Design and Implementation of a Floating-Point Application Specific Instruction Set Processor for Asset Simulations

Jacqueline Stratmann

BACHELORTHESIS

Department of Computer Science
University of Kaiserslautern

May 2012

Supervisors:
Christian de Schryver
Prof. Dr. Klaus Schneider
Prof. Dr. Norbert Wehn

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mir beim Verfassen dieser Arbeit geholfen haben, in erster Linie meinen Betreuern Christian de Schryver, Prof. Dr. Klaus Schneider und Prof. Dr. Norbert Wehn.
Ganz besonders danken möchte ich außerdem meinen Eltern, Schwestern und meinem Freund. Sie haben mich immer unterstützt und waren jederzeit für mich da.

Erklärung / Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

I declare, that this thesis and the code has been composed by myself, that I did not use other sources than the cited ones and that I marked all passages which were taken from other sources.

Kaiserslautern, 31.05.2012

Jacqueline Stratmann

Zusammenfassung / Abstract

Diese Abschlussarbeit behandelt den Designvorgang und die Implementierung eines applikationsspezifischen Prozessors mit speziellem Befehlssatz (englisch: Application Specific Instruction Set Processor (ASIP)) für Optionspreisberechnungen. Da Optionen ein wichtiger Teil des Börsenalltags sind und deren Preise schnell und möglichst effizient berechnet werden sollen, bieten sich Hardwarebeschleuniger zur Berechnung an. ASIPs sind hier besonders interessant, da sie schneller und effizienter arbeiten als generische Prozessoren, trotzdem aber programmier- und damit erweiterbar sind.
Zuerst werden in dieser Arbeit einige Grundlagen abgehandelt. Der Begriff Option in der Finanzmathematik wird kurz erläutert, Gleitkommazahlen mit doppelter Präzision und die Grundrechenarten mit diesen werden erklärt und außerdem wird eine kleine Einführung in VHDL angeboten.
Anschließend wird näher auf den Ablauf des Designvorgangs eingegangen:
Der Referenzcode, also die Implementierung des Algorithmus für den der Prozessor entworfen wird, zeigt, welche Operationen und Methoden genutzt werden. Diese Operationen müssen auch vom ASIP ausgeführt werden können und bilden deshalb den ersten vorläufigen Befehlssatz. Für die genutzen Methoden werden Näherungsverfahren bzw., wenn dies möglich ist, genaue Verfahren untersucht. Um sie testen zu können und die dort verwendeten Operationen identifizieren zu können, werden diese Verfahren in C++ implementiert. Nachdem die endgültige Auswahl der Verfahren abgeschlossen ist, werden die Programme auf Operationen untersucht, die noch nicht im Befehlssatz stehen. Diese werden hinzugefügt und bilden so den endgültigen Befehlssatz.
Nachdem die Befehle feststehen, können die arithmetischen Einheiten bestimmt werden, die dann zusammen mit Kontrolleinheiten die Recheneinheit bilden. Um den Datenpfad zu vollenden, wird anschließend die Registerbank implementiert. Diese ist in Bezug auf Anzahl der Register und deren Bitbreite komplett generisch aufgebaut, um spätere Erweiterungen und auch den erneuten Nutzen in anderen Prozessoren zu ermöglichen.

This thesis considers the design and implementation of an application specific instruction set processor (ASIP) for option pricing. Due to the fact that options are an important task at the stock market and their prices have to be calculated as fast and as efficient as possible, hardware accelerators are very suitable. ASIPs are of a particular interest, because they are able to calculate faster and more efficient as generic processors. But nevertheless they are programmable and in the wake of that they are expandable.
First some basics are explained. The concept of a financial option is briefly summarized, double precision floating point numbers and the calculation with them are explained. Furthermore, a short introduction to VHDL is given.
Adjacent the design flow is examined:
The reference code, i. e. the implementation of the algorithm, the ASIP is designed for, shows the required operations and mathematical methods. The ASIP has to be able to execute these operations and therefore, they build the first instruction set. The methods are implemented with algorithms that approximate or, if possible, compute the exact result. Addicted to the required operations, the performance and the memory requirements, the final implementation of each method is chosen. Operations that are used within these implementations and that are missing in the instruction set are added to it.
After assigning the final instruction set, the required arithmetic units that build die arithmetic logical unit (ALU) in combination with control units are identified. To complete the data path, the registerfile is implemented. It is fully generic with respect to the number of registers and their bitwidth to allow subsequent extensions and the employment for another processor possible.

Contents

1 Acronyms

ABS	Absolute Value
ADD	Addition
ALU	Arithmetic Logic Unit
ASIP	Application Specific Instruction Set Processor
CEIL	Round Up (to infinity resp. $\lceil x \rceil$)
CPU	Central Processing Unit
DIV	Division
FA	Full Adder
FAC	Faculty
FPGA	Field Programmable Gate Array
GPGPU	General-Purpose Computing On Graphics Processing Units
GHz	Gigahertz (10^9 Hz)
GPU	Global Processing Unit
HA	Half Adder
IFF	If And Only If
LISA	Language for Instruction Set Architecture
MAC	Multiply and Accumulate
MAX	Maximum Value
MUL	Multiplication
MUX	Multiplexer
PC	Personal Computer
POW	Power
RAM	Random Access Memory
SDE	Stochastic Differential Equation
SQRT	Square Root
SUB	Subtraction
VHDL	Very High Speed Integrated Circuit Hardware Description Language

2 Introduction

2.1 Motivation

The financial business is very dynamic and the fast calculation of option prices is a common but nevertheless a non-trivial issue.
In order to evaluate options and calculate their prices, hardware accelerators are a beneficial alternative to Personal Computers (PCs), Global Processing Unit (GPU) implementations on clusters or general-purpose computing on graphics processing units (GPGPU) implementations. Application Specific Instruction Set Processors (ASIPs), which are designed for a specific calculation with respect to throughput, required area or power consumption, are particularly suitable for this task. ASIPs are designed for a specific algorithm, but they are programmable with their specific instruction set and so the implementation can be expanded if additional or different requirements are desired.
Furthermore, the multilevel Monte Carlo simulation implemented in Chapter 6.1 is pretty new and in combination with the Heston model it is not been used ad nauseam, instead it is a rewarding research to analyze the potential of a hardware implementation of this algorithm.

2.2 Contribution

This thesis presents the design and implementation of an ASIP for a financial algorithm. Unfortunately, only the implementation of the data path is completely done due to lack of time and the implementation of the control path had to be planned as future work.
It starts with some small introductions in financial options, the required language VHDL and double precision floating point numbers. Accordingly the main part of the thesis succeeds. After describing the design flow of an ASIP in general, a side trip to the reference code gives a view of the structure for the hardware designer. Thereafter, the implemented components are described and the instruction set is documented. Subsequently, the algorithms used to approximate the used mathematical methods are analyzed and the final decisions are announced.

2.3 Related Work

The number of hardware accelerators for option pricing with the Heston model in combination with the Monte Carlo simulation is pretty small. Henning Marxen, Christian de Schryver et al. present the algorithm in "Algorithmic Complexity in the Heston Model: An Implementation View" [18] and describe the hardware implementation in "An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model" [10]. The Heston model

itself is used in more publications, e. g. in "The Little Heston Trap" by Hansjörg Albrechter et al. [2]. There exist conspicuously more publications that give attention to accelerators that use the Monte Carlo simulation, e.g. "Reconfigurable acceleration for Monte Carlo based financial simulation" by G. L. Zhang et al. [31], which depicts a generic implementation of a hardware accelerator for the Monte Carlo simulation and "Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer " by Xiang Tian and K. Benkrid [27], which describes "the fastest ever reported FPGA implementation of this model" [27]. The multilevel Monte Carlo simulation is shown in "Multi-level Monte Carlo path simulation" by Michael B. Giles [11].
To sum up, it can be said that the topic of hardware accelerators for financial simulations, especially the combination of Heston model and Monte Carlo simulation, is certainly not a trite topic, in my opinion is has a high potential instead.

2.4 Organization of this Thesis

This thesis is organized as follows:
Chapter 1 provides a list of the used acronyms while Chapter 2 presents the motivation to write this thesis and the related work.
The actual theme starts with Chapter 3 and introductions to financial options, the hardware description language VHDL and floating point numbers. Chapter 4 covers the main part of this thesis and is split into several sections, which contain a description of the design flow, the reference code, the implemented components and the instruction set of the ASIP and the algorithms required to implement the methods used in the reference code.
Finally, Chapter 5 demonstrates the status quo of the project and the future work that is planned for the time after delivering this bachelorthesis.

3 Preliminaries

3.1 Financial Option

Options are a common bargain in the financial business.

> "An option is the right either to buy or to sell a specified amount or value of a particular underlying interest at a fixed exercise price by exercising the option before its specified expiration date." [1]

The definition above specifies American options while the ASIP is intended to calculate European options. The difference between these two is that the European option can only be redeemed at the expiration date and the American option can be used the whole time until the expiration date.
The right to buy is called *Call Option* while the right to sell is called *Put Option* [1]. The buyer of an option is called the *Long Position* while the seller of an option is called *Short Position*. Figure 3.1 depicts when the exercise price of an option is profitable. The exercise price of the Put Option is 400 while the exercise price of the Call Option is 600. The reward of both options is 100.

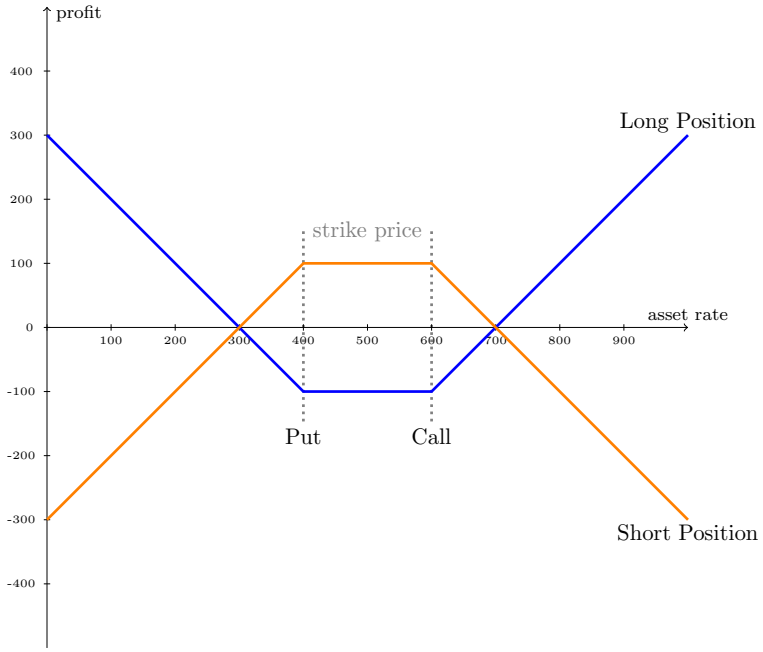

Figure 3.1: Profit Diagram Of Put And Call Option

The presented ASIP can handle call options as well as put options. In addition, it can handle so-called barrier options. Barrier option means that the option is activated or deactivated by the incidence of a certain event. There are two diffent types of barrier options the ASIP can differentiate: Knock-In options and Knock-Out options. In case of the former, the option is deactivated if a predefined barrier is crossed and in case of the latter one, the option is activated if a predefined barrier is reached at least one time. Within this differentiation it can be distinguished between Up, which means that the strike price has to be higher than the barrier price and Down, which means that the strike price has to be lower than the barrier price. Every combination of Down/Up Knock-In/Knock-Out Call/Put is valid. Figure 3.2 shows the progress of two barrier options. The blue one is an Up Knock-In Call option that is deactivated because it crosses the Lower Barrier. The orange line represents a Down Knock-In Put option that is activated because it does not cross the Upper Barrier and its value is always lower than the Upper Barrier.
Furthermore, the option can be a digital option. That means that the payoff is a fixed value, if the option is not deactivated.

