $2^k - 1$ if the characteristic polynomial P is primitive, $s_0 \neq 0$, and s is coprime to p.

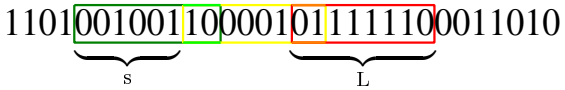


Figure 3.4.: Example showing three numbers generated by a Tausworthe generator using word-length $L = 8$ and a shift value $s = 6$.

The direct use of this generator would, however, require to perform s steps for each random number u_n. L'Ecuyer [35] proposed the use of recurrences with a primitive trinomial as characteristic polynomial:

$$P(z) = z^k - z^q - 1 \tag{3.18}$$

with $0 < 2q < k, 0 < s \leq k - q < k \leq L,\ \gcd(s, 2^k - 1) = 1$.

A generator of this form can be realized by the logic shown in figure 3.5. It has, however, two flaws. First, the characteristic polynomial has few nonzero coefficients which leads to statistical defects and second, the periodicity cannot exceed 2^L, which, depending on the word-length, might lead to repetitions of the same sequences during Monte-Carlo simulations.

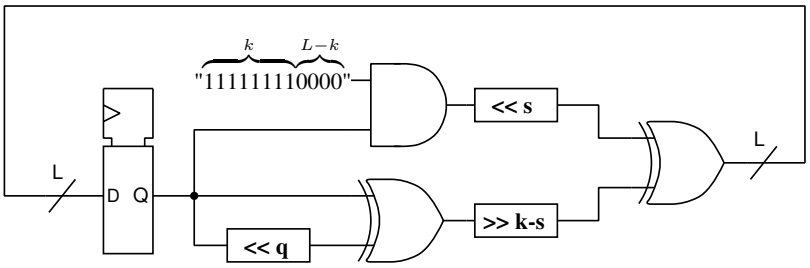


Figure 3.5.: General form of a Tausworthe generator using a primitive trinomial.

L'Ecuyer [35] has shown how the combination of the output of three such Tausworthe generators with a three-way XOR gate solves these problems. Figure 3.6 shows such a combined Tausworthe generator. It is also the one used in the architecture. The combined characteristic polynomial now takes the form:

$$P_{comb}(z) = P_0(z) \cdot P_1(z) \cdot P_2(z) \tag{3.19}$$

where $P_i(z)$ is the characteristic polynomial of a single Tausworthe generator. It has now many nonzero coefficients, eliminating the first flaw. The second flaw of low periodicity is solved by using Tausworthe generators of different periodicity. Through this the periodicity of the combined Tausworthe generator becomes the product of the periodicities of the used generators. In the shown example this gives a periodicity of 2^{88}.

3.4. Pricing

Figure 3.7 depicts the process flowchart of a single path simulation. When a new path simulation starts, the current prices of all underlying assets are set to one, representing their relative initial value. A loop is then started which, if not interrupted, is repeated

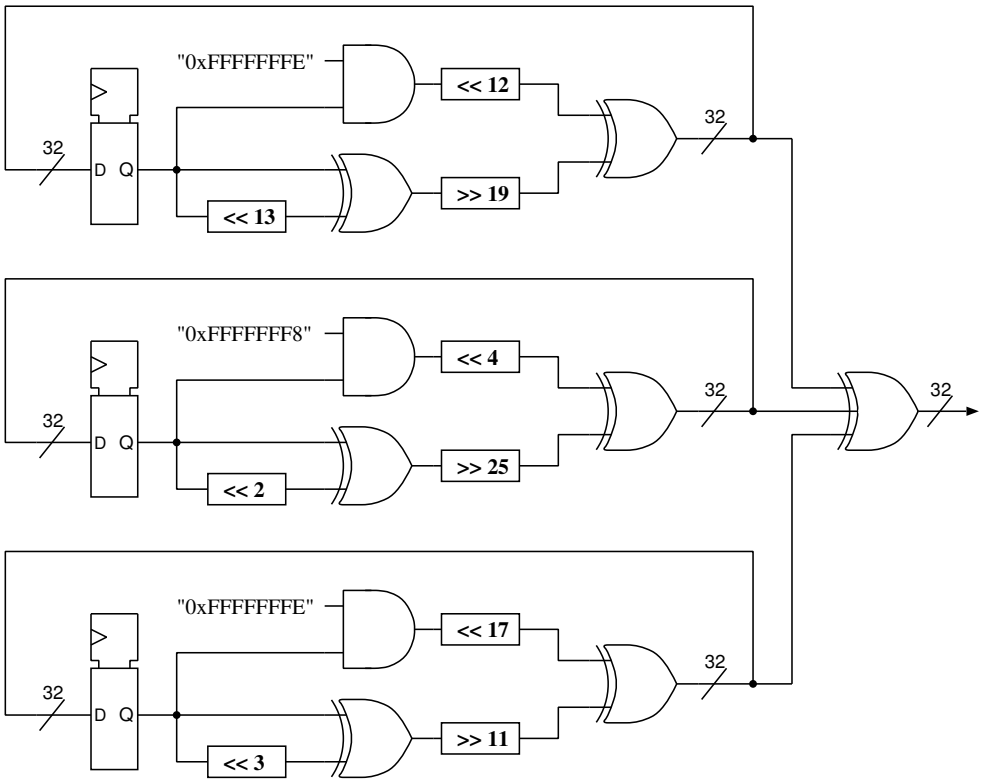


Figure 3.6.: Combined Tausworthe generator with periodicity 2^{88} and bit width 32.

until the product reaches maturity. Depending on Δt and the time to maturity the number of steps which have to be performed is:

$$N = \frac{\text{time to maturity}}{\Delta t} \tag{3.20}$$

As N has to be an integer, Δt is determined from of the predefined time to maturity and number of steps and not the other way round.

This iteration loop is, however, immediately stopped if at any iteration step any price of an asset falls on or below the predefined barrier level. When this happens a break signal is sent out which represents a zero payoff for the current path simulation and a new simulation is started. If this doesn't happen another check is performed which tests whether the price of the worst performing asset lies on or below the predefined strike price. If this is the case, again a break signal is sent out again representing a zero payoff. If this is not the case the the simulated product has a payoff which is then sent out for further processing.

To determine the actual price of the product the mean payoff of all simulated paths, including zero payoffs, has to be taken.

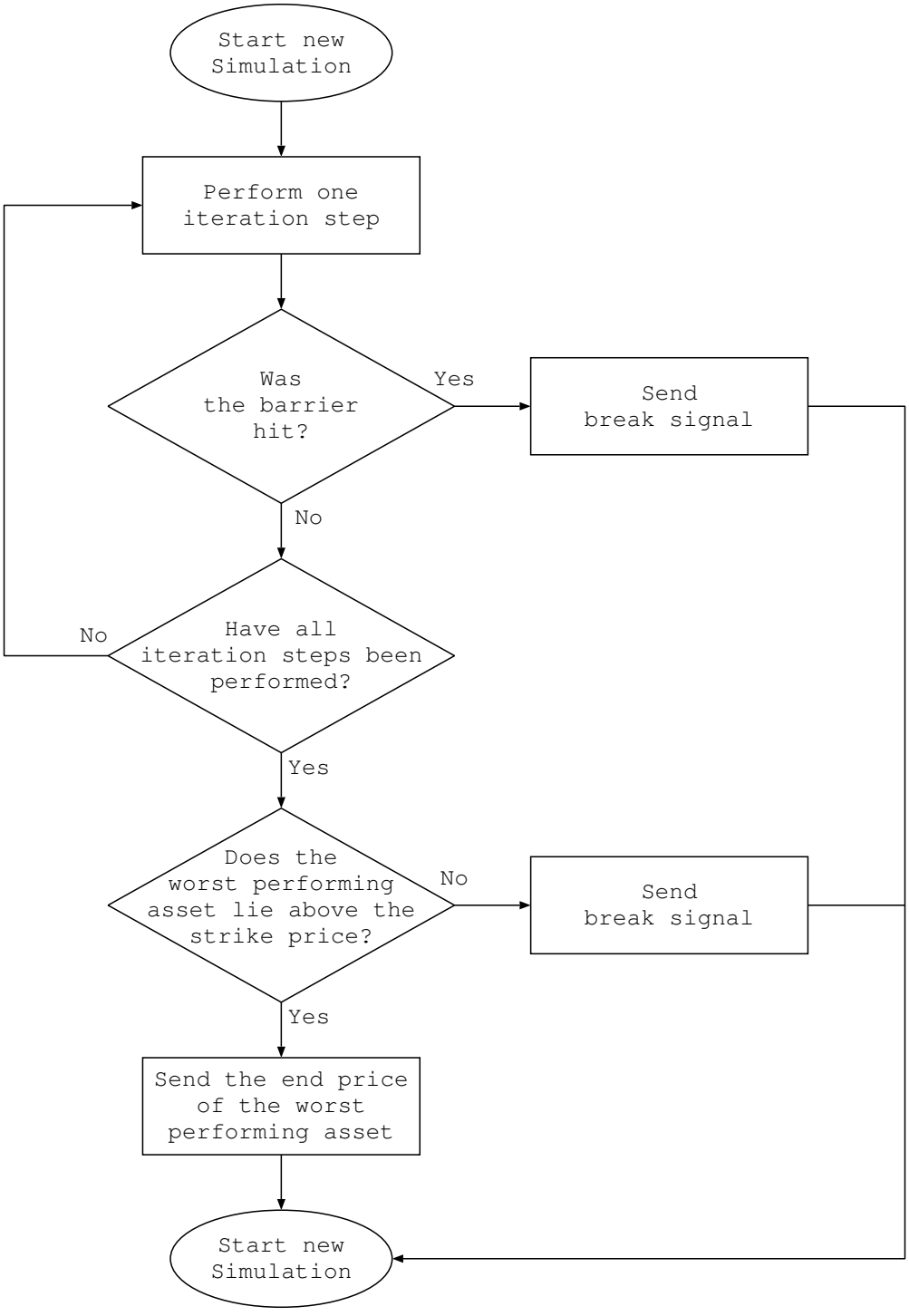


Figure 3.7.: Flowchart of the product simulation

3.5. Summary

The model chosen to simulate the underlying assets is the Black-Scholes model, as it facilitates the evaluation of the GRNG by its simpler calibration than more sophisticated models. It was implemented as a multivariate geometric Brownian motion.

As simulations of the pricing engine had shown that a generator using the central limit theorem with three added URNs already fulfilled the required pricing precision, this very slim GRNG was chosen. To generate the necessary URNs a combined Tausworthe generator built of three Tausworthe generators was chosen as it has good statistical properties and a very high periodicity.

Chapter 4

Hardware Architecture

This chapter describes the individual components used in the architecture on a detailed level. A simplified overview of the architecture is given in figure 4.1, for a more detailed block diagram refer to appendix B. The complete design was implemented on a Xilinx Zynq-7020 all processing system on a chip (SoC) with a developing environment featuring a programmable logic (PL) and a processing system (PS).

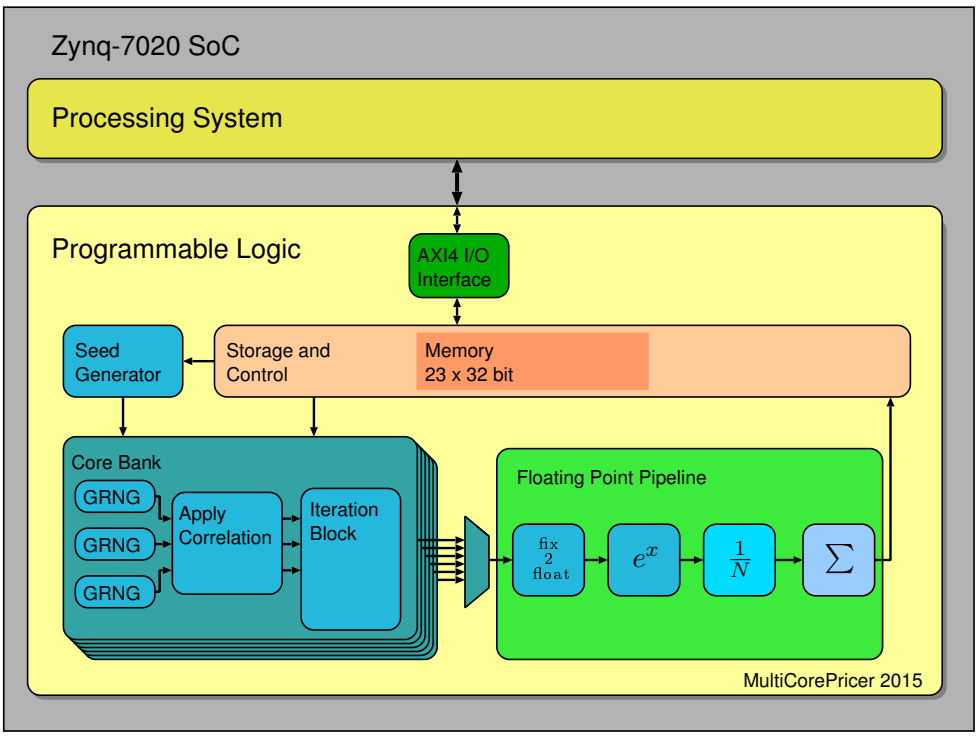


Figure 4.1.: Simplified design overview

4.1. Programmable Logic

The programmable logic contains the pricing engine which performs the Monte-Carlo pricing described in 3.4 and runs autonomous after the model parameters and start signal have been sent by the processing system. This section gives a detailed description of its components, along with their design considerations.

4.1.1. I/O Interface and Storage

To handle communication between the PS and PL an AXI4-Lite protocol is used with a word width of 32-bit. The protocol was chosen as it is a more area-efficient subset of the well established and portable AXI4 protocol, which in contrast to the latter sends only one data word per transaction. The AXI4-Lite is a better fit as there is only data transfer at the start and end of a pricing. Another reason is that, after the initial setup, most of the time only one parameter has to be changed for each new pricing.

The implemented storage, necessary to hold all pricing parameters, consists of a RAM of 23 32-bit words (92 bytes) with the following memory distribution:

- `36B`: 9 starting seeds for the seed generator
- `44B`: constant model parameters
- `4B`: request and acknowledge between prog. system and prog. logic
- `8B`: output value

More details on the I/O interface and storage are presented in appendix A.

4.1.2. Core

The Core block is the main building block of the architecture, with the full design containing 26 instantiations. Figure 4.2 shows a simplified block diagram of a single core. Each block generates a potential price for the defined product by simulating its evolution using the iteration formula derived from the GBM model. As shown in chapter 3, each simulation step of three correlated underlyings requires the generation of three GRNs and correlating them.

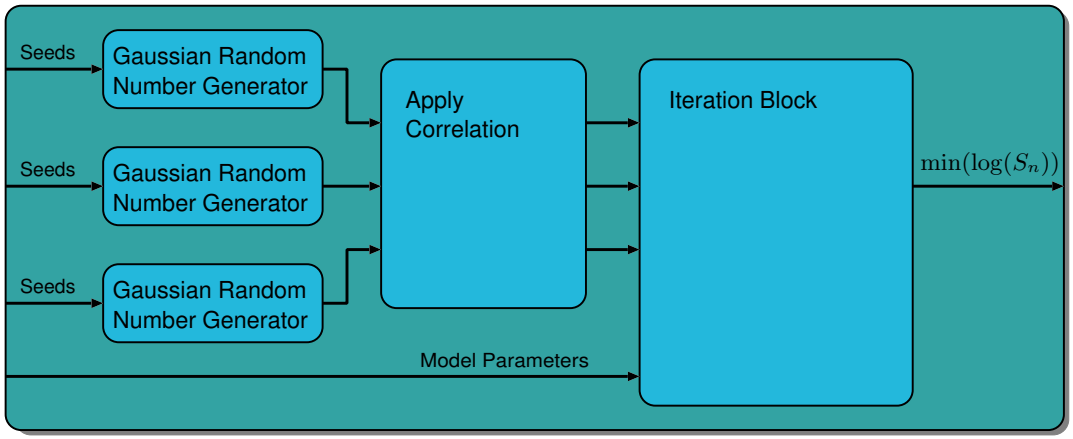


Figure 4.2.: Simplified Core block diagram

Bit Widths

To determine the ideal bit widths of all signals, various tests have been performed where exterior or interior signals where quantized to certain bit widths. Tests were first performed for different bit widths of the GRNs. To evaluate the precision, the convergence point of six different worst-of barrier options were determined by running a high number of pricings with the corresponding number of simulated paths per pricing. The mean value of these pricings was then taken as the ground truth and the same was done using quantized GRNs. For all quantizations, three integer and one sign bit were used, the number of fractional bits is therefore the bit width minus four.

