Master's Thesis

# Hardware Approximation of the Square Root Function

Dalia Mihaiela Iurascu
Alejandro Vázquez Bofill


Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, January 2014.

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Master Thesis

# Hardware Approximation of the Square Root Function

*Authors:*
Dalia Mihaiela Iurascu
Alejandro Vázquez Bofill

*Supervisors:*
Peter Nilsson
Erik Hertz
Rakesh Gangarajaiah

*A thesis submitted in fulfillment of the requirements for the degree of Master of Science*

*in the*

Digital ASIC Group
Department of Electrical and Information Technology

January 2014

# Declaration of Authorship

We, Dalia Mihaiela Iurascu and Alejandro Vázquez Bofill, declare that this thesis titled, "Hardware Approximation of the Square Root Function" and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.
- Where the thesis is based on work done by ourselves jointly with others, we have made clear exactly what was done by others and what we have contributed ourselves.

Signed: _____

Date: _____

Signed: _____

LUNDS UNIVERSITET

# *Abstract*

Lunds Tekniska Högskola
Department of Electrical and Information Technology

Master of Science

**Hardware Approximation of the Square Root Function**

by
Dalia Mihaiela Iurascu
Alejandro Vázquez Bofill

The aim of this master thesis research work focus on the development of a revolutionary methodology in which *Parabolic Synthesis* and *Second Degree Interpolation* methods are used for designing and implementing a novel approximation method of the Square Root function. This algorithm is implemented in VHDL, synthesized and mapped to an ASIC, using a 65 nm Standard $V_T$ technology library, with respect to timing, chip area and power consumption. This methodology is also suitable for calculating other different unary functions such as trigonometric, logarithmic and division functions and the main advantage of the approach is the simple hardware implementation using just elementary operations like addition, shifting and multiplication. The results reveal that the high proportion of parallelism used in the architecture of the design makes this algorithm implementation feasible for fast computation applications.

# *Acknowledgments*

First we would like to express our deepest gratitude to our Examiner Professor, PhD, Docent Peter Nilsson and Supervisor Tech. Lic. Erik Hertz for guiding our steps in the research work and encouragements throughout all the discoveries and forward proceedings. This Master Thesis work would not exist without their enduring support and persisting assistance.

Our enormous gratefulness goes to Digital ASIC research group, for their tremendously valuable hints, tips and encouragements and EIT Department for providing us the high research accessibility.

Special thanks goes to our System on Chip classmates for their amiability, friendliness, consideration and precious help and particularly appreciation to our dears Christoph and Steffen for all their patience and hints regarding the latest versions of different tools and scripting approach guidelines for automation of our work, among others.

Dalia's distinguished thankfulness goes to her family, especially her sister for all the encouragements, inspiration, love and valuable guidance.

Alejandro is deeply thankful to his closest and loved ones and in particular to his daughter for all the love and encouragement she inspires.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **DSP** | **D**igital **S**ignal **P**rocessing |
| **GPU** | **G**raphic **P**rocessing **U**unit |
| **CORDIC** | **C**oordination **R**otation **D**igital **C**omputers |
| **RTL** | **R**egister **T**ransfer **L**ogic |
| **VHSIC** | **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits |
| **VHDL** | **V**HSIC **H**ardware **D**escription **L**anguage |
| **LPHVT** | **L**ow **P**ower **H**igh **T**hreshold **V**oltage |
| **LPLVT** | **L**ow **P**ower **L**ow **T**hreshold **V**oltage |
| **GPSVT** | **G**eneral **P**urpose **S**tandard **T**hreshold **V**oltage |
| **CPU** | **C**entral **P**rocessing **U**nit |
| **SDF** | **S**tandard **D**elay **F**ormat |
| **SDC** | **S**ynopsis **D**esign **C**onstraint |
| **DC** | **D**irect **C**urrent |
| **VLSI** | **V**ery **L**arge **S**cale **I**ntegration |
| **PS** | **S**tatic **P**ower |
| **CMOS** | **C**omplementary **M**etal **O**xide **S**emiconductor |
| **IC** | **I**ntegrated **C**ircuits |
| **PT** | **T**ransient **P**ower |

| | |
|---|---|
| **SAIF** | **S**witching **A**ctivity **I**nterchange **F**ormat |
| **VCD** | **V**alue **C**hange **D**ump |
| **SPEF** | **S**tandard **P**arasitic **E**xchange **F**ormat |
| **SOC** | **V**alue **C**hange **D**ump |
| **IO** | **I**nput **O**utput |
| **ASIC** | **A**pplication **S**pecific **I**ntegrated **C**ircuit |

# Preface

In this thesis work, Alejandro Vazquez Bofill has been working with Dalia Mihaiela Iurascu.

Dalia Mihaiela Iurascu has been working on writing the Introduction(Chapter 1), theoretical approach for the Square Root Function(Chapter 2), Error Estimation(Chapter 6) and Synthesis together with the Place and Route of the design (Chapter 10 and Chapter 11). She has been writing also the Results chapter(Chapter 12) in which all the data were collected and discussed and finally she worked on the Conclusion and Improvements(Chapter 13).

Alejandro Vazquez Bofill has been working with Parabolic Synthesis (Chapter 3) and Second Degree Interpolation (Chapter 4). He also worked on Combining the Parabolic Synthesis and Second Degree Interpolation Methodologies(Chapter 5), Square Root Function Approximation(Chapter 7) and he discussed the Architecture of the design(Chapter 8) and finally he wrote on the Error Analysis(Chapter 9).

Abstract and Bibliography parts have been done together by Dalia and Alejandro.

Chapter 1

# Introduction

Nowadays, when gadgets and computers are present in everyday aspect of our life, more advanced algorithms for shorter computational timing are tremendously important.

Algebraic functions, for instance square root, logarithm, as well as trigonometric functions embrace the main source of algorithm implementation in domains like digital signal processing (DSP), wireless communication, graphic processing units (GPU), image processing, communication systems and medical robotics.
The performance of only software implementations of these algorithms is not satisfactory all the time, thus in order to improve the functionality, a translation of the software into hardware is desired.

The CORDIC (Coordination Rotation Digital Computers) algorithm [1], [2], [3] has constituted the fundamental method for unary functions implementations in the past. The CORDIC algorithm [4] is based on elementary and

uncomplicated mathematical operations such as shifting and addition operations and it does not use any multiplier elements, nevertheless it is a repetitive method, henceforth slow for this type of applications.

The easiest approach for implementing such functions is by applying single look-up table techniques such the ones described in [5] and [6]. The drawback of this method anyhow, is the big size of the tables thus long execution time.

An innovative approach, which is using *Parabolic Synthesis* and *Second Degree Interpolation* for calculating different unary functions, has been proposed by Erik Hertz and Peter Nilsson in [7],[8], [9],[10].

The key element of this original methodology is the usage of approximations of unary functions as is detailed in [7].
The objective of this master thesis is to reduce the execution time of unary functions such as square root by applying Parabolic Synthesis and Second Degree Interpolation methods [7] in the hardware implementation of the square root function.

Our goal is to implement the design and compare it for accuracy, error behaviour, power consumption and performance to an optimized square root algorithm implementation. The success features of the proposed approach of Erik Hertz and Peter Nilsson is the parallelism used in the hardware architecture together with the usage of low complexity operations.

Other unary functions such as logarithmic, exponential and trigonometric functions has been implemented already using this methodology as is described in [11], [8] and the results are showing that is a very efficient method with better accuracy, shorter critical path and higher throughput compared with other algorithms.

This thesis is organized as follows:

Chapter 1 dispenses the reader with the motivation for reading the material in this thesis and provides a brief introduction.

Chapter 2 describes the square root function approximation as well as higher square roots.

Chapter 3 and 4 familiarize the lecturer with Parabolic Synthesis and Second Degree Interpolation techniques.

Chapter 5 details the combination process between Parabolic Synthesis and Second Degree Interpolation methodologies.

Chapter 6 is talking about the error approximation and the tools used in the characterization of the error.

Chapter 7 is outlining the square root function approximation using the Parabolic Synthesis and Second Degree Interpolation method, followed by chapter 8 which details the architecture of the algorithm implementation.

In chapter 9 a continuation of the error analysis is exposed.

In chapter 10 the synthesis of the algorithm is characterized, followed by chapter 11 and 12, which include the place-and-route process description and the results of implementation and verification of the algorithm, respectively.

Chapter 13 outlines the conclusions and advice for attainable future improvements.

Chapter 2

# Square Root Function

## 2.1 Introduction

In this chapter the algorithm approximation for the square root and higher square roots are introduced to the reader.

## 2.2 Square Root Function

The conversion of a floating-point number into base 4 using binary number representation is the key element of the algorithm approximation for computing the square root of a number $x$. As explained in (2.1), by making use of the base 4 and binary representation, is easier to make the integer part of the mantissa to be expressed in 2 bits $MM$.

$$v = MM.mmmm....mm \cdot 4^{exp} \tag{2.1}$$

As it will be detailed further on the following chapters, the number of bits from the fractional part (fractional bits), $mm....mm$ is determined according with the accuracy needed on the input.

For calculating the square root of the floating point number represented in (2.1), the square root is executed on the mantissa and the exponent individually as it will be depicted in the equation (2.2), where the final output is called $z$.

$$z = \sqrt{MM.mmmm...mm} \cdot \sqrt{4^{exp}} = \sqrt{MM.mmmm...mm} \cdot 2^{exp} \tag{2.2}$$

The same approach as described above, can be used for computing the $n^{th}$ root, where $n$ represent a natural number, for example the cubic root of a floating point number. For the approximation of the algorithm for computing the cubic root of a number, $x$ is adjusted into a floating point number with the base 8 instead of 4 and binary representation number as shown in (2.3).

$$v = MMM.mmmm....mm \cdot 8^{exp} \tag{2.3}$$

By using the base 8 and binary representation of numbers, makes the integer part of the mantissa to consist of three bits $MMM$. As in the case of square root of a number, the number of fractional bits of the cubic root depends on the accuracy of the input as well. In the same manner as the square root, for performing the cubic root of the floating point numbers shown in (2.3), the mantissa and the exponent will be considered separately as shown in (2.4).

$$z = \sqrt[3]{MMM.mmmm...mm} \cdot \sqrt[3]{8^{exp}} = \sqrt[3]{MMM.mmmm...mm} \cdot 2^{exp} \tag{2.4}$$

Chapter 3

# Parabolic Synthesis

## 3.1 Introduction

The Parabolic Synthesis methodology proposed and described by Erik Hertz and Peter Nilsson in [9], is a method for calculating *unary functions*[1] approximations, based on the recombination of parabolic functions. Parabolic functions, also called *sub-functions*, are employed to provide fast convergence and rather simple hardware implementation to the design. This methodology also proposes a sub-functions recombination process, based on multiplication of the factors, instead of addition as in other methods. This allows higher levels of parallelism and further improves the convergence.

[1] functions that take one argument, such as *trigonometric*, *logarithmic* and *exponential* functions, among others

## 3.2 The Methodology

This methodology is based on that, the normalized function $f_{org}(x)$ of a specific unary function $f(x)$ can be perfectly recreated, by recombining an infinite number of parabolic functions, based on multiplications, as in (3.1a), where $s_1(x)$, $s_2(x)$ up to $s_\infty(x)$ are the parabolic functions. Since the main purpose of this methodology is a hardware implementation, the number of sub-functions to be implemented must be finite, thus this methodology relies on the development of a set of sub-functions to achieve an approximation to the unary function with a required accuracy, as in (3.1b).

$$f_{org}(x) = s_1(x) \cdot s_2(x) ... s_\infty(x) \tag{3.1a}$$

$$f_{org}(x) \approx s_1(x) \cdot s_2(x) ... s_n(x) \tag{3.1b}$$

## 3.3 Development of the first sub-function

To develop the first sub-function, the unary function is first *normalized*, with the goal of decreasing the hardware complexity of the design by limiting the numerical range of the calculations. This normalization is closely related to the unary function to be implemented, thus a different transformation is to be applied to a different unary function to normalize it. To assure proper functionality of the methodology, the normalized function, also called the original function $f_{org}(x)$, has to fulfil three important conditions:

- *must be strictly concave or convex through the interval to be approximated*

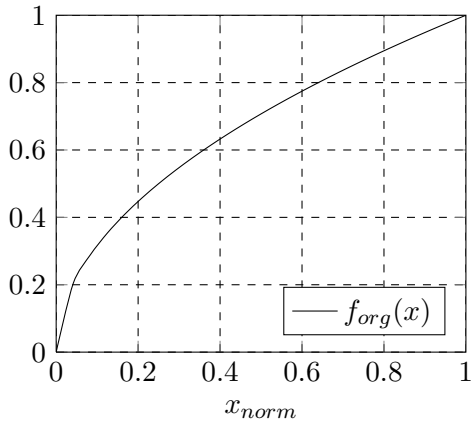



FIGURE 3.1: Concave or Convex Normalized Function

- *the limit of the original function divided by the linear function $x$, when $x$ tends to zero, subtracted with one, must be equal to a real number*

$$\lim_{x \to 0} \frac{f_{org}(x)}{x} - 1 = c_1, \text{ where } c_1 \in \Re$$

- *the absolute value of the real value obtained in the previous requirement must be smaller than one*

If $f_{org}(x)$ meets the exposed requirements, then the first sub-function $s_1(x)$ is obtained. To ease this analysis, the usual mathematical representation of quadratic functions in (3.2a) is transformed to (3.2b). In this equation, the coefficient $l_n$ represents the starting point of the function to be approximated, which can be calculated as in (3.2c). In the same manner, the coefficient $k_n$ represents the gradient of the same function and it is given by the slope of the linear function formed by its starting and ending points, which it can obtained as in (3.2d). Lastly, the coefficient $c_n$ represents the quadratic part of the sub-function, which it is obtained differently for each sub-function, therefore

explained with more details for each case.

$$y = d + e \cdot x + f \cdot x^2 \tag{3.2a}$$

$$s_n(x) = l_n + k_n \cdot x + c_n \cdot (x - x^2) \tag{3.2b}$$

$$l_n = f(x_{start}) \tag{3.2c}$$

$$k_n = \frac{f(x_{end}) - f(x_{start})}{x_{end} - x_{start}} \tag{3.2d}$$

From Fig. 3.1, it is given that the coefficient $l_1$ is equal to zero, as $f_{org}(0) = 0$ and the coefficient $k_1$ is equal to one, since the normalized function starts at the coordinates $(0;0)$ and ends at the coordinates $(1;1)$. Then, the sub-function $s_1(x)$ can be re-written as in (3.3), where the coefficient $c_1$ is obtained to satisfy the condition in (3.4a), which provide a simple implementation of the second sub-function $s_2(x)$.

$$s_1(x) = x + c_1(x - x^2) \tag{3.3}$$

Since the term $x^2$ in (3.4b) goes faster towards zero than the term $x$, it can be excluded for the calculations, as in (3.4c). Regrouping the term $(1 + c_1)$,

it is obtained (3.4d). Thus, the coefficient $c_1$ must be obtained as in (3.4e).

$$1 = \lim_{x \to 0} \left[ \frac{f_{org}(x)}{x + c_1(x - x^2)} \right] \tag{3.4a}$$

$$1 = \lim_{x \to 0} \left[ \frac{f_{org}(x)}{x(1 + c_1) - c_1 \cdot x^2} \right] \tag{3.4b}$$

$$1 = \lim_{x \to 0} \left[ \frac{f_{org}(x)}{x(1 + c_1)} \right] \tag{3.4c}$$

$$1 + c_1 = \lim_{x \to 0} \left[ \frac{f_{org}(x)}{x} \right] \tag{3.4d}$$

$$c_1 = \lim_{x \to 0} \left[ \frac{f_{org}(x)}{x} - 1 \right] \tag{3.4e}$$

## 3.4 Help Functions

In order to develop the higher order sub-functions to increase the accuracy of the approximation, help functions are first obtained to ease this process. These help functions provide curves, which later on, are approximated by the sub-functions.