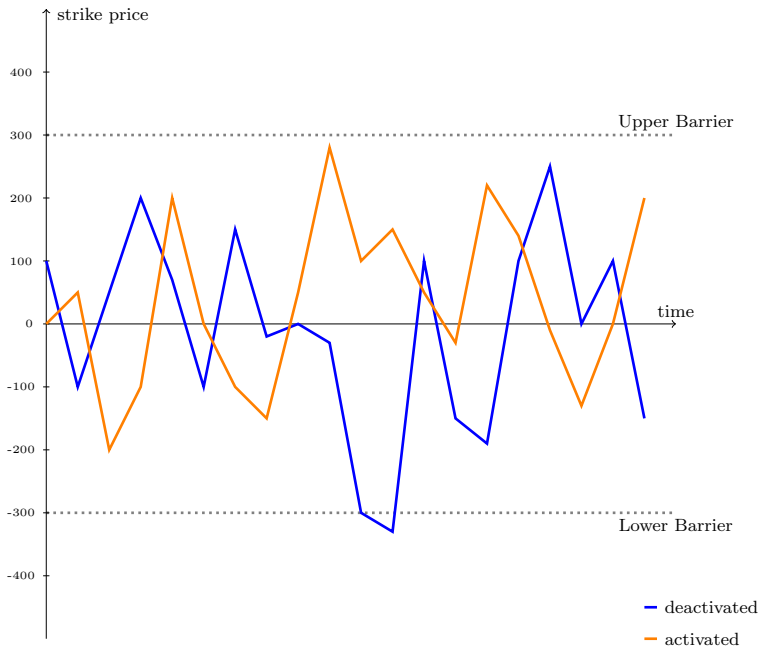

Figure 3.2: Barrier Options

There exist many different ways to calculate the asset price, also called payoff, and the variance of an option. The ASIP designed in this thesis uses the Monte Carlo Simulation, described in Section 3.1.2, of the Heston Model, explained in Section 3.1.1.

3.1.1 Heston Model

The Heston Model resembles the Black-Scholes Model, which is often used for option pricing [3]. But it has the disadvantage that the volatility can be deterministic. This problem cannot occur within the Heston Model, where the the volatility can neither be constant nor deterministic, but follows a stochastic differential equation (SDE).
Figure 3.3 and 3.4 show the price process of the Heston model [2].

$$dX(t) = rX(t)dt + \sqrt{V(t)}X_t dW_X(t)$$

Figure 3.3: Calculation Of Payoff With Heston Model

$$dV(t) = \kappa(\theta - V(t))dt + \sigma\sqrt{V(t)}dW_V(t)$$

Figure 3.4: Calculation Of Variance With Heston Model

W_X and W_V are Brownian motions, r is the rate of the asset, κ is the rate with which $V(t)$ relapses to θ, θ is the expected value of $\lim_{t\to\infty} V(t)$ and σ is the volatility, i. e. the variance of $V(t)$.

3.1.2 Monte Carlo Simulation

The singlelevel Monte Carlo path simulation will not be described in more detail because it is not calculated by the ASIP itself. That is done by an external component, called the Kernel, that is connected to the ASIP.

The multilevel Monte Carlo path simulation attempts to calculate the SDE with the usage of only a few discretization steps. Figure 3.1.2 shows the multilevel Monte Carlo algorithm [3,11] implemented in the reference code in Chapter 6.1.

L is the current simulation level, N_L is the sample size for a given simulation level L, $\hat{P}$ is the temporary payoff, $\hat{V}$ is the temporary variance, N_L^* is the optimal sample size, M is the multilevel constant and ϵ is the required precision.

In addition to the described simulations, the ASIP can handle the multilevel Monte Carlo path simulation with an optimized starting level. As the name tells, this is mainly the multilevel Monte Carlo path simulation described before, but the value of the starting level is not $L = 0$ but optimized before the calculation of the paths begins.
The both multilevel Monte Carlo simulations are detailed described in [18].

1. Set $L = 0$
2. Generate an initial $N_L = 10^4$ samples $\hat{P}_0^{(i)}$ or $\hat{P}_L^{(i)} - \hat{P}_{L-1}^{(i)}$ of the payoff of one asset price simulation. Use this to evaluate $\hat{Y}_L = \frac{1}{N_L} \sum_{i=1}^{N_L} (\hat{P}_L^{(i)} - \hat{P}_{L-1}^{(i)})$ and $\hat{Y}_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \hat{P}_0^{(i)}$, respectively, and estimate their variance $\hat{V}_L$.
3. Set the optimal sample size N_L^* by $N_L^* = \lceil 2\epsilon^{-2} (\sum_{l=0}^{L} \sqrt{V_l h_l^{-1}}) \sqrt{V_l h_l} \rceil$ for $l = 0, 1, ..., L$.
4. If $N_L^* > N_L$, then evaluate the additional $N_L^* - N_L$ samples and update the estimators $\hat{Y}_l$ and $\hat{V}_l$.
5. Set $N_L = N_L^*$.
6. if $L < 2$ increase L by one otherwise
7. Test if the bias criterion $(max\{(M^{2\alpha} - M^{\alpha})^{-1} |\hat{Y}_{L-1}|,\ (M^{\alpha} - 1)^{-1} |\hat{Y}_L|\} < \frac{1}{\sqrt{2}} \epsilon)$ is fulfilled. If it is then the algorithm has finished and the estimator is $\sum_{l=0}^{L} \hat{Y}_l$ if not increase L by one and continue from 2.

Figure 3.5: Multilevel Algorithm Used By Giles [11]

This section introduced some formulas that are going to be calculated on the ASIP presented in this thesis. Since all variables appearing in these formulas are real numbers, the ASIP must be capable of handling floating point numbers. The next Section gives an introduction and overview of this number representation.

3.2 Double Precision Floating Point Numbers

My ASIP works with double precision floating point numbers [16,19,20].
A double precision floating point number is a binary number (base 2) that is built up of 64 bits segmented into a sign, which requires one bit, an exponent, which requires eleven bits and a mantissa, which requires the remaining 52 bits. The sectioning is depicted in Figure 3.6.

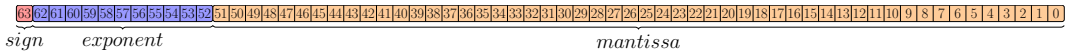

Figure 3.6: Double Precision Floating Point Format

In the following I use the generic notation shown in Figure 3.7, where B stands for the base, e for the number of digits used to represent the exponent and m for the number of digits used to represent the mantissa. According to the IEEE 754 standard that describes double precision floating point numbers, the usage of the so-called hidden bit increases the accuracy of the floating point numbers by an additional bit. Since most numbers start with a "1", this bit is neglected such that an additional bit of the mantissa can be stored.

$$\langle [x_n, x_{n-1}, ..., x_0] \rangle^{\mathbb{R}}_{B,m,e}$$

Figure 3.7: Notation Of Floating Point Numbers [24]

To get the decimal value of a given floating point number, the formula depicted in Figure 3.8 can be used. Within the formula there is a β used. This $\beta = (B^e - 1)\ div\ 2$ is called the bias and is required to be able to represent negative exponents.

$$\langle [x_n, ..., x_0] \rangle^{\mathbb{R}}_{B,m,e} = (-1)^{x_n} \cdot (\langle [x_{m-1}, ..., x_0] \rangle^{\mathbb{N}}_{B} \cdot B^{1-m}) \cdot B^{\langle [x_{n-1}, ..., x_m] \rangle^{\mathbb{N}}_{B} - \beta}$$

Figure 3.8: Conversion Of Floating Point Numbers Into Decimal Numbers [24]

3.2.1 Addition and Subtraction

To add or subtract two floating point numbers, the calculation rules depicted in Figure 3.9 are used. In the following, the sum of x and y is calculated. The mantissas are called M_x and M_y, the exponents are called E_x and E_y and the signs are called S_x, respectively S_y. The precondition for this calculation is that $E_y \leq E_x$ holds. If that is not the case, the floating point numbers are just interchanged so that $y+x$ is computed instead of $x+y$. This yields the same result since the commutative law holds for additions and subtractions of real numbers, that are actually handled here.

$$
\begin{aligned}
& M_x \cdot B^{E_x-\beta} + M_y \cdot B^{E_y-\beta} \\
= \quad & M_x \cdot B^{E_x-\beta} + (M_y \cdot B^{E_y-E_x}) \cdot B^{E_x-\beta} \\
= \quad & (M_x + M_y \cdot B^{E_y-E_x}) \cdot B^{E_x-\beta}
\end{aligned}
$$

Figure 3.9: Addition Of Floating Point Numbers [24]

Figure 3.9 shows that the mantissa M_y has to be shifted about $E_x - E_y$ digits to the right before it can be added to the mantissa M_x. After the calculation is done, the floating point number has to be normalized and rounded to adapt it to the required format.
An elaborate example is given in Figure 3.13 of Section 3.2.4.

3.2.2 Multiplication

The multiplication of two floating point numbers is quite easy in comparison to the addition. Figure 3.10 demonstrates that the mantissas only have to be muliplied and the exponents have to be added. In the end, the bias has to be subtracted from the sum of the exponents to avoid that it is added twice. After the calculation is done, the floating point number has to be normalized to adapt it to the required format [25]. Finally, the sign is determined by applying the XOR operation to the signs of the inputs: $S_{\sum} = S_x \oplus S_y$.

$$
\begin{aligned}
& (M_x \cdot B^{E_x-\beta}) \cdot (M_y \cdot B^{E_y-\beta}) \\
= \quad & M_x \cdot M_y \cdot B^{E_x-\beta} \cdot B^{E_y-\beta} \\
= \quad & M_x \cdot M_y \cdot B^{E_x-\beta+E_y-\beta} \\
= \quad & M_x \cdot M_y \cdot B^{E_x+E_y-2\beta}
\end{aligned}
$$

Figure 3.10: Multiplication Of Floating Point Numbers

The example shown in Figure 3.14 of Section 3.2.4 exemplifies the previously declared rules.

3.2.3 Division

The division of two floating point numbers is accomplished similar to the multiplication.
In contrast to the multplication the mantissas have to be divided and the exponents have to be subtracted. Furthermore the bias must not be subtracted from the new exponent, it have to be added instead, because the subtraction of the exponents leads to the extinction of the bias. After the calculation is done, the floating point number has to be normalized to adapt it to the required format [25].
The previously claimed is formally evidenced in Figure 3.11.
The example shown in Figure 3.12 of Section 3.2.4 exemplifies the previously declared rules.

$$\begin{aligned} & \frac{M_x \cdot B^{E_x-\beta}}{M_y \cdot B^{E_y-\beta}} \\ = \; & \frac{M_x}{M_y} \cdot B^{E_x-\beta-(E_y-\beta)} \\ = \; & \frac{M_x}{M_y} \cdot B^{E_x-E_y} \end{aligned}$$

Figure 3.11: Division Of Floating Point Numbers

3.2.4 Examples

The figures 3.13, 3.14 and 3.12 demonstrate the previously explained operations on floating point numbers. To simplify their understanding and to keep the operations generic, the numbers do not contain a hidden bit.

The two floating point numbers $x = \langle[0,1,2,2,1]\rangle^{\mathbb{R}}_{3,2,2}$ and $y = \langle[0,0,2,1,1]\rangle^{\mathbb{R}}_{3,2,2}$ have to be divided.
The first step is to divide the mantissas.

$$\begin{array}{cccccccc} 2 & 1 & : & 1 & 1 & = & 1 & 2 \quad ... \\ 1 & 1 & & & & & & \\ \hline 1 & 0 & 0 & & & & & \\ 0 & 2 & 2 & & & & & \\ \hline & 0 & 1 & & & & & \end{array}$$

$$\Rightarrow M_{\sum} = \langle[2,1]\rangle^{\mathbb{N}}_3 : \langle[1,1]\rangle^{\mathbb{N}}_3 = \langle[1,2]\rangle^{\mathbb{N}}_3$$

The next step is to subtract the exponents and to add the bias.

$$\begin{array}{rl} 1\ 2 & \textit{exponent of } x + \textit{bias} \\ -\ 0\ 2 & \textit{exponent of } x + \textit{bias} \\ \hline 1\ 0 & \\ +\ 1\ 1 & \textit{bias} \\ \hline 2\ 1 & \end{array}$$

$$\Rightarrow E_{\sum} = \langle[1,2]\rangle^{\mathbb{N}}_3 - \langle[0,2]\rangle^{\mathbb{N}}_3 - \langle[1,1]\rangle^{\mathbb{N}}_3 = \langle[2,1]\rangle^{\mathbb{N}}_3$$

The last step is to determine the sign:

$$S_{\sum} = S_x \oplus S_y = 0 \oplus 0 = 0$$

Finally, the computed segments can be put together:

$$\Rightarrow \frac{\langle[0,1,2,2,1]\rangle^{\mathbb{R}}_{3,2,2}}{\langle[0,0,2,1,1]\rangle^{\mathbb{R}}_{3,2,2}} = \langle[0,2,1,1,2]\rangle^{\mathbb{R}}_{3,2,2}$$

Figure 3.12: Example: Division Of Floating Point Numbers

The two floating point numbers $x = \langle [0,2,2,1,0] \rangle^{\mathbb{R}}_{4,2,2}$ and $y = \langle [1,2,1,1,2] \rangle^{\mathbb{R}}_{4,2,2}$ have to be added.
As it can be seen, they have both the base four, and the mantissas and the exponents require two digits each.
In this case, the condition $E_y < E_x$ holds, so it can directly be calculated

$$E_x - E_y = \langle [2,2] \rangle^{\mathbb{N}}_4 - \langle [2,1] \rangle^{\mathbb{N}}_4 = \langle [1] \rangle^{\mathbb{N}}_4.$$

Therefore the mantissa of y has to be shifted one digit to the right. In order to not to loose the precision induced by the coming addition, x is shifted one bit to the left instead.
Although the actual operation is an addition, the fact that $S_x \oplus S_y = 0 \oplus 1 = 1$ holds leads to the consequence that the mantissas have to be subtracted.