Figure 4.3 shows the convergence error for different GRN quantizations and bit widths. In the left-hand plot it can be seen that the convergence errors for different bit widths lie very close to each other. A slightly larger error can be seen for the 8 bit quantized GRN, having only four fractional bits, implying that the low resolution in the Gaussian distribution only has a small impact on the precision. The right-hand plot shows the relative standard deviation of the pricing for different numbers of simulated paths and quantization. Here a perfect overlap occurs, showing that the standard deviation of the pricing engine does not depend on the quantization of the GRNs.

The GRNs were the only signals whose bit widths were determined by statistical evaluation. All other bit widths were determined by performing single path simulations with quantized parameters, at first with zero drift, then with zero diffusion, and then with nonzero parameters. For the drift part an acceptable error of 0.01% was pursued and achieved, to have leeway to examine the GRNG's impact on precision.

A list of the input signals and bit widths can be found in appendix A. The signal bit widths inside a core block can be seen in figure 4.6.

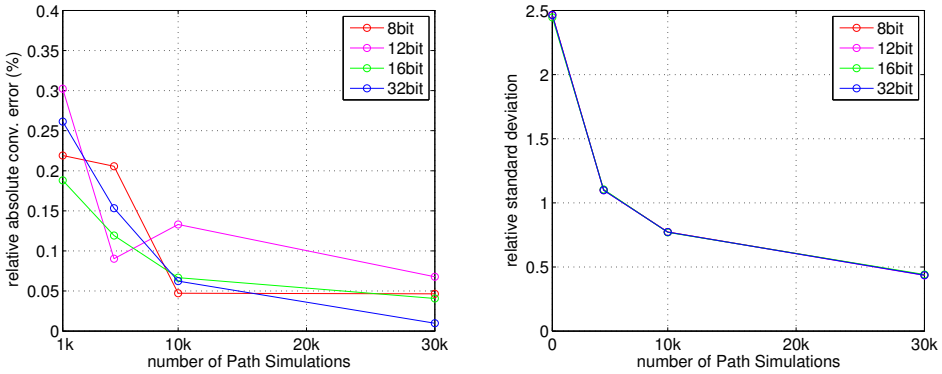


Figure 4.3.: Mean convergence error of six products using different GRN bit widths

4.2. Combined Tausworthe Generator

Every core contains nine Tausworthe generators as URNGs, three for each GRNG. Figure 4.4 shows the implemented combined Tausworthe generator used for generating the URNs. As described in chapter 3 the combined Tausworthe generator consists of three Tausworthe generators and therefore needs three 32-bit seeds to start.

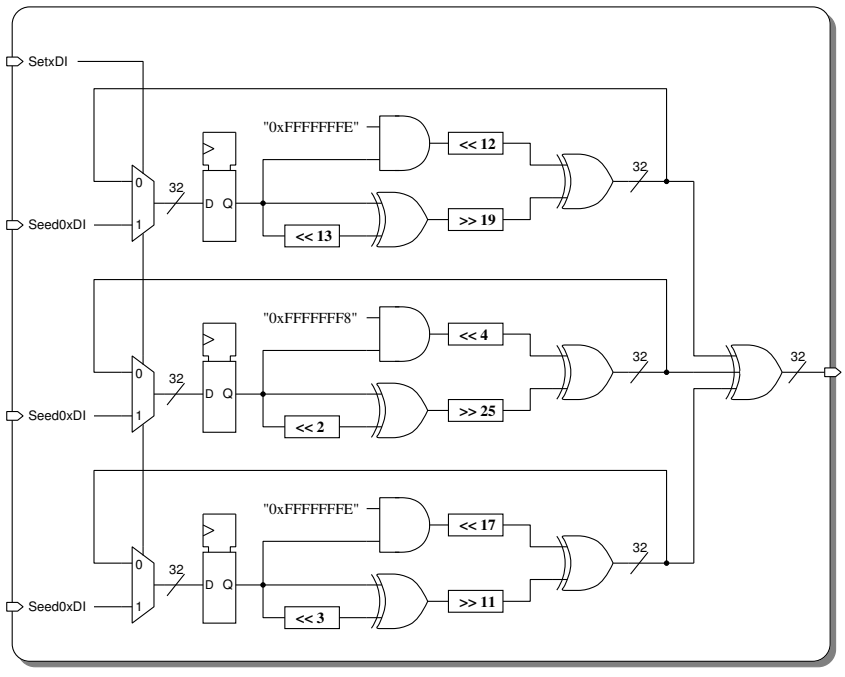


Figure 4.4.: Combined Tausworthe generator

CLT Gaussian Random Number Generator

Every core contains three CLT GRNGs, one for each asset price simulation. As explained in chapter 3 the GRNG works by adding three uniform RNs generated by the combined Tausworthe generators. The implementation of the GRNG can be seen in figure 4.5.

To have a mean value of zero it is necessary to subtract the maximally possible value divided by two. In the case of n=3 the maximum value is a odd number making it necessary to append a fractional bit. In the case of the topmost Tausworthe generator in figure 4.5 this requires the addition of three bits, one because the subtracted value is 1.5 times the value 13 bits can hold, one for the additional fractional bit and one for sign. Since the output of the upper adder is always negative, the bit width isn't increased over the last adder. To have unit variance, the output has to be interpreted as having two integer bits, 13 fractional bits and a sign bit, this gives the GRNG a range from -2.9996337890625 to 2.9996337890625.

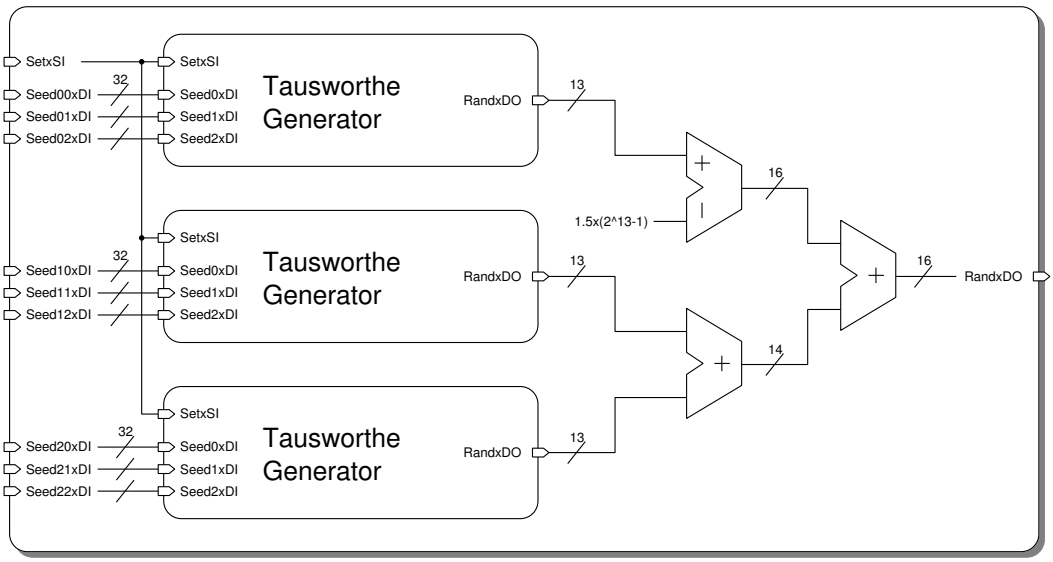


Figure 4.5.: Central Limit Theorem GRNG using 3 additions

Correlator

The correlator block performs the correlation of the GRNs by applying the Cholesky matrix to the three GRNs, as shown in chapter 3, which, for the case of three underlying assets, results in five multiplications.

Iteration Block

The iteration block simulates the hypothetical evolution of the defined product by performing the predefined number of iteration steps of the GBM model. As shown in 3.3, the exponential's input can be split into a drift and a diffusion part. Substituting the Wiener process increment W_n for $\sqrt{\Delta t}\,\varepsilon_n$, where ε_n is a correlated GRN, the iteration formula then becomes:

$$S_n = S_{n-1} \exp\left(\left(\mu - \tfrac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\,\varepsilon_n\right) \tag{4.1}$$

Implementing this formula would however be very hardware costly and slow as for each iteration step a multiplication has to be performed. To avoid this, the whole model simulation is performed using the logarithm of the formula. An iteration step is then defined by the following formula:

$$\ln(S_n) = \ln(S_{n-1}) + \underbrace{(\mu - \tfrac{\sigma^2}{2}\Delta t)}_{drift} + \underbrace{(\sigma\sqrt{\Delta t}\,\varepsilon_n)}_{diffusion} \tag{4.2}$$

With the exception of ε_n, which changes every clock-cycle, all formula variables are constant. One iteration step now only consists of two additions and one multiplication. Since all asset prices are computed relative to their initial prices, all $\ln(S_0)$ values can be set to zero representing a price of one or 100%.

Figure 4.6 shows the block diagram of the iteration block. The iteration block consists of three iteration cells each consisting of a multiplier and two adders, a finite state machine denoted as FSM and logic for testing for the payout scenarios described in chapter 3. The FSM is responsible for checking whether a barrier event has occurred, all iteration steps have been performed and handling the corresponding request and acknowledge signals, as well as sending the worst performing asset's price to the floating point pipeline if its end price lies above the strike price. After receiving an acknowledge signal the FSM starts a new simulation.

The number of iteration steps is defined by the input signal NStpxDI. The SetxSI signal informs the FSM that all seeds of the corresponding GRNG have been set and an initialization count is started which tells the FSM when the first GRNs arrive. Finally there is the RestartxSI signal which let's the core immediately start a new simulation. This functionality is needed to start a complete new product pricing and is normally sent after a new parameter set has been written to the programmable logic.

4.2.1. Controller

The Controller block initializes the cores after all input parameters have been written and is responsible for handling requests and acknowledges from all cores. Break requests (signaling a barrier event) and normal requests (sent with a simulation price) are handled

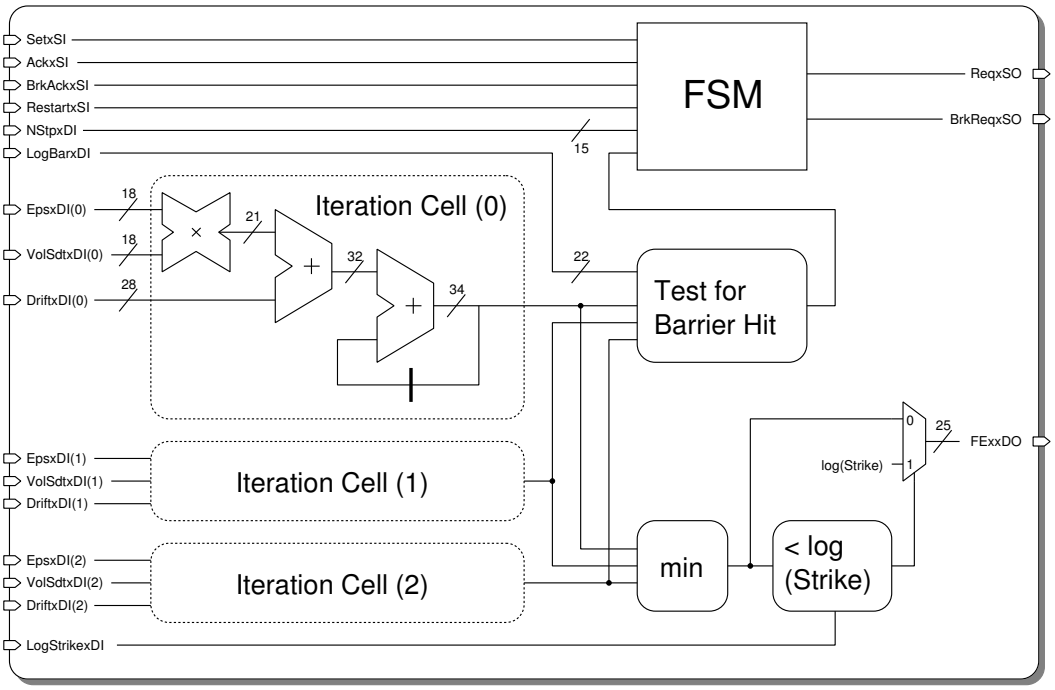


Figure 4.6.: Iteration Block

separately with their own counter. Both request types are added up to have a total simulation count. Break requests simply increase the break counter whereas a normal request gets the simulated price sent to the floating-point pipeline. If the total simulation count reaches the predefined number of path simulations, the controller block writes the result and output request into the corresponding registers.

Seed generator

Simulations of the pricing engine, using the combined Tausworthe generators, showed that the choice of starting seeds for each generator is of significant importance for the performance of the Monte-Carlo pricing. The first attempt of simply increasing the starting seeds of each generator by a fixed number showed a very poor performance for small increments with the price converging to a completely incorrect value. Larger increments of each seed returned better results but were still not satisfying. It was therefore deemed necessary to develop a more sophisticated method.

To keep the number of starting seeds low and save startup time, an onboard seed generation scheme was chosen. Instead of using a complete new URNG, the existing one was modified. Three equal URNGs were stacked similar to the GRNG block, therefore making nine starting seeds necessary. After the first URNs are generated, instead of continuing the sequence, two new seeds are introduced from the URNs generated by the

two other URNGs for each URNG. A more easily understandable block diagram of the seed generator can be seen in figure 4.7.

Global Counters

To check whether the predefined number of path simulations has been performed, two global counters are implemented in the controller. A break counter which counts the paths stopped by barrier events and worst performing assets ending below the strike price, and an add counter which counts the number of added paths. The count of both is summed up and checked against the requested number of path simulations. When the total count surpasses it, an additional 45 clock cycles are run to make sure that the floating point pipeline is flushed out. This is necessary because both the break and add count are increased when a core sends the corresponding request, which means that the sent price from a core has to go through the floating point pipeline before the final price is updated.

Floating Point Pipeline

The floating-point pipeline consists of three Xilinx floating-point IP blocks. The first block computes the exponential function and has a single-precision floating point (SP-FP) number as input and output. As all simulation prices up to this point have been computed as the logarithm of the price it is necessary to compute the exponential function of them. The Xilinx floating-point IP represents a slim and fast solution that doesn't occupy any costly DSP slices of the FPGA. Logic before the first block converts the price from a fixed-point number to a SP-FP number. The mantissa of 23 bits is a more than sufficiently high precision to store the computed simulation prices.

The second block consists of a SP-FP subtracter which is used to subtract the strike price, as the payoff is the difference of the end price to the strike price.

The third and last block is a double-precision FP adder which sequentially sums up all weighted prices. Since the number of simulated paths is always set to a power of two, the division of a simulation price by the total number of simulations is reduced to a decrease of the exponent of the FP number. This subtraction and the conversion to a DP-FP number are performed through logic between the second and third block. The conversion to a DP-FP number is necessary as otherwise almost all fractional information of a simulated path's price would be lost during addition for a high number of simulated paths ($2^{20} \approx$ 1e6).