The original function $f_{org}(x)$ is *divided* by its first approximation or the sub-function $s_1(x)$, when calculating the first help function $f_1(x)$. It is important to notice that the help functions are obtained by means of divisions, instead of subtractions, which improves the convergence of the approximation. To obtain help functions of order $n$, when $n > 1$, the division occurs between the order $n - 1$ help function and the order $n$ sub-function, as in (3.5).

$$f_n(x) = \begin{cases} \frac{f_{org}(x)}{s_1(x)}, & \text{for } n = 1 \\ \frac{f_{(n-1)}(x)}{s_{(n)}(x)}, & \text{for } n > 1 \end{cases} \tag{3.5}$$

However, help functions of higher order than one are not strictly convex or concave through the whole interval. Therefore, these help functions are split into sub-intervals, where every sub-interval is either concave or convex. Then, a *help partial function* is obtained for each sub-interval as in (3.6).

$$f_n(x) = \begin{cases} f_{n,0}(x), & 0 \leq x < \frac{1}{2^{(n-1)}} \\ f_{n,1}(x), & \frac{1}{2^{(n-1)}} \leq x < \frac{2}{2^{(n-1)}} \\ \quad \vdots & \quad \vdots \\ f_{n,[2^{(n-1)}-1]}(x), & \frac{2^{(n-1)}-1}{2^{(n-1)}} \leq x < 1 \end{cases} \tag{3.6}$$

## 3.5 Developing higher order sub-functions

### 3.5.1 Second Sub-function

The second sub-function $s_2(x)$ is developed as an approximation to the first help function $f_1(x)$, as in Fig. 3.2. This sub-function is developed also applying the quadratic function in (3.2b).

From Fig. 3.2, it is given that the coefficient $l_2$ is equal to one, since $f_1(0) = 1$ and the coefficient $k_2$ is equal to zero, since the $f_1(x)$ starts at the coordinates $(0; 1)$ and ends at the coordinates $(1; 1)$. Then, the second sub-function $s_2(x)$ can be re-written as in (3.7a) and the coefficient $c_2$ can be obtained as in (3.7b). Its graphical representation can be seen in Fig. 3.2.

$$s_2(x) = 1 + c_2 \cdot (x - x^2) \tag{3.7a}$$

$$c_2 = 4 \cdot [f_1(0.5) - 1] \tag{3.7b}$$

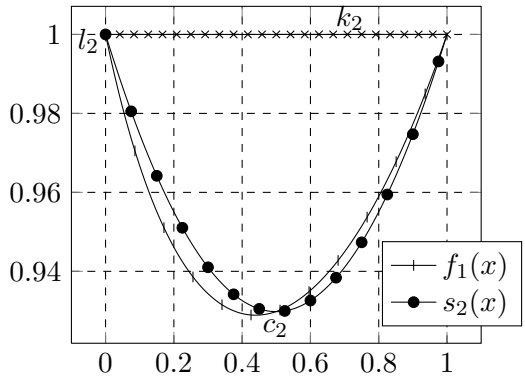



FIGURE 3.2: Approximation to the first help function

### 3.5.2 Sub-functions of Higher Order

For sub-functions of order $n$, where $n > 1$, a *partial sub-function* is developed for every help partial function obtained in (3.6). The partial sub-functions are represented as in (3.8a), where every interval is normalized. Therefore, the starting point or coefficient $l_n$ for each one of them equals one and the gradient is zero, resulting in (3.8b). The quadratic part is calculated as in (3.8c).

$$s_{n,m}(x) = l_{n,m} + k_{n,m} \cdot x + c_{n,m} \cdot (x_n - x_n^2) \tag{3.8a}$$

$$s_{n,m}(x) = 1 + c_{n,m} \cdot (x_n - x_n^2) \tag{3.8b}$$

$$c_{n,m} = 4 \left\{ f_{n-1,m} \cdot \left[ \frac{2(m+1)-1}{2^{(n-1)}} \right] \right\} \tag{3.8c}$$

After that, every partial sub-function is obtained as in (3.9).

$$s_{n,m}(x) = \begin{cases} s_{n,0}(x_n), & 0 \leq x < \frac{1}{2^{(n-2)}} \\ s_{n,1}(x_n), & \frac{1}{2^{(n-2)}} \leq x < \frac{2}{2^{(n-2)}} \\ \vdots & \vdots \\ s_{n,[2^{(n-2)}-1]}(x_n), & \frac{2^{(n-2)}-1}{2^{(n-2)}} \leq x < 1 \end{cases} \tag{3.9}$$

The term $x_n$ is employed instead of $x$, because the normalization of $x$ in the intervals is performed. This transformation is done as in (3.10), where the $frac()$ function performs a truncation of the integer part of the number. This integer part is used to index the partial sub-function to be used.

$$x_n = frac\{x \cdot [2^{(n-2)}]\} \tag{3.10}$$

## 3.6 Hardware Implementation

To implement designs based on this methodology, it is recommended to apply the two's complement binary representation and to divide the process in three different stages, *pre-processing*, *processing* and *post-processing*, as in Fig. 3.3, where the *operand* $v$ is the input value, the *operand* $x$ is the normalized or optimized input value, the *operand* $y$ is the normalized or optimized result and the *operand* $z$ is the final result. Thus, in the first stage, the normalization transformation is performed, along with other possible optimizations to decrease the overall complexity of the design. If the calculation of the approximations of the unary function is to be implemented as a module of a larger design, this stage can be merged into previous blocks and therefore be excluded.

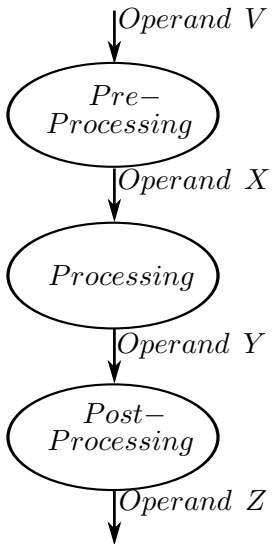



FIGURE 3.3: Design Stages

In the second stage, the calculations of the approximations take place, where the partials are obtained and recombined. A parallel implementation of this stage is represented in Fig. 3.4 and an iterative one in Fig. 3.5. The trade-off between these two implementations is given by the silicon area versus computational time, making the parallel approach more suitable for fast applications and the iterative one for lower area costs. Pipeline stages can also be introduced to the design to increase the throughput, which also negatively affects the latency.

Since this methodology is based on quadratic functions, a squaring unit is needed for calculating the values $x^2$ and $x_n^2$, where the latter ones are partial results of the first one. Thus, these calculations can be performed in the same *squaring unit*, applying the algorithm proposed by Erik Hertz and exposed in table 3.1, which involves only shifting, simple multiplications and simple additions, to achieve these results. To further decrease the hardware complexity of this unit and to improve the computational times of these calculations an

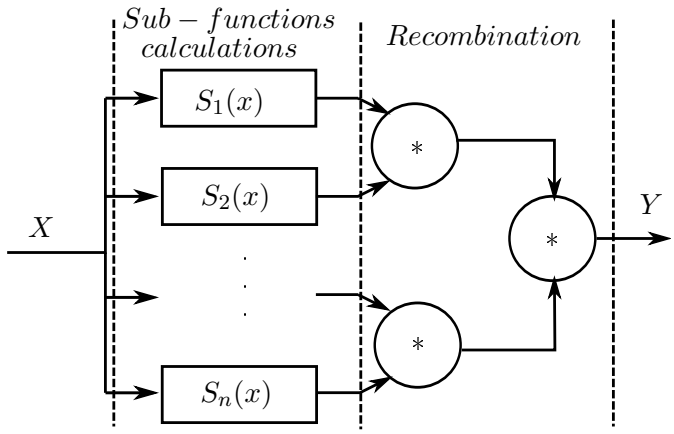



FIGURE 3.4: Parabolic Synthesis. Parallel Approach

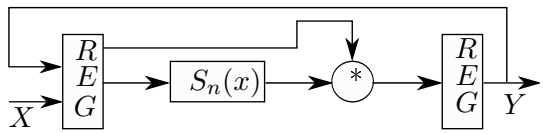



FIGURE 3.5: Parabolic Synthesis. Iterative Approach

improved version[2] of this algorithm is implemented, which is shown in table 3.2. In table 3.1 and table 3.2, the partial products $p$, $q$ and $r$ are the results of squaring the two, three and four least significant bits of the input value, respectively. Since, $x_n$ is the $(word\_length - truncation\_bits)$ least significant bits of $x$, this algorithm is very appropriate for obtaining $x^2$ and $x_n^2$ simultaneously.

A more detailed architecture of the parallel approach can be seen in Fig. 3.6, in which it is possible to see how the squaring unit produces the different squared values simultaneously, how the sub-functions are implemented in hardware and, how the different partials are then recombined based on *multiplications*, following a tree structure. In this way, the original function is recreated up to an accuracy level given by the four sub-functions implemented. In the final

[2]This improved version was also proposed by Erik Hertz.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $x_3$ | $x_2$ | $x_1$ | $x_0$ | |
| | | | | $*$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | |
| | | | | | | | | $x_0 \cdot x_0$ | |
| | | | | | | | | $p_0$ | **p** |
| | | | | $+$ | | | $x_1 \cdot x_0$ | | |
| | | | | $+$ | | $x_1 \cdot x_1$ | $x_0 \cdot x_1$ | | |
| | | | | | $q_3$ | $q_2$ | $q_1$ | $p_0$ | **q** |
| | | $+$ | | | | $x_2 \cdot x_0$ | | | |
| | | $+$ | | | $x_2 \cdot x_1$ | | | | |
| | | $+$ | | $x_2 \cdot x_2$ | $x_1 \cdot x_2$ | $x_0 \cdot x_2$ | | | |
| | | | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ | **r** |
| $+$ | | | | | $x_3 \cdot x_0$ | | | | |
| $+$ | | | | $x_3 \cdot x_1$ | | | | | |
| $+$ | | | $x_3 \cdot x_2$ | | | | | | |
| $+$ | | $x_3 \cdot x_3$ | $x_2 \cdot x_3$ | $x_1 \cdot x_3$ | $x_0 \cdot x_3$ | | | | |
| | $s_7$ | $s_6$ | $s_5$ | $s_4$ | $s_3$ | $s_2$ | $s_1$ | $s_0$ | **s** |

TABLE 3.1: Squaring Algorithm

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $x_3$ | $x_2$ | $x_1$ | $x_0$ | |
| | | | | $*$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | |
| | | | | | | | | $x_0 \cdot x_0$ | |
| | | | | | | $x_1$ | | $p_0$ | **p** |
| | | | | $+$ | | $x_1 \cdot x_0$ | 0 | | |
| | | | | $x_2$ | $q_3$ | $q_2$ | $q_1$ | $p_0$ | **q** |
| | | $+$ | | $x_2 \cdot x_1$ | $x_2 \cdot x_0$ | | | | |
| | | $x_3$ | $r_5$ | $r_4$ | $r_3$ | $r_2$ | $r_1$ | $r_0$ | **r** |
| | $+$ | $x_3 \cdot x_2$ | $x_3 \cdot x_1$ | $x_3 \cdot x_0$ | | | | | |
| | $s_7$ | $s_6$ | $s_5$ | $s_4$ | $s_3$ | $s_2$ | $s_1$ | $s_0$ | **s** |

TABLE 3.2: Improved Squaring Algorithm

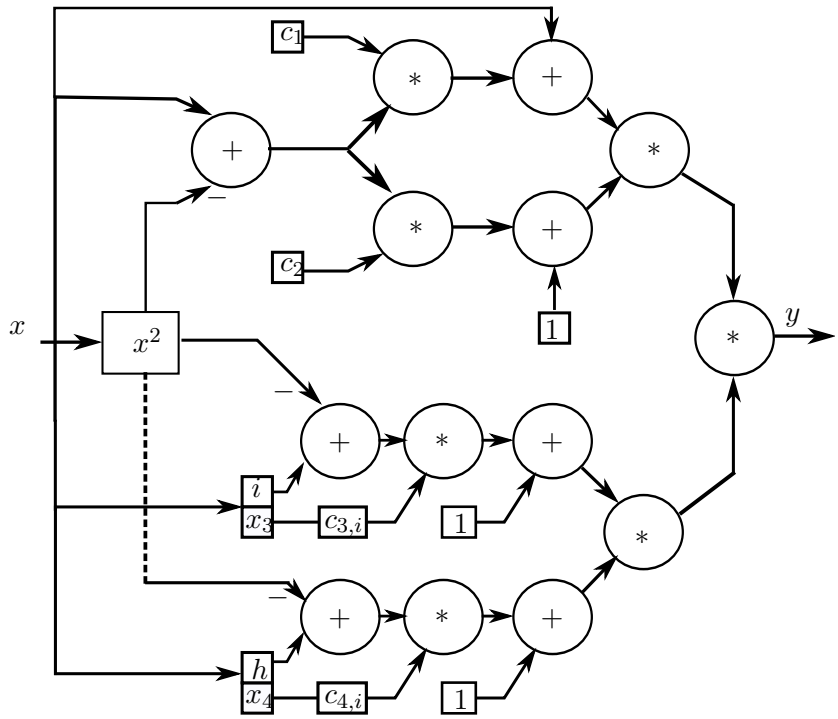



FIGURE 3.6: Parabolic Synthesis. Detailed Parallel Approach

stage, the transformations applied in the first stage are reversed to obtain the final result. As in the first stage, if the calculation of the approximations of the unary function is to be implemented as a module of a larger design, this stage can be merged into following blocks and therefore can be excluded.

## 3.7 Conclusions

This methodology allows for high parallelism, as seen in Fig. 3.4 and its main drawback is when high precision is required. In order to achieve high accuracy, a large amount of sub-functions is required, increasing the hardware complexity of the design. Therefore, a new method was integrated to this methodology.

Chapter 4

# Second Degree Interpolation

## 4.1 Introduction

The Second Degree Interpolation methodology is also a methodology based on parabolic functions, for calculation of approximations on unary functions. This methodology divides the function to be approximated into sub-intervals and parabolic functions of the same family are developed to approximate them, as in Fig. 4.1. In this figure, it is possible to see, how the parabolic approximation of the first sub-interval is somehow inaccurate, but the parabolic approximations of the higher order sub-intervals are very similar to the sub-interval to be approximated. If the approximation to one of the sub-intervals cannot achieve the required accuracy then, the function must be divided into a larger number of sub-intervals.