	1 0 0	*mantissa of x, expanded about one digit*
−	1 2	*mantissa of y*
	0 2 2	

The first step to gain the final mantissa is to normalize the result of the subtraction. Normalization means to shift the result such that the mantissa requires exactly m digits, i. e. 2 digits in this example. The direction of the required shifting operation is determined by the operation that has been applied. Particularly, the given example requires the mantissa to be shifted to the left. Hence, the mantissa of the result is

$$M_{\sum} = \langle [2,2] \rangle^{\mathbb{N}}_4.$$

The normalization affects the exponent, because the mantissa of the result was shifted to the left, the resulting exponent must be corrected. Because of that the exponent of the sum is equal to the exponent of x minus 1.

$$E_{\sum} = E_x - 1$$

The last thing to do is to determine the sign of the sum. The mantissa did not become negative and x was a positive floating point number, so the sum is also positive. Then the results can be put together to get the desired sum.

$$S_{\sum} = 0$$

$$\Rightarrow \langle [0,2,2,1,0] \rangle^{\mathbb{R}}_{4,2,2} + \langle [1,2,1,1,2] \rangle^{\mathbb{R}}_{4,2,2} = \langle [0,2,1,2,2] \rangle^{\mathbb{R}}_{4,2,2}$$

Figure 3.13: Example: Addition Of Floating Point Numbers

The two floating point numbers $x = \langle[0,1,1,2,1,0]\rangle^{\mathbb{R}}_{3,3,2}$ and $y = \langle[1,0,2,1,0,2]\rangle^{\mathbb{R}}_{3,3,2}$ have to be multiplied.
The first step is to multiply the mantissas.

$$\begin{array}{ccccccc} 2 & 1 & 0 & \cdot & 1 & 0 & 2 \\ \hline & & 2 & 1 & 0 & & \\ & & & & & 0 & \\ & & & 1 & 1 & 2 & 0 \\ \hline & & 2 & 2 & 1 & 2 & 0 \end{array}$$

$$\Rightarrow M_{\sum} = \langle[2,1,0]\rangle^{\mathbb{N}}_3 \cdot \langle[1,0,2]\rangle^{\mathbb{N}}_3 = \langle[2,2,1,2,0]\rangle^{\mathbb{N}}_3$$

The resulting mantissa has already exactly one digit on the left of the decimal point, and therefore, it does not have to be normalized. Nevertheless, the mantissa consists of more than the required three digits and needs to be rounded. Using *round to nearest even*, which is state-of-the-art rounding method in floating point, the final mantissa is exposed to be

$$\Rightarrow M_{\sum} = \langle[2,2,2]\rangle^{\mathbb{N}}_3$$

The next steps are to add the exponents and to subtract the bias.

$$\beta = (3^2 - 1)\ div\ 2 = (9-1)\ div\ 2 = 4 = \langle[1,1]\rangle^{\mathbb{N}}_3$$

$$\begin{array}{rl} 1\ 1 & \textit{exponent of } x \textit{ + bias} \\ +\ 0\ 2 & \textit{exponent of } y \textit{ + bias} \\ \hline 2\ 0 & \\ -\ 1\ 1 & \textit{bias} \\ \hline 0\ 2 & \end{array}$$

$$\Rightarrow E_{\sum} = \langle[1,1]\rangle^{\mathbb{N}}_3 + \langle[0,2]\rangle^{\mathbb{N}}_3 - \langle[1,1]\rangle^{\mathbb{N}}_3 = \langle[0,2]\rangle^{\mathbb{N}}_3$$

The last step is to determine the sign:

$$S_{\sum} = S_x \oplus S_y = 0 \oplus 1 = 1$$

Finally, the computed segments can be put together:

$$\Rightarrow \langle[0,1,1,2,1,0]\rangle^{\mathbb{R}}_{3,3,2} + \langle[1,0,2,1,0,2]\rangle^{\mathbb{R}}_{3,3,2} = \langle[1,1,1,2,2,2]\rangle^{\mathbb{R}}_{3,3,2}$$

Figure 3.14: Example: Multiplication Of Floating Point Numbers

3.3 VHDL

By the reason that many components of the ASIP designed in this thesis, which more precisely means all that will not be constructed with LISA, a processor designer tool from Synopsys [21], are written in VHDL [4,5,7], the following chapter will give a small introduction to this hardware description language.

VHDL is used to describe algorithms that are intended to run on hardware. It is based on an event driven model of computation, which means that it supports the detection, modification of events and the reaction to them. An event can be seen as a change in the state of a signal, e. g. the clock signal. To test the implementation, it can be synthesized and simulated [6,17]. The verification is an very important topic e. g. in avionics, to ensure the correctness of an implemented hardware design [23,28], especially in embedded system. Later adjustments on the completed hardware is a very expensive task, if it is even possible without changing the entire hardware structure. In many cases it is easier and cheaper to discard the old hardware and start from scratch again. Due to lack of time, the components of my processor are not formally verified, but thoroughly tested.

in_1	in_2	*carry*	*sum*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 3.15: Truth Table Of Half Adder [22, page 189]

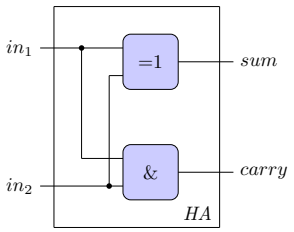

Figure 3.16: Half Adder [22, page 189]

Figure 3.15 depicts the truth table of a half adder. It will serve as a running example to explain how to implement digital circuits in VHDL. The function respresented by this truth table takes two arguments and returns a tuple consisting of two values. The outputs are gained by applying an XOR and an AND to the inputs as shown in the digital circuit in Figure 3.16. The truth table shows that the half adder has two input ports, in_1 and in_2, and two output ports, *carry* and *sum*. Furthermore, it can be seen, that the boolean formula $in_1 \wedge in_2$ represents the *carry* and the boolean formula of the *sum* is $in_1 \oplus in_2$. Hence, the digital circuit of this half adder can be built by using one AND gate and one XOR gate (see Figure 3.16).

A VHDL implementation of this digital circuit is shown in Listing 3.1.
The four ports of the circuit and one port for the clock signal are defined within the entity. The clock signal and the two bits that have to be added are inputs, which is illustrated by the “in” while *carry* and *sum* are outputs, which is illustrated by the “out”. In this case, only bits are required, therefore “STD_LOGIC” can be used as type of the ports.
The behavior of the implemented component is described within the architecture part of the VHDL code. To compute the required operations AND and XOR and connect the results to

the output ports, a process has to be defined. If and only if the clock signal changes from *low* to *high*, the result of $in_1 \wedge in_2$ is connected to the port representing the carry and $in_1 \oplus in_2$ is connected with the port representing the sum.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity half_adder is
5    Port ( clock : in  STD_LOGIC;
6      in1 : in  STD_LOGIC;
7      in2 : in  STD_LOGIC;
8      carry : out  STD_LOGIC;
9      sum : out  STD_LOGIC);
10 end half_adder;
11
12 architecture Behavioral of half_adder is
13
14 begin
15 calc: process (clock) is
16 begin
17   if rising_edge(clock) then
18     carry := in1 and in2;
19     sum := in1 xor in2;
20   end if;
21 end process calc;
22
23 end Behavioral;
```

Listing 3.1: Code Of Half Adder

in_1	in_2	$carry_{in}$	$carry_{out}$	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.17: Truth Table Of Full Adder

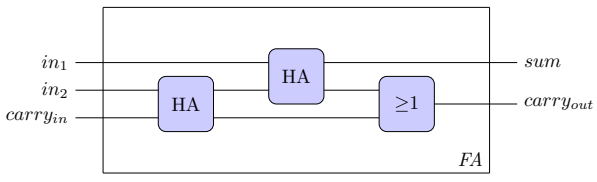

Figure 3.18: Full Adder [22, page 190]

Figure 3.17 shows the truth table of a full adder and Figure 3.18 shows a corresponding digital circuit using two half adders and one OR gate.
An implementation of this digital circuit is shown in Listing 3.2.
As done before, the five ports depicted in the truth table, respectively the digital circuit, and the port for the clock signal are defined within the entity, where in_1, in_2, $carry_{in}$ and *clock* are marked as inputs while $carry_{out}$ and *sum* are marked as outputs.

The half adder implemented before is used twice and so it has to be included as a component inside the architecture of the full adder. Furthermore, it has to be instantiated twice because two half adders are required. The wiring shown in Figure 3.18 is defined within the particular port map of each half adder.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity full_adder is
5    Port ( clock : in  STD_LOGIC;
6      in1 : in  STD_LOGIC;
7      in2 : in  STD_LOGIC;
8      carry_in : in  STD_LOGIC;
9      carry_out : out  STD_LOGIC;
10     sum : out  STD_LOGIC);
11 end full_adder;
12
13 architecture Behavioral of full_adder is
14
15 ----------------------------------------------------
16 --            COMPONENT HALF-ADDER                --
17 ----------------------------------------------------
18 component half_adder
19   port (
20     clock : in  STD_LOGIC;
21     in1 : in  STD_LOGIC;
22     in2 : in  STD_LOGIC;
23     carry : out  STD_LOGIC;
24     sum : out  STD_LOGIC
25   );
26 end component;
27
28 ----------------------------------------------------
29 --                  SIGNALS                       --
30 ----------------------------------------------------
31 signal carry0_internal : STD_LOGIC;
32 signal carry1_internal : STD_LOGIC;
33 signal summand_internal : STD_LOGIC;
34
35 begin
36
37 ----------------------------------------------------
38 --       INSTANTIATION OF HALF-ADDERS             --
39 ----------------------------------------------------
40 ha0 : half_adder
41   port map (
42     clock => clock,
43     in1 => in1,
44     in2 => in2,
45     carry => carry0_internal,
46     sum => summand_internal
47   );
```

```
48 ha1 : half_adder
49   port map (
50     clock => clock,
51     in1 => summand_internal,
52     in2 => carry_in,
53     carry => carry1_internal,
54     sum => sum
55   );
56
57 calc: process (clock) is
58 begin
59   if rising_edge(clock) then
60     carry_out := carry0_internal or carry1_internal;
61   end if;
62 end process calc;
63
64 end Behavioral;
```

Listing 3.2: Code Of Full Adder

4 Design and Implementation

4.1 Design Process

The design flow of an ASIP is shown in Figure 4.1. Each step has to be completed considering the optimization goal, which is typically either the throughput or the energy consumption or the required area. The ASIP of this thesis is designed and optimized with respect to the throughput.

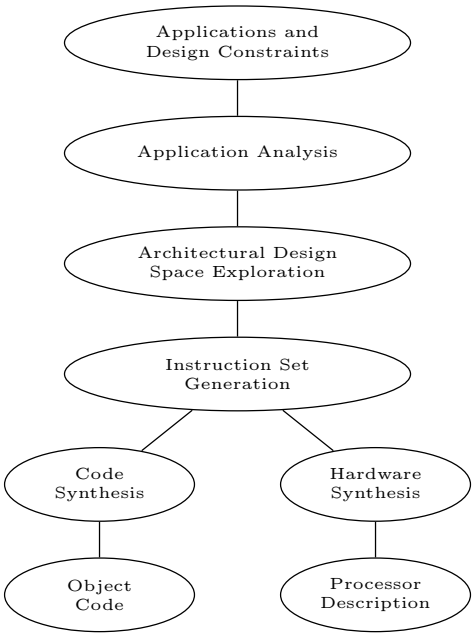

Figure 4.1: ASIP Design Flow [15]

First of all the algorithm has to be implemented in a high level language, e.g. C++ in this thesis, to indentify the required operations. The algorithm was already implemented, but the initially implementation had some small bugs and the naming of the variables was not really meaningful and therefore it had to be reimplemented. The resulted implementation is attached to this thesis in Chapter 6.1. The course of the implementation is described in more detail in Chapter 4.2.
After the implementation is done, the real design process starts. I analyzed the code in reference to the used operations, especially frequently used combinations of operations, required methods and the memory requirements. The operations lead to the instruction set presented

in Chapter 4.4. Since I did not found combinations of operations that are used many times I decided not to include operations for any combination in the instruction set. The instruction set then leads to the required arithmetic units that build the main part of the ALU, which is presented in Chapter 4.3.2. The memory requirements lead to the necessary number of registers which are organized in the register file presented in Chapter 4.3.1.