4.3. Processing System

The processing system runs a C program with pricing parameters (model simulation parameters and product information) which are written to its header file using a Matlab script. The program handles the input and output to the programmable logic and sends the pricing results to the host computer via UART after all parameter sets have been priced.

4.3.1. Batch Pricing

To reproduce a practical application, two parameter change functionalities have been implemented. The first changes all pricing parameters, allowing the reconfiguration to a different asset set, barrier level, strike price etc. The second changes only one parameter, the barrier level. Using the second method of only changing one parameter saves writing cycles and therefore time. The barrier level parameter was chosen as it is, besides time to maturity and strike price, a parameter which is defined by the issuer, with all other parameters being derived from historical data. Other single parameter change functions, including parameters derived from historical data, could be easily implemented. Such single parameter change functions are useful to compute the so called "Greeks". These are quantities representing the sensitivity of the price to different parameter fluctuations as volatility, time, risk-free rate and others.

4.3.2. Pricing Flow

After programming the PS and PL, the following steps are performed by the PS:

- Write starting seeds to PL
- Write pricing parameters to PL
- Write start signal to PL
- Poll the result ready register until it is one
- Read the result register
- Loop for the number of parameter sets -1:
  - Write new parameter set
  - Poll the result ready register until it is one
  - Read result
- Return prices to host computer

The initial writing process is broken into two stages, as the PL starts generating seeds for the RNGs as soon as all starting seeds have been written. The pricing parameters are distributed over multiple input words as all model parameters have different optimized bit widths. Additionally the last bit of the last written input word acts as a ready bit, signaling the PL that all pricing parameters have been written and that it can start the pricing.

After finishing pricing, the PL writes a result ready signal to a designated communication register which is being polled by the PS. If other parameter sets are still left, the PS writes a single or multiple parameters to the corresponding registers and an acknowledge signal to the communication register, acting as a starting signal for the PL to commence a new pricing.

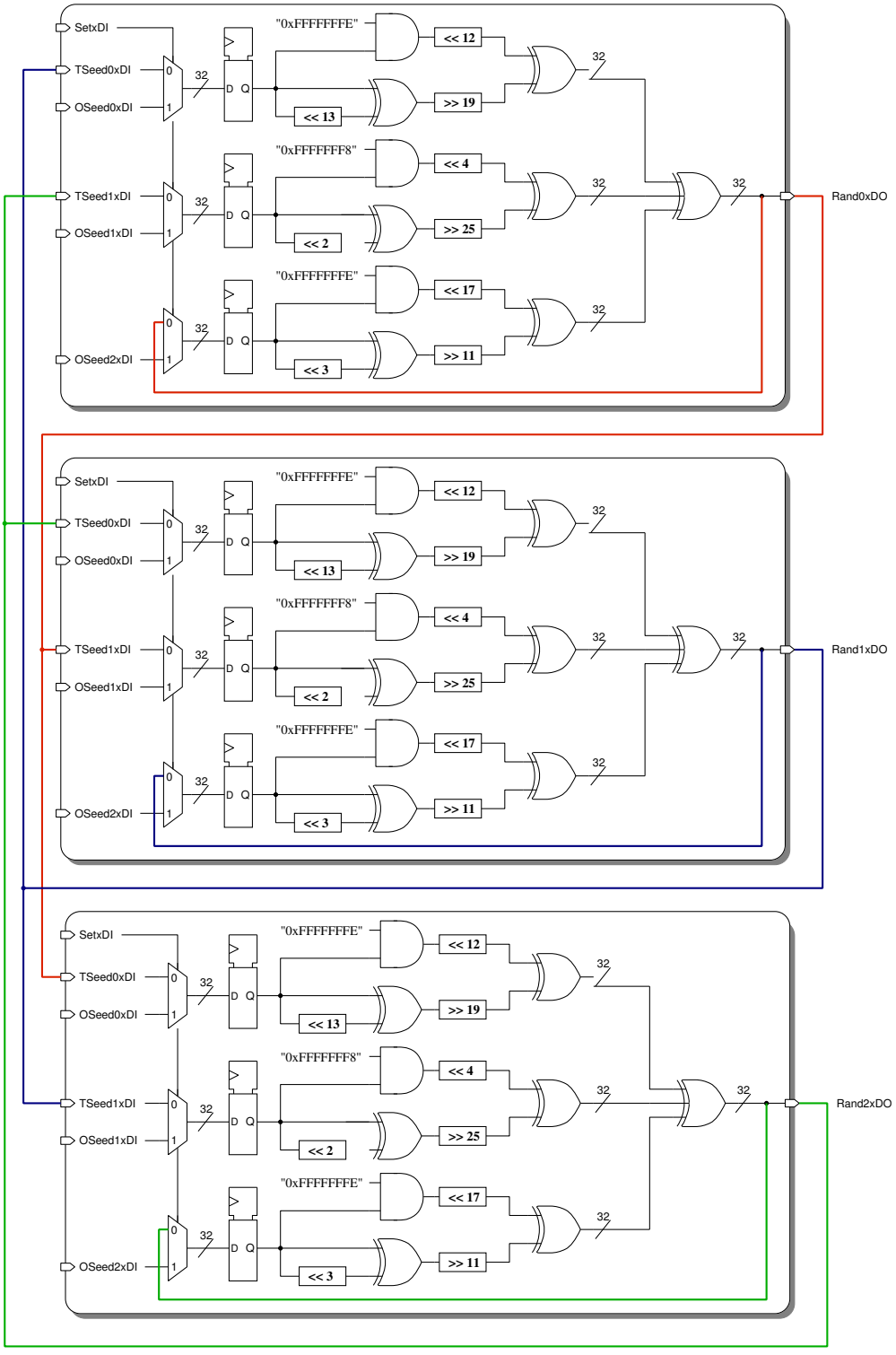


Figure 4.7.: Seed generator built out of three combined Tausworthe generators

Chapter 5

Results

In this chapter the project results are presented. This includes a hardware utilization analysis, the methods chosen to test precision and speed-up as well as a listing of the achieved results.

5.1. Utilization

The design presented in this thesis was implemented on a Zynq-7020 SoC using the Xilinx Vivado Design Suite v2014.1 with synthesis optimized for speed. Table 5.1 shows the number and percentage of resources used by different components. Looking at the top figure for the full architecture it can be seen that the maximum number of cores is limited by the available DSP units on the Zynq-7020. Since the DSP slices of each core are subdivided to the Correlator and the Iteration Block no absolute number of DSP usage can be given for these components. Considering that the correlator performs five multiplication operations and the iteration block only one, though with a higher bit width, it can be assumed that the correlator uses about 80% of the 8 DSP slices. The last row shows the figures for the AXI4-Lite communication block and the control logic from which can be seen that the overhead of hardware not used for pricing is less than 5% of the available hardware resources.

5.2. Precision

To evaluate the precision of the design, the convergence points (CP) of pricings of six different products performed by the FPGA have been compared to the corresponding ground truth prices. The CP of the Monte-Carlo pricings have been determined by simulating a total number of 2^{25} paths for 2^{14} and 2^{17} path pricings. This means the

	DSP48		LUTs		Registers	
	No.	**Pct.**	**No.**	**Pct.**	**No.**	**Pct.**
Full Architecture	211	95.9%	41394	77.8%	41895	39.4%
All Cores (26)	208	94.5%	35663	67.0%	36868	34.7%
Single Core	8	3.6%	1372	2.6%	1418	1.3%
Single GRNG	0	0.0%	298	0.6%	306	0.3%
Correlator			68	0.1%	154	0.1%
Iteration Block			410	0.8%	346	0.3%
Float P. Pipeline	3	1.4%	2212	4.2%	3260	3.1%
Seed Generator	0	0.0%	2302	4.3%	267	0.3%
Comm. & Control	0	0.0%	1217	2.9%	1500	3.6%

Table 5.1.: Hardware usage of different components

average price of 2^{11} pricings using 2^{14} paths and the average price of 2^{7} pricings using 2^{17} paths. The ground truth for the Black-Scholes comparison is the analytical solution computed using Matlab's `blsprice` function. The ground truth for the worst-of barrier options were computed by running a Matlab pricing script for a total of 2^{24} paths, a reduced version of the script can be found in appendix B.

5.2.1. Black-Scholes Comparison

Because of its design the architecture cannot price a single asset call option. To test it against the analytical solutions of Black-Scholes pricings the design had therefore to be modified to ignore the output of the latter two simulated assets and ignore barrier events. The errors of the CPs were computed for pricings using 2^{14} and 2^{17} simulated paths. The simulation parameters were computed using selected historical data of the SIX Swiss Exchange from 2010 to 2014 and a risk-free rate of 2% was used. Strike prices were chosen from a range of 70% to 120% of the initial price. Each product setup was evaluated with six different stocks to derive the average error. For an extensive list of the average error for different product parameters and a list of the stocks used, see appendix C.

Figure 5.1 shows different average errors for pricings using 2^{14} and 2^{17} simulated paths and different strike prices. The left-hand plots show the average absolute error and the right-hand plots show the average error. It can be seen in all figures that there is the tendency of a higher error for higher strike prices. This tendency will also be seen for the worst-of barrier option for high barrier levels in the next section and will be further explained there. Comparing the upper 2^{14} path pricings against the lower 2^{17} path pricings, it can also be seen that this higher error is reduced for pricings using

more simulated paths. As with higher strike prices only a few simulated paths actually contribute to the final pricing, since most end below the strike price and count as zero, this observation points to statistical defects in the combined Tausworthe generator. Shorter option maturity times also lead to fewer simulation paths surpassing the strike price amplifying this effect, as can be seen for the half year maturity call option.

The right-hand plots reveal that there is a tendency for the CP error to be negative. This behavior, if still existent after improvements in the URNG, could be characterized to improve pricing by a post programmable logic correction. The overall average CP error for the Black-Scholes pricing is 0.071%/0.047% for pricings using $2^{14}/2^{17}$ paths.

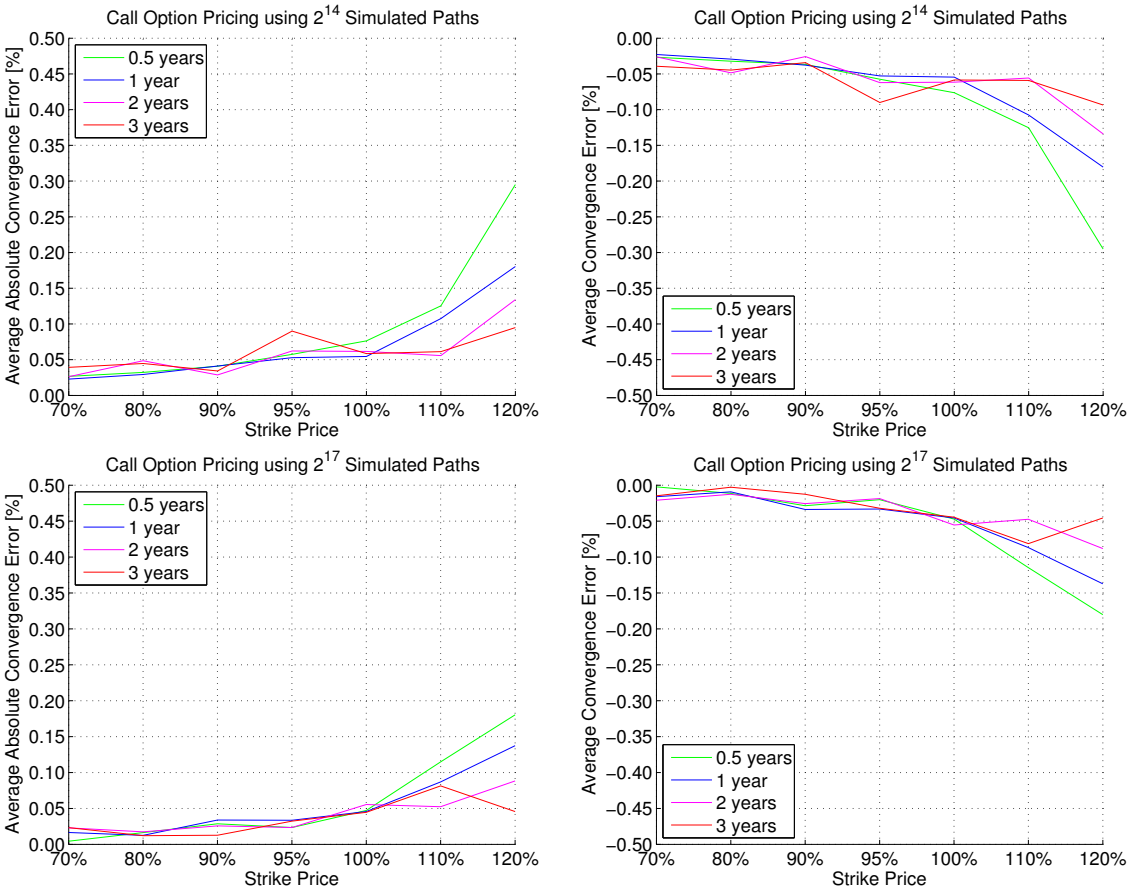


Figure 5.1.: Average absolute error and average error for call options

5.2.2. Worst-of Barrier Call Option Comparison

Figure 5.2 shows different average error plots for pricings using 2^{14}, 2^{17} and 2^{20} simulated paths for different barrier levels. The same historical data was used as with the Black-Scholes pricings. An extensive list of the average errors for different types and a list of the stock-sets used can be found in appendix C.

As with the Black-Scholes pricing there is a higher error for pricings where only few simulation paths contribute to the price as is here the case with higher barrier levels. Contrary to Black-Scholes the CP error is higher for longer maturity times, this is because a longer duration enables more barrier hits reducing the number of simulations contributing to the product price.

Despite the statistical defects of the Tausworthe generator and the use of a CLT Gaussian distribution the standard deviation of pricings performed by the FPGA and software were practically equal. The average standard deviation for different product parameters can be seen in the appendix list. It should first be noted that barrier values above 80% are practically never used in any real products, but are useful to analyze the architecture's behavior, the same can be said about barrier values below 50%. Because of this, they are excluded in the overall precision calculation. Second, for the hypothetical 90% barrier products a very high number of path simulations is necessary to have a low standard deviation over multiple pricings. E.g. the relative standard deviation of a pricing using 2^{17} paths, a 90% barrier and maturity of 2 or 3 years was always above 1%, a multiple of the CP error. This means that pricings using multiple millions of simulated paths would be necessary to have a more approximate CP ground truth and error figure. Taking a realistic range into account, the average CP error for a pricing with a barrier between 50% and 80% and maturity of 1 to 3 years is 0.074%/0.067%/0.059% for pricings using $2^{14}/2^{17}/2^{20}$ paths.

As with the Black-Scholes pricing a certain error characteristic was found for the average error, as can be seen on the right-hand plots. This characteristic, as mentioned in the last section, stems mainly from the Tausworthe generator, leading to the conclusion that shorter sequences of generated random numbers show stronger statistical defects. Software simulations of the model, using seeds generated with Matlab's `randi` function instead of the seed generator, have shown that these statistical defects are caused by the generated seeds. The reason for this is most likely, that the seed generator is a modified version of the combined Tausworthe generator. A completely different uniform random number generator as seed generator or other way of providing seeds could therefore significantly improve precision.

These model simulations have further shown that even when using "ideal" seeds, the GRNG using three summing terms from combined Tausworthe generators, does not achieve the same precision as a GRNG using three Matlab random numbers. This demonstrates that a statistically ideal uniform distribution is highly important in a CLT GRNG.