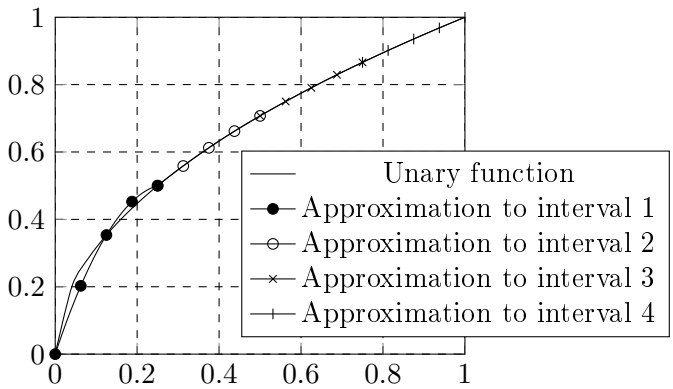



FIGURE 4.1: 4 Intervals Approximations

## 4.2 Developing the interval approximations

The parabolic function shown in (4.1) is employed to develop the interval approximations, where the sub-index $i$ denotes the order of the interval and the term $x_w$ represent the normalized value of $x$ for the interval $i$.

$$s_{2,i}(x) = l_i + k_i \cdot x_w + c_i \cdot (x_w - x_w^2) \tag{4.1}$$

To simplify the indexing of the sub-intervals in hardware, the amount of sub-intervals $I$ needs to satisfy the condition in (4.2), where $(w \in \mathbb{N})$. This condition allows the indexing of the sub-intervals using the most significant $w$ bits of the input value $x$. In this way, the normalized value $x_w$ is given by the remaining or least significant $(world\_length - w)$ bits of $x$.

$$I = 2^w \tag{4.2}$$

The sub-functions are developed according to the diagram in Fig. 4.2. In this diagram it is possible to see how the approximation equals the function in the initial, middle and end points to minimize the error. It is also possible to see

how the approximation from the starting point to the middle point approaches the function on the opposite manner than the approximation from the middle to the end point. In a similar manner to Parabolic Synthesis Interpolation,

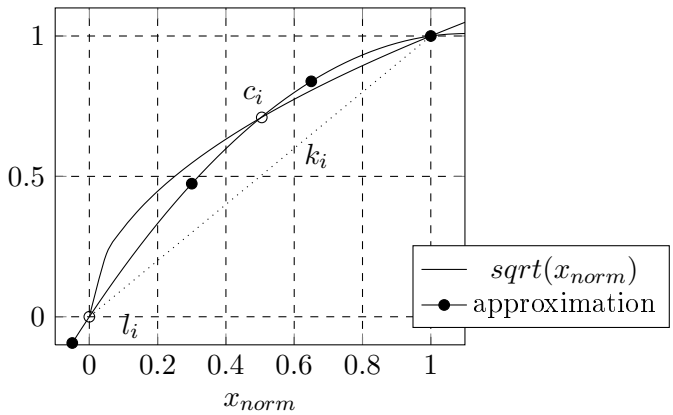



FIGURE 4.2: Intervals Approximations

the coefficients $l_i$ represent the starting point of each interval. In Fig. 4.2, it is possible to see this point for $i = 1$, where $x = 0$. To obtain these values, the function to be approximated is evaluated for the initial values of each interval, $x_{start,i}$, as in (4.3).

$$l_i = f(x_{start,i}) \tag{4.3}$$

The coefficients $k_i$ are obtained as the gradient of each interval, mathematically expressed in (4.4), where the denominator is always equal to one due to the intervals are normalized. The gradient is given by the slope of the linear function that crosses the starting and ending points of the interval, represented with a dotted line in Fig. 4.2.

$$k_i = \frac{f(x_{end,i}) - f(x_{start,i})}{x_{end,i} - x_{start,i}} \tag{4.4}$$

The coefficients $c_i$ are developed in a way, so that the resemblances, or parabolic functions, cut the intervals in the middle point, as can be seen in Fig. 4.2. The

mathematical expression to obtain these coefficients is represented in (4.5).

$$c_i = 4[f(x_{middle,i}) - l_i - 0.5k_i] \tag{4.5}$$

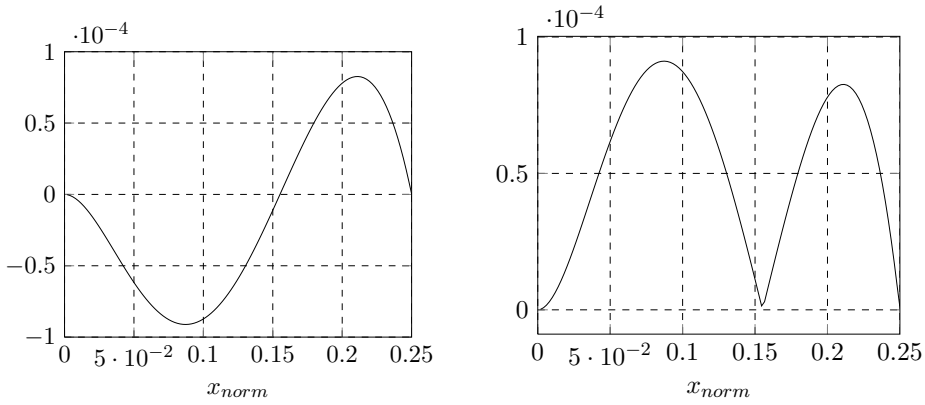
To calculate the values $x_{start,i}$, $x_{final,i}$ and $x_{middle,i}$ for each interval $i$, the inverse value of $I$ (number of Intervals) is multiplied with the order of the interval, where the starting point is $0 \leq i < I$ and the ending point is $0 < i \leq I$. The middle point is obtained as an average of these values. Table 4.1 resumes how to calculate these values.

| | | |
|---|---|---|
| $x_{start,i}$ | $i(I^{-1})$ | $0 \leq i < I$ |
| $x_{end,i}$ | $i(I^{-1})$ | $0 < i \leq I$ |
| $x_{middle,i}$ | $\frac{x_{end,i}+x_{start,i}}{2}$ | – |

TABLE 4.1: Relation of initial, middle and final values

## 4.3 Optimizations

By developing the sub-intervals resemblances as explained in the previous section, the achieved error equals zero at the starting, middle and ending points of each sub-intervals. However, it present a combined concave-convex behaviour through the interval, as it is shown in Fig. 4.3(a). In occasions, the maximum positive error is much larger than the negative one, or vice versa, as shown in Fig. 4.3(b). This undesired result can be avoided, by manually correcting the coefficients $c_i$, to a value where both errors are equal. This step constitutes the first optimization of designs based on this methodology. Another optimizations is achieved by further simplifying the sub-function $s_{2,i}(x)$. Precomputing the coefficients $j_i$, as shown in (4.6a) which yields (4.6b), resulting




(a) Maximum Error

(b) Absolute Error

FIGURE 4.3: Error Behavior

in a reduced amount of calculations.

$$j_i = k_i + c_i \tag{4.6a}$$

$$s_{2,i}(x) = l_i + j_i x_p - c_i x_p^2 \tag{4.6b}$$

## 4.4 Hardware Implementation

From a hardware point of view, by utilizing parabolic functions of the same family, it is possible to re-use the hardware arithmetic modules, as shown in Fig. 4.4. In this figure it is also possible to see that by indexing different sets of coefficients, which are stored in look-up tables, the calculations of the right approximation are performed, allowing hardware re-use. In this figure, it is also possible to see that the $w$ most significant bits of the input value or $x_m$, can be used to address these look-up tables and that the $(word\_length - w)$ least significant bits or $x_w$, are used for the calculations.

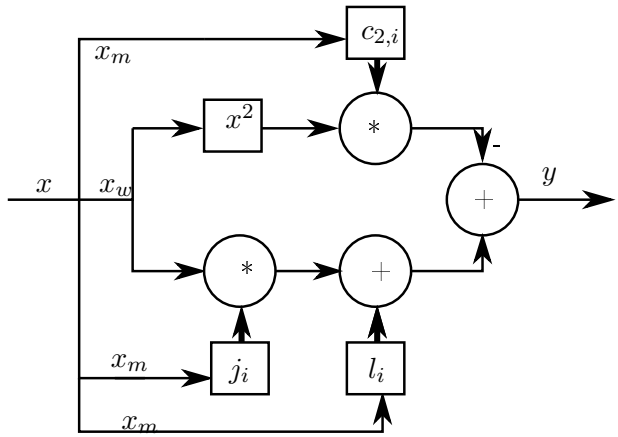



FIGURE 4.4: Second Degree Interpolation. Hardware Representation

## 4.5 Conclusions

This methodology allows for lower hardware complexity in the designs, but its main drawback is that it introduces lower levels of parallelism. To achieve an improved performance in the calculations of the approximations of unary functions, a combination of these two methodologies has been proposed.

Chapter 5

# Combining the Parabolic Synthesis and Second-Degree Interpolation Methodologies

## 5.1 Introduction

The combination of these two methodologies, proposed by Erik Hertz and Peter Nilsson, is based on the development of a first sub-function $s_1(x)$ by the Parabolic Synthesis Methodology and of a second sub-function $s_{2,i}(x)$ by applying the Second Degree Interpolation Methodology. Their recombination is then, performed by means of multiplication, as shown in (5.1). The first sub-function $s_1(x)$ is an initial approximation of the whole interval. The second one $s_{2,i}(x)$ is a set of fine-tuned approximations of the set of sub-intervals. The division between the original function and the first approximation is divided into.

$$f_{org}(x) = s_1(x) \cdot s_{2,i}(x) \tag{5.1}$$

## 5.2 Obtaining the sub-functions

When obtaining the first sub-function, $s_1(x)$, the expression shown in (5.2a) is employed, where the coefficient $c_1$ is obtained as in (5.2b). These expressions have already been presented and explained in chapter 3.

$$s_1(x) = x + c_1(x - x^2) \tag{5.2a}$$

$$c_1 = \lim_{x \to 0} \frac{f_{org}(x)}{x} - 1 \tag{5.2b}$$

Then, the help function $f_1(x)$ is obtained by applying (5.3a) and it is divided into a number of sub-interval $I$, which satisfy the condition in (5.3b).

$$f_n(x) = \frac{f_{org}(x)}{s_1(x)} \tag{5.3a}$$

$$I = 2^w \tag{5.3b}$$

Then, a number equal to $I$ of sub-intervals approximations are obtained as in (5.4a), to form the second sub-function, $s_{2,i}(x)$. To this end, the coefficients $l_{2,i}$ are obtained applying (5.4b); the coefficients $j_{2,i}$ are obtained by applying the optimization represented in (5.4c); the coefficients $k_{2,i}$ are obtained as the gradient between the initial and final points of each interval or as in (5.4d) and the coefficients $c_{2,i}$ are obtained by applying (5.4e), as explained in chapter 4.

$$s_{2,i}(x) = l_{2,i} + j_{2,i}x_w - c_{2,i}x_w^2 \tag{5.4a}$$

$$l_{2,i} = f_1(x_{start,i}) \tag{5.4b}$$

$$j_{2,i} = k_{2,i} + c_{2,i} \tag{5.4c}$$

$$k_{2,i} = \frac{f_1(x_{end,i}) - f_1(x_{start,i})}{x_{end,i} - x_{start,i}} \tag{5.4d}$$

$$c_{2,i} = 4[f_1(x_{middle,i}) - l_{2,i} - 0.5k_{2,i}] \tag{5.4e}$$

## 5.3 Conclusions

The implementation of this methodology results in a more compact, efficient and effective architecture. As it is possible to see in Fig. 5.1, only two sub-functions and a set of look-up tables are implemented in hardware. At the same time, the hardware complexity remains the same for higher accuracy levels, because only a larger set of intervals is needed, which only increases the size of the look-up tables. Furthermore, the recombination step is decreased to a single multiplication.

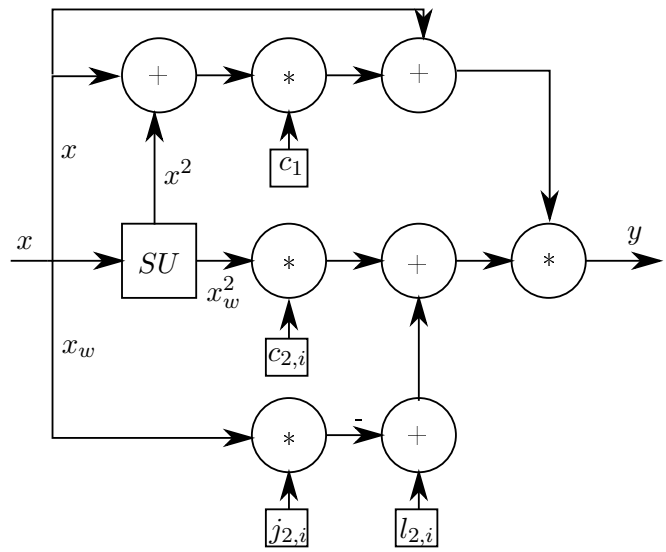



FIGURE 5.1: Second Degree Interpolation. Block Diagram

Chapter 6

# Error Estimation

## 6.1 Introduction

In this section the tools are analysed for characterizing the error of the approximation of our algorithm.

## 6.2 Error Performance

In order to describe and characterize the error estimation five metrics are used as described in [12]. The five metrics are maximum absolute error, mean error, median, standard deviation and root mean square error.

### 6.2.1 Maximum Absolute Error

The absolute error $\Delta x_i$ is the magnitude of the difference between the exact value $x_i$ and the approximation $\hat{x}$ as detailed in (6.1). As the name suggests,

the maximum absolute error represents the maximum value of the absolute error from the specific interval that is analysed.

$$\Delta x_i = |\hat{x_i} - x_i| \tag{6.1}$$

### 6.2.2 Mean Error

The mean error $\bar{x}$ of $n$ different values is an average of the absolute errors $\hat{x_i} - x_i$ as defined in (6.2). The following equation express that the average of a sequence of $n$ numbers represents the sum of the numbers divided by $n$.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n}(\hat{x_i} - x_i) = \frac{(\hat{x_1} - x_1) + (\hat{x_2} - x_2) + .... + (\hat{x_n} - x_n)}{n} \tag{6.2}$$

### 6.2.3 Median Error

The median error $\tilde{x}$, gives the middle value and $\tilde{x}$ of a sample of errors.

### 6.2.4 Standard Deviation

Standard deviation, $\sigma$ is computed as shown below in equation (6.3).

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}[\hat{x_i} - \bar{x}]^2 \tag{6.3}$$

### 6.2.5 Root Mean Square (RMS)

The RMS of a continuous time waveform represents the square root of the arithmetic average of the squares of the original values as is depicted in the

following formula (6.4).

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \hat{x_i} - {x_i}^2} \tag{6.4}$$

Basically, RMS quantifies how far the signal oscillates from the mean. The variance produces the power of this oscillation. Note that the RMS value measures both the AC and DC components. However if the signal has no DC components then the RMS value is similar to the standard deviation $x_{rms}$.

### 6.2.6 Decibel (dB)

When showing the accuracy of a function, the logarithm scale has the main advantage of providing better resolution and a simple understanding of the results. When referring to measurements of power or intensity the dB is expressed using the formula as shown in (6.5a). When referring to measurements of the amplitude, however the formula is the one used in (6.5b). In (6.5a) and (6.5b), $x_0$ is a specified value with the same units as $x$.

$$x_{dB} = 10 \log_{10} \frac{x}{x_0} \tag{6.5a}$$

$$x_{dB} = 20 \log_{10} \frac{x}{x_0} \tag{6.5b}$$

It is very suitable to use decibel(dB) in combination with binary numbers given the fact that 20log(2)=20*(0.301)=6dB and 6dB represents 1 bit resolution. Therefore, by using dB scale to display the results, we simplify the understanding of it.