4.2 Reference Code

The processor is designed for the algorithm realized by the code shown in Listing 6.1. The complete processor design concerned in this thesis is based on this reference code.

As it is described in Chapter 4.1, the algorithm was already implemented in C++. Due to some bugs and a lack of meaningful naming of variables, a reimplementation was unavoidable to provide a reliable base for the presented processor.
In course of this reimplementation, the purely procedural description is replaced by an object-oriented one. The classes allow a simplified derivation of the hardware components in the given context.

There are three different ways to calculate the option price and variance: singlelevel calculation, multilevel calculation and optimized multilevel calculation (see Chapter 3.1).

If singlelevel calculation has to be used, the kernel computes the results and no pre- or postcalculations are necessary (see Listing 6.1, lines 13-20).
This case is not of interest for the design of the processor, because there is no chance to optimize this part within the control-unit - every optimization concerns the code of the kernel, which is an external element and not part of this thesis.

The implementation of the multilevel Monte Carlo simulation can be seen in Listing 6.1, lines 22 to 154.
The multilevel Monte Carlo simulation with optimized starting level is implemented in the lines 158 to 255 of Listing 6.1.

All the optimizations done in this thesis refer to the implementations of the multilevel Monte Carlo simulation with and without optimized starting level. The operations used in this implementation lead to the instruction set depicted in Chapter 4.4 and approximations for all methods that are used within the reference code are analyzed in Chapter 4.5.

4.3 Components

4.3.1 Registerfile

Every processor has to store values in some way. This can be realized with different mediums, which are described in the following.
In order to avoid the slowdown of a calculation, as it is shown in Figure 4.2, registers are the best way to store data. But registers are not only the fastest medium, they are also the most expensive medium that is available. Therefore, the number of registers is kept as small as possible and so the capacity is quite small. The next fastest memory medium is the processor's cache, which is mainly used to reduce the access time of the most frequently used data. The random access memory (RAM) is the third medium relating to the throughput. It is a volatile memory that has a worst case throughput of constant time. It is considerably faster than the hard disk, which is the fourth medium in matters of the random access time. The hard disk has a very high latency in comparison to the previously introduced mediums. But it has a benefit, the others do not have: the capacity. Hard disks are a pretty profitable way to store a lot of data.
Figure 4.2 visualizes the correlation between throughput and capacity in reference to the previous described memory mediums.

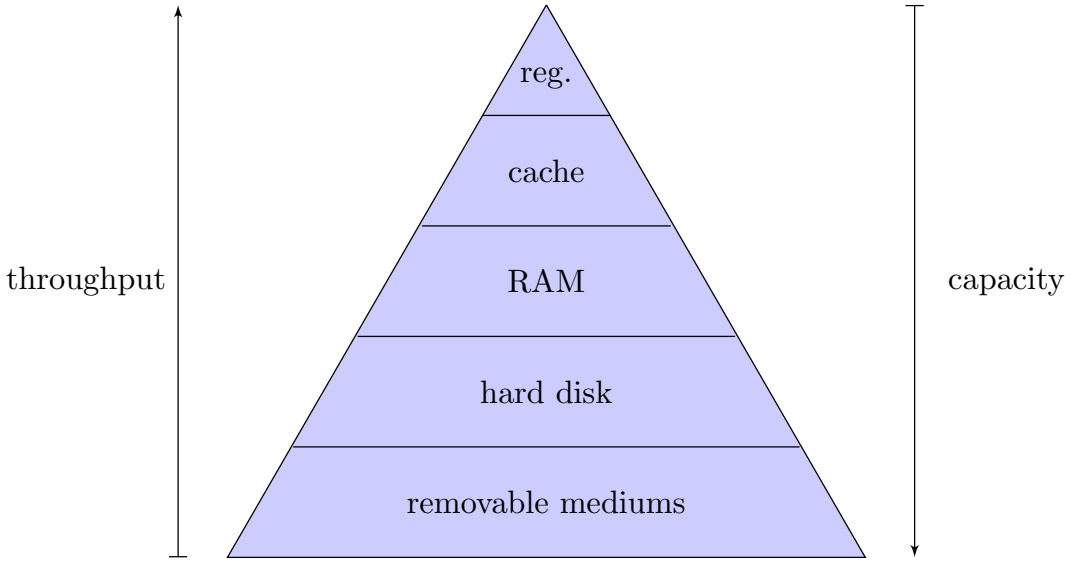

Figure 4.2: Storage Comparison

By the reason that registers are the fastest way to store data, the ASIP designed in this thesis requires this medium. Other methods would decrease the performance enormously and the main purpose of my ASIP design is to afford a fast calculation.
Registers are collocated in a register file that is directly connected to the ALU and allows to access the data in one clock cycle.
Figure 4.3 depicts the register file implemented in this thesis. My implementation is fully

generic with respect to the number of registers and their bit width to make future enhancements of the processor and its data path possible. Currently, only registers with 64 bit are required.

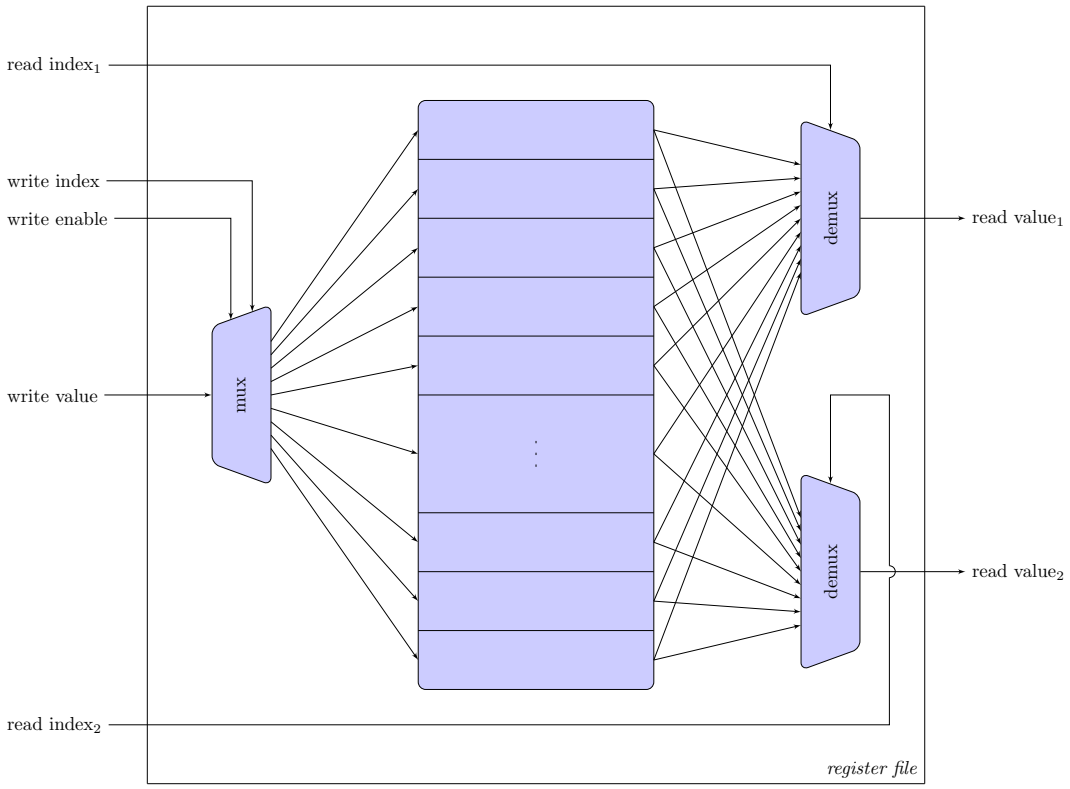

Figure 4.3: Register File

The inputs are three bitvectors with $\lceil log_2(number\_of\_registers) \rceil$ bits for addressing called “read index$_1$”, “read index$_2$” and “write index”, one bitvector with an arbitrary width which is called “write value” and two control bits, one used for activating the write mode called "write enable" and one called “clock”. The last one is, self-explaining by its name, required for the clock signal. The more precise usage of the other ones will be described in the following.

The register file supports the simultaneous reading of the content of two registers and the writing into one register per clock cycle.
To write into a register, the bit called “write enable” in Figure 4.3 has to be set on *high*. The double-precision floating point number called “write value” is written into the register addressed by the bitvector called “write index” in Figure 4.3.
To read the content of two registers, the registers have to be addressed by the bitvectors called “read index$_1$” and “read index$_2$” in Figure 4.3. The desired data is then available at the ports called “read value$_1$” and “read value$_2$” after one clock cycle.

4.3.2 Arithmetic Logic Unit

The previous section describes how information can be stored. The component that can deal with this data is introduced in this section. This component is called Arithmetic Logic Unit (ALU) and is a central element of the processor. To this end, the design of the ALU was a main aspect during the project.

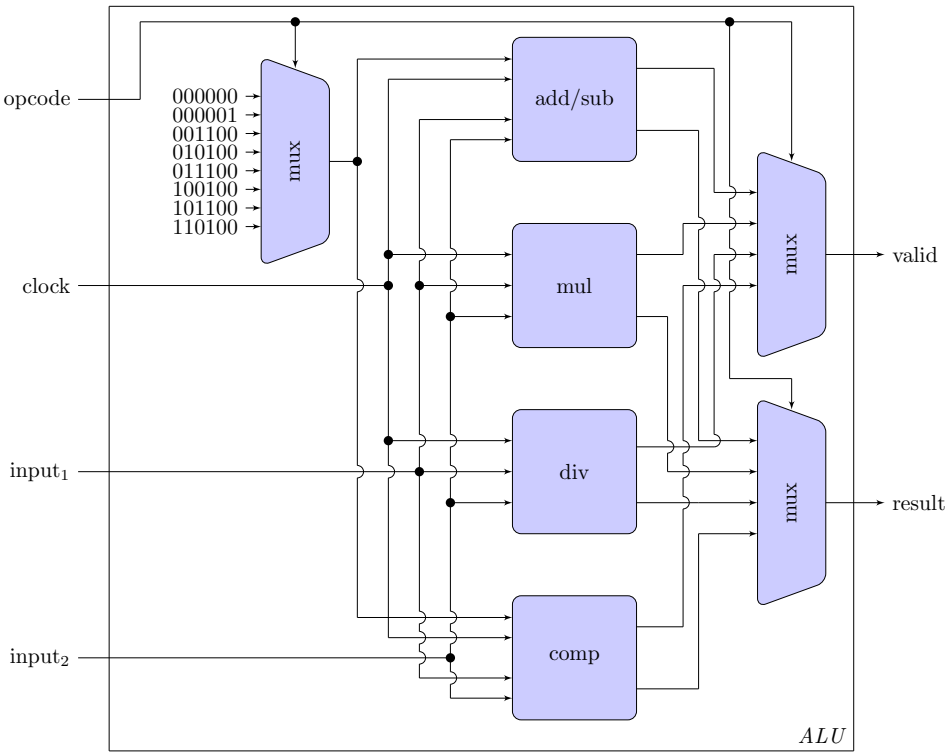

Figure 4.4: First ALU Concept

The reference code shown in Listing 6.1 is analyzed and it is searched for recurring combinations of operations that could be combined into one special arithmetic unit. There are some candidates but in the end the costs would exceed the benefits and so I decided to use the instruction set presented in Chapter 4.4.
Depending on the instruction set, I determined the required arithmetic units and started to work out some concepts of the ALU design. These concepts are presented in the following.

One of the considered concepts of the ASIP's ALU is depicted in Figure 4.4.
The inputs are the opcode given as a bitvector, a clock signal given as one bit and two double-precision-floating-point numbers given as bitvectors with 64 bit each.
The ALU contains four arithmetic units provided by XILINX: one for addition and subtraction ("add/sub"), one for multiplication ("mul"), one for division ("div") and one for comparison ("comp").
Furthermore, there are three multiplexers within the ALU whose relevance will be explained

in the accurate description below.