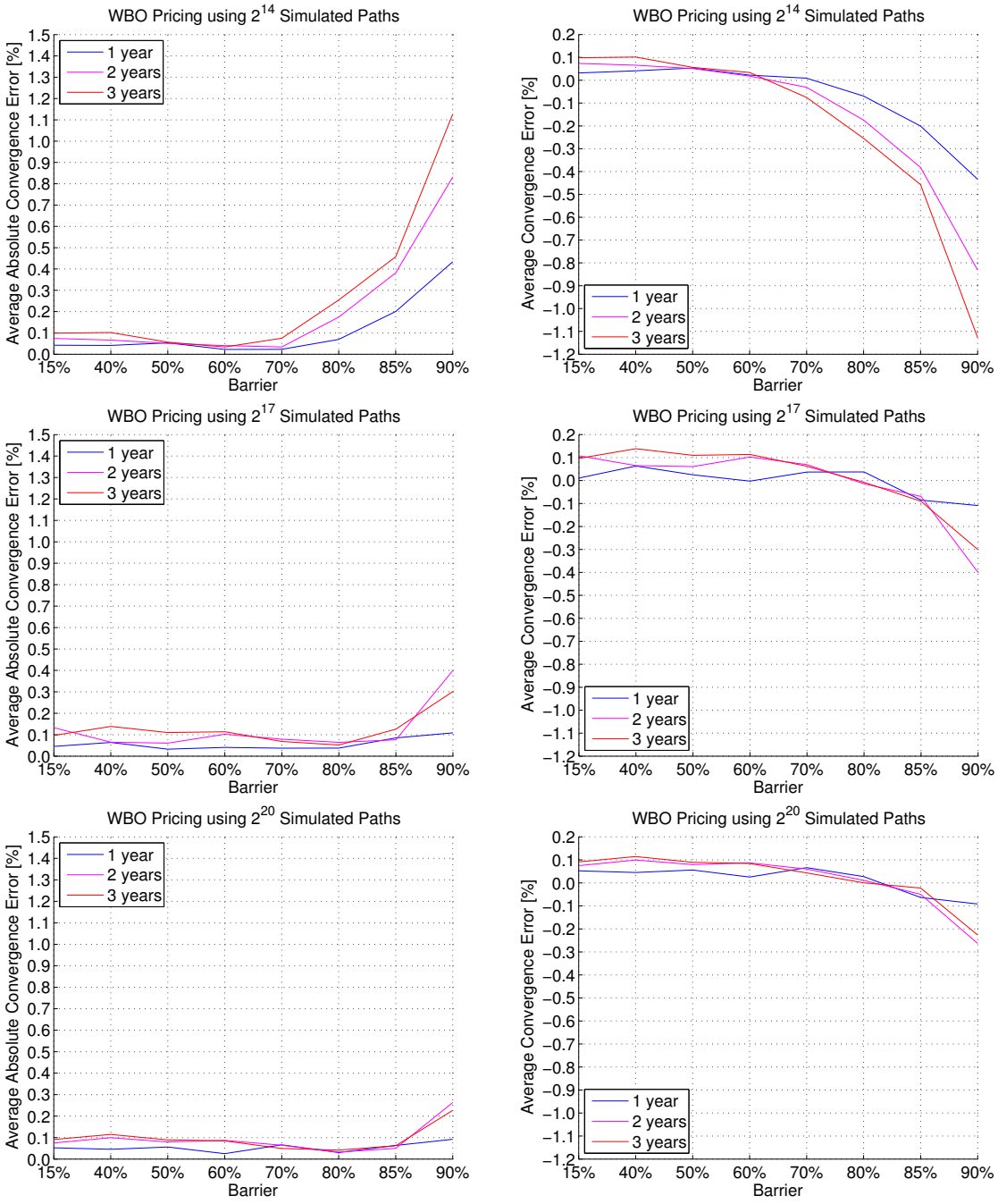


Figure 5.2.: Average absolute error and average error for worst-of barrier call options

5.3. Speed-Up

The speed-up was determined by comparing the time the FPGA needed to price a product with the time needed by the Matlab script running on a single Intel i5-4200U Core running at 1.60GHz with 8GB RAM in Matlab R2013b 64-bit on Ubuntu 12.04. The clock frequency on the FPGA was limited to 100MHz as the input/output pins only supported velocities up to this region. Using a separate clock-domain for the communication block could further increase computation speed.

Table 5.2 shows the average computation times on CPU and FPGA and the resulting speed-ups for a selected group of products. A more extensive list and a list of the stock-sets used can be found in appendix C. The speed-up can be split into two parts, one resulting from the hardware acceleration and one from an optimized path simulation scheme. The first part which is realized by using parallelism and bit width reduction gives a speed up of about 550, which can be seen for very low barrier products in which almost no barrier events occur. The second part is the optimized path simulation scheme which wastes no time continuing the path simulation after a barrier event has occurred and immediately starts a new path simulation. This gives for high barrier products of 80%, speed-ups ranging from 850 to 1450. Summarizing, the pricing of a three year product using 2^{14} simulated paths requires less than 20ms, the pricing of a two year product less than 13ms and the pricing of a one year product less than 7ms.

Depending on the parameter set the resulting pricing has a smaller or bigger standard deviation, it lies therefore in the responsibility of the issuer to choose the number of simulated paths required to determine an "accurate" price. As the computation times scale linearly with the number of simulated paths, they can readily be multiplied by a factor to determine the computation time for other numbers of simulated paths.

Barrier Price	Time to Maturity	CPU Comp. Time	FPGA Comp. Time	Speed-up
40%	1 year	3.5s	6.4ms	550
60%	1 year	3.5s	6.1ms	574
80%	1 year	3.5s	4.1ms	852
40%	2 years	6.9s	12.5ms	554
60%	2 years	6.9s	10.9ms	638
80%	2 years	6.9s	6.0ms	1171
40%	3 years	10.4s	18.3ms	568
60%	3 years	10.4s	14.6ms	713
80%	3 years	10.4s	7.2ms	1462

Table 5.2.: Speed-ups and computation times for pricings using 2^{14} simulated paths

Chapter 6

Conclusion & Outlook

This project has shown that Gaussian distributions generated using the central limit theorem are well suited for Monte-Carlo pricing of multi-asset financial products. Even with the smallest necessary number of three addition terms a very accurate pricing is possible reducing the necessary hardware space considerably.

The used AXI4-Lite communication protocol allows for seamless implementation in other systems and the introduced parameter/result storage enables batch pricing of multiple products with the possibility of only changing a single parameter for a more efficient processing flow.

The implemented combined Tausworthe URNG has proven to be a fast solution which requires a sophisticated seed selection, as using poorly chosen seeds leads to strong statistical defects in the CLT GRNG. Using the presented seed generator these statistical defects could be reduced but not completely eliminated as software simulations have shown.

The use of independently running cores being handled by a single controlling block and using a single floating-point pipeline allows for an easy scaling of the design onto larger FPGAs and the code versatility makes it easy to manipulate signal bit widths within the architecture.

The desired precision of a relative convergence point error of 1% or smaller was achieved and could be further improved, foremost by eliminating the statistical defects of the combined Tausworthe generator, either through better seeds or using another URNG. An increase of the input and internal signal bit widths could additionally improve precision.

The high potential of specialized architectures to accelerate the pricing of exotic financial products has been demonstrated and could be seen especially strong in this product, as it becomes void if the barrier level is touched. This benefit can be seen in the high speed-up achieved especially for high barrier options. The speed-up of 550 without

considering barrier events, is surprisingly high, even when considering the fact that the comparison was made to a single CPU core, as the design was also run on a FPGA board with comparatively small hardware capabilities. Top speed-ups for three year and 80% barrier products went over 1450. Overall the design realizes the pricing of a three year product in less than 20ms, a two year product in less than 13ms and a one year product in less than 7ms.

Outlook

Further development of the architecture presented in this thesis could include the implementation of the Heston model or other more sophisticated models. As the scheduling process is handled by the controller, the exchange of the model would break down to exchanging the iteration block and changing the number of GRNGs. In case of the Heston model with changing volatility three additional GRNGs would be needed for each core.

As the architecture's clock frequency was bottle-necked by the I/O pins to 100MHz, a significant speed-up could simply be achieved by introducing a second clock domain solely for the communication block. The use of the antithetic variates method, which uses for each generated random number sequence also its mirrored negative sequence, could also significantly increase speed. The already correlated GRNs can simply be reversed to gain a second GRNs set, this would effectively half the used area of all the core logic excluding the iteration block. A thorough quantitative analysis on whether this has a negative impact on the Monte-Carlo pricing would have to be performed to confirm its applicability.

Appendix A

MultiCorePricer 2015 Datasheet

Architecture for the Monte-Carlo pricing of worst-of barrier call options, built from three underlying assets, realized on the Xilinx Zynq-7020 SoC.

A.1. Electrical characteristics

Supply voltage	3.3 V
Clock rate	100 MHz
Total On-Chip Power	1.123 W

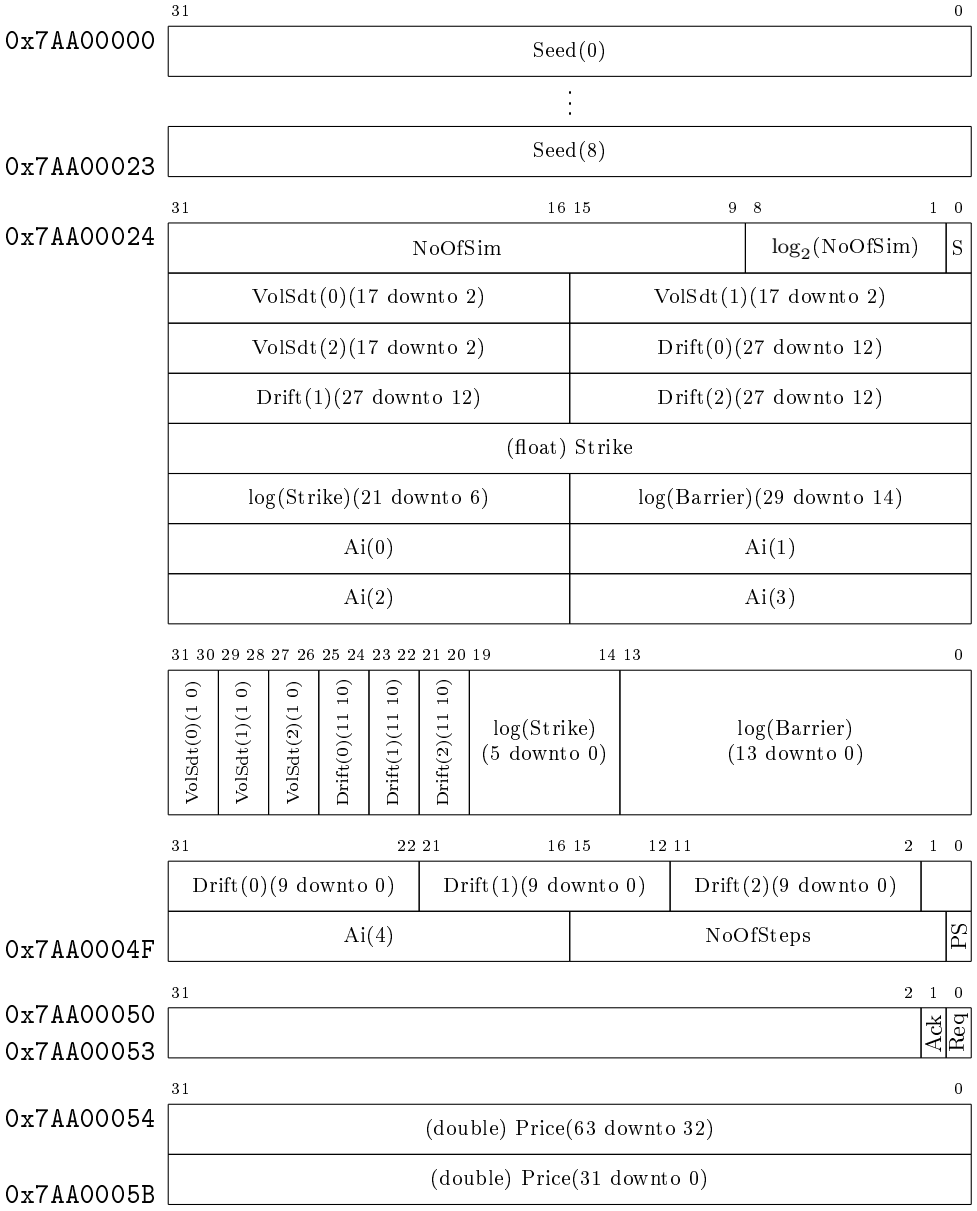
A.2. Applications information

A.2.1. I/O interface

The transfer of data between the processing system and logic is handled using the AXI4-Lite protocol [36].

A.2.2. Storage address map

The I/O interface can be used to directly write to the internal RAM. The address space is structured as shown below:



Data	**Format (int,frac,s/u)**	**Content**
Seed(0-8)	(32,0,u)	32-bit starting seeds for the seed generation
S	(1,0,u)	Bit signaling that all seeds have been written
NoOfSim	(33,-10,u)	Total number of simulated paths (used by the simulation counter)
$\log_2$(NoOfSim)	(8,0,u)	Total Number of simulated paths (used to divide single simulation price)
VolSdt	(-2,20,u)	$= \sigma \cdot \sqrt{\Delta t}$, constant factor in the model
Drift	(-3,30,s)	$= (\mu - \frac{\sigma^2}{2}) \cdot \Delta t$, constant factor in the model
(float) Strike	float	Strike price in single precision floating point format
log(Strike)	(4,17,s)	Natural logarithm of the strike price (used for payoff check)
log(Barrier)	(3,26,s)	Natural logarithm of the barrier level in percent (used for barrier event check)
Ai(0-4)	(0,16,u)	Coefficients of the Cholesky decomposition of the correlation matrix of the assets
NoOfSteps	(15,0,u)	Number of steps per simulation (used by the Iteration Block counter)
PS	(1,0,u)	Bit signaling that all pricing parameters have been written
Ack	(1,0,u)	Bit signaling the PL that the price has been read by the PS
Req	(1,0,u)	Bit signaling the PS that the price can be read
(double) Price	double	Price in double precision floating point format

Table A.1.: List of input signals, their content and format (integer and fractional bit width (negative value means fractional point outside of the word), signed or unsigned)

A.2.3. Basic usage

Before any products can be priced, the necessary header file has to be generated. This is done by defining the following variables in the Matlab script "ParameterSet.m" and running it:

- `assets`: Array containing the address numbers in the Tickerlist, where the historical data of the three underlying assets is stored (can be a matrix containing different asset sets in each row)
- `barrierInPercent`: The barrier value in percent (can be an array with different barrier values)
- `strike`: The strike price as a percentage of the initial price
- `timeToMaturity`: The time to maturity in years (can be a fractional)
- `stepsPerYear`: Number of iteration steps per year (defines the Δt in the model)
- `riskFreeRate`: The risk-free interest rate
- `noOfSim`: The number of simulated paths used to price the product (will be rounded up to the next power of two)

The GenerateHeader.m Matlab script then uses the historical data to compute the volatilities and correlation matrix of the underlying assets. The historical data of the various assets has to be stored in a struct named "Tickerlist" where each asset has the fields "time" and "value" with the corresponding data.

Appendix B

Detailed Block Diagram and Code Overview

B.1. Block diagram

A detailed block diagram of the top level of the PL is presented in figure B.1.

B.2. VHDL code overview

A coarse overview of what is where in the code file hierarchy is given here to aid readers, eager to understand the details of the VHDL code, find what they want.