## 6.3 Error Distribution

The biggest challenge of the polynomial approximation consists of establishing an effective approximation that can be accommodated to the function that is approximated in the chosen interval. For working with approximations there are two main strategies, one for minimizing the average error, named *least square approximations* and one for minimizing the worst case error, termed *least maximum approximations* to the approximated function [12]. Depending on the requirements of the design, the strategy is chosen. If the error of the approximation is to get the best fitting to the function to be approximated, then least squares approximations is to be used, otherwise the worst case error is used. For the examination of the approximation there are couple of statistical tools that have been used and described below.

### 6.3.1 Probability distribution

For judging the deviation of the error the most helpful tool could be the diagram of the probability distribution since it is very easy to visualize the distribution of the absolute error. Therefore it would help to get a better understanding of the standard deviation and root mean square values.

Chapter 7

# Square Root Function Approximation

## 7.1 Introduction

The square root function is widely used in QR decomposition, which is the de-facto method for detecting MIMO-OFDM systems and other wireless applications. The speed required for these applications is in increase, therefore faster, more reliable and more efficient ways of implementing the calculations needed. Thus, the main task of this thesis work, is the implementation in hardware of the calculations of the approximations of the square root function, applying the explained methodologies.

## 7.2 Design Flow Methodology

As Fig. 7.1 depicts, the hardware implementation followed the top-down digital design flow methodology. The first step was to define the specifications

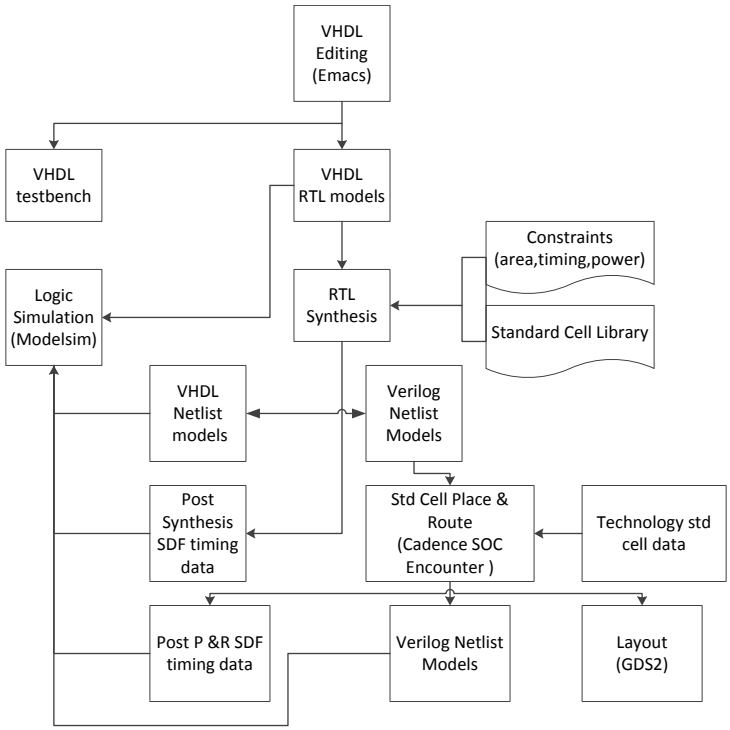



FIGURE 7.1: Top Down Design Flow

and requirements of the project, followed by the designing of the suitable architecture. The behavioural description of the different units of the design was next, succeeded by the behavioural simulations. After assuring that the simulations provided the correct waveforms, the design flow continued with the synthesis process, which converted the Register Transfer Level (RTL) description into a Verilog net-list with a timing information model. Later on, the *place-and-route* process took place to produce the final layout of the circuit, followed by the analysis of the power consumption.

## 7.3 Requirements and Specifications

The objective is to design and implement an approximation of the square root function. An algorithm to calculate the square root has been developed in which Parabolic Synthesis combined with Second-Degree Interpolation is used. The implementation shall approximate the square root function from a 16 bit fix point number to a 16 bit fix point number with an 8 bit integer part and an 8 bit fractional part. The design should be simulated and compared for accuracy, error behaviour, power consumption and performance. The core area should also be estimated. Synthesized VHDL is to be used in the project. As a starting point, Low Power High VT, and Low Power Low VT, and General Purpose Standard VT transistors should be used, in separate designs, where Low Power High VT is a good starting point. Three different supplies, VDD = 0.8, 1.0, 1.2 volts is to be used, where 1.0 volt, is a good starting point. The power and energy consumption, both static and dynamic, should be estimated for different clock frequencies.

## 7.4 Mathematical Modelling

Based on the requirements given by the supervisors and shown in the previous section, a mathematical modelling followed. The first goal of this model was to determine the value of the different coefficients involved in every sub-function and by a *trial-and-error* process, the amount of sub-intervals needed to provide the required accuracy. This model also helped to determine with exactitude the word length of every signal and of every hardware module in the design. Furthermore, this model was also used with verification purposes,

to corroborate that the values obtained by the hardware implementation coincided with the ones obtained by this model. To achieve this mathematical model, the algorithm was divided into the three different stages proposed in Fig. 7.2, *pre-processing*, *processing* and *post-processing*.

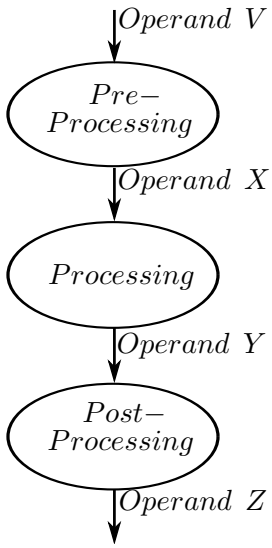



FIGURE 7.2: Design Stages

### 7.4.1 Pre-processing

In the pre-processing stage two transformations were performed in order to simplify the calculations. The first one was to represent the input value $x$, in floating point base four notation, as explained in Chapter 2. In this chapter, it is explained how this notation reduces the hardware complexity in the calculations of the square root. Therefore, the calculations are reduced to a calculation of the square root of a real value, $x_{base4}$, with a numerical range between one and four and to a shifting operation. The shifting operation is

easily achievable in hardware and the calculation of the square root of $x_{base4}$ is to be performed by applying the proposed methodologies.

To this end, the second optimization was applied and consisted in the normalization of $x_{base4}$. As it is possible to see in Fig. 7.3(a), the function $\sqrt{x_{base_4}}$ ranges from one to four, for the $x$ axis and from one to two for the $y$ axis. Since the normalization is a transformation in which the function becomes only defined between the values zero and one in both axes, the $\sqrt{x_{base_4}}$ function needs to be moved towards the $y$ axis, as in Fig. 7.3(b). This movement is represented mathematically as a subtraction with the value one. After this step, the function in question is defined between zero and three for the $x$ axis. By dividing it with the value three, the function becomes defined between the values zero and one for both axes, as in Fig. 7.3(c). This function, called $f_{org}(x)$ from now on, is the function to be approximated by the combination of the proposed methodologies. The expression represented in (7.1) resumes the normalization transformation from a mathematical point of view.

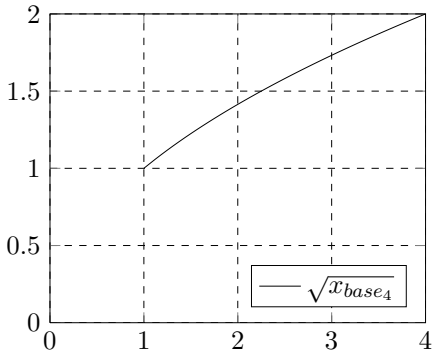
$$f_{org}(x) = \sqrt{1 + 3 \cdot x} - 1 \tag{7.1}$$

### 7.4.2 Processing

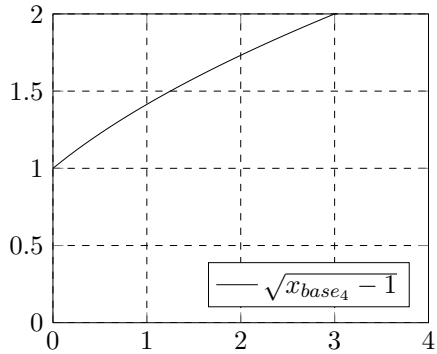
In the processing stage, the first step was to determine the coefficients of the first sub-function $s_1(x)$, based on the recently obtained $f_{org}(x)$. Applying (7.2a), it is given that the coefficient $c_1$ equals 0.5, thus the first sub-function $s_1(x)$ can be represented as in (7.2b).

$$c_1 = \lim_{x \to 0} \frac{f_{org}(x)}{x} - 1 \tag{7.2a}$$

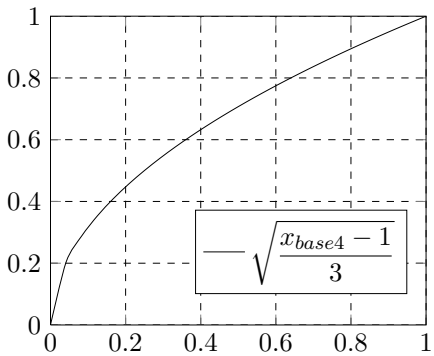
$$s_1(x) = x + 0.5(x - x^2) \tag{7.2b}$$




(a) Base four Square root function




(b) Subtracting




(c) Dividing

FIGURE 7.3: Normalization

A graphical representation of $f_{org}(x)$ and its first approximation, the first sub-function $s_1(x)$, can be seen in Fig. 7.4. As it is possible to see in this figure, this first approximation is somehow inaccurate, therefore the need of a second sub-function. Thus, the first help function $f_1(x)$ was needed and it was obtained utilizing the expression in (7.3). Its graphical representation can be seen in Fig. 7.5.

$$f_1(x) = \frac{f_{org}(x)}{s_1(x)} \tag{7.3}$$

To approximate this help function applying the Second Degree Interpolation Methodology, a $trail-and-error$ process took place to determine the amount

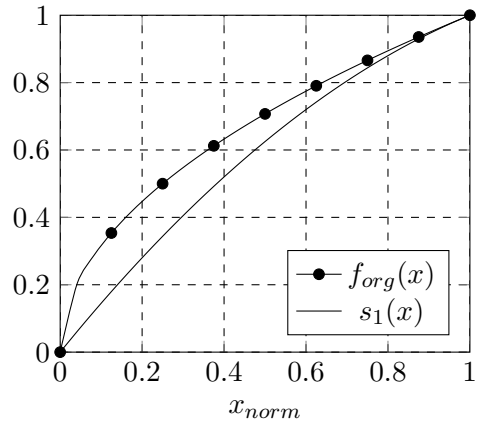



FIGURE 7.4: First sub-function $s_{1,1}(x)$

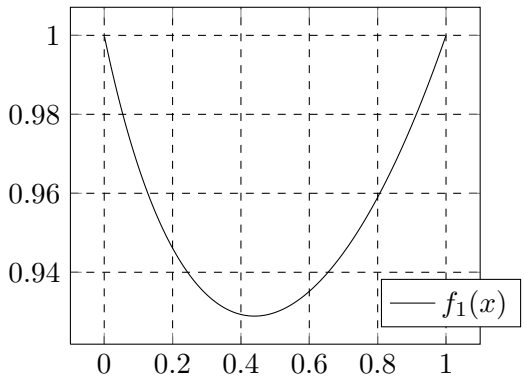



FIGURE 7.5: Approximation to the first help function

of intervals and their approximation coefficients. As it is explained in more detail in Chapter 9, it was proven that for eight intervals, the required accuracy was provided by the approximations, thus the help function $f_1(x)$ was divided into eight sub-intervals and their approximation coefficients obtained as in (7.4a), (7.4b), (7.4c) and (7.4d). In table 7.1 all the obtained coefficients for the second sub-function are shown and the second sub-function $s_{2,i}(x)$ is

represented in (7.5).

$$l_{2,i} = f_1(x_{start,i}) \tag{7.4a}$$

$$j_{2,i} = k_{2,i} + c_{2,i} \tag{7.4b}$$

$$k_{2,i} = \frac{f_1(x_{end,i}) - f_1(x_{start,i})}{x_{end,i} - x_{start,i}} \tag{7.4c}$$

$$c_{2,i} = 4[f_1(x_{middle,i}) - l_{2,i} - 0.5k_{2,i}] \tag{7.4d}$$

| $Interval(i)$ | $l_{2,i}$ | $j_{2,i}$ | $c_{2,i}$ |
|---|---|---|---|
| 0 | 1.000000000000000 | −0.050626667552115 | −0.011205116002104 |
| 1 | 0.960578448449989 | −0.028386869455018 | −0.007083055280797 |
| 2 | 0.939274634275768 | −0.014364452778615 | −0.005097130170301 |
| 3 | 0.930007311667454 | −0.004253765183449 | −0.003906250000000 |
| 4 | 0.929822128134704 | 0.003836343188820 | −0.003547417729198 |
| 5 | 0.937205889052722 | 0.010899360255743 | −0.003332543559085 |
| 6 | 0.951437792867549 | 0.017538352398746 | −0.003332589083664 |
| 7 | 0.972308734349959 | 0.024180532433370 | −0.003510733216671 |

TABLE 7.1: Set of Coefficients for 8 intervals

$$s_{2,i}(x) = \begin{cases} 1.000000000 - 0.05062666 \cdot x_w + 0.011205116 \cdot x_w^2, & \text{for } i = 0 \\ 0.960578448 - 0.02838686 \cdot x_w + 0.007083055 \cdot x_w^2, & \text{for } i = 1 \\ 0.939274634 - 0.01436445 \cdot x_w + 0.005097130 \cdot x_w^2, & \text{for } i = 2 \\ 0.930007311 - 0.00425376 \cdot x_w + 0.003906250 \cdot x_w^2, & \text{for } i = 3 \\ 0.929822128 + 0.00383634 \cdot x_w + 0.003547417 \cdot x_w^2, & \text{for } i = 4 \\ 0.937205889 + 0.01089936 \cdot x_w + 0.003332543 \cdot x_w^2, & \text{for } i = 5 \\ 0.951437792 + 0.01753835 \cdot x_w + 0.003332589 \cdot x_w^2, & \text{for } i = 6 \\ 0.972308734 + 0.02418053 \cdot x_w + 0.003510733 \cdot x_w^2, & \text{for } i = 7 \end{cases} \tag{7.5}$$

It is important to mention that in this stage the recombination of the sub-functions was performed, based on a single multiplication. At the end of this stage, the optimized and normalized result was produced.