The opcode defines the operation which should be executed and it has to be converted into a six-digit-code if an addition, subtraction or comparison should be accomplished. This is done by the multiplexer in the top left corner of the structure shown in Figure 4.4. The list of all available opcodes is given in Figure 4.5.

operation	code
add	00000000
subtract	00000001
unordered	00000100
less than	00001100
equal	00010100
less than or equal	00011100
greater than	00100100
not equal	00101100
greater than or equal	00110100

Figure 4.5: Scheme For Operation Conversion [30, page 8]

No matter what calculation has to be done, all units receive the same input data. The multiplication and the division unit only get the clock-signal and the two floating-point numbers for calculation. They do not need an opcode, because they can only accomplish one operation. The addition/subtraction unit and the comparison unit get the same inputs plus the converted opcode.

Each of these units has two outputs: one result-signal and one valid-signal, which shows if the signal at the result-output is valid.
As explained before, all four arithmetic units provide a result- and a valid-signal, but only one of these four result- and one of these four valid-signals is required. To choose which of them has to be transported to the output, I introduced the two multiplexers on the right side of Figure 4.4, that are driven by the opcode.

The drawback of this concept is that every arithmetic unit has an individual runtime but it would be necessary that an external component determines, when a result can be read from the output ports. The input ports have to hold their configuration until the required result is available at the output ports. That leads to an inadmissible slow down of the processor. I therefore started to optimize this draft with respect to achievable performance and easy controlling. This new concept is also the final one and depicted in Figure 4.6.

This ALU has six input ports and one output port. The inputs are one control bit, called "clock", three bitvectors with 64 bit, called "$input_1$", "$input_2$" and "load_value" and two bitvec-

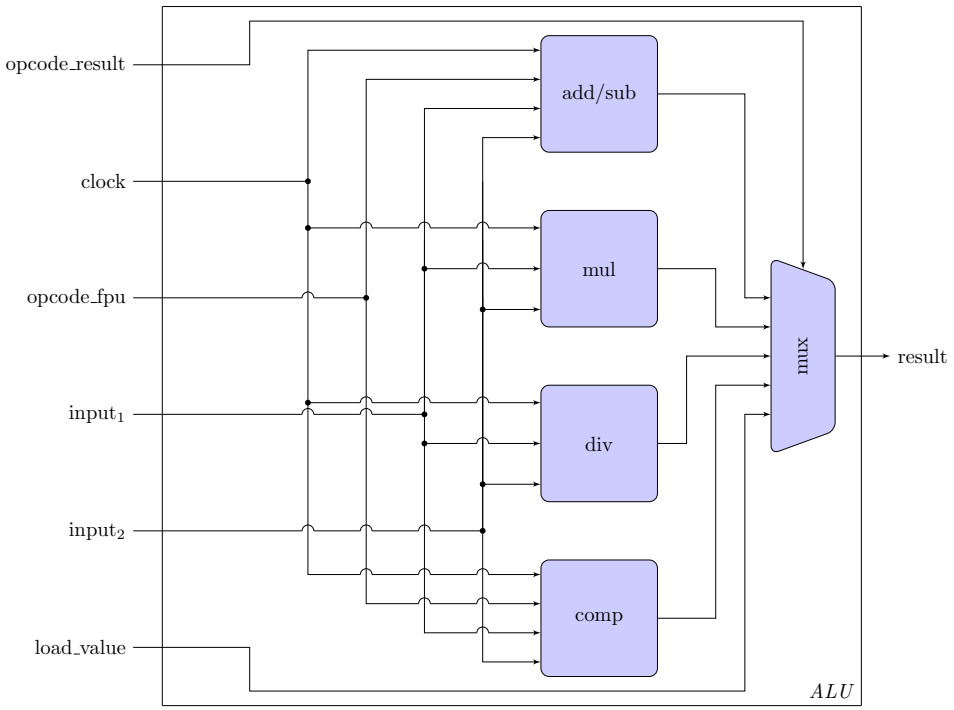

Figure 4.6: Final ALU Concept

tors with eight bits, called "opcode_fpu" and "opcode_result". The output port called "result" holds the result of the computation.
There are four arithmetic units, one for addition and subtraction, one for multiplication, one for division and one for comparison within the ALU and one multiplexer. Each arithmetic unit is able to execute a specific set of operations. These units are pipelined and each of them has a specific pipeline length. To reduce the control complexity, the pipeline length of the addition and comparison unit are adapted to the pipeline length of the multiplication unit. To realize this, a buffer with two stages is attached to the addition and substraction unit and a buffer with eight stages is attached to the comparison unit. After that modification it is known that every arithmetic unit, except the division unit, requires exactly ten clock cylces to compute a result.
The addition and subtraction unit and the comparison unit need the specific opcode shown in Figure 4.5 as before. But instead of converting the actual opcode, I decided to use an opcode that has the same coding for the operations, this two units are responsible for.
After ten clock cycles, the opcode of the operation, the result is available at the result port of the particular unit, is connected to the port called "opcode_result" in Figure 4.6. The multiplexer is driven by this opcode and connects the port "result" to the result port of the right arithmetic unit.
If a value has to be loaded into the registers (see Figure 4.3), the value that has to be loaded has to be connected to the port called "load_value" and the port "opcode_result" has to be connected to the opcode used for coding the load instruction at the same time. Another special

case is the division. The arithmetic unit provided by XILINX for the division takes 68 clock cycles [30] to compute a result. If a division has to be accomplished, the first pipeline stage of the control unit has to be empty 58 clock cylces after the division has been loaded into it, i. e. that the 58th instruction after a division has to be a NOP. If that would not be done, it could happen that two arithmetic units are ready with their computation and offer their results at the same time. This would lead to the problem that at least one of the results get lost.

4.4 Instruction Set

Figure 4.7 shows the instruction set of the ASIP designed in this thesis.

Instruction		Opcode	Description
ADD	$reg_z\ reg_x\ reg_y$	00000000	addition: $reg_z = reg_x + reg_y$
SUB	$reg_z\ reg_x\ reg_y$	00000001	subtraction: $reg_z = reg_x - reg_y$
MUL	$reg_z\ reg_x\ reg_y$	01000000	multiplication: $reg_z = reg_x * reg_y$
DIV	$reg_z\ reg_x\ reg_y$	10000000	division: $reg_z = \frac{reg_x}{reg_y}$
SL	$reg_z\ reg_x\ i$	00000010	shift value of register reg_x about i bits to the left
SR	$reg_z\ reg_x\ i$	00000010	shift value of register reg_x about i bits to the right
LFP	$reg_z\ i$	11000000	load the floating point number i in register reg_z
LOAD	$reg_z\ address$	11100000	load value stored at address *address* in register reg_z
SAVE	$reg_x\ address$	11110000	load value of register reg_x and store it at adress *address*
BLT	$reg_x\ reg_y\ label$	00001100	branch to *label* if $reg_x < reg_y$
BEQ	$reg_x\ reg_y\ label$	00010100	branch to *label* if $reg_x == reg_y$
BLE	$reg_x\ reg_y\ label$	00011100	branch to *label* if $reg_x <= reg_y$
BGT	$reg_x\ reg_y\ label$	00100100	branch to *label* if $reg_x > reg_y$
BNE	$reg_x\ reg_y\ label$	00101100	branch to *label* if $reg_x \neq reg_y$
BGE	$reg_x\ reg_y\ label$	00110100	branch to *label* if $reg_x >= reg_y$
JMP	*label*	00111111	unconditional jump to *label*

Figure 4.7: Instruction Set

As it is described in Section 4.3.2, the reference code depicted in Chapter 6.1 is analyzed with respect to the used operations. That leads to the operations that are definitely necessary. Furthermore, it is searched for recurring combinations of operations so that they can be put together in one as it is done with the multiply/accumulate (MAC) operations [14].
Since I did not found any combination of operations that came into consideration, the instruction set is built from the required standard operations inside the reference code and the implementations of the mathematical methods in Chapter 4.5. At the moment, the these methods are implemented in the ASIPs assembler language, but maybe they will be defined as instructions later.

The opcode consists of eight bits, which is quite more than necessary to encode the defined operations. The reason for this choice is the opcode of the arithmetic units provided by XILINX, which is shown in Figure 4.5. If the ASIPs opcode would not correspond the special opcode by XILINX, it had to be converted for some of the arithmetic units provided by XILINX. This conversion would decrease the performance of the designed ALU. Another advantage of the longer opcode is the ability for an easy enhancement of the supplied instructions.

4.5 Algorithms

Within the reference code shown in Listing 6.1 there are some methods used. To be able to compute them with my ASIP, I analyzed different algorithms with respect to the required arithmetic operations.
To validate the algorithms, I implemented them in C++ [13,29]. Later they will be written in the ASIP's assembler language and maybe they can be called with an pseudo instruction in the future.

In the following the several algorithms are described and the reasons for the final choices are explained.

4.5.1 Absolute Value

The C++ method *abs* returns the absolute value of a given number.

$$abs\ (x) = \begin{cases} -x & if\ x < 0 \\ x & else \end{cases}$$

By the reason that floating-point numbers have a bit representing the sign, it is no problem to get the positive value of a floating-point number. It is gained by simply cleaning the sign bit, i. e. setting it on *low*. Particularly, the function *abs* is implemented by using the bitwise AND with a mask where all bits, but the most-significant, are set on $high$.

4.5.2 Round Up

The C++ method *ceil* returns the least integer number that is not less than the given one, what is called round to infinity.

$$ceil\ (x) = \lceil x \rceil$$

An implementation of the ceil function is shown in Figure 4.1.
Three cases can be distinguished: If the floating point number that has to be rounded has an exponent that is less than zero, the number has to be rounded to one if it is a positive number respectively to zero if it is a negative number. This is implemented in lines eight and nine of the implementation shown in Figure 4.1. The next case is that the number is already an integer, what is checked in lines eleven and twelve. If so, the number can be taken unchanged. The last case that can occur is that the absolute value of the number is greater than one. In this case, the fractional bits, which means the digits standing on the right side of the decimal point, have to be removed. If the floating point number is negative, the modified number is the result. If it is positive, the modified number has to be increased by one. This is done in lines 14 to 19 of Figure 4.1.

```
1  double ceil (double Number)
2  {
3    int Bias = 1023;  // double precision floating point
4
5    uint64_t input;
6     memcpy(&input, &Number, 8);
7
8     int exponent = ((input >> 52) & 2047) - Bias;
9     if (exponent < 0) return (Number > 0);
10
11    int fractional_bits = 52 - exponent;
12    if (fractional_bits <= 0) return Number;
13
14    uint64_t integral_mask = 0xfffffffffff << fractional_bits;
15    uint64_t output = input & integral_mask;
16
17    memcpy(&Number, &output, 8);
18    if (Number > 0 && output != input) ++Number;
19    return Number;
20 }
```

Listing 4.1: Ceil Implementation

4.5.3 Faculty

The C++ method fac returns the faculty of a given number. The faculty of n is the product of the first n natural numbers excluding zero.

$$fac\,(x) = x! = \prod_{i=1}^{x} i$$

The easiest way to compute the faculty is to multiply the numbers sequentially, as it is implemented in Listing 4.2.

```
1  double faculty (double Number)
2  {
3    double Result = 1;
4    while (Number>0)
5    {
6      Result = Result * Number;
7      Number--;
8    }
9    return Result;
10 }
```

Listing 4.2: Simple Faculty Implementation

Figure 4.8 clarifies the seriality that results from the linear dependencies.