Top level chip entities	
top.vhd	Toplevel entity with AXI4-Lite interface and RAM

Control	
Controller.vhd	Entity containing all internal components and controlling data flow
Seeder.vhd	Seed generator for all Cores
Core.vhd	Sub-entity of the Controller containing Core components

Seeder.vhd	
SeedTausworthe.vhd	Modified combined Tausworthe generator (figure 4.7)

Core.vhd	
GRNG3.vhd	Gaussian random number generator presented in chapter 4
Correlator.vhd	Applies the Cholesky matrix
IterationBlock.vhd	Simulates the GBM model as explained in chapter 4

GRNG3.vhd	
Tausworthe.vhd	Combined Tausworthe generator (figure 4.4)

Other files	
constants.vhd	Constants used throughout the project
types.vhd	Type declarations used throughout the project
RealARITH.vhd	Arithmetic operations library designed by IIS
top_tb.vhd	Testbench (not necessary for synthesis)
tb_util.vhd	Testbench utilities by IIS (not necessary for synthesis)

B.3. Matlab pricing script

```
%% Input parameters
% s0                : initial price (set to 1 = 100% for all)
% strike            : strike price (as percentage of s0)
% barrier           : barrier level (as percentage of s0)
% timeToMaturity    : product duration in years
% rf                : risk-free rate
% corrMat           : correlation Matrix of the underlyings
%                     = (corr(price2ret(timeSeriesPrice)))
% vol               : volatility of the underlyings
%                     = std(price2ret(timeSeriesPrice),1)*sqrt(255)
% noOfSim           : number of paths to simulate
% noOfSteps         : number of steps per simulation
function[optionPrice] = multiAssetBarrierAny(s0,strike,barrier,...
  timeToMaturity,rf,corrMat,vol,noOfSim,noOfSteps)


%% Prellocation

  % allocate variables
  optionPrice = [];

  % specify precision
  Spec.precision = 'double';


%% Preprocessing

  deltaT = timeToMaturity/noOfSteps;

  % Calculate Chol Decomposition
  colDecCorr = sqrtm(corrMat);


%% Calculate independent normal distribution path

  noOfAssets = size(s0, 1);

  randomSamples = zeros(noOfAssets, noOfSteps, noOfSim, Spec.precision);

  for ii = 1:noOfAssets
    randomSamples(ii,:,:) = randn(noOfSim, noOfSteps)';
  end


%% Apply correlation matrix to independent simulations

  for ii = 1 : noOfSim
    randomSamples(:,:,ii) = reshape(colDecCorr*randomSamples(:,:,ii),...
      [noOfAssets,noOfSteps]);
  end
```

```
%% Build price pathes

  % calculate drift matrix
  driftMatrix = repmat(exp((rf-1/2*vol.^2)*deltaT),[1,noOfSteps]);

  % calculate diffusion matrix
  diffusionMatrix = exp(repmat(vol,[1,noOfSteps, noOfSim]).*...
    randomSamples*sqrt(deltaT));

  % calulate sample paths
  pathMatrix = repmat(s0,[1,noOfSteps,noOfSim]).*...
    cumprod(repmat(driftMatrix,[1,1,noOfSim]).* diffusionMatrix,2);


%% Calculate Option Price

  % down option, check barrier hit
  if any( any(s0 <= repmat(barrier,[1,size(s0,2)]),1) )
    index = true(1,T,NoOfSim);
  else
    index = pathMatrix <= repmat(barrier,[1,noOfSteps,noOfSim]);
  end

  % out option
  indexBH = ~any( any(index,2),1 );
  sum(sum(indexBH));

  % set hits to zero and calculate average
  optionPrice = exp(-rf*deltaT*noOfSteps) / (noOfSim) *...
    sum(double(indexBH) .* min( max( pathMatrix(:,end,:) -...
    repmat(strike,[1,1,noOfSim]),zeros(noOfAssets,1,noOfSim)),[],1));
```

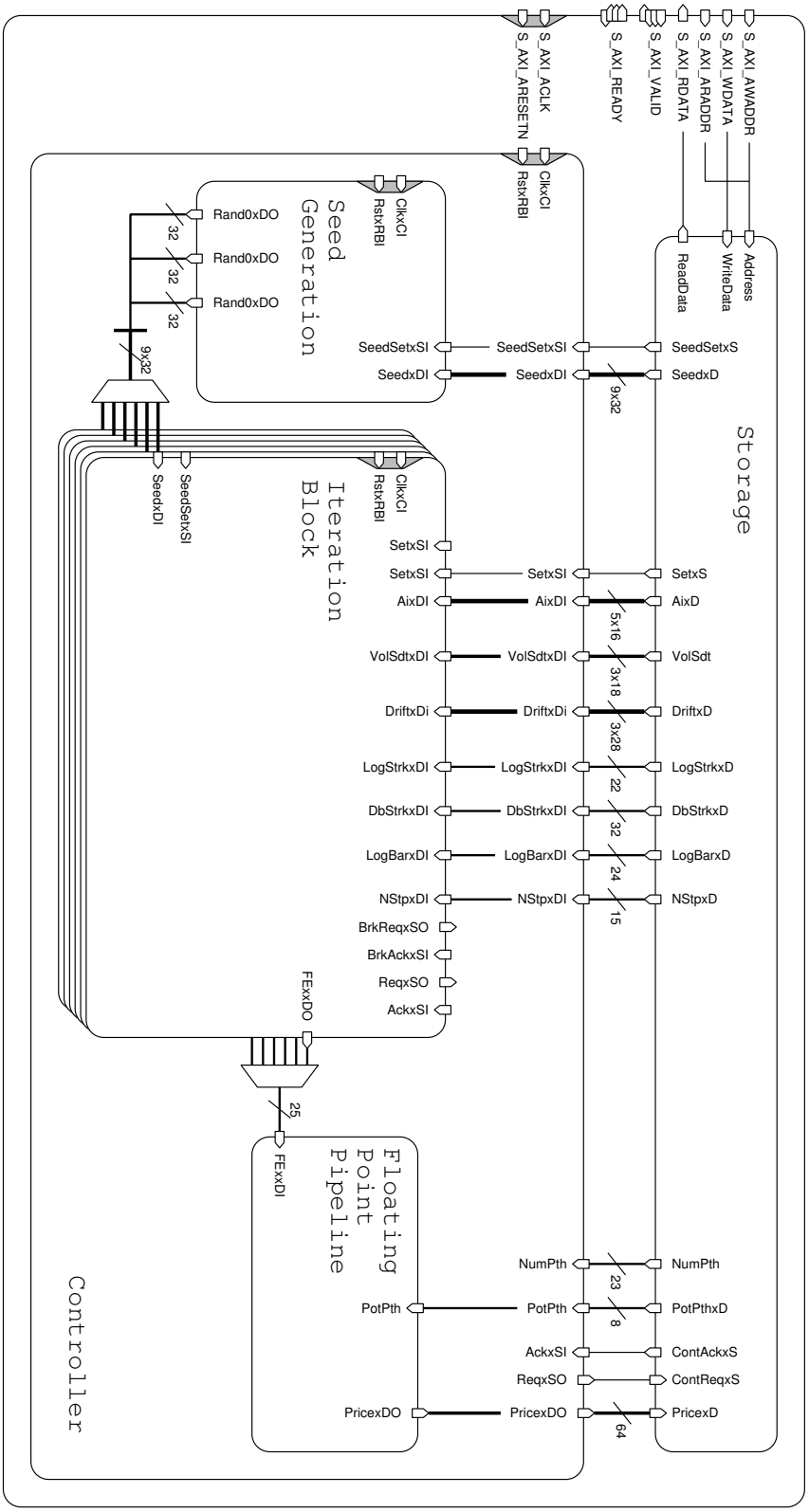


Figure B.1.: Detailed block diagram

Appendix C

MultiCorePricer 2015 Simulation Data

This appendix lists the underlying assets used for the test products to analyze the precision and speed-up of the architecture. Extensive lists of the average convergence point errors and speed-ups are given. For all pricings a Δt of $\frac{1\,year}{1000}$ was used.

C.1. Black-Scholes Comparison

The historical data between 2010 and 2014 of the following stocks, shown in figure C.1, were used to determine the input parameters and compute the average convergence point error:

- ABB (ABBN)
- Nestle (NESN)
- Novartis (NOVN)
- Swatch (UHR)
- Swisscom (SCMN)
- UBS (UBSN)

Table C.1 shows a list of the maximum and average error of the architecture's convergence point for different call option parameters over the six above listed stocks. Monte-Carlo pricings with 2^{14} and 2^{17} simulated paths were used. The strike price is given as a percentage of the initial price.

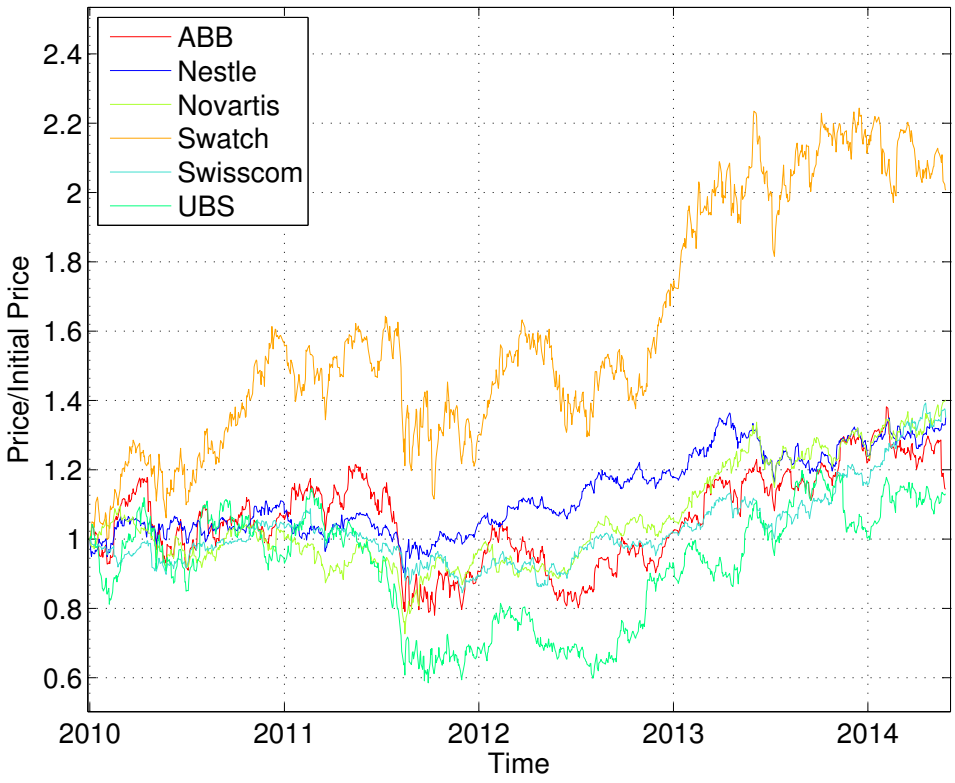


Figure C.1.: Time series of the historical data used for the Black-Scholes comparison

C.2. Worst-of Barrier Call Option Comparison

The historical data between 2010 and 2014 of the following stock-sets were used to determine the input parameters and compute the average convergence point error (the rightmost percentage shows the approximate average correlation of the stocks with each other):

- ABB (ABBN), CIE Financiere Richemont (CFR), Swatch (UHR) (70%)
- SGS (SGSN), swissRE (SREN), UBS (UBSN) (52%)
- Actelion (ATLN), Novartis (NOVN), Swisscom (SCMN) (35%)

Table C.2 shows a list of the maximum and average error of the architecture's convergence point for different product parameters over the three above listed stock-sets with strike prices of 100% and 95% of the initial price. Monte-Carlo pricings with 2^{14}, 2^{17} and 2^{20} paths were used. The barrier price is given as a percentage of the initial price. The

error values and standard deviations are given as a percentage relative to the computed ground truth price.

C.3. Speed-Up

Speed-up was determined by comparing the time the FPGA needed to price a product with the time needed by the Matlab script running on a single Intel i5-4200U Core running at 1.60GHz with 8GB RAM in Matlab R2013b 64-bit on Ubuntu 12.04.

Table C.3 shows the computation times on CPU and FPGA and the resulting speed-ups for different product setups. The CPU and FPGA computation times were determined by running the corresponding product pricing multiple times and taking the average computation time.

Strike Price	Time to Maturity	2^{14} Path Pricing Max. Error	2^{14} Path Pricing Avg. Error	2^{17} Path Pricing Max. Error	2^{17} Path Pricing Avg. Error
120%	0.5 years	0.448%	0.295%	0.332%	0.180%
120%	1 year	0.237%	0.180%	0.237%	0.137%
120%	2 years	0.205%	0.134%	0.180%	0.088%
120%	3 years	0.122%	0.095%	0.075%	0.046%
110%	0.5 years	0.204%	0.125%	0.201%	0.115%
110%	1 year	0.164%	0.108%	0.116%	0.087%
110%	2 years	0.116%	0.056%	0.107%	0.052%
110%	3 years	0.121%	0.061%	0.101%	0.081%
100%	0.5 years	0.112%	0.076%	0.081%	0.047%
100%	1 year	0.075%	0.054%	0.077%	0.046%
100%	2 years	0.088%	0.062%	0.080%	0.055%
100%	3 years	0.077%	0.059%	0.076%	0.045%
95%	0.5 years	0.162%	0.057%	0.047%	0.023%
95%	1 year	0.145%	0.053%	0.060%	0.033%
95%	2 years	0.139%	0.062%	0.049%	0.023%
95%	3 years	0.127%	0.090%	0.050%	0.032%
90%	0.5 years	0.071%	0.041%	0.055%	0.029%
90%	1 year	0.078%	0.041%	0.068%	0.034%
90%	2 years	0.052%	0.029%	0.042%	0.026%
90%	3 years	0.091%	0.034%	0.028%	0.013%
80%	0.5 years	0.041%	0.032%	0.038%	0.016%
80%	1 year	0.060%	0.029%	0.024%	0.012%
80%	2 years	0.095%	0.048%	0.047%	0.017%
80%	3 years	0.078%	0.045%	0.031%	0.012%
70%	0.5 years	0.035%	0.027%	0.014%	0.006%
70%	1 year	0.031%	0.023%	0.034%	0.016%
70%	2 years	0.035%	0.026%	0.039%	0.023%
70%	3 years	0.052%	0.039%	0.052%	0.023%

Table C.1.: Pricing precision for a single asset call option

Barrier Price	Time to Maturity	2^{14} Path Pricing			2^{17} Path Pricing			2^{20} Path Pricing		
		Max. Error	Avg. Error	Avg. Std.	Max. Error	Avg. Error	Avg. Std.	Max. Error	Avg. Error	Avg. Std.
15%	1 year	0.120%	0.042%	1.799%	0.097%	0.045%	0.649%	0.089%	0.052%	0.217%
40%	1 year	0.063%	0.041%	1.797%	0.126%	0.064%	0.628%	0.097%	0.045%	0.220%
50%	1 year	0.068%	0.053%	1.767%	0.056%	0.032%	0.623%	0.097%	0.056%	0.230%
60%	1 year	0.049%	0.023%	1.797%	0.126%	0.041%	0.631%	0.040%	0.025%	0.226%
70%	1 year	0.049%	0.023%	1.808%	0.075%	0.037%	0.673%	0.129%	0.066%	0.237%
80%	1 year	0.120%	0.069%	1.952%	0.080%	0.038%	0.686%	0.066%	0.030%	0.235%
85%	1 year	0.290%	0.201%	2.225%	0.205%	0.085%	0.749%	0.091%	0.063%	0.279%
90%	1 year	0.501%	0.434%	2.883%	0.176%	0.109%	1.042%	0.136%	0.092%	0.351%
15%	2 years	0.146%	0.074%	1.902%	0.275%	0.133%	0.658%	0.118%	0.075%	0.241%
40%	2 years	0.154%	0.066%	1.885%	0.153%	0.065%	0.674%	0.130%	0.100%	0.226%
50%	2 years	0.080%	0.051%	1.930%	0.129%	0.061%	0.658%	0.100%	0.080%	0.229%
60%	2 years	0.102%	0.040%	1.902%	0.162%	0.102%	0.685%	0.119%	0.087%	0.231%
70%	2 years	0.099%	0.034%	2.014%	0.124%	0.079%	0.721%	0.151%	0.064%	0.242%
80%	2 years	0.255%	0.174%	2.407%	0.121%	0.065%	0.826%	0.049%	0.033%	0.299%
85%	2 years	0.487%	0.382%	2.870%	0.181%	0.076%	1.012%	0.109%	0.050%	0.376%
90%	2 years	0.952%	0.832%	3.883%	0.560%	0.400%	1.343%	0.398%	0.263%	0.504%
15%	3 years	0.189%	0.099%	1.976%	0.221%	0.097%	0.676%	0.112%	0.091%	0.245%
40%	3 years	0.148%	0.102%	1.980%	0.177%	0.139%	0.741%	0.161%	0.114%	0.262%
50%	3 years	0.146%	0.057%	2.002%	0.162%	0.110%	0.698%	0.144%	0.089%	0.242%
60%	3 years	0.074%	0.034%	2.038%	0.294%	0.114%	0.719%	0.119%	0.084%	0.245%
70%	3 years	0.124%	0.075%	2.229%	0.118%	0.068%	0.777%	0.083%	0.049%	0.257%
80%	3 years	0.318%	0.254%	2.795%	0.084%	0.052%	0.954%	0.072%	0.042%	0.327%
85%	3 years	0.512%	0.457%	3.427%	0.394%	0.126%	1.202%	0.119%	0.062%	0.408%
90%	3 years	1.365%	1.128%	4.739%	0.385%	0.301%	1.647%	0.365%	0.227%	0.578%
50-80%	1 year		0.042%			0.037%			0.044%	
50-80%	2 years		0.075%			0.077%			0.066%	
50-80%	3 years		0.105%			0.086%			0.066%	
50-80%	1-3 years		0.074%			0.067%			0.059%	
15-90%	1-3 years		0.198%			0.102%			0.081%	