### 7.4.3 Post-Processing

In the post-processing stage the transformations applied in the *pre-processing stage* were reversed to obtain the final result. As it can be seeing in (7.6), the values are first *de-normalized* and then shifted *exp* times.

$$z(x) = [y(x) + 1] \cdot 2^{exp} \tag{7.6}$$

### 7.4.4 Bit True Simulation

At this point of time, the mathematical model was computing the square root values achieving the requirements. The square root values were calculated

and compared with those of Matlab square root function and the required accuracy was provided by the approximations. Furthermore, the coefficients belonging to the sub-functions were obtained. However, the purpose of this mathematical model was to provide an ideal mathematical representation of the hardware implementation and since the computational power of Matlab[1] exceeds the needs of our design, this model was not complete. A technique called *bit true simulation* was employed to determine what portion of this computational power was needed in our design. This technique was proposed by Erik Hertz and is based on truncating and/or rounding the variables of the mathematical model to resemble the word length of the signals of the hardware implementation. It is based on shifting, achieved in Matlab by multiplications with base two numbers, truncating or rounding and then, shifting back the same amount of bits. In table 7.2, it is possible to see an example applying this technique, where a nine bits signal is truncated to a five bits signal. This process was applied to every variable of the mathematical model and the amount of bits shifted was tested until it provided the smallest amount of bits to be kept while still providing the required accuracy. It is important to mention that this method determines the smallest chip area for a given combination of signals word length, but it fails in determine which combination is the best. The authors aimed to reduce the sizes of the multipliers and decided to implement a *good enough* combination of signals word length that provided a good hardware implementation, but it might not be best one.

| Representation | Number | Shifting | Truncating | Shifting back |
|---|---|---|---|---|
| Decimal | 5.6406 | $5.6456 \cdot 2^2 = 22.5824$ | 22 | 5.5 |
| Binary | 101.100001 | 10110.0001 | 10110 | 101.10 |

TABLE 7.2: Bit True Simulation

[1] Tool used to develop this mathematical model

## 7.5 Conclusions

A mathematical model of the hardware implementation has been developed. The coefficients, intervals and word length of the signals has been determined. The code of this mathematical model can be found in the Chapter Appendix of this Master Thesis Report. The next step is the Hardware Implementation of the design.

Chapter 8

# Architecture

## 8.1 Introduction

In this chapter, the proposed architecture of each of the stages is presented, as well as the optimizations applied to the algorithm to minimize the hardware resources and execution times. Similarly to the mathematical model approach, the hardware implementation of the design was partitioned in the three different stages, pre-processing, processing and post-processing.

## 8.2 Pre-Processing

The architecture of the pre-processing stage can be seen in Fig. 8.1, where $V$ is the input value and $X$ is $x_{norm}$. The first step was the implementation of the floating-Point conversion in base four module. It was achieved by searching for the most significant *digital one* in pairs of bits of the input signal. The exponent value equalled the order of the bit pair times two, starting from the

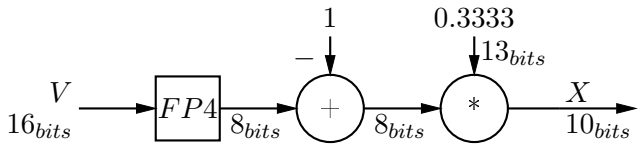



FIGURE 8.1: Normalization Stage

least significant pair with a value of zero. As it possible to see in table 8.1, the integer part of the base four value takes the value of the bit pair where the most significant digital one was found and the fractional part takes the value of the bits to the right of this bit pair.

| Decimal | Binary | Pairs of bits<br>7 6 5 4 3 2 1 0 | Base 4 |
|---|---|---|---|
| 50456 | 1100010100011000 | 11 00 01 01 00 01 10 00 | $11.00010100011000 \cdot 2^{14}$ |
| 387 | 0000000110000011 | 00 00 00 01 10 00 00 11 | $01.10000011000000 \cdot 2^{8}$ |

TABLE 8.1: Floating Point Base 4 Conversion

The second step was the *Normalization* process and it was performed on the base four value. First, *one* was subtracted from its integer part, as it can be seen in table 8.2. Then, this result was divided by *three*, what in hardware was achieved by multiplying by $\frac{1}{3}$ or by $0.33\overline{33}$. The output results of this stage $X$ or $x_{norm}$ was then rounded, instead of truncated, due to the extensive use of this value in the rest of the design, impacting thus the overall precision.

| | |
|---|---|
| 11.00010100011000 | |
| $-1.$ | |
| 10.00010100011000 | |
| $*\ 0.010101010101010$ | $(0.33\overline{33})$ |
| 0.10110001011101 | |
| 0.101100011 | *rounding to 10 bits* |

TABLE 8.2: Normalization

## 8.3 Processing

The implementation of the sub-functions occurred in this stage. Its architecture can be seen in Fig. 8.2, where the inherent parallelism of the algorithm is well represented. The calculations of the sub-function $s_1(x)$ (on the top of the figure) and the sub-function $s_{2,i}(x)$ (on the bottom) occurred simultaneously. In this figure, $Y$ represented the normalized final result.

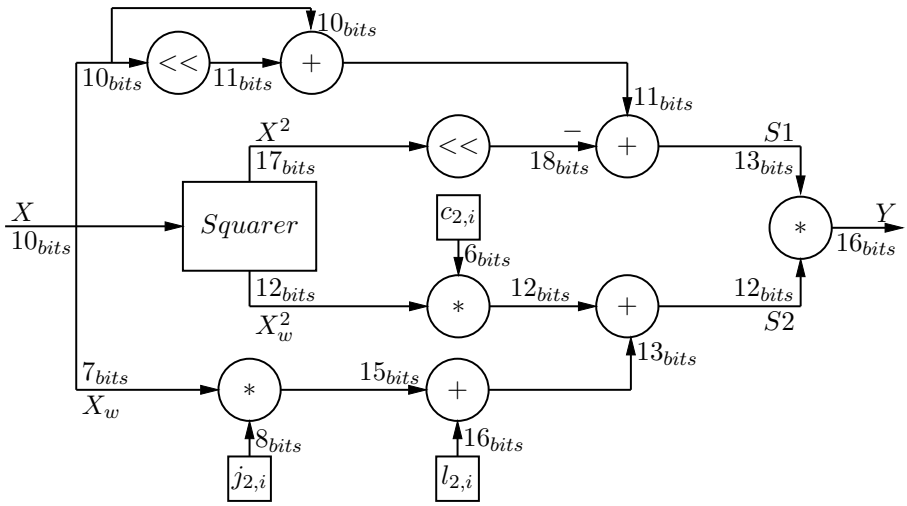



FIGURE 8.2: Processing Stage

When approaching the hardware implementation of this stage, four optimizations were performed to decrease its hardware complexity. The first one was to regroup the calculations to be executed for the first sub-function $s_1(x)$. This sub-function initially represented as in (8.1a) was regrouped as in (8.1b), which decreased the complexity of the calculations to two shifting operations, two additions/subtractions operations and one squaring unit.

$$s_1(x) = x + 0.5(x - x^2) \tag{8.1a}$$

$$s_1(x) = 1.5x - 0.5x^2 \tag{8.1b}$$

The second one, was to scrutinize the coefficients to be stored in the different look-up tables to extract leading sequences of zeros to even further decrease the use of silicon resources. These values were re-arranged later on in hardware all at once. The third optimization constituted the implementation of the *Squarer* following the methodology described in Chapter 3. The fourth optimization was the rounding of the input signals of the recombination multiplier to decrease the size of this unit, which is the largest in the design.

## 8.4 Post processing

In the post-processing stage of the design, the final result $Z$ was extracted from $Y$ as shown in Fig. 8.3. This was the smallest stage of the design, since the addition was implemented as a *concatenation* due to that the input value was known to be lesser than one and a right shifting operation, which is very simple implemented in hardware, as in Fig. 8.4.

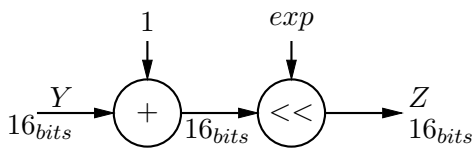



FIGURE 8.3: Post-Processing Stage

## 8.5 RTL Implementation

The Register-Transfer-Level was described utilizing the hardware description language VHDL'93. It consisted of two packages files *comp_pack.vhd* and *constants.vhd*, where the parametric modules and all the signals length and parameter of the design were implemented and declared, respectively. Thus,

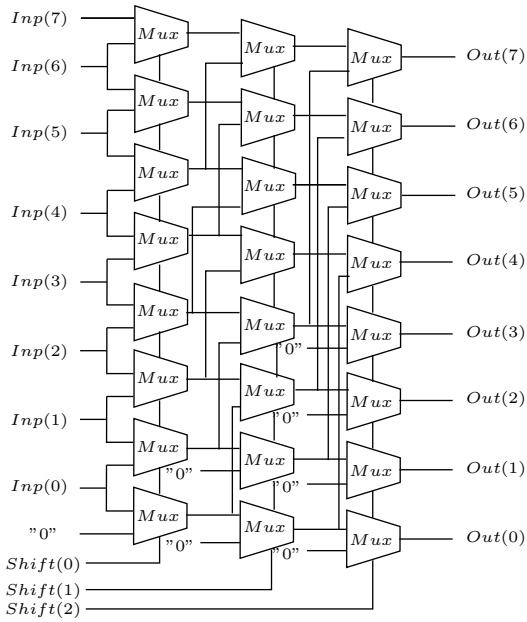



FIGURE 8.4: Right Shift Operation

if the input or output value word length is altered in the requirements, there is no need to re-write the entire design. By determining the signals word length in the mathematical model and changing the respective values in the file *constants.vhd*, the hardware will be re-generated to the new conditions. The description also consisted of a *top.vhd* file, where all the components declared in the *comp_pack.vhd* file were instantiated and interconnected. This decreased the coding-debugging time, because it allowed the testability of every module separately and later on, their interconnections, going from a simpler to a more complex implementation, where the earlier step was known to be working. For testability purposes another file was created *testbench.vhd*, to determine the correct functioning of the modules and of the entire design. These codes are listed in the Chapter Appendix of this Master Thesis Report.

Chapter 9

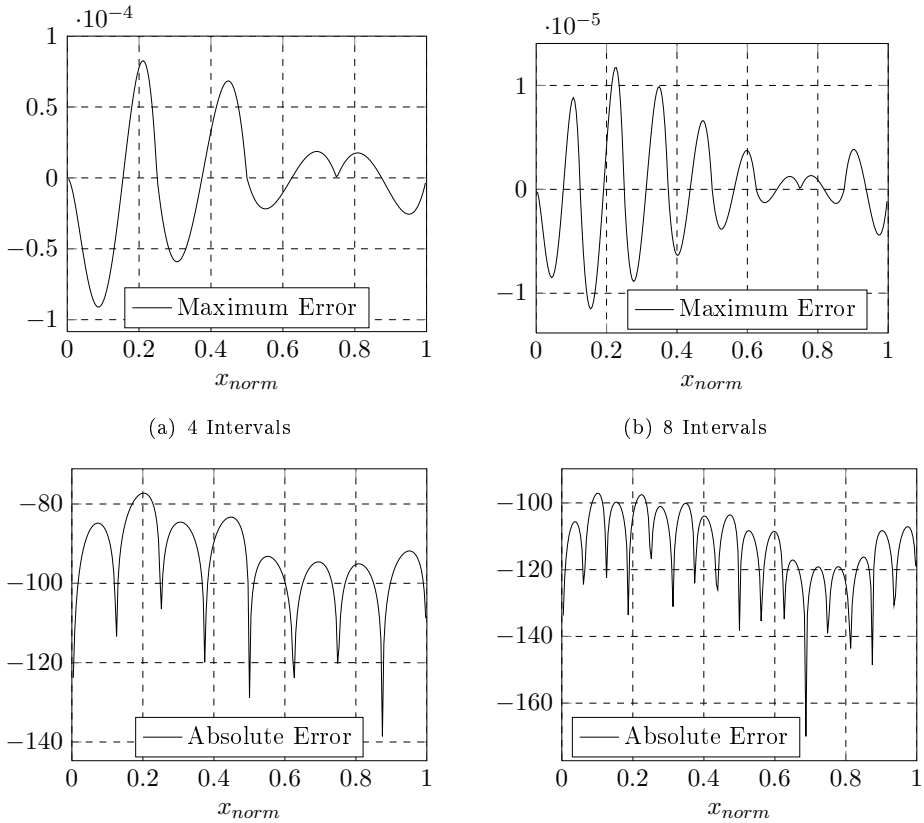
# Error Analysis

When analysing the error behaviour of the mathematical model, the results achieved by the proposed methodology were compared to those of the Matlab[1] square root function. In Fig. 9.1(a) and Fig. 9.1(b) it is possible to see the difference between these two set of results, when applying the proposed methodology employing four and eight intervals, respectively. These graphs show that the results achieved by the proposed methodology present a maximum error in the order of $0.8 * 10^{-4}$ and $1.2 * 10^{-5}$.

However, the dB scale is a preferred method when analysing accuracy in hardware, because of (9.1). In Fig. 9.1(c) and Fig. 9.1(d), it is possible to see these differences expressed in dB,

where the maximum absolute error for four intervals is approximately 80 dB, which implies an accuracy of 13.3 bits, and the maximum absolute error for

[1] De facto standard high-level language and interactive environment for numerical computation, visualization, and programming, developed by MathWork




(a) 4 Intervals

(b) 8 Intervals

(c) 4 Intervals

(d) 8 Intervals

FIGURE 9.1: Error Behaviour

eight intervals is approximately 95 dB, thus achieving an accuracy of 15.8 bits.

$$6.02\ dB \approx 1\ bit \tag{9.1}$$

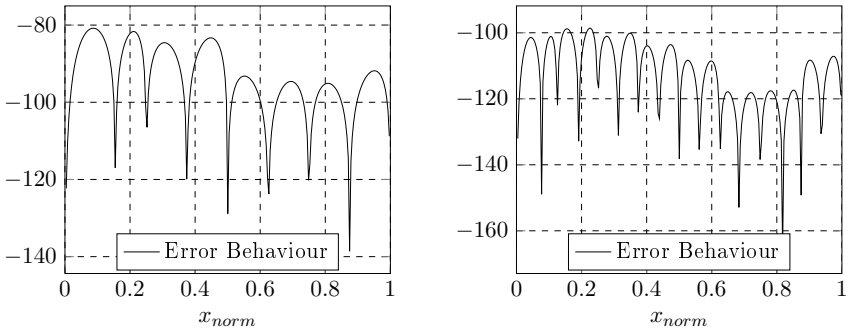
None of this results satisfies the accuracy requirement, but the eight intervals implementation is really close to it. By further scrutinizing the results shown in Fig. 9.1(d), it is possible to see that the absolute maximum error is achieved in the second peak from the right, while the first peak presents a much lower maximum value. Since, the proposed methodology calculates the approximations by utilizing parabolic functions that cut the original function

in the start, middle and end points, it is given that the error achieved between the start and middle points is related to the error achieved between the middle and end points, see Fig 4.2. By changing the value of the coefficient $c_{2,i}$, it is possible to change the point where the approximation cuts the middle value of the original function. This can be used to decrease the absolute error achieved in the start-to-middle approximation while increasing the absolute error in the middle-to-end approximation, or vice versa or in other words, to decrease the absolute error achieved in the second peak from the right, while increasing the absolute error in the first one.

### 9.0.1 Coefficients Optimization

By re-accommodating the coefficients $c_{2,i}$ of the sub-functions $s_{2,i}$, it was possible to alter the error behaviour of the algorithm. In Fig. 9.2(a) and Fig. 9.2(b) it is possible to see that the error behaviour decreased after optimizing the coefficients. This optimization was done by adding or subtracting a small amount to the coefficients, to change the points where the approximation crosses the original function. This operation *moves* the error achieved in one side of the sub-interval approximation to the other, reducing thus the maximum error. As it is possible to see in these graphs, the maximum absolute error is now in the order of 90 dB or 14.9502 bits of accuracy for four sub-intervals and 99 dB or 16.4452 bits of accuracy for eight sub-intervals, achieving thus the required accuracy. The optimized coefficients for eight sub-intervals can be seen in table 9.1.