The fact, that every multiplication depends on the previous one, makes it impossible to pipeline this computation. That would lead to the problem that the pipeline of my processor has to let run out empty after every multiplication, which would decrease the performance enourmously. To avoid that and to use the benefits of a pipeline, the computation is split

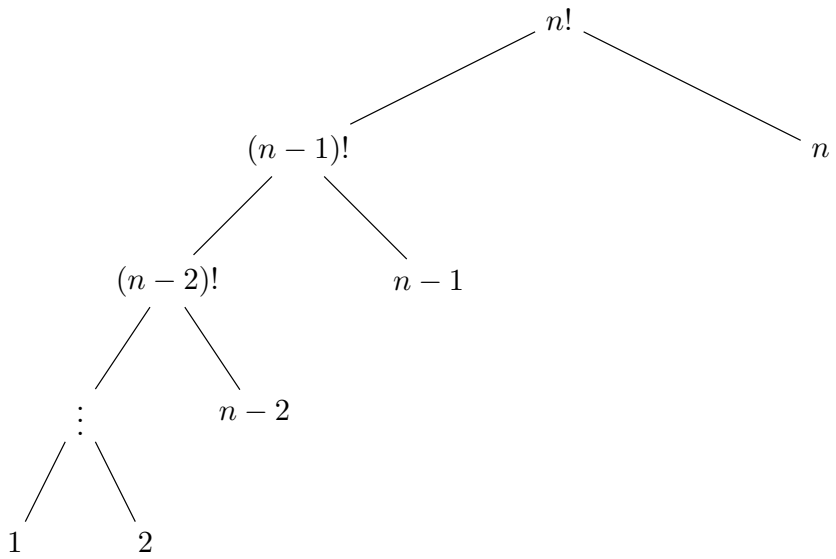

Figure 4.8: Scheme Of Simple Faculty Computation

in as many independent computations as possible and that leads to the structure shown in Figure 4.9 and implemented in C++ in Listing 4.3.

```
1  double faculty_tree (int Number)
2  {
3    double Result[Number];
4    int NumberOfElemets = Number;
5    for (int Index = 0; Index < NumberOfElemets; Index++)
6    {
7      Result[Index] = Index+1;
8    }
9    while (NumberOfElemets>1)
10   {
11     int ResultIndex = 0;
12     int Index = 0;
13     for (Index = 0; Index < NumberOfElemets-1; Index+=2)
14     {
15       Result[ResultIndex] = Result[Index]*Result[Index+1];
16       ResultIndex++;
17     }
18     if (Index = NumberOfElemets)
19       Result[ResultIndex] = Result[NumberOfElemets-1];
20     NumberOfElemets = NumberOfElemets/2 + NumberOfElemets%2;
21   }
22   return Result[0];
23 }
```

Listing 4.3: Fast Faculty Implementation

The first step is to store the leaves of the tree shown in Figure 4.9. This is realized with an array in my implementation (see lines 3 to 8 of Listing 4.3).
After that initialization my algorithm always multiplies two results of the previous tree depth

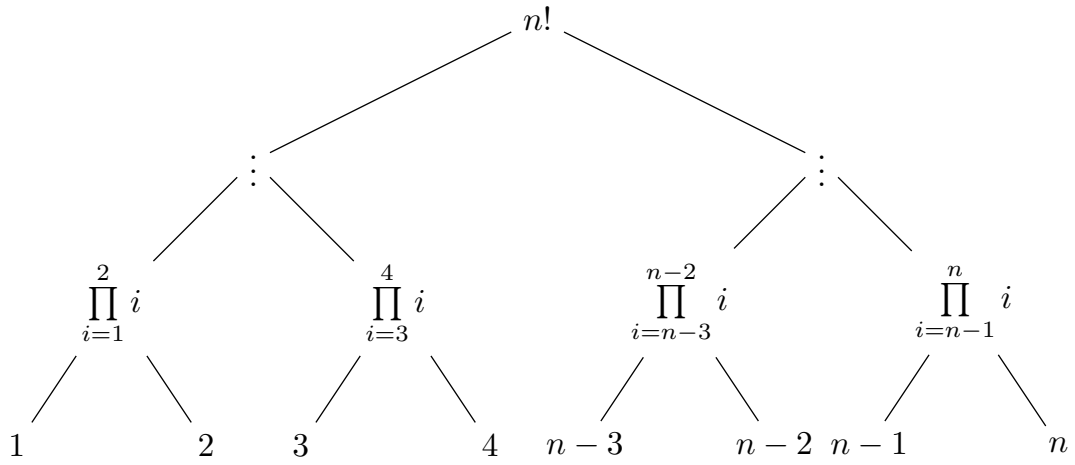

Figure 4.9: Scheme Of Fast Faculty Computation

until only one result is left (see lines 9 to 21 of Listing 4.3). The last result is the faculty of the given parameter.

System	faculty	faculty_tree	Speedup
Intel(R) Core(TM)2 CPU T5500 1.66GHz	8430 ms	880 ms	9.57
Intel(R) Pentium(R) 4 CPU 2.60GHz	17840 ms	780 ms	22.87
Intel(R) Xeon(TM) CPU X5680 3.33GHz	5410 ms	400 ms	13.53
Intel(R) Xeon(TM) CPU 3.60GHz	14230 ms	670 ms	21.24
Intel(R) Core(TM) i7 CPU 860 2.80GHz	5280 ms	480 ms	11
Mean	10238 ms	642 ms	15.95

Figure 4.10: Runtimes Of Faculty Implementations

Figure 4.10 shows the runtimes, several systems need to compute the faculties of all numbers between one and $10,000$ using the two implementations described before.
It can be seen that the algorithm that realizes the tree-based computation of the faculty is much faster than the simple implementation.
The only problem the implementation brings is the additional memory requirement. If the numbers for which the faculty has to be computed of, become large, this algorithm loses some of its benefit because of the slow access time to the memory. The registers of the ASIP can only be used with a couple of numbers, if they become numerous, the values have to be stored

somewhere else. And as it is described in Chapter 4.3.1, every access to another memory type is slower than an access to the registers.

4.5.4 Maximum

The C++ method *max* returns the greater number of two given ones.

$$max\ (x,y) = \begin{cases} x & if\ x > y \\ y & else \end{cases}$$

The instruction set of the ASIP designed in this thesis provides comparance and so it is easy to found out which of the two given numbers is the greater one. Therefore the C++ implementation shown in Listing 4.4 can be transfered to the ASIP's assembler language without substantial changes.

```
1  double max (double Number1, double Number2)
2  {
3    if (Number1 > Number2)
4    {
5      return Number1;
6    }
7    else
8    {
9      return Number2;
10   }
11 }
```

Listing 4.4: Maximum Implementation

4.5.5 Power

The C++ method *pow* returns the power of a given number with a given exponent.

$$pow\ (x,y) = x^y$$

The fact that the exponents used in the reference code are always natural numbers, simplifies the implementation of this function, because all difficult calculations can be disregarded.
A very simple implementation that calculates the power iteratively is shown in Listing 4.5.

```
1  double pow_simple (double Base, int Exponent)
2  {
3    double Result = 1;
4    for (; Exponent > 0; Exponent--)
5    {
6      Result = Result * Base;
7    }
8    return Result;
9  }
```

Listing 4.5: Simple Power Calculation

This implementation has two disadvantages that decrease the performance enourmously: The

first is that every multiplication depends on the previous one, which makes it impossible to pipeline the calculation and the second is that the number of multiplications is much higher than necessary. To reduce the number of multiplications and to deploy the benefits of the pipelined ALU, the algorithm builds packages, thereby decreasing the complexity from linear ($O(n)$) to logarithmic ($O(log_2(n))$) as already done for the faculty function. The resulting implementation is shown in Listing 4.6. The mathematical description of this idea is depicted in Figure 4.11 and Figure 4.12 shows an example.

```
1  double pow_fast (double Base, int Exponent)
2  {
3    double Result = 1;
4    while (Exponent > 0)
5    {
6      if (Exponent % 2 == 1)
7        Result = Result * Base;
8      Base = Base*Base;
9      Exponent = Exponent/2;
10   }
11   return Result;
12 }
```

Listing 4.6: Fast Power Calculation

$$
\begin{aligned}
\forall n \in \mathbb{N} : x^n &= \prod_{i=1}^{n} x \\
&= x^{n \bmod 2} \cdot \prod_{i=1}^{n \operatorname{div} 2} x^2 \\
&= x^{n \bmod 2} \cdot x^{(n \operatorname{div} 2) \bmod 2} \cdot \prod_{i=1}^{(n \operatorname{div} 2) \operatorname{div} 2} x^{2^2} \\
&= \ldots
\end{aligned}
$$

Figure 4.11: Formal Description Of Power Calculation

$$
\begin{aligned}
\text{example: } 2^7 &= \prod_{i=1}^{7} 2 \\
&= 2^{7 \bmod 2} \cdot \prod_{i=1}^{7 \operatorname{div} 2} 2^2 = 2 \cdot \prod_{i=1}^{3} 4 \\
&= 2 \cdot 4^{3 \bmod 2} \cdot \prod_{i=1}^{3 \operatorname{div} 2} 4^2 = 2 \cdot 4 \cdot \prod_{i=1}^{1} 16 \\
&= 2 \cdot 4 \cdot 16 = 128
\end{aligned}
$$

Figure 4.12: Example Of Power Calculation

Figure 4.13 shows the runtimes several systems need to compute all 2^n with $n = 1, ..., 100000$. It can be seen that the optimized implementation is much faster than the simple one. Especially the runtime of the Pentium 4 is very interesting, because this processor is keenly

System	pow	pow_fast	Speedup
Intel(R) Core(TM)2 CPU T5500 1.66GHz	822540 ms	190 ms	4329.16
Intel(R) Pentium(R) 4 CPU 2.60GHz	1719840 ms	370 ms	4648.22
Intel(R) Xeon(TM) CPU X5680 3.33GHz	498240 ms	110 ms	4529.45
Intel(R) Xeon(TM) CPU 3.60GHz	1355440 ms	300 ms	4518.13
Intel(R) Core(TM) i7 CPU 860 2.80GHz	501700 ms	110 ms	4560.91
Mean	979552 ms	216 ms	4534.96

Figure 4.13: Runtimes Of Power Implementations

pipelined [26]. So the effort of the implementation with respect to pipelining is conspicuously demonstrated. In contrast to the faculty function, the optimized pow function does not require additional memory. Therefore, it does not suffer from the problems of calculating the function value for large n.

4.5.6 Square Root

The C++ method *sqrt* returns the square root of a given number.

$$sqrt\ (x) = \sqrt{x}$$

I analyzed five different methods to calculate the square root of a given number: The Babylonian method, also called Heron's method [8], the exponential identity, the high/low method [12], the Bakhshali Approximation [9] and a Taylor series. These five methods are particularly explained in the following. After the explaination of the algorithms, an evaluation is provided.

Babylonian Method or Heron's Method

To calculate the square root of a given number S by using the Babylonian method, which is also called Heron's method, the steps shown in Figure 4.14 have to be accomplished.
An implementation in C++ of this method is depicted in Listing 4.7.

1. Start with a positive start value if you search for a positive square root or a negative start value if you search for a negative square root. This is called x_0 in the following steps.
2. Calculate $x_{n+1} = 0.5 \cdot (x_n + \frac{S}{x_n})$.
3. Go back to step 2 until the desired accuracy is achieved. If it is achieved, the last x_n is the desired square root.

Figure 4.14: Pseudocode Of Square Root Calculation Using The Babylonian/Heron's Method

```
1  double sqrtHeron (double Number, double Accuracy)
2  {
3    if (Number < 0)
4    {
5      cout << "ERROR: calculation of the square root of negative numbers is not
           possible!!!" << endl;
6      return std::numeric_limits<float>::quiet_NaN();
7    }
8    double Result = Number;
9    double SqrResult = Result*Result;
10   do
11   {
12     Result = 0.5 * (Result + Number/Result);
13     SqrResult = Result*Result;
14   } while (SqrResult-Accuracy>Number || SqrResult+Accuracy<Number);
15   return Result;
16 }
```

Listing 4.7: Implementation Of Square Root Calculation Using The Babylonian/Heron's Method

This implementation only computes positive square roots, as the C++ method does. The number for which the square root has to be computed of, is always taken as start value (see Listing 4.7, line 8). After setting the start value, the second step of the pseudo code, implemented in line 12 of the C++ implementation shown in Listing 4.7, has to be executed until the desired accuracy is achieved, what is checked in line 14 inside the C++ implementation.

Exponential Identity

Calculating the square root of a given number by using the exponential identity is done by many pocket calculators. These often have very good functions to approximate the exponential function and the natural logarithm. These two functions are required to compute the square root by using the exponential identity.
An implementation in C++ of the exponential identity is depicted in Listing 4.8.

```
1 double sqrtExponential (double Number)
2 {
3   return exp(0.5*ln(Number));
4 }
```

Listing 4.8: Implementation Of Square Root Calculation Using The Exponential Identity

High/Low Method

The high/low method is the only method I tested that works without using a division which is computation expensive. To calculate the square root of a given number S, the three steps shown in Figure 4.15 have to be accomplished.

1. Determine a number that is preferably close to the desired square root.
2. Check if the square of this number is greater or less than S.
3. Adjust the number by considering the result and go back to step 2 until the desired accuracy is achieved.