Table C.2.: Pricing precision for the worst-of barrier call option (errors in absolute value)

Barrier Price	Time to Maturity	CPU Comp. Time	FPGA Comp. Time	Speed-up
15%	1 year	3.50442s	0.00638s	549.371
40%	1 year	3.50384s	0.00637s	550.047
50%	1 year	3.50731s	0.00632s	553.994
60%	1 year	3.51637s	0.00611s	573.574
70%	1 year	3.50272s	0.00547s	640.598
80%	1 year	3.48017s	0.00413s	851.542
85%	1 year	3.48866s	0.00315s	1119.573
90%	1 year	3.50583s	0.00201s	1757.184
15%	2 years	6.92515s	0.01271s	546.040
40%	2 years	6.90814s	0.01254s	553.657
50%	2 years	6.87407s	0.01204s	576.917
60%	2 years	6.93533s	0.01089s	638.239
70%	2 years	6.97458s	0.00888s	784.736
80%	2 years	6.96409s	0.00597s	1170.708
85%	2 years	7.00048s	0.00428s	1635.074
90%	2 years	6.93196s	0.00256s	2726.818
15%	3 years	10.44938s	0.01905s	545.113
40%	3 years	10.44546s	0.01828s	568.089
50%	3 years	10.37941s	0.01693s	613.772
60%	3 years	10.39047s	0.01460s	712.742
70%	3 years	10.44707s	0.01127s	926.177
80%	3 years	10.26342s	0.00715s	1462.227
85%	3 years	10.34386s	0.00498s	2099.776
90%	3 years	10.39921s	0.00290s	3600.175

Table C.3.: Speed-ups and computation times of pricings using 2^{14} simulated paths for different product setups

Appendix D

Presentation Slides

MultiCorePricer: Derivative Pricing

using hardware acceleration on the Zynq

Miguel Guerrero
Harald Kröll, Donnacha Daly,
Marcus Hildmann, Lukas Bruderer
Prof. Qiuting Huang, Prof. Didier Sornette

ETH Zurich

January 29, 2015

Outline

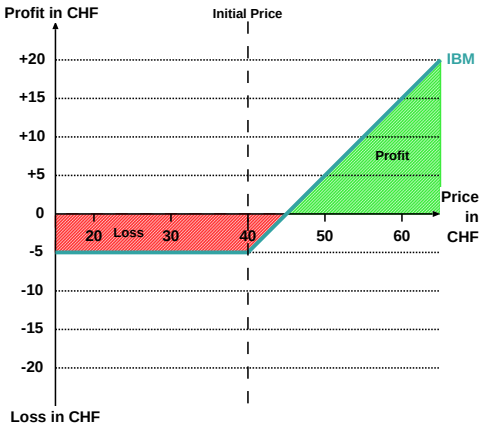
Introduction

Project Background

- A **derivative** is a contract that *derives* its value from the performance of an underlying entity, which can be an asset, index, or interest rate and is often called an underlying.
- Analytical evaluation of prices and risk figures generally not possible → Monte-Carlo simulations
- Goal: Investigate advantages of a dedicated architecture on an FPGA over CPU based one.

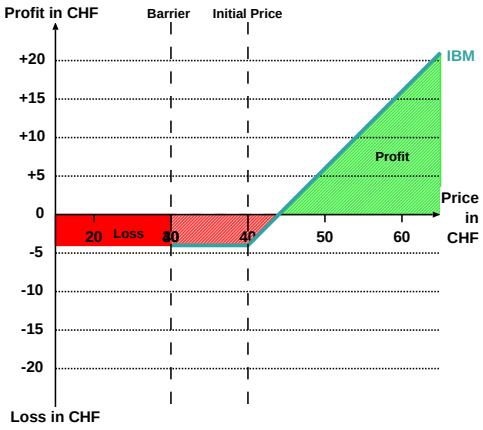
Call Option

- Contract which gives the buyer the right, but not the obligation, to buy an underlying at a specified date for a specified price.



Barrier Call Option

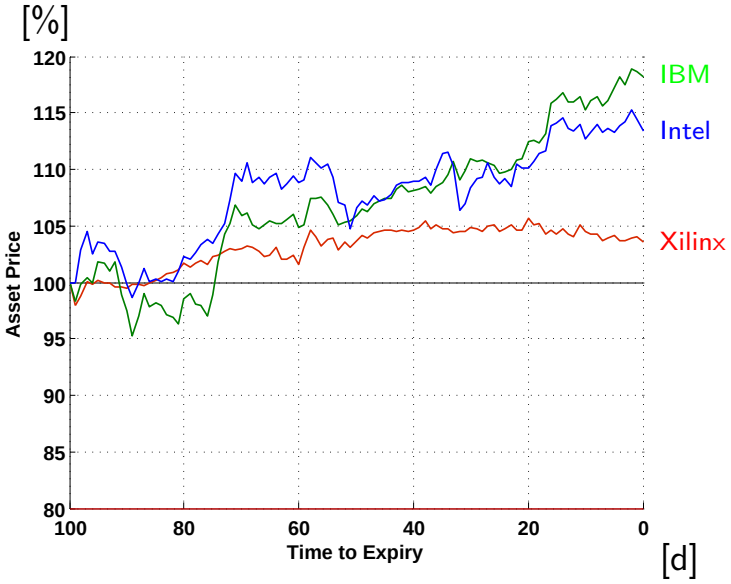
- A Call option that nullifies if the price drops on or below the **barrier level** during it's lifetime.



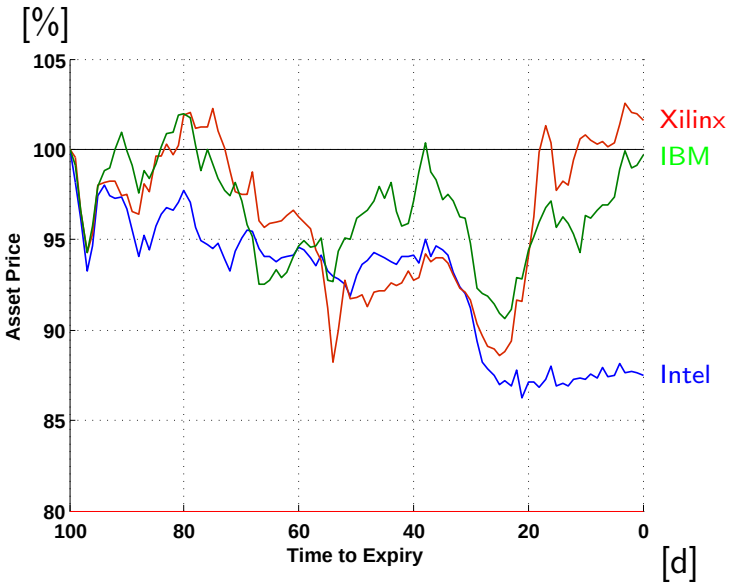
Worst-of Barrier Option

- Multiple underlyings instead of one
- 3 underlyings fixed for this architecture
- Payout is measured on worst performing underlying

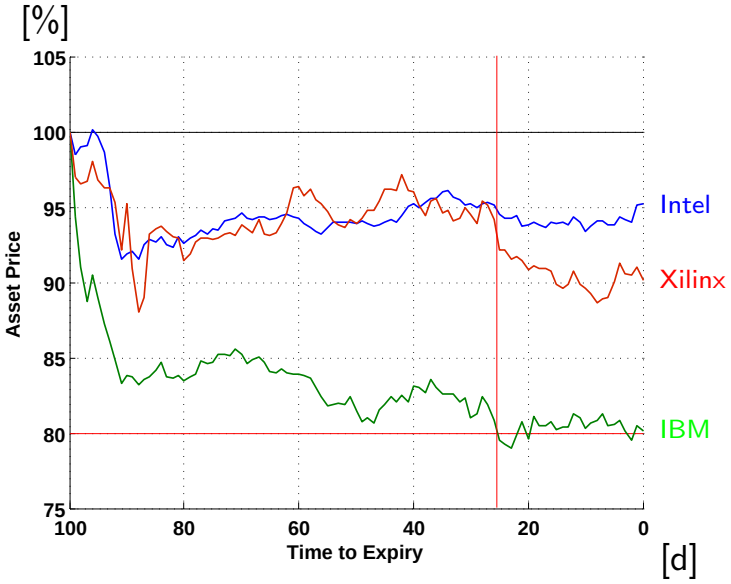
Worst-of Barrier Option



Worst-of Barrier Option



Worst-of Barrier Option



Pricing the Product

- Price is the mean of all simulated outcomes
- Three possible simulation outcomes:
  1. No barrier event & lowest underlying's price $>$ Initial price
     $\rightarrow$ return price of lowest underlying
  2. No barrier event & lowest underlying's price $\leq$ Initial price
     $\rightarrow$ return 0
  3. Barrier event
     $\rightarrow$ return 0

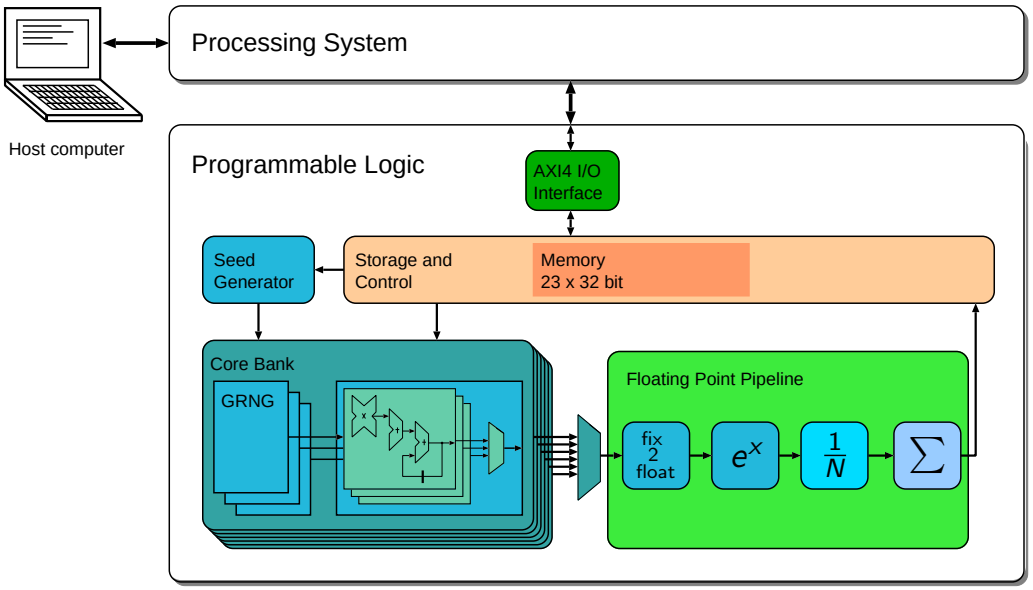
Simulating Underlying Paths

- Model used: Geometric Brownian motion
- $dS_t = \overbrace{\mu S_t dt}^{drift} + \overbrace{\sigma S_t dW_t}^{diffusion}$

  $S(t)$ price at time t
  $\mu(t)$ percentage drift
  $\sigma(t)$ percentage volatility (std)
  $W(t)$ Wiender process or Brownian motion
- W(t) starts at 0, moves along normally distributed increments
- Applying one time step:
  $S_t = S_0 \cdot \exp((\mu - \frac{\sigma^2}{2})t + \sigma W_t)$
- Correlated Wiener process $\Rightarrow$ correlated increments
- $W(t) - W(t - \Delta t) \propto \begin{bmatrix} 1 & 0 & 0 \\ a_0 & a_1 & 0 \\ a_2 & a_3 & a_4 \end{bmatrix} \begin{bmatrix} Grd_0 \\ Grd_1 \\ Grd_2 \end{bmatrix} = \begin{bmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{bmatrix}$