However, when analysing the error behaviour of the real hardware implementation, see Fig. 9.3, it is possible to see a different behaviour. This is caused by the truncation or rounding of the internal signals of the design, which was




(a) 4 Intervals

(b) 8 Intervals

FIGURE 9.2: Error Behaviour with Optimized Coefficients

| $Interval(i)$ | $l_{2,i}$ | $j_{2,i}$ | $c_{2,i}$ |
|---|---|---|---|
| 0 | 1.000000000000000 | −0.050537109375 | −0.0112304687500 |
| 1 | 0.960571289062500 | −0.028320312500 | −0.0070800781250 |
| 2 | 0.939270019531250 | −0.014404296875 | −0.0051269531250 |
| 3 | 0.929992675781250 | −0.004150390625 | −0.0040283203125 |
| 4 | 0.929809570312500 | −0.003906250000 | −0.0035400390625 |
| 5 | 0.937194824218750 | −0.010986328125 | −0.0032958984375 |
| 6 | 0.951446533203125 | −0.017578125000 | −0.0032958984375 |
| 7 | 0.972320556640625 | −0.024169921875 | −0.0035400390625 |

TABLE 9.1: Optimized Set of Coefficients

carefully calculated with the bit-true simulation to avoid absolute maximum error values above the requirements.

### 9.0.2 Statistics

The probability distribution for the whole interval of $x$ can be seen in Fig. 9.4. In this figure, it is possible to see that for a error value of zero the probability is the highest and vanishes with increasing values of the error.

The standard deviation (std) was calculated, as well as the mean value (mean), the root mean square value (rms) and the median error (median). These values

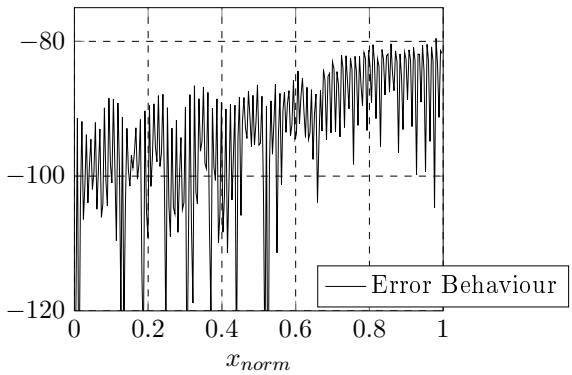



FIGURE 9.3: Error Behaviour of Real Hardware Implementation

can be seen in table 9.2. It is important to notice how the standard deviation value and the root mean square value are in the same order and that the mean value and median error are really small.

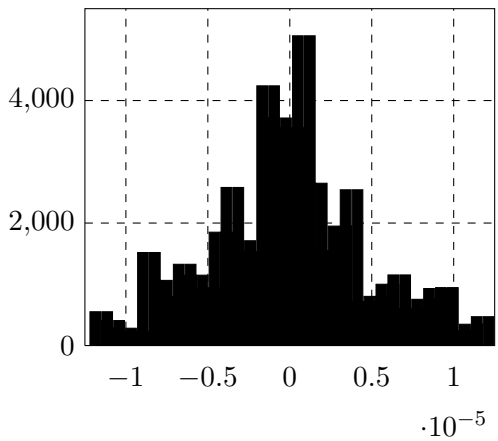



FIGURE 9.4: Probability Distribution

| | |
|---|---|
| std | $1.0305967893090 \cdot 10^{-2}$ |
| mean | $6.149186432794 \cdot 10^{-3}$ |
| rms | $1.2000993597086 \cdot 10^{-2}$ |
| median | $1.182310703335965 \cdot 10^{-4}$ |

TABLE 9.2: Statistics

Chapter 10

# Place and Route

The final step in the digital design flow was to take the verilog netlist of the design generated by the synthesis tool and convert it to a layout representation using a place and route tool, in this case being SOC Encounter from Cadence.

The place and route process starts by importing the design, followed by defining the global power nets (*vdd* and *gnd*). After that, the floor planning was done, in our case having a floor plan of 133.73 $\mu m^2$ on 141 $\mu m^2$ for our logic.

Given the fact that for our design, it was decided not to include the input output pads (because it would have taken too much area, comparing with the area of our logic), the IO placement step was skipped from our script file.

The next accomplishment was to add the power rings and power stripes. The power ring was added around the core of the design and we used metal 3 for the horizontal wires and metal 4 for the vertical ones and the same for the power stripes.

Following the power planning, the placement of the standard cells was carried out, succeeded by the placement of the filler cells and the special routing, the

final layout looking as in Fig. 10.1. All the steps were organized in a tcl script, which can be found in the appendix for further utilization.

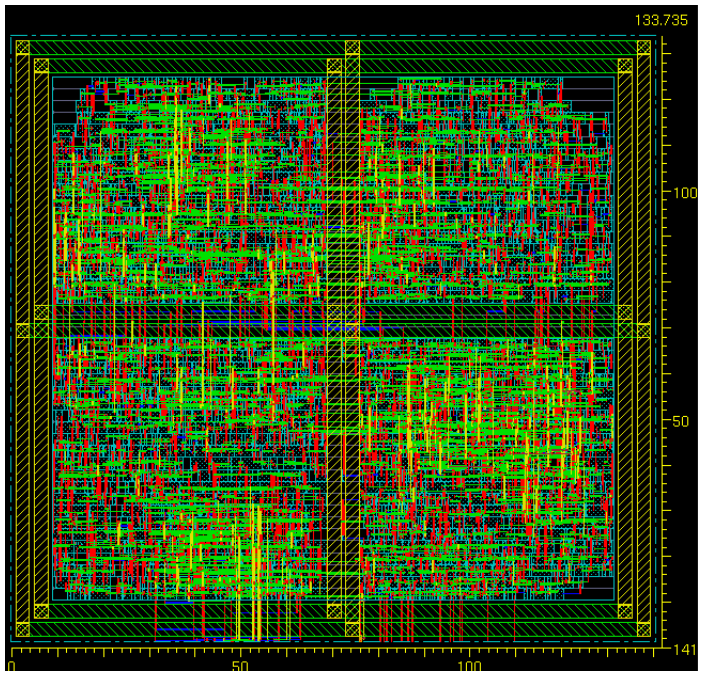



FIGURE 10.1: Layout of the Design

Modelsim simulations were performed after the routing of the design, in order to assure that the design is still working according with the specifications. For the simulation after place and route, we have included the verilog netlist file, the test bench of the design, the Standard Delay Format (.sdf) file and the corresponding library. Simulations after the placement of the design are functioning correct, the only unwanted consequence, being the glitches, which happens sometimes as signals propagate through combinational logic.

The reason why this happened was because the different signal paths have different delays, so when one input reaches the input of a gate while the other input is delayed, a wrong output may occur. The output signal is however

corrected once all of the correct inputs were propagated through, but the dynamic power dissipation can increase significantly, due to the glitches.

# Chapter 11

# Results

## 11.1 Area Results reported by Design Vision Synthesis Tool

Fast computations, small silicon area and low power consumption are the challenges of every single digital design, however sometimes, compromises has to be done as well regarding at least one of the requirements.

In our case, the square root calculation algorithm has been constrained for minimum computational time, giving it priority over the silicon area. As a start point, we synthesised our RTL code using Low Power High Threshold Voltage library [15], which is a Standard Cell library for the 65nm technology used in VLSI digital design platform and is optimized for use in low power applications.

For a better observability, the area results are synthesised in the graph below:

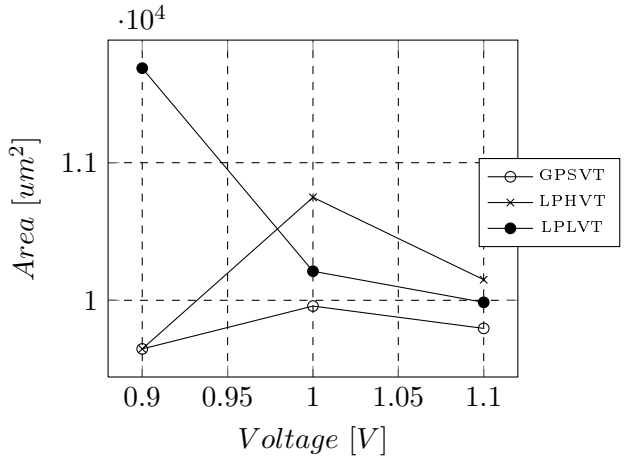



FIGURE 11.1: Area representation for LPHVT, LPLVT,GPSVT

For the Low Power High Threshold Voltage library operating at 1.00 V, the area of our design was resumed in table 11.1. Given the fact that our design does not have any sequential logic, the total area of it is given by the combinational logic, which in this case is 10748.399840µm$^2$.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 10748.399840 | µm$^2$ |
| Noncombinational area | 0.000000 | µm$^2$ |
| Total cell area | 10748.399840 | µm$^2$ |

TABLE 11.1: Area Low Power High Threshold Voltage 1.00V

The number of resources used for implementing the combinational logic of our design can be retrieved from table 11.2. As can be observed from the table below, only 432 cells had been used, which translate in a very small logic.

In the case of Low Power High Threshold Voltage library operating at 1.10V, the area of the design has decreased as can be observed in table 11.3, now the design occupying 10149.879847µm$^2$ but the number of the used resources increased as is discernible from table 11.4.

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 589 |
| Number of cells | 432 |
| Number of combinational cells | 421 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 147 |
| Number of references | 120 |

TABLE 11.2: Used Resources

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 10149.879847 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 10149.879847 | $\mu m^2$ |

TABLE 11.3: Area Low Power High Threshold Voltage 1.10V

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 583 |
| Number of cells | 426 |
| Number of combinational cells | 415 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 126 |
| Number of references | 121 |

TABLE 11.4: Used Resources for 1.10V

For the Low Power High Threshold Voltage library operating at 0.9V, the area and resources are depicted in table 11.5 and table 11.6

Similarly with Low Power High Threshold Voltage, Low Power Low Threshold Voltage (LPLVT) [16] library is also a Standard Cell library for the 65 nm technology, used for VLSI digital design platform which is optimized for use in low power applications.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 9646.519829 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 9646.519829 | $\mu m^2$ |

TABLE 11.5: Area Low Power High Threshold Voltage 0.9V

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 595 |
| Number of cells | 438 |
| Number of combinational cells | 427 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 153 |
| Number of references | 122 |

TABLE 11.6: Used Resources for 0.9V

Area results for Low Power Low Threshold Voltage Library operating at 1.00 V are shown in table 11.7, where it can be observed that using this library we have almost the same area ($10211.239852\mu m^2$) as the one taken by Low Power High Threshold Voltage, operating at the same voltage supply.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 10211.239852 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 10211.239852 | $\mu m^2$ |

TABLE 11.7: Area for Low Power Low Threshold Voltage 1.00V

The number of used resource is detailed in table 11.8.

The area for the Low Power Low Threshold Voltage library 1.10 V is shown in table 11.9, and the resources used for the implementation of our combinational logic is laid out in table 11.10. From table 11.9 and table 11.7, we can conclude

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 652 |
| Number of cells | 495 |
| Number of combinational cells | 484 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 198 |
| Number of references | 115 |

TABLE 11.8: Used Resources for LPLVT 1.00 V

that the area of our design is just slightly decreasing by changing the voltage supply of the used library.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 9985.039862 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 9985.039862 | $\mu m^2$ |

TABLE 11.9: Area Low Power Low Threshold Voltage 1.10V

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 620 |
| Number of cells | 463 |
| Number of combinational cells | 452 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 153 |
| Number of references | 131 |

TABLE 11.10: Used Resources for LPLVT 1.10 V

Similarly, the area for Low Power Low Threshold Voltage operating at 0.9V are described below:

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 11686.999874 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 11686.999874 | $\mu m^2$ |

TABLE 11.11: Area Low Power Low Threshold Voltage 0.9V

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 652 |
| Number of cells | 495 |
| Number of combinational cells | 484 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 167 |
| Number of references | 108 |

TABLE 11.12: Used Resources for LPLVT 0.90 V

The General Purpose Standard Threshold Voltage Library (GPSVT) [17] also represents a Standard Cell library for the 65 nm technology, used in general purpose applications and has 866 Combinational and Sequential cells. Is optimized for high performance, low power or general purpose use.

Area results for General Purpose Standard Threshold Voltage Library operating at 1.00 V are pointed in table 11.13, in this case our design occupying an area of 9957.999843 $\mu m^2$. The used resources for the implementation of

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 9957.999843 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 9957.999843 | $\mu m^2$ |

TABLE 11.13: Area General Purpose Standard Threshold Voltage 1.00V

our combinational logic are retrieved from table 11.14, and comparing it with

Low Power High Threshold Voltage library, it can be seen that the number of cells decreased.

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 608 |
| Number of cells | 451 |
| Number of combinational cells | 440 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 151 |
| Number of references | 117 |

TABLE 11.14: Used Resources for GPSVT 1.00 V

For the same library, but operating at 1.10 V, the area is described in table 11.15, and the usage of resources in table 11.16.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 9795.759843 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 9795.759843 | $\mu m^2$ |

TABLE 11.15: Area General Purpose Standard Threshold Voltage 1.10V

The number of resources that has been used are given below:

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 574 |
| Number of cells | 417 |
| Number of combinational cells | 406 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 126 |
| Number of references | 121 |

TABLE 11.16: Used Resources for GPSVT 1.10 V

In the case of General Purpose Standard Threshold Voltage with $V_{dd} = 0.90V$ the area reports are given in table 11.17 and table 11.18.

| Type | Area | Unit measure |
|---|---|---|
| Combinational area | 9646.519829 | $\mu m^2$ |
| Noncombinational area | 0.000000 | $\mu m^2$ |
| Total cell area | 9646.519829 | $\mu m^2$ |

TABLE 11.17: Area for GPSVT library 0.90 V

| Name | Size |
|---|---|
| Number of ports | 32 |
| Number of nets | 595 |
| Number of cells | 438 |
| Number of combinational cells | 427 |
| Number of sequential cells | 0 |
| Number of macros | 0 |
| Number of buf/inv | 153 |
| Number of references | 122 |

TABLE 11.18: Used Resources for GPSVT 0.90 V

However, the area of the design given by using different voltages, can not be compared, given the fact that, for the different voltages we did not used the same conditions. Note that for all three libraries, it was used for the simulation 0.90V-worst condition, 1.00V-nominal condition and 1.10V-best condition, thus the incomparable values.

## 11.2 Power Consumption results from Design Vision

Reduction of power consumption makes a device more reliable and power management is one of the important design challenges as the technology advances. As known, the two very important components which determine the

power consumption in a CMOS design are the static power consumption and dynamic power consumption.

Static power consumption is the product of the device leakage current and the supply voltage. Total static power consumption, PS, can be obtained as shown in equation 11.1.

$$PS = \sum(leackage\_current) \times (supply\_voltage) \tag{11.1}$$

The dynamic power consumption of a CMOS IC is calculated by adding the transient power consumption (PT), and capacitive-load power consumption (PL), as in equation 11.2.

$$P_{dyn} = P_{internal} + P_{switching} \tag{11.2}$$

The transient power consumption is due to the current that flows only when the transistors are switching from one logic state to another and the equation is depicted in 11.3.

$$P_{internal} = Vdd * I_{internal} + \frac{1}{2} * C_{intL} * V_{dd}^2 * Tr \tag{11.3}$$

The capacitive load power consumption (PL) is the additional power, which is consumed in charging external load capacitance, and is dependent on switching frequency, as shown in equation 11.4.

$$P_{switching} = \frac{1}{2} * C_L * V_{dd}^2 * Tr \tag{11.4}$$

Where:

$T_r$=toggle rate, number of switching of data per unit time.