Figure 4.15: Pseudocode Of Square Root Calculation Using The High/Low Method

An implementation in C++ of the high/low method is depicted in Listing 4.9.

```
1  double sqrtHighLow (double Number, double Accuracy)
2  {
3    double Higher = Number+1;
4    double Lower = 0;
5    double Result;
6    while (Higher-Lower >= Accuracy)
7    {
8      Result = 0.5*(Lower+Higher);
9      if (Result*Result > Number)
10     {
11       Higher = Result;
12     }
13     else
14     {
15       Lower = Result;
16     }
17   }
18   return Result;
19 }
```

Listing 4.9: Implementation Of Square Root Calculation Using The High/Low Method [12]

To guarantee that I have a number that is greater and a number that is less than the desired square root, we start with 0 as lower bound and $n + 1$ as upper bound with n is the number for which the square root has to be calculated. This is done in lines three and four of the C++ implementation shown in Listing 4.9.
After setting the start values, the algorithm always searches for the arithmetic mean of the

number that is greater than the square root, called *Higher*, and the number that is less than the square root, called *Lower*. Than it has to be determined, if the mean is greater or less than the desired square root and depending on that, the mean value becomes the variable *Higher*, respectively the variable *Lower*. This is done, until the desired accuracy is reached.

Bakhshali Approximation

Calculating the square root of a given number S by using the Bakhshali Approximation, the steps depicted in Figure 4.16 have to be accomplished.

1. Let N^2 be the nearest perfect square to S.
2. $d = S - N^2$
3. $P = \frac{d}{2N}$
4. $A = N + P$
5. $sqrt(S) = A - \frac{P^2}{2A}$

Figure 4.16: Square Root Calculation Using The Bakhshali Approximation

Instead of calculating all these intermediate values, the square root can be computed with the following formula: $sqrt(S) = \frac{N^4+6SN^2+S^2}{4N^3+4SN}$
An implementation of the Bakhshali Approximation is shown in Listing 4.10.

```
1  double sqrtBakhshali (double Number)
2  {
3    double N;
4    for (N=1; N*N<Number; N++)
5    {
6    }
7    if (N*N-Number > Number-((N-1)*(N-1)))
8      N--;
9    return (N*N*N*N + 6*Number*N*N + Number*Number) / (4*N*N*N + 4*Number*N);
10 }
```

Listing 4.10: Implementation Of Square Root Calculation Using The Bakhshali Approximation

The nearest perfect square to the given number is detected within the lines 3 to 8 of the C++ implementation depicted in Listing 4.10. After that is done, the founded number can be put in the formula $sqrt(S) = \frac{N^4+6SN^2+S^2}{4N^3+4SN}$, which is done in line 9 of the C++ code.

Taylor Series

To calculate the square root of S by using a Taylor series, the steps shown in Figure 4.17 must be accomplished.

Listing 4.11 shows an implementation in C++ of the steps depicted in Figure 4.17.

1. Guess a number that is close to the searched square root. This approximation is called N in the following.
2. The usage of the following Taylor series leads to a better approximation: $sqrt(N^2+d) = \sum_{n=0}^{\infty} \frac{(-1)^n * (2n)! * d^n}{(1-2n) * n!^2 * 4^n * N^{2n-1}} = N + \frac{d}{2N} - \frac{d^2}{8N^3} + \frac{d^3}{16N^5} - \frac{5d^4}{128N^7} + ...$

Figure 4.17: Pseudocode Of Square Root Calculation Using A Taylor Series

```
1  double sqrtTaylor (double Number, double NumberOfTerms)
2  {
3    double N;
4    for (N=1; N*N<Number; N++)
5    {
6    }
7    if (N*N-Number > Number-((N-1)*(N-1)))
8      N = N-1;
9    double Result = 0;
10   double Difference = Number - N*N;
11   while (NumberOfTerms > 0)
12   {
13     NumberOfTerms--;
14     Result += pow((-1),NumberOfTerms) * faculty(2*NumberOfTerms) * pow(
           Difference,NumberOfTerms)/((1-2*NumberOfTerms) * pow(faculty(
           NumberOfTerms),2) * pow(4,NumberOfTerms) * pow(N,(2*NumberOfTerms-1)));
15   }
16   return Result;
17 }
```

Listing 4.11: Implementation Of Square Root Calculation Using A Taylor Series

The approximation of the square root is calculated in lines 3 to 8 of the C++ code shown in Listing 4.11. After that is done, as many terms as told are calculated. This is done in lines 11 to 15 of the code.

Final Choice

I decided not to use the exponential identity, because this method requires implementations of the exponential function and the natural logarithm and neither of them is used somewhere else in the algorithm.
Figure 4.18 shows some results of the remaining algorithms described before. To show the correctness of these results, the results of the GENIE 742GC, a pocket calculator, are also listed in Figure 4.18. In case of Heron's method and the high/low method, 0.00001 is chosen as accuracy and in the case of the Taylor series, ten terms are computed.

It can be seen that the results of all algorithms are close to each other, i. e. each algorithm provides a good accuracy. The runtimes of my implementations are depicted in Figure 4.19. They show how long the calculation of all square roots between one and 10,000,000 takes, if

n	Heron's Method	High/Low Method	Bakhshali Approximation	Taylor Series	GENIE 742GC
2	1.41422	1.41421	1.41667	1.4192	1.414213562
3	1.73205	1.73205	1.73214	1.73205	1.732050808
4	2	2	2	2	2
6523	80.7651	80.7651	80.7651	80.7651	80.76509147
32876	181.317	181.317	181.317	181.317	181.3174013

Figure 4.18: Results Of Algorithms

0.00001 is the required precision for the Heron's method and the high/low method, and the Taylor Series has to compute ten terms. It can be seen that the implementation of the Heron's method is the fastest one, followed by the high/low method.

System	Heron's Method	High/Low Method	Bakhshali Approximation	Taylor Series
Intel(R) Core(TM)2 CPU T5500 1.66GHz	5820 ms	7420 ms	140220 ms	188560 ms
Intel(R) Pentium(R) 4 CPU 2.60GHz	5740 ms	12480 ms	225390 ms	315140 ms
Intel(R) Xeon(TM) CPU X5680 3.33GHz	2100 ms	4080 ms	75390 ms	101570 ms
Intel(R) Xeon(TM) CPU 3.60GHz	3980 ms	9830 ms	171090 ms	237630 ms
Intel(R) Core(TM) i7 CPU 860 2.80GHz	2020 ms	3960 ms	73300 ms	98540 ms
Mean Runtime	3932 ms	7554 ms	137078 ms	188288 ms

Figure 4.19: Runtime Of Algorithms

The Heron's method requires one division per loop cycle (see line 12 of Listing 4.7) while the high/low method does not require any divisions. The division is the slowest operation my ASIP can compute and so I choose the high/low method to compute square roots.

5 Conclusion and Future Work

5.1 Conclusion

This thesis presents the design and implementation of an ASIP for a financial algorithm.

The starting point was reference code. After some preparation of this code, it was analyzed, which led to the first part of the instruction set. I also used the reference code to indicate the required mathematical methods and analyzed approximations for them. These approximations were implemented in C++ to see which instructions are required. The operations that were not already in the instruction set were added to it.
After the instruction set was defined, the ALU was build. The required arithmetic units were determined by dint of the instruction set. To complete the data path, the registerfile was implemented and connected to the ALU.

Due to the thorough analysis and conscientious preparation, only the implementation of the data path is completely done due to lack of time. The control path is already roughly designed and planned to be implemented after the disposal.

5.2 Future Work

After the disposal of this thesis, I will accomplish the design of the contol path and implement it afterwards. The current plan for the future is to implement this part with LISA, a processor designer tool offered by Synopsys [21].
After that is done, the components will be interconnected and the project will be validated by the simulation of a standardized Heston benchmark set.
Furthermore, I have some optimizations of the ALU in mind. These have to be scrutinized with respect to their performance, their practicability and their costs. If any of these optimizations will be implemented, the instruction set has to be adjusted contingently.