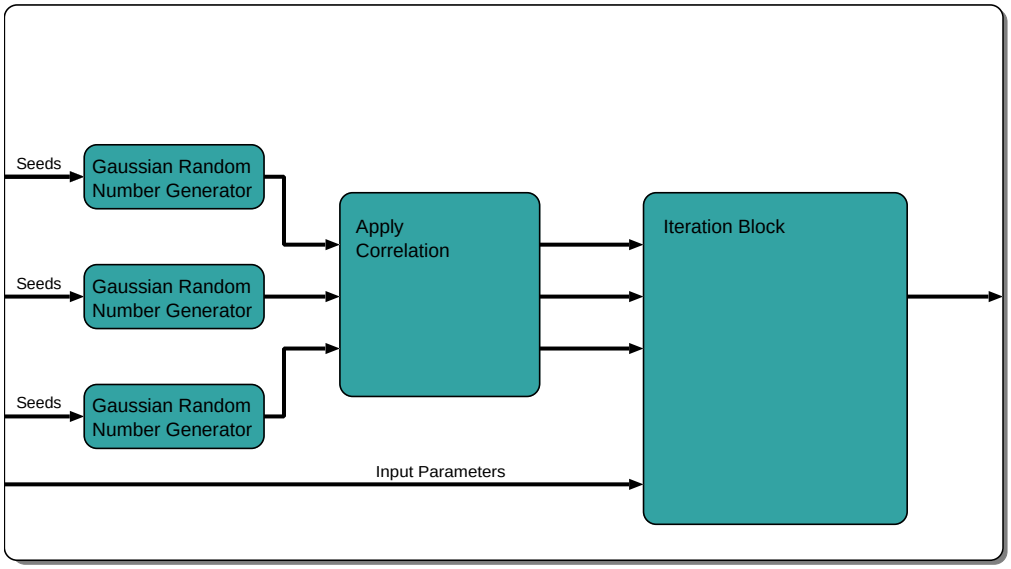
Design

Design Overview

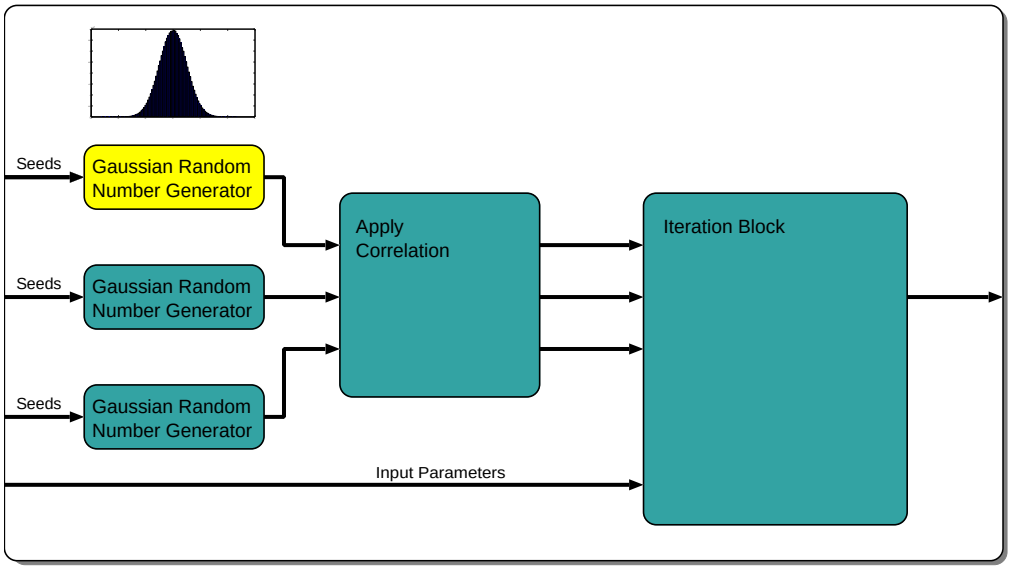


MultiCorePricer 2015

Core

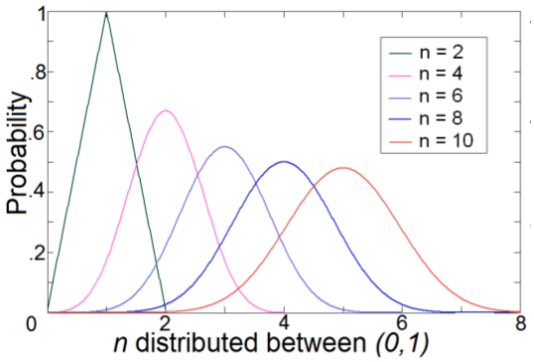

Seeds
Gaussian Random Number Generator
Seeds
Gaussian Random Number Generator
Seeds
Gaussian Random Number Generator
Apply Correlation
Iteration Block
Input Parameters

Core


Seeds
Gaussian Random Number Generator
Seeds
Gaussian Random Number Generator
Seeds
Gaussian Random Number Generator
Apply Correlation
Iteration Block
Input Parameters

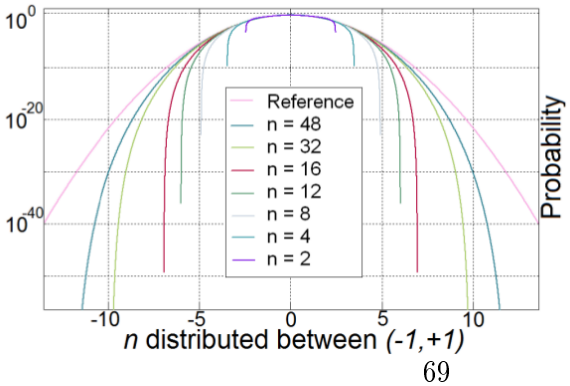
Central Limit Theorem

- Arithmetic mean of a sufficiently large number of independent random variables will be approximately normally distributed



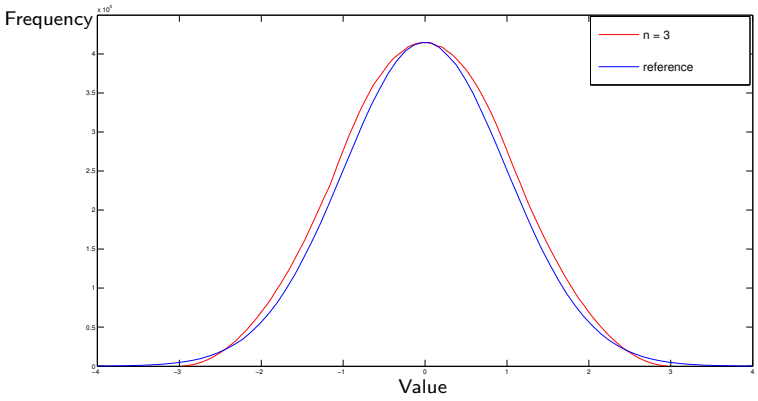
Central Limit Theorem

- Arithmetic mean of a sufficiently large number of independent random variables will be approximately normally distributed
- Seldom used for high accuracy normally distributed random numbers
- Finite number of additions $\rightarrow$ error in tail region



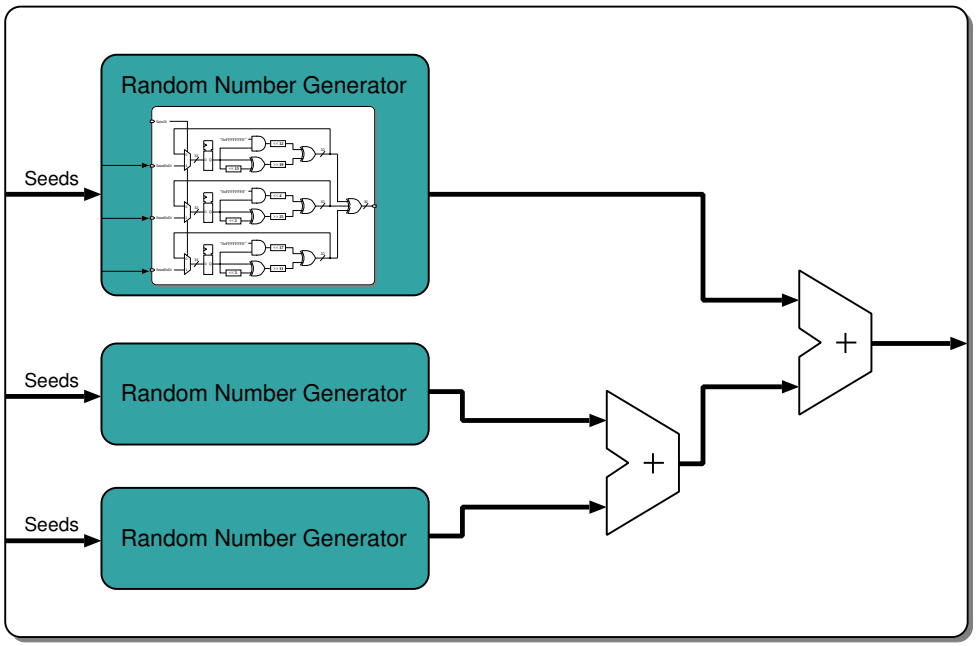
Central Limit Theorem

- Low accuracy distribution performs well
- For this application → relative error of up to 1% acceptable
- n = 3 already fulfills this condition



Matlab (randn) vs. CLT (n=3)

Gaussian Random Number Generator



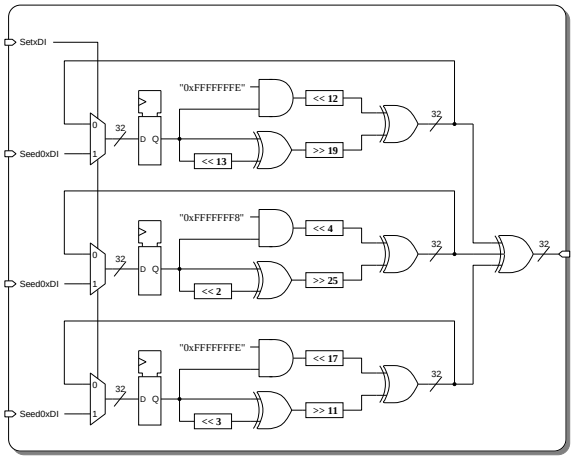
Uniform Random Number Generator

- Proposed by L'Ecuyer et al.
- Combined generator

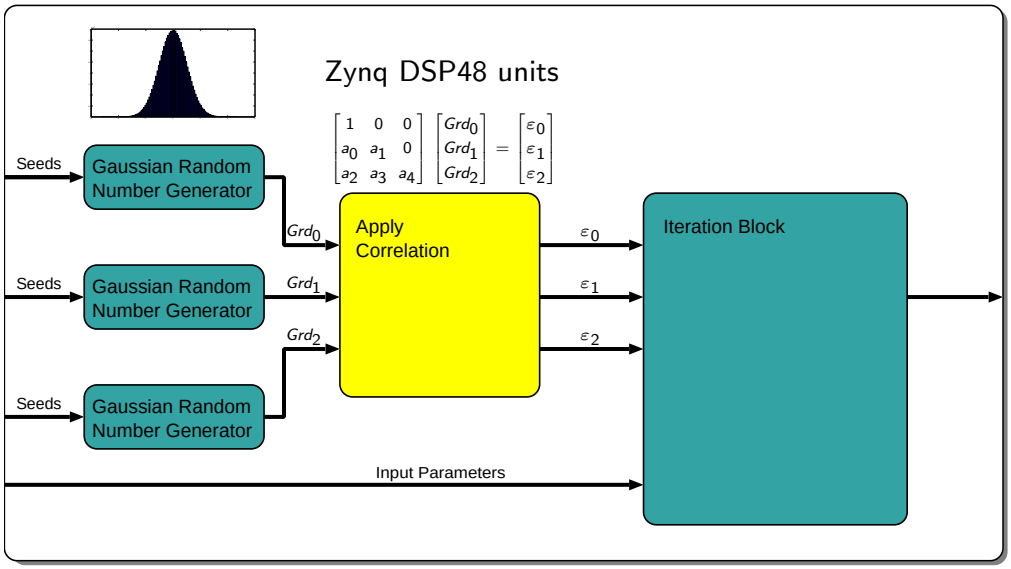
$$P_i(z) = z^k - z^q - 1 \; mod \; 2$$

$$P_{comb}(z) = P_0(z) \cdot P_1(z) \cdot P_2(z)$$

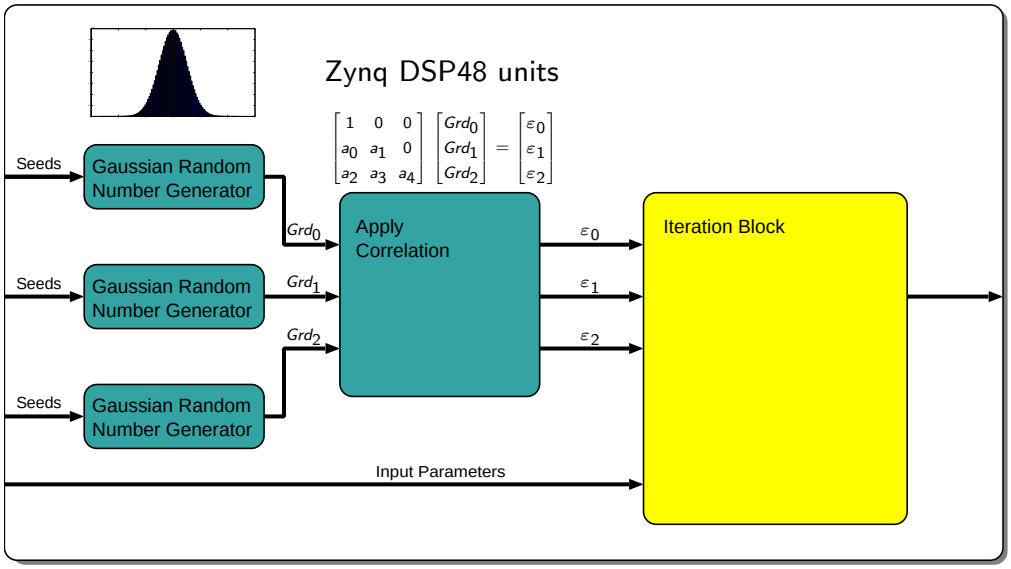
- Few nonzero coefficients
- Periodicity of 2^{88}



Core



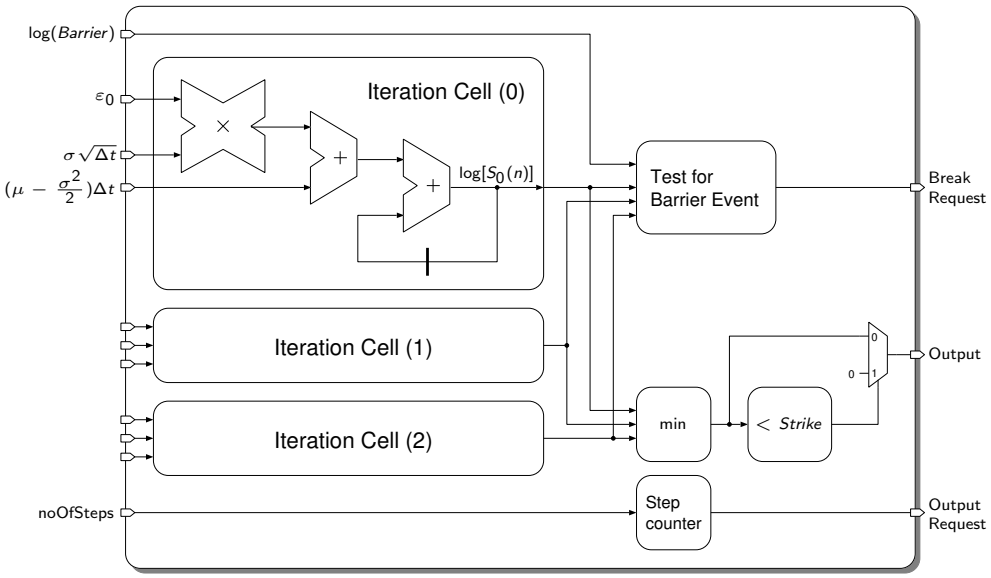
Core



Iteration Block

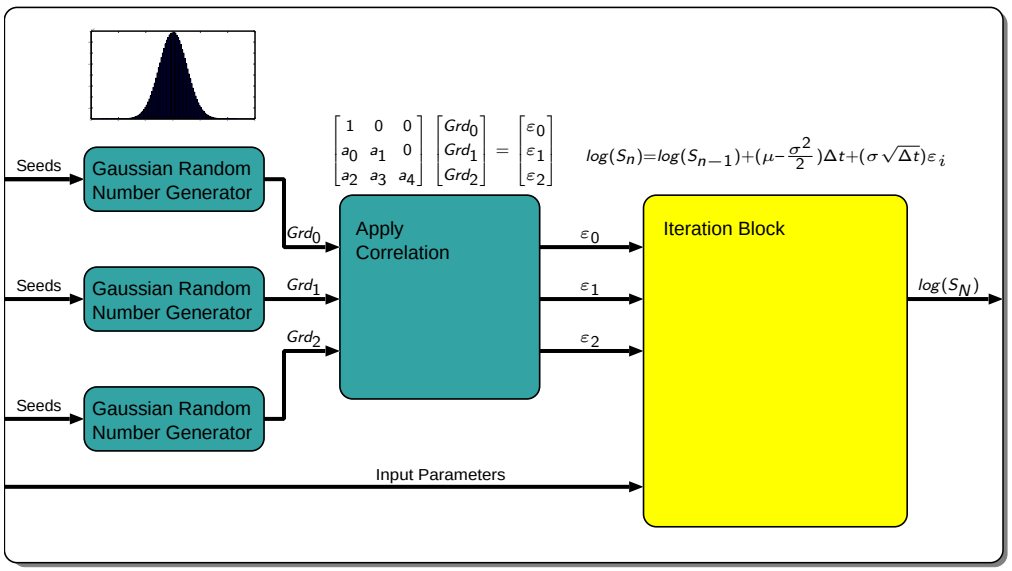
- $S_i(n) = S_i(n-1) \cdot \exp((\mu - \frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}\,\varepsilon_{n-1})$
- $\log[S_i(n)] = \log[S_i(n-1)] + \underbrace{(\mu - \frac{\sigma^2}{2})t}_{drift=const.} + \underbrace{\sigma\sqrt{\Delta t}\,\varepsilon_{n-1}}_{Diffusion}$
- $\log[S_i(0)] = 0 \rightarrow S_i(0) = 1$, instead of $S_i(0) =$ Starting price
- $\log[S_i(n)] \leq \log(\textit{Barrier in percent})$