$C_{intL}$=internal load capacitance

$C_L$=load capacitance

The power consumption results shown in the tables below depicted in the following graph.

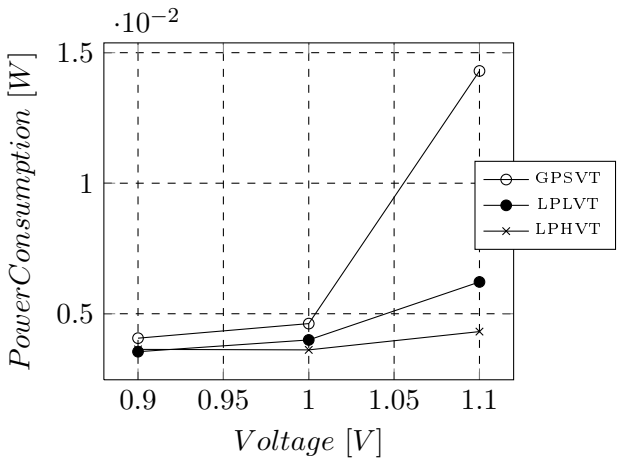



FIGURE 11.2: Power Consumption for LPHVT, LPLVT,GPSVT

The design has been analysed for low effort and the synthesis results for the power consumption using the Low Power High Threshold Voltage Library at 1.00V are the ones depicted in table 11.19.

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.0543 | 1.5610 | 0.2650e-06 | 3.6154 |
| Total | 2.0543 | 1.5610 | 0.2650e-06 | 3.6154 |

TABLE 11.19: Power Report in mW

Dynamic power consumption for Low Power High Threshold Voltage library operating at 1.00V is shown below in the table 11.20, and it contains Internal Power and Switching Power.

The Internal Power is 2.0543 mW, which is the power consumed within a cell for charging and discharging internal cell capacitances and it also includes short-circuit power. The Switching Power is 1.5610 mW, and it represents the power dissipated for charging and discharging the load capacitance from the cell output.

Static Power (Leackage Power) is 0.2650e µW and is the power consumed by the gate when is not switching and is caused by the currents that flow through the transistors even though they are turned off.

The total power using Low Power High Threshold Voltage for $V_{dd} = 1.00V$ is 3.6154 mW and it represents the sum of the preceding power values.

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.0543 | mW | 57% |
| Net Switching Power | 1.5610 | mW | 43% |
| Total Dynamic Power | 3.6154 | mW | 100% |
| Cell Leakage Power | 0.2650 | µW | |

TABLE 11.20: Dynamic and Static Power Consumption

Using the same library, but for $V_{dd} = 1.10V$, the power consumption has increased as can be seen from the table 11.21.

The details about total power consumption for the Low Power High Threshold Voltage for VDD=1.1V are depicted in table 11.22.

In the case of Low Power High Threshold Voltage Library for 0.9V the reported power by Design Vision is shown in table 11.23. The power consumption for

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.4116 | mW | 56% |
| Net Switching Power | 1.8937 | mW | 44% |
| Total Dynamic Power | 4.3160 | mW | 100% |
| Cell Leakage Power | 2.0697 | µW | |

TABLE 11.21: Dynamic and Static Power Consumption for 1.10V

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.4116 | 1.8937 | 1.0697e-06 | 4.3160 |
| Total | 2.4116 | 1.8937 | 1.0697e-06 | 4.3160 |

TABLE 11.22: Power Report for 1.10V in mW

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.1911 | mW | 60% |
| Net Switching Power | 1.4425 | mW | 40% |
| Total Dynamic Power | 3.6344 | mW | 100% |
| Cell Leakage Power | 0.078317 | µW | |

TABLE 11.23: Dynamic and Static Power Consumption for 0.9V

the combinational logic of our design is shown below in table 11.24.

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.1911 | 1.4425 | 0.078317e-06 | 3.6344 |
| Total | 2.1911 | 1.4425 | 0.078317e-06 | 3.6344 |

TABLE 11.24: Power Report for 0.9V in mW

For the given Low Power Low Threshold Voltage Library operating at $V_{dd} = 1.00V$, power consumption is detailed in table 11.25.

The dynamic and static power consumption for the Low Power Low Threshold Voltage at 1.00V as is shown in table 11.26 and in table 11.27 is detailed the

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.1404 | 1.8489 | 0.75076e-06 | 3.9968 |
| Total | 2.1404 | 1.8489 | 0.75076e-06 | 3.9968 |

TABLE 11.25: Power Report for LPLVT 1.00V in mW

power report for Low Power Low Threshold Voltage operating at 1.10V. Just

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.1404 | mW | 54% |
| Net Switching Power | 1.8489 | mW | 46% |
| Total Dynamic Power | 3.9968 | mW | 100% |
| Cell Leakage Power | 0.75076 | µW | |

TABLE 11.26: Power consumption LPLVT 1.00V

by changing the voltage supply from 1.00 V to 1.10 V , the power consumption of the design has almost doubled.

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.8996 | 2.1916 | 1.1265 | 6.2177 |
| Total | 2.8996 | 2.1916 | 1.1265 | 6.2177 |

TABLE 11.27: Power Report for LPLVT 1.10V in mW

Total dynamic and static power consumption for Low Power Low Threshold Voltage 0.90 V, are detailed further in table 11.28 and table 11.29.

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 1.9762 | mW | 56% |
| Net Switching Power | 1.5323 | mW | 44% |
| Total Dynamic Power | 3.5457 | mW | 100% |
| Cell Leakage Power | 0.37188 | µW | |

TABLE 11.28: Power consumption LPLVT for 0.90V

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 1.9762 | 1.5323 | 0.37188e-06 | 3.5457 |
| Total | 1.9762 | 1.5323 | 3.7188e-06 | 3.5457 |

TABLE 11.29: Power Report for LPLVT 0.90V in mW

The power consumption results using the General Purpose Standard Threshold Voltage operating at 1.00V are presented in table 11.30, and the dynamic and static power consumption for the specified library are shown in table 11.31.

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.5994 | 1.8233 | 0.1995e-06 | 4.6222 |
| Total | 2.5994 | 1.8233 | 0.1995e-06 | 4.6222 |

TABLE 11.30: Power Report for GPSVT 1.00V in mW

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.5994 | mW | 59% |
| Net Switching Power | 1.8233 | mW | 41% |
| Total Dynamic Power | 4.6222 | mW | 100% |
| Cell Leakage Power | 1.995 | µW | |

TABLE 11.31: Dynamic and static power consumption GPSVT for 1.00V

For the same library, but 1.10 V voltage supply, the static power consumption and dynamic power consumption results are displayed in table 11.32 and 11.33. When changing the voltage supply of the General Purpose Standard Threshold Voltage library from 1.00 V to 1.10 V, the power consumption increased very much, now having a total power consumption of 14.3000 mW.

For General Purpose Standard Threshold Voltage operating at 0.90 V the power consumption is illustrated in table 11.34, and 11.35.

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 3.5431 | 1.9739 | 8.7830 | 14.3000 |
| Total | 3.5431 | 1.9739 | 8.7830 | 14.3000 |

TABLE 11.32: Power Report for GPSVT 1.10V in mW

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 3.5431 | mW | 64% |
| Net Switching Power | 1.9739 | mW | 36% |
| Total Dynamic Power | 14.300 | mW | 100% |
| Cell Leakage Power | 8.7830 | µW | |

TABLE 11.33: Dynamic and static power consumption GPSVT for 1.10V

| Power Group | Internal Power | Switching Power | Leackage Power | Total Power |
|---|---|---|---|---|
| combinational | 2.0643 | 1.4416 | 0.5580e-06 | 4.0639 |
| Total | 2.0643 | 1.4416 | 0.5580e-06 | 4.0639 |

TABLE 11.34: Power Report for GPSVT 0.90V in mW

| Name | Value | Unit Measure | Percentage |
|---|---|---|---|
| Cell Internal Power | 2.0643 | mW | 59% |
| Net Switching Power | 1.4416 | mW | 41% |
| Total Dynamic Power | 4.0639 | mW | 100% |
| Cell Leakage Power | 0.5580 | µW | |

TABLE 11.35: Dynamic and static power consumption GPSVT for 0.90V

As mentioned in the previous section, the values post simulation for the power consumption can not be compared between the different voltages that were used due to the fact that were used in different operating conditions.

## 11.3 Timing Results from the Synthesis Tool

The critical path is the slowest logic between any two registers, therefore is the limiting factor preventing us from decreasing the clock period constraint. Given the fact that our design does not have a clock signal, in order to calculate the minimum critical path, we were interested to constrain the design for setting the maximum delay from the input signal to the output signal to 0 ns. As shown below, the critical path takes a total of 8.73 ns which is bigger than 0 ns that we constraint the design for, consequently the violated slack.

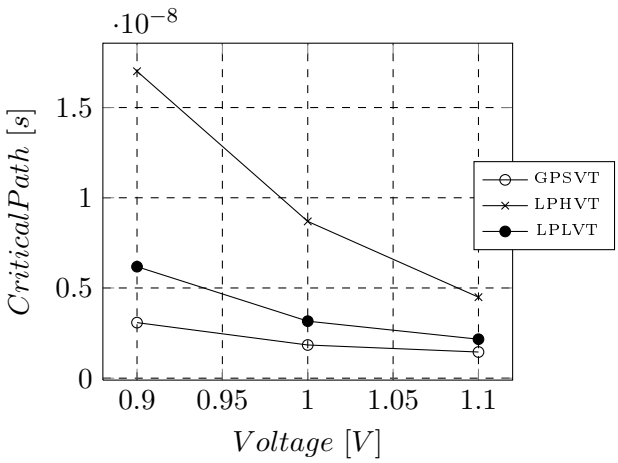



FIGURE 11.3: Critical path for LPHVT, LPLVT,GPSVT

As mentioned before, the design has been constraint for minimum execution time in the synthesis script, and the critical path for our design using Low Power High Threshold Voltage library 1.00V has been detailed in table 11.36. Further details about internal signals timing can be seen in table 11.37.

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[13] (input port) | 0.00 | ns |
| Endpoint | z_mt[12] (output port) | 8.69 | ns |

TABLE 11.36: Critical path

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[12] (out) | 0.02 | 8.69 | ns |
| data arrival time | | 8.69 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -8.69 | ns |
| slack | | -8.69 | ns |

TABLE 11.37: Timing Information for 1.00V

The timing report for Low Power High Threshold Voltage Library operating at 1.10 V are described in table 11.38. The critical path for the library operating

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[8] (out) | 0.00 | 4.49 | ns |
| data arrival time | | 4.49 | ns |
| max_delay | 0.00 | 0.00 | ns |
| output external delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -4.49 | ns |
| slack | | -4.49 | ns |

TABLE 11.38: Timing Information for 1.10V

at the 1.10 V is detailed in table 11.39 and is taking 4.49 ns. Internal signals

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[10] (input port) | 0.00 | ns |
| Endpoint | z_mt[8] (output port) | 4.49 | ns |

TABLE 11.39: Critical path for 1.10V

timing information for the Low Power High Threshold Voltage library for 0.9 V are detailed in the following tables 11.40. The critical path can be identified from the table 11.41.

The longest critical path of our design is 17.01 ns, using the Low Power High Threshold Voltage library for a voltage supply of 0.9V. Using Low Power Low

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[3] (out) | 0.00 | 17.01 | ns |
| data arrival time | 0.00 | 17.01 | ns |
| max_delay | 0.00 | 0.00 | ns |
| output external delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -17.01 | ns |
| slack | | -17.01 | ns |

TABLE 11.40: Timing Information for 0.90V

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[14] (input port) | 0.00 | ns |
| Endpoint | z_mt[3] (output port) | 17.01 | ns |

TABLE 11.41: Critical path for 0.90V

Threshold Voltage library for 1.00V, we have a critical path of 3.16 ns for our design, given in table **??**. Timing details regarding internal signals for Low

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[6] (input port) | 0.00 | ns |
| Endpoint | z_mt[0] (out) (output port) | 3.16 | ns |

TABLE 11.42: Critical path for LPLVT 1.00V

Power Low Threshold Voltage operating at 1.00 V are detailed below, in table 11.43. The critical path in this case, using Low Power Low Threshold Voltage

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[0] (out) | | 3.16 | ns |
| data arrival time | | -3.16 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -3.16 | ns |
| slack | | -3.16 | ns |

TABLE 11.43: Timing Information for LPLVT 1.00V

library with $V_{dd} = 1.10V$ is shown in table 11.44 and the timing details in table 11.45.

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[6] (input port) | 0.00 | ns |
| Endpoint | z_mt[0] (out) (output port) | 2.16 | ns |

TABLE 11.44: Critical path for LPLVT 1.10V

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[0] (out) | 0.00 | 2.16 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -2.16 | ns |
| slack | | -2.16 | ns |

TABLE 11.45: Timing Information for LPLVT 1.10V

Using the Low Power Low Threshold voltage for 0.9 V we have the following critical path:

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[6] (input port) | 0.00 | ns |
| Endpoint | z_mt[0] (out) (output port) | 6.18 | ns |

TABLE 11.46: Critical path for LPLVT 0.9V

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[0] (out) | 0.00 | 6.18 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -6.18 | ns |
| slack | | -6.18 | ns |

TABLE 11.47: Timing Information for LPLVT 0.90V

For the case of General Purpose Standard Threshold Voltage library, $V_{dd} = 1.0V$ critical path has been detailed in table 11.48.

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[5] (input port) | 0.00 | ns |
| Endpoint | z_mt[0] (output port) | 1.84 | ns |

TABLE 11.48: Critical path for GPSVT 1.00V

Internal signals timing details are shown below in table 11.49. Using the same

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[0] (out) | | 1.84 | ns |
| data arrival time | | 1.84 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -1.84 | ns |
| slack | | -1.84 | ns |

TABLE 11.49: Timing Information for GPSVT 1.00V

library but at different voltage, we have the critical path of 1.45 ns as can be observed in table 11.50. Timing details about internal signals, using General

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[11] (input port) | 0.00 | ns |
| Endpoint | z_mt[1] (output port) | 1.45 | ns |

TABLE 11.50: Critical path for GPSVT 1.10V

Purpose Standard Threshold Voltage library for $V_{dd} = 1.10V$, in table 11.51.

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[1] (out) | | 1.45 | ns |
| data arrival time | | 1.45 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -1.45 | ns |
| slack | | -1.45 | ns |

TABLE 11.51: Timing Information for GPSVT 1.10V

For the General Purpose Standard Threshold Voltage operating at 0.90 V the critical path and internal signal's timing information are set out in table 11.52 and 11.53.

| Name | Signal | Timing Info | Unit Measure |
|---|---|---|---|
| Startpoint | v_mt[12] (input port) | 0.00 | ns |
| Endpoint | z_mt[8] (output port) | 3.08 | ns |

TABLE 11.52: Critical path for GPSVT 0.90V

| Point | Incr | Path | Unit Measure |
|---|---|---|---|
| z_mt[8](out) | | 3.08 | ns |
| data arrival time | | 3.08 | ns |
| max_delay | 0.00 | 0.00 | ns |
| data required time | | 0.00 | ns |
| data arrival time | | -3.08 | ns |
| slack | | -3.08 | ns |

TABLE 11.53: Timing Information for GPSVT 0.90V

## 11.4 Prime Time Power Analysis Tool

Prime Time PX is a power analysis tool from Synopsys, which can perform supremely power analysis because it is taking the advantage of the timing and signal righteousness engines of the PrimeTime solutions for processing our design data and parasitic back annotation. Furthermore, it supports multiple signal activity formats.