6 Appendix

6.1 Reference Code

```
1  HestonPricerResult HestonPricerMontecarloMultilevel::ComputeNewResult(const BarrierOption& option,
       const HestonMarket& market, const MontecarloSimulation& algorithm)
2  {
3    // check if a kernel is connected
4    if (_pKernel == 0)
5    {
6      cout << "no kernel connected!" << endl;
7      assert(false);
8    }
9
10   // check, which calculation should be used (singlelevel / multilevel / optimized multilevel)
11   switch (algorithm.Type)
12   {
13   case Singlelevel:
14   {
15     MultilevelMontecarloKernelResult KernelResult;
16     KernelResult = _pKernel->Run(option, market, algorithm);
17     Result.Price = KernelResult.Singlelevel.Payoff;
18     Result.Variance = KernelResult.Singlelevel.Variance;
19   }
20   break;
21
22   case Multilevel:
23   {
24     bool CalculationCompleted = false;
25     int CurrentSimulationLevel = -1;
26     int* NumberOfIterationsForAllLevels = new int[algorithm.MaximalLevel+1];
27     int* NumberOfIterationsForAllLevelsPreviousRun = new int[algorithm.MaximalLevel+1];;
28
29     HestonPricerResult* IntermediateResult = new HestonPricerResult[algorithm.MaximalLevel+1];
30
31     MontecarloSimulation CurrentAlgorithm = algorithm;  // needed for kernel usage
32
33     MultilevelMontecarloKernelResult KernelResult;
34
35     while (!CalculationCompleted)
36     {
37       CurrentSimulationLevel++;
38
39       cout << "Calculating level " << CurrentSimulationLevel << " ..." << endl;
40
41       NumberOfIterationsForAllLevels[CurrentSimulationLevel] = 10000;
42
43       // do the initial calculation on this level
44       CurrentAlgorithm.NumberOfIterations = NumberOfIterationsForAllLevels[CurrentSimulationLevel];
45       CurrentAlgorithm.NumberOfTimeSteps = std::pow((double)algorithm.MultilevelConstant,(double)(
             CurrentSimulationLevel+algorithm.StartLevel));
46       if (CurrentSimulationLevel>=1)
47       {
48         CurrentAlgorithm.Type = Multilevel;
49       } else
50       {
51         CurrentAlgorithm.Type = Singlelevel;
52       }
53
54       KernelResult = _pKernel->Run(option, market, CurrentAlgorithm);
55
56       if (CurrentAlgorithm.Type == Singlelevel)
57       {
58         IntermediateResult[CurrentSimulationLevel].Price = KernelResult.Singlelevel.Payoff;
59         IntermediateResult[CurrentSimulationLevel].Variance = KernelResult.Singlelevel.Variance;
60       }
61       else
62       {
63         IntermediateResult[CurrentSimulationLevel].Price = KernelResult.Multilevel.Payoff;
```

```
64          IntermediateResult[CurrentSimulationLevel].Variance = KernelResult.Multilevel.Variance;
65        }
66
67        cout << " Payoff = " << IntermediateResult[CurrentSimulationLevel].Price << ", var = " <<
              IntermediateResult[CurrentSimulationLevel].Variance << endl;
68
69        double TemporaryVariance = 0;
70
71        // renewing the number of iterations on each level
72        for (int l=0;l<=CurrentSimulationLevel;l++)
73        {
74          NumberOfIterationsForAllLevelsPreviousRun[l] = NumberOfIterationsForAllLevels[l];
75          TemporaryVariance += sqrt(IntermediateResult[l].Variance*std::pow((double)algorithm.
                MultilevelConstant,(double)l+algorithm.StartLevel));
76
77        }
78
79        for (int l=0;l<=CurrentSimulationLevel;l++)
80        {
81          NumberOfIterationsForAllLevels[l] = (int)(ceil(2*1/algorithm.Precision*1/algorithm.Precision
                *sqrt(IntermediateResult[l].Variance/std::pow((double)algorithm.MultilevelConstant,
                double(l+algorithm.StartLevel)))*TemporaryVariance));
82          NumberOfIterationsForAllLevels[l] = std::max(NumberOfIterationsForAllLevels[l],10000);
83        }
84
85        // calculating the missing simulations and updating the value and the variance
86        for (int l=0;l<=CurrentSimulationLevel;l++)
87        {
88          int delta_N = NumberOfIterationsForAllLevels[l]-NumberOfIterationsForAllLevelsPreviousRun[l
                ];
89          if (delta_N > 0)
90          {
91            delta_N = max(delta_N,2);
92
93            double OldVariance = IntermediateResult[l].Variance;
94            double OldPrice = IntermediateResult[l].Price;
95
96            CurrentAlgorithm.NumberOfIterations = delta_N;
97            CurrentAlgorithm.NumberOfTimeSteps = std::pow((double)algorithm.MultilevelConstant,(double
                  )(l+algorithm.StartLevel));
98            if (l>=1)
99            {
100             CurrentAlgorithm.Type = Multilevel;
101           } else
102           {
103             CurrentAlgorithm.Type = Singlelevel;
104           }
105
106           KernelResult = _pKernel->Run(option, market, CurrentAlgorithm);
107
108           if (CurrentAlgorithm.Type == Singlelevel)
109           {
110             IntermediateResult[l].Price = KernelResult.Singlelevel.Payoff;
111             IntermediateResult[l].Variance = KernelResult.Singlelevel.Variance;
112           }
113           else
114           {
115             IntermediateResult[l].Price = KernelResult.Multilevel.Payoff;
116             IntermediateResult[l].Variance = KernelResult.Multilevel.Variance;
117           }
118
119           IntermediateResult[l].Variance = 1/(double) (NumberOfIterationsForAllLevels[l]-1)*( (
                  NumberOfIterationsForAllLevelsPreviousRun[l]-1)*OldVariance + (delta_N -1)*
                  IntermediateResult[l].Variance + (NumberOfIterationsForAllLevelsPreviousRun[l]*delta_N
                  )/ (double) NumberOfIterationsForAllLevels[l] * std::pow(OldPrice - IntermediateResult
                  [l].Price,2));
120           IntermediateResult[l].Price = 1/ (double) NumberOfIterationsForAllLevels[l] * (
                  NumberOfIterationsForAllLevelsPreviousRun[l]*OldPrice + delta_N*IntermediateResult[l].
                  Price);
121         }
122
123       }
124
125       if ((CurrentSimulationLevel>= 2) && ( (max( abs(IntermediateResult[CurrentSimulationLevel-1].
              Price/algorithm.MultilevelConstant),abs(IntermediateResult[CurrentSimulationLevel].Price))
              ) < (1/sqrt((double)2)*(algorithm.MultilevelConstant-1)*algorithm.Precision)) )
126       {
127         CalculationCompleted = true;
128       }
129
130       if ((CurrentSimulationLevel == algorithm.MaximalLevel) && (!CalculationCompleted) )
131       {
```

```
132         CalculationCompleted = true;
133         cout << "!!!Maximal level attained has insufficient accuracy.!!!" << endl;
134       }
135
136     }
137
138     Result.Price = 0;
139     Result.Variance = 0;
140     Result.RuntimeInSeconds = 0;     // not implemented
141
142     for (int l=0;l<=CurrentSimulationLevel;l++)
143     {
144       cout << l << ": price = " << IntermediateResult[l].Price << ", var = " << IntermediateResult[l
              ].Variance << endl;
145       Result.Price += IntermediateResult[l].Price;
146       Result.Variance += IntermediateResult[l].Variance/NumberOfIterationsForAllLevels[l];
147     }
148
149     delete[] NumberOfIterationsForAllLevels;
150     delete[] NumberOfIterationsForAllLevelsPreviousRun;
151     delete[] IntermediateResult;
152   }
153
154   break;
155   // end of multi level
156
157
158   case MultilevelWithOptimizedStartlevel:
159     // begin of optimized multi level
160   {
161     bool CalculationCompleted = false;
162     bool UseMultilevelCalculation = false;
163     int CurrentSimulationLevel = 0;
164     int* N = new int[algorithm.MaximalLevel];
165     int* N_old = new int[algorithm.MaximalLevel];
166     MultilevelMontecarloKernelResult* IntermediateResult = new MultilevelMontecarloKernelResult[
            algorithm.MaximalLevel];
167
168     int StartLevel = 0;
169
170     while (!CalculationCompleted)
171     {
172       CurrentSimulationLevel++; // it is sufficient to start with CurrentSimulationLevel=1
173       N[CurrentSimulationLevel] = 10000;
174
175       // calculating on each level and given out the result
176       cout << "Calculating level " << CurrentSimulationLevel << " ...";
177
178       MontecarloSimulation CurrentAlgorithm = algorithm;
179       CurrentAlgorithm.NumberOfIterations = N[CurrentSimulationLevel];
180       CurrentAlgorithm.NumberOfTimeSteps = std::pow((double)algorithm.MultilevelConstant,
              CurrentSimulationLevel);
181       if (CurrentSimulationLevel>=1)
182       {
183         CurrentAlgorithm.Type = Multilevel;
184       }
185       else
186       {
187         CurrentAlgorithm.Type = Singlelevel;
188       }
189
190       IntermediateResult[CurrentSimulationLevel] = _pKernel->Run(option, market, CurrentAlgorithm);
191       cout << "price = " << IntermediateResult[CurrentSimulationLevel].Multilevel.Payoff << ",
              variance singlelevel = " << IntermediateResult[CurrentSimulationLevel].Singlelevel.
              Variance << ", variance multilevel = " << IntermediateResult[CurrentSimulationLevel].
              Multilevel.Variance << endl;
192
193       for (int l=1;l<=CurrentSimulationLevel;l++)
194       {
195         N_old[l] = N[l];
196       }
197
198       // testing whether the optimal start level is already reached
199       if ( std::pow((1-1/(double) sqrt((double)algorithm.MultilevelConstant)),2)*IntermediateResult[
              CurrentSimulationLevel].Singlelevel.Variance >= IntermediateResult[CurrentSimulationLevel
              ].Multilevel.Variance)
200       {
201         StartLevel = CurrentSimulationLevel-1;
202         CalculationCompleted = true;
203         UseMultilevelCalculation = false;
204       }
205
```

```
206       if ((CurrentSimulationLevel>= 2) && ( max( abs(IntermediateResult[CurrentSimulationLevel-1].
              Multilevel.Payoff/algorithm.MultilevelConstant),abs(IntermediateResult[
              CurrentSimulationLevel].Multilevel.Payoff) ) < 1/sqrt(2.0)*(algorithm.MultilevelConstant
              -1)*algorithm.Precision) )
207       {
208         CalculationCompleted = true;
209         UseMultilevelCalculation = true;
210         StartLevel = CurrentSimulationLevel;
211         N[CurrentSimulationLevel] = ceil(2*IntermediateResult[CurrentSimulationLevel].Singlelevel.
                Variance/std::pow(algorithm.Precision,2));
212       }
213
214       if (CurrentSimulationLevel==algorithm.MaximalLevel)
215       {
216         CalculationCompleted = true;
217         cout << "!!!Maximal level attained has insufficient accuracy.!!!" << endl;
218         return Result;
219       }
220     }
221
222     if (UseMultilevelCalculation)
223     {
224       cout << "Using single level calculation on level " << StartLevel << " (" << N[StartLevel] << "
              simulations)" << endl;
225
226       MontecarloSimulation CurrentAlgorithm = algorithm;
227       CurrentAlgorithm.Type = Singlelevel;
228       CurrentAlgorithm.StartLevel = StartLevel;
229       CurrentAlgorithm.NumberOfIterations = N[StartLevel];
230       CurrentAlgorithm.NumberOfTimeSteps = std::pow((double)algorithm.MultilevelConstant,StartLevel)
              ;
231
232       MultilevelMontecarloKernelResult KernelResult;
233       KernelResult = _pKernel->Run(option, market, CurrentAlgorithm);
234
235       Result.Price = KernelResult.Singlelevel.Payoff;
236       Result.Variance = KernelResult.Singlelevel.Variance;
237     }
238     else
239     {
240       cout << "Using multilevel calculation with start level " << StartLevel << endl;
241
242       MontecarloSimulation CurrentAlgorithm = algorithm;
243       CurrentAlgorithm.Type = Multilevel;
244       CurrentAlgorithm.StartLevel = StartLevel;
245       Result = ComputeNewResult(option, market, CurrentAlgorithm);
246     }
247
248     delete[] N;
249     delete[] N_old;
250     delete[] IntermediateResult;
251
252   }
253   Result.RuntimeInSeconds = 0;  // not implemented
254
255   break;
256   // end of optimized multi level
257   default :
258     // wrong entry
259     cout << "ERROR: wrong level type, use \"SL\" for single-level, \"ML\" for multi-level or \"MLopt
              \" for optimized multi-level calculation" << endl;
260     break;
261   }
262
263   Result.RuntimeInSeconds = 0; // no timing measurement implemented for whole pricer
264   return Result;
265 }
```

Listing 6.1: Reference Code

Bibliography

[1] Characteristics And Risks Of Standardized Options, February 1994.

[2] Hansjörg Albrecher, Philipp Mayer, Wim Schoutens, and Jurgen Tistaert. The Little Heston Trap. Technical report, Department Of Mathematics Katholieke Universiteit Leuven, January 2007.

[3] John A. D. Appleby, David C. Edelman, and John J. H. Miller. *Numerical Methods for Finance*. Chapman & Hall/CRC Financial Mathematics Series, 2008.

[4] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, 3rd edition, 2008.

[5] P.J. Ashenden. The VHDL Cookbook. University of Adelaide, South Australia, 1990. ftp://www.cs.adelaide.edu.au/pub/VHDL-Cookbook/.

[6] Prithviraj Banerjee, Rajesh K. Gupta, and Venkatram Krishnaswamy. A Procedure for Software Synthesis from VHDL Models. In *Proceedings of the Asia and South Pacific Design Automation Conference 1996*, 1997.

[7] D.R. Coelho. *The VHDL Handbook*. Kluwer, 1989.

[8] Eleanor Robson David Fowler. Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context. *Historia Mathematica*, 25:366–378, 1998.

[9] Jonathan M. Borwein David H. Bailey. Ancient Indian Square Roots: An Exercise in Forensic Paleo-Mathematics. October 2011.

[10] Christian de Schryver, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model. 2011.

[11] Michael B. Giles. Multi-level Monte Carlo path simulation. 56(3):607–617, 2008.

[12] Prof. Dr.-Ing. habil. Alexander Potchinkov. Digitale Signalverarbeitung: Algorithmen und Implementierung. Presentation of Course, Chapter 2, April 2010.

[13] Peter A. Henning and Holger Vogelsang. *Taschenbuch Programmiersprachen*. Carl Hanser Verlag, 2nd edition, 2007.

[14] Chung-Yih Ho, Karl J. Molnar, and Daniel A. Staver. Architecture to implement floating point multiply/accumulate operations, June 1989.

[15] Manoj Kumar Jain, M. Balakrishnan, and Anshul Kumar. ASIP Design Methodologies: Survey and Issues. 2001.

[16] W. Kahan. IEEE Standard 754 for Binary Floating-Point Arithmetic. October 1997.

[17] Venkatram Krishnaswamy, Rajesh Gupta, and Prithviraj Banerjee. Implications of VHDL timing models on simulation and software synthesis. *J. Syst. Archit.*, 44(1):23–36, October 1997.

[18] H. Marxen, C. de Schryver, A. Kostiuk, R. Korn, S. Wurm, I. Shcherbakov, and N. Wehn. Algorithmic Complexity in the Heston Model: An Implementation View. pages 5–12, November 2011.

[19] Russel W. Mason and Craig A. Heikes. Common Format For Encoding Both Single And Double Precision Floating Point Numbers, September 1992.

[20] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehle, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[21] A. Nohl, F. Schirrmeister, and D. Taussig. Application specific processor design: Architectures, design methods and tools. November 2010.

[22] Prof. Dr. Peter Rechenberg and Prof. Dr. Gustav Pomberger. *Informatik-Handbuch*. Carl Hanser Verlag, 1997.

[23] R. Reetz, K. Schneider, and T. Kropf. Formal Specification in VHDL for Formal Hardware Verification. In *Design, Automation and Test in Europe (DATE)*, pages 257–263. IEEE Computer Society, 1998.

[24] Prof. Dr. rer. nat. Klaus Schneider. Computer Systems 1 & 2, 2011. Lecture Notes.

[25] Dr. Bernd Schürmann. Begleitschrift zu den Vorlesungen "Rechnersysteme 1 und 2", 2011. Lecture Notes.

[26] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. pages 25–34, May 2002.

[27] Xiang Tian and K. Benkrid. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer. pages 81–88, December 2008.

[28] J.P. van Tassel and D. Hemmendinger. Toward Formal Verification of VHDL Specifications. In *Applied Formal Methods for Correct VLSI Design*, pages 409–418. North-Holland, 1989.

[29] André Willms. *Spielend C++ lernen ... oder wie man Käfern Beine macht*. Galileo Press, 1st edition, 2010.

[30] XILINX. *DS816 Floating-Point Operator v6.0, Data Sheet*, January 2012.

[31] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. pages 215–222, December 2005.