Iteration Block

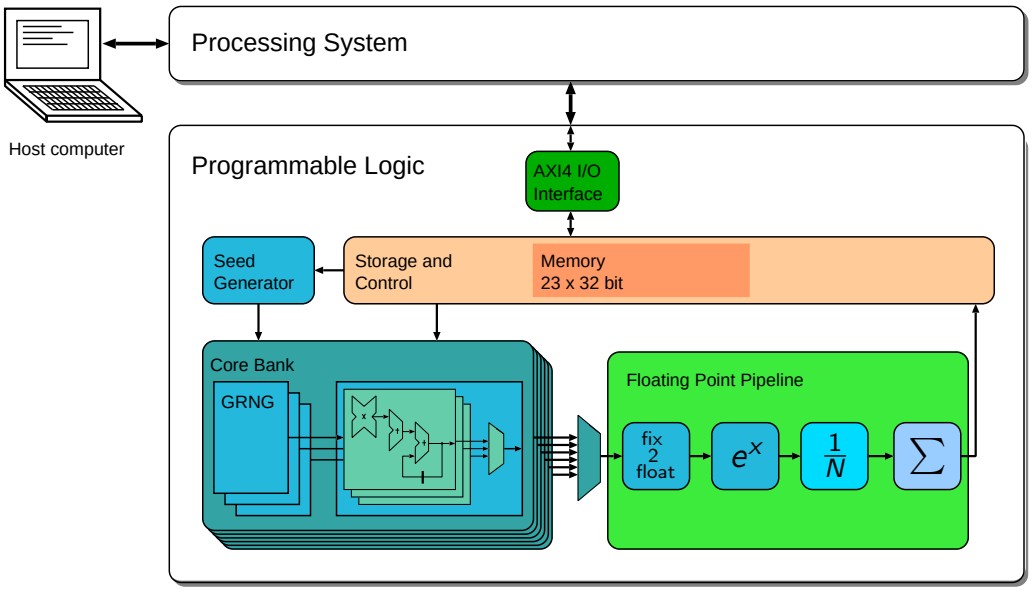


$$\log[S_i(n)] = \log[S_i(n-1)] + \underbrace{(\mu - \frac{\sigma^2}{2}\Delta t)}_{Drift=const.} + \underbrace{(\sigma\sqrt{\Delta t}\cdot\varepsilon_i)}_{Diffusion}$$

Core



Design Overview



MultiCorePricer 2015

Results

Precision

- Relative error for convergence point
- Standard deviation for Zynq and Matlab the same
- Higher rel. error for high barrier and long product lifetime
  → Non-ideal Gauss bell

	Barrier [%]		
Duration	40	60	80
1 year	0.0005	0.0005	0.0012
2 years	0.0005	0.0005	0.0025
3 years	0.0014	0.0014	0.0031

Zynq vs. Matlab, $\Delta t = \frac{1year}{1000}$

Speed-Up

- Higher Barrier
  → more simulation breaks
- No barrier
  → Speed-Up ≈ 550

	Barrier [%]		
Duration	40	60	80
1 year	552	581	891
2 years	559	685	1250
3 years	607	773	1551

Zynq vs. Matlab, $\Delta t = \frac{1year}{1000}$

Utilization

- Maximum clock frequency: 100MHz

	Full Arch.	Single Core	Exp, Add	Cont.+AXI
DSP				
DSP48	95.91%	3.64%	1.36%	0%
Slice Logic				
Slice LUTs	81.02%	2.70%	4.15%	6.67%
Slice Registers	39.35%	1.33%	3.06%	1.66%

Utilization

	Zynq-7020 XC7Z020	Zynq MMP XC7Z045
DSP Slices	220	2020
Logic Cells	85k	444k
Cores	26	135
Speed-up	600-900	3120-46800
Price	495.00$	1,495.00$

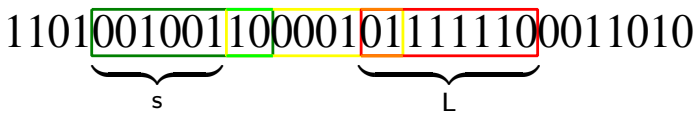
Conclusion

- General speed-up of 600x to 900x
- Higher speed-up for longer product lifetime
- First architecture for financial products using CLT Gaussian RNG
- Overhead has been held minimal by using a slim AXI4-Lite communication protocol and dedicated Exp() and Add IPs
- Limiting factor on Zynq are the DSP Slices

Appendix

Tausworthe Generator

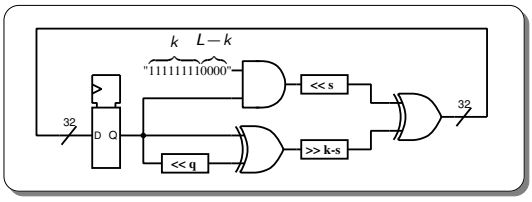
- Sequence of bits from a linear recurrence modulo 2
  $x_n = a_1 x_{n-1} + ... + a_k x_{n-k} \ mod\ 2, \quad x_i, a_i \in \{0,1\}$
  $P(z) = z^k - a_1 z^{k-1} - ... - a_k$
- Fractional numbers formed by taking blocks of bits
  $u_n = \sum_{i=1}^{L} x_{ns+i-1} 2^{-i}$

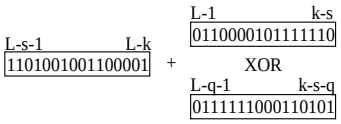


- System deterministic ⇒ Produces pseudorandom numbers
- Periodicity of $p = 2^k - 1$ if P is primitive, $s_0 \neq 0$, and s is coprime to p
- Requires s steps for each u_n

Tausworthe Generator

- Faster implementation for special cases.
- Primitive trinomial of the form $P(z) = z^k - z^q - 1$
  $0 < 2q < k, 0 < s \leq k - q < k \leq L,\ gcd(s, 2^k - 1) = 1$

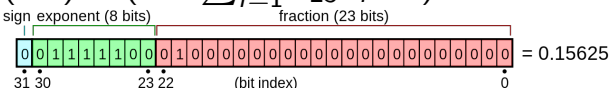




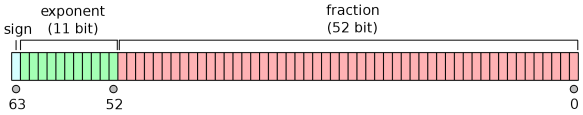
- Too few nonzero coefficients lead to statistical defects
- Period length cannot exceed 2^L

Exponential & Adder IP

- Single-precision floating point exponential function IP
- $(-1)^{sign}(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}) \times 2^{e-127}$



- Number of simulated paths is always a 2's potence
  $\rightarrow$ division becomes subtraction of exponent
- Double-precision floating point adder
- $(-1)^{sign}(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}) \times 2^{e-1023}$



Finite field

- A Finite field or Galois Field is a field that contains a finite number of elements, called its order.
- Common but not the only examples of finite fields are the integers modulo a prime.
- Finite fields only exist when the order is a prime power p^k (no zero divisors allowed).
- The Characteristic of a field of order p^k is p (this means that adding p copies of any element always results in zero)

Finite field examples

F_2

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

F_3

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

×	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

F_4

+	0	1	A	B
0	0	1	A	B
1	1	0	B	A
A	A	B	0	1
B	B	A	1	0

×	0	1	A	B
0	0	0	0	0
1	0	1	A	B
A	0	A	B	1
B	0	B	1	A

F_9

```
Field of 9 elements represented as matrices
integers are modulo 3

element (0)     element (1)     element (2)

  0  0            1  0            0  1
  0  0            0  1            1  1

element (3)     element (4)     element (5)

  1  1            1  2            2  0
  1  2            2  0            0  2

element (6)     element (7)     element (8)

  0  2            2  2            2  1
  2  2            2  1            1  0

+/  (0) (1) (2) (3) (4) (5) (6) (7) (8)
(0)  0   1   2   3   4   5   6   7   8
(1)  1   5   3   8   7   0   4   6   2
(2)  2   3   6   4   1   8   0   5   7
(3)  3   8   4   7   5   2   1   0   6
(4)  4   7   1   5   8   6   3   2   0
(5)  5   0   8   2   6   1   7   4   3
(6)  6   4   0   1   3   7   2   8   5
(7)  7   6   5   0   2   4   8   3   1
(8)  8   2   7   6   0   3   5   1   4

x/  (0) (1) (2) (3) (4) (5) (6) (7) (8)
(0)  0   0   0   0   0   0   0   0   0
(1)  0   1   2   3   4   5   6   7   8
(2)  0   2   3   4   5   6   7   8   1
(3)  0   3   4   5   6   7   8   1   2
(4)  0   4   5   6   7   8   1   2   3
(5)  0   5   6   7   8   1   2   3   4
(6)  0   6   7   8   1   2   3   4   5
(7)  0   7   8   1   2   3   4   5   6
(8)  0   8   1   2   3   4   5   6   7
```


Picture References

- CLT for different n values, courtesy: Revisiting Central Limit Theorem, Malik et al.
- Single-precision floating point diagram, courtesy: Wikipedia, Fresheneesz
- Double-precision floating point diagram, courtesy: Wikipedia, Codekaizen
- Finite field examples, courtesy Wikipedia

Bibliography

[1] "SIX structured products," http://www.six-structured-products.com, accessed: 2015-01-27.

[2] S. Tolle, B. Hutter, P. Rüthemann, and H. Wohlwend, *Structured Products in Wealth Management.* John Wiley & Sons, 2012.

[3] "Schweizerische Nationalbank: Statistisches Monatsheft Dezember 2014," http://www.snb.ch/ext/stats/statmon/pdf/defr/D5_1_Wertschriftenb_in_Kundendepots.pdf, accessed: 2015-01-07.

[4] "SIX Structured Products Exchange AG: Marktreport Nr.11 November 2014," www.six-structured-products.com/arcmsdownload/c664fb0c10cb46957c9fb2337a7f7226/CONTENT.pdf/SIX_SPE_Marktreport_1114_Final.pdf, accessed: 2015-01-07.

[5] P. Brandimarte, *Numerical methods in finance and economics: a MATLAB-based introduction.* John Wiley & Sons, 2013.

[6] A. J. McNeil, R. Frey, and P. Embrechts, *Quantitative risk management: concepts, techniques, and tools.* Princeton university press, 2010.

[7] J. Hull, *Options, futures and other derivatives.* Pearson education, 2009.

[8] D. Matteson and D. Ruppert, "Time-series models of dynamic volatility and correlation," *Signal Processing Magazine, IEEE*, vol. 28, no. 5, pp. 72–82, Sept 2011.

[9] P. Boyle, M. Broadie, and P. Glasserman, "Monte carlo methods for security pricing," *Journal of economic dynamics and control*, vol. 21, no. 8, pp. 1267–1321, 1997.

[10] M. Haugh, "Advanced variance reduction techniques," *Lecture Notes IEOR E4703, Center for Financial Engineering, Columbia University*, 2004.

[11] B. Spiers and D. Wallez, "High-performance computing on wall street," *Computer*, vol. 43, no. 12, pp. 53–59, 2010.

[12] L. Hong, L. Zhong-hua, and C. Xue-bin, "The applications and trends of high performance computing in finance," in *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on.* IEEE, 2010, pp. 193–197.

[13] E. Jäger, "High performance computing and statistical modeling of financial returns," Master's thesis, ETH Zurich, 2010.

[14] M. Smelyanskiy, "Challenges of mapping financial analytics to many-core architecture," in *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on.* IEEE, 2008, pp. 1–1.

[15] R.-S. Liu, Y.-C. Tsai, and C.-L. Yang, "Parallelization and characterization of GARCH option pricing on GPUs," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–10.

[16] G. W. Morris, D. B. Thomas, and W. Luk, "FPGA accelerated low-latency market data feed processing," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on.* IEEE, 2009, pp. 83–89.

[17] S. Weston, J.-T. Marin, J. Spooner, O. Pell, and O. Mencer, "Accelerating the computation of portfolios of tranched credit derivatives," in *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on.* IEEE, 2010, pp. 1–8.

[18] D. B. Thomas and W. Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on.* IEEE, 2008, pp. 229–238.

[19] A. H. Tse, D. B. Thomas, and W. Luk, "Option pricing with multi-dimensional quadrature architectures," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on.* IEEE, 2009, pp. 427–430.

[20] Q. Jin, D. B. Thomas, and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on.* IEEE, 2009, pp. 73–78.

[21] G. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. Cheung, D.-U. Lee, R. C. Cheung, and W. Luk, "Reconfigurable acceleration for monte carlo based financial simulation," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 2005, pp. 215–222.

[22] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. S. Trew, A. McCormick, G. Smart *et al.*, "Maxwell-a 64 FPGA supercomputer." *AHS*, vol. 7, pp. 287–294, 2007.

[23] X. Tian and K. Benkrid, "Design and implementation of a high performance financial monte-carlo simulation engine on an FPGA supercomputer," in *ICECE Technology, 2008. FPT 2008. International Conference on.* IEEE, 2008, pp. 81–88.

[24] N. A. Woods and T. VanCourt, "FPGA acceleration of quasi-monte carlo in finance," in *Field programmable logic and applications, 2008. FPL 2008. International Conference on.* IEEE, 2008, pp. 335–340.

[25] R. Sridharan, G. Cooke, K. Hill, H. Lam, and A. George, "FPGA-based reconfigurable computing for pricing multi-asset barrier options," in *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on.* IEEE, 2012, pp. 34–43.

[26] R. C. Cheung, D.-U. Lee, W. Luk, and J. D. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, no. 8, pp. 952–962, 2007.

[27] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An energy efficient FPGA accelerator for monte carlo option pricing with the heston model," *Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 468–474, 2011.

[28] C. de Schryver, D. Schmidt, N. When, E. Korn, H. Marxen, and R. Korn, "A new hardware efficient inversion based random number generator for non-uniform distributions," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on.* IEEE, 2010, pp. 190–195.

[29] X. Tian, K. Benkrid, and X. Gu, "High performance monte-carlo based option pricing on fpgas." *Engineering Letters*, vol. 16, no. 3, pp. 434–442, 2008.

[30] H. Edrees, B. Cheung, M. Sandora, D. B. Nummey, and D. Stefan, "Hardware-optimized ziggurat algorithm for high-speed gaussian random number generators." in *ERSA*, 2009, pp. 254–260.

[31] J. S. Malik, A. Hemani, J. N. Malik, B. Silmane, and N. D. Gohar, "Revisiting central limit theorem: Accurate gaussian random number generation in vlsi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 99, 2014.

[32] "Maximal length lfsr feedback terms," http://users.ece.cmu.edu/~koopman/lfsr/index.html, accessed: 2015-01-10.

[33] P. Greisen, "Flexible digital emulator for a wireless mimo channel," Master's thesis, ETH Zurich, 2007.

[34] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965.

[35] P. L'Ecuyer, "Maximally equidistributed combined tausworthe generators," *Math. Comp.*, vol. 65, pp. 203–213, 1996.

[36] R. Griffin, "Designing a custom axi-lite slave peripheral," http://www.silica.com/fileadmin/02_Products/Productdetails/Xilinx/designing_a_custom_axi_slave_rev1.pdf, accessed: 2015-01-09.