Altogether, Prime Time PX provides the best platform to accurately use and analyse all of the key inputs required for explicit power analysis. As illustrated in Fig. 11.4, in order to analyse the power consumption of our design using Prime Time PX, we had to take into account the net list of the design, which determines the design connectivity and the type of cells used in the design.

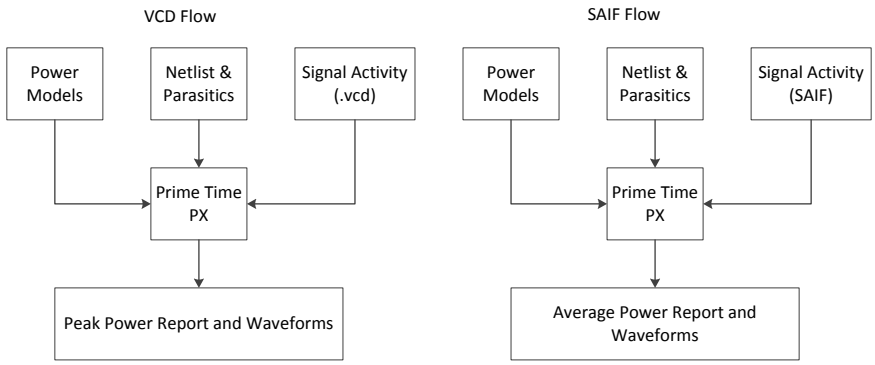



FIGURE 11.4: Power Analysis Flow

To compute the internal power of the CMOS cells used in the design, we used Low Power High Threshold Voltage Library from STMicroelectronics, 65 nm and the signal activity of the design, which affects both the static and dynamic power consumption of the design and parasitics (resistance and capacitance).

The power analysis depends on accurate signal activity. The PrimeTime PX power analysis supports both average and peak power analysis based on the signal activity provided.

For average power analysis, PrimeTime PX supports the propagation of switching activity derived from the logic simulation (gate-level), saved as a Switching Activity Interchange Format (SAIF) file. For peak power analysis, PrimeTime PX requires a timing logic simulation and generation of the Value Change Dump (VCD) that takes in the activity and time of every event on each net. Power analysis was performed using the familiar PrimeTime commands, user interface, reports, and attributes all incorporated in a Perl script file.

As depicted in Fig. 11.5, the first step in the analysis of power consumption using Prime Time PX was to set up the power analysis mode. There are two options for that, either *Average Power Analysis*, in which the tool performs a vector free power analysis, using the default toggle rate or *Time Based*

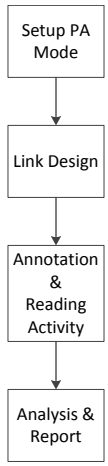



FIGURE 11.5: Prime Time PX Flow

*Analysis*, in which were taken into account all factors contributing to power consumption.

As in the synthesis script, the next step was to set-up the search path and the link path for the used library. After that, the netlist of the design and the libraries were read by the tool and then the design was linked. As part of the script was also the annotation of the parasitics (reading the Standard Parasitic Exchange Format file) and switching activities (reading the Value Change Dump file). In the end, the power was analysed and reported.

For our design, we have set up the time based power analysis and a frequency of 100 MHz. The power consumption report generated by Prime Time PX is detailed further in table 11.54 and table 11.55 and the units info are in table 11.56.

| Hierarchy | Switch Power | Int Power | Leak Power | Total Power | Units |
|---|---|---|---|---|---|
| top_level | 2.80e-06 | 2.59e-06 | 8.02e-09 | 5.39e-06 | W |

TABLE 11.54: Power Information Prime Time PX

| Hierarchy | Peak Power | Peak Time | Glitch Power | X-Tran Power | Units |
|---|---|---|---|---|---|
| top_level | 2.77e-02 | 15360.808999 – 15360.809000 | 1.21e-08 | 0.0000 | W |

TABLE 11.55: Power Information

| Name | Units |
|---|---|
| Voltage Units | 1 V |
| Capacitance Units | 1 pF |
| Time Units | 1 ns |
| Dynamic Power Units | 1 W |
| Leakage Power Units | 1 W |

TABLE 11.56: Power-specific unit information

We have also analysed the power consumption of our design at 200 MHz and the results were summarised in table 11.57 and table 11.58.

| Hierarchy | Peak Power | Peak Time | Glitch Power | X-Tran Power | Units |
|---|---|---|---|---|---|
| top_level | 5.03e-02 | 122880.949000 – 122880.949001 | 9.44e-08 | 0.00 | W |

TABLE 11.57: Power Information-part2

| Hierarchy | Switch Power | Int Power | Leak Power | Total Power | Units |
|---|---|---|---|---|---|
| top_level | 1.35e-05 | 1.06e-05 | 2.22e-8 | 2.40e-05 | W |

TABLE 11.58: Power Information

Similarly with the power, the energy efficiency is a very important design requirement this days. The energy consumption is given by the formula from equation 11.6 and 11.5.

$$Energy = \int_0^T P(t)\,dt. \tag{11.5}$$

$$Energy = Power * Time \tag{11.6}$$

| Simulation Time | Total Power Consumption | Total Energy Consumption [Unit] |
|---|---|---|
| 75535e-09 | 5.39e-06 | 4.071e-10 [ J ] |

TABLE 11.59: Energy Consumption

For our design, the energy consumption has been calculated and is shown in table 11.59 and table 11.60.

| Simulation Time | Total Power Consumption | Total Energy Consumption [Unit] |
|---|---|---|
| 75535e-09 | 2.40e-05 | 1.81e-11 [ J ] |

TABLE 11.60: Energy Consumption-part2

In table 11.61 and 11.62 are synthesised all the data results from the three used libraries with three different voltages.

| Lib | $V_{dd}(V)$ | $Area(\mu m^2)$ | $Cells$ | $t_p$(ns) |
|---|---|---|---|---|
| HVT | 0.9 | 11686.999874 | 495 | 17.01 |
| HVT | 1.0 | 10748.399840 | 432 | 8.69 |
| HVT | 1.1 | 10149.879847 | 426 | 4.49 |
| LVT | 0.9 | 10753.599861 | 463 | 6.18 |
| LVT | 1.0 | 10753.599861 | 495 | 3.16 |
| LVT | 1.1 | 9985.039862 | 463 | 2.16 |
| SVT | 0.9 | 9646.519829 | 438 | 3.08 |
| SVT | 1.0 | 9957.999843 | 451 | 1.84 |
| SVT | 1.1 | 9795.759843 | 417 | 1.45 |

TABLE 11.61: Results-part 1

| Lib | $V_{dd}(V)$ | $P_{DV}$(mW) | $P_{inter}$(mW) | $P_{leak}$(µW) |
|---|---|---|---|---|
| HVT | 0.9 | 3.6344 | 2.1911 | 0.078317 |
| HVT | 1.0 | 3.6154 | 2.0543 | 0.2650 |
| HVT | 1.1 | 4.3160 | 2.4116 | 1.0697 |
| LVT | 0.9 | 3.5457 | 1.9762 | 0.37188 |
| LVT | 1.0 | 3.9968 | 1.8489 | 0.75076 |
| LVT | 1.1 | 6.2177 | 2.1916 | 1.1265 |
| SVT | 0.9 | 4.0639 | 2.0643 | 0.5580 |
| SVT | 1.0 | 4.6222 | 2.5994 | 1.995 |
| SVT | 1.1 | 14.300 | 3.5431 | 8.7830 |

TABLE 11.62: Results -part 2

Chapter 12

# Conclusions

Our master thesis demonstrates the hardware implementation approach of the original algorithm for approximation of unary functions, focusing on the implementation of the square root function. In this thesis report, using the cutting-edge *Parabolic Synthesis* and *Second Degree Interpolation* methods, we proved that square root of integer numbers between $0 < x < 65536$ are computed, but also other elementary functions such as logarithm and trigonometric functions are suitable for implementation of the same algorithm.

Among other methods that exist these days for calculating the square root of a number, our implementation has the main advantage of using the same algorithm approach and architecture for computing other different unary functions, just by changing the sets of the coefficients. The design can be reused in many other application without any major changes. The Parabolic Synthesis method is very flexible due to the big advantage of conferring high performance of the algorithm by using a high parallelism in the architecture.

Another major advantage of this implementation is the usage of parallelism in the architecture, which represents the main factor in the reduction of the execution time and critical path making this algorithm implementation the fastest one between the two's compared in the previous chapter.

The last but not the least benefit of using this method is the usage of only low complexity operations for the implementation of the logic design like shifting, adding and multiplication, among which the multiplication is more often used which translates into a very easy hardware implementation.

## 12.1 Improvements

As explained in chapter 2, the normalized interval $0<x<1$, was split in 8 subintervals, where the parabolic sub-function was developed as an approximation of the help function. However, given the fact that the number of intervals sets up the size of the look up table needed in the last sub-function for storing the coefficients, an enlargement of the number of intervals is equivalent with an increase in the accuracy of the algorithm.

Therefore, as an improvement we could consider is increasing the number of intervals from 8 used in this implementation to 16 intervals. With 16 intervals, the algorithm implementation is getting even more accurate and faster but at the price of a bigger area.

Another future improvement could be considering to implement the algorithm in floating point, which we assume it would reduce the area of the design, since with the floating point representation the error gets a better accuracy.

One imminent advancement could be the implementation and design of the cubic root function approximation, using the same architecture but changing the set of coefficients and taking into account also the logarithm implementation.

Due to the high degree of parallelism inherent in the design architecture, a possible enhancement would be to pipeline the design, for faster computational time and higher throughput.

An interesting continuation of the project would be to prototype it on FPGA for reporting the used resources, chip area, power consumption and critical path.

Appendix A

# Appendix

Listing A.1: Mathematical Model (Matlab)

```
function [x_mt, v_val, y, db, z, error, y1, y2, y3] = full_rounding(
    v_mt, intervals)

clc
close all
format long

%% DEFAULT VALUES

v_mt = bit_true(v_mt,16,'t');

C1 = 0.5;
exp = 0;
i = v_mt;
I = intervals;
c1 = 0.5;
small_step = 1/I;
x_st=0:small_step:(I-1)*small_step;
x_end=x_st+small_step;
x_middle = (x_end + x_st)/2;
%% BASE 4 FP

if v_mt >= 1
 while(i>=4)
     i = i/4;
     exp = exp + 1;
 end
else
     i = i*4;
     exp = exp - 1;
end

%% Normalizing


x_mt = (bit_true(i,6,'t')-1)*bit_true(1/3, 13, 'r');
x_mt = bit_true(x_mt,10, 'r');

%% Calculating Sub-function S1 %%%

%s1 = 1.5*x_mt-0.5*x_mt^2

s1_p1 = x_mt^2;
s1_p1 = bit_true(s1_p1, 17, 't');

s1_p2 = 2^-1*s1_p1;
s1_p2 = bit_true(s1_p2, 18, 't');

s1_p3 = x_mt + bit_true(x_mt*2^-1, 11, 't');
s1_p3 = bit_true(s1_p3, 11, 't');

s1 = s1_p3 - s1_p2;
%s1 = bit_true(s1, 17, 't');
s1 = bit_true(s1, 13, 'r');

%% Calculating Sub-function S2 %%%

for g=1 : I
    if g == 1
        f1_st(g) = 1;
    else
        f1_st(g)=(sqrt(1 + 3 * x_st(g)) -1)./(x_st(g)+c1*(x_st(g)-x_st
         (g).^2));
    end
    f1_end(g)=(sqrt(1 + 3 * x_end(g)) -1)./(x_end(g)+c1*(x_end(g)-
     x_end(g).^2));
```

```
    f1_middle(g)=(sqrt(1 + 3 * x_middle(g)) -1)./(x_middle(g)+c1*(
     x_middle(g)-x_middle(g).^2));
end

l = f1_st;
k = f1_end - f1_st;
c2 = 4*(f1_middle - l - k*c1);

%% Coefficients Optimization

if I == 4

    c2(1) = c2(1) + 2^-9.57;    %4 intervals

else if I == 8

    c2(1) = c2(1) + 2^-12;    %8 intervals
    c2(2) = c2(2) + 2^-15;    %8 intervals
    c2(6) = c2(6) - 2^-20;    %8 intervals
    c2(7) = c2(7) - 2^-19.5;    %8 intervals

    else

        c2(1) = c2(1) + 2^-14.6;    %16 intervals
        c2(2) = c2(2) + 2^-17;    %16 intervals
        c2(3) = c2(3) + 2^-18;    %16 intervals
        c2(12) = c2(12) - 2^-24;    %16 intervals
        c2(13) = c2(13) - 2^-23.5;    %16 intervals
    end
end

%% Xp and j %%

j = k + c2;
```

```
for ind2 = 1 : I
    if x_mt < small_step*(ind2)
        index1 = ind2;
        x_p = (x_mt - small_step*(ind2-1))*2^(log2(I)); % Subtraction
         and shifting
        %x_p1 = bit_true(x_p, 8, 't');
        x_p = bit_true(x_p, 7, 't');
        break
    else if x_mt == small_step*(ind2)
            index1 = ind2+1;
            x_p = (x_mt - small_step*(ind2))*2^(log2(I));
            x_p = bit_true(x_p, 7, 't');
            break
        end
    end
end

%% Second-Sub Function S2 %%%

%s2=(l(index1) + j(index1) * x_p - c2(index1) * x_p^2);
% First Operation
j = bit_true(j,12,'r');
multi1 = j(index1) * x_p;
multi1 = bit_true(multi1,15, 't');

% Second Operation
l = bit_true(l, 15, 'r');%18
s2_p1=l(index1) + multi1;
s2_p1 = bit_true(s2_p1,13, 't');

% Third Operation
c2 = bit_true(c2, 13, 'r'); %22
x_p2 = bit_true(x_p^2, 12, 't');
s2_p2 = c2(index1) * x_p^2;
```

```
s2_p2 = bit_true(s2_p2,12, 't');

% Last Operation
s2 = s2_p1 - s2_p2;
s2 = bit_true(s2,12, 'r');

%% Final Result %%%%%%

y = s1*s2;
y = bit_true(y, 16, 't');
z = (y+1)*2^(exp);
if v_mt == 0
    z = 0;
else
    z = bit_true(z, 8, 't');
end

%% Comparisson %%%%%%%%%%

v_val=sqrt(v_mt);
error = (sqrt(1+3*x_mt)-1) - y;
db = 20*log10(abs(error));
y1 = 20*log10(abs((s1*s2)-(sqrt(1+3*x_mt)-1))+10^-15);
y3 = (sqrt(1+3*x_mt)-1) - (s1*s2);
y2 = y3+10^-15;
```

LISTING A.2: Error Estimation Matlab Script

```
clc
clear all
close all
format long

% Default Values
intervals = 8;
max_error = 0;
e1 = 0;

% Call to the Mathematical Model for every input value
%for e =  1 : 1 : 2^16-1
%for e =  0 : 2^-8 : 1
for e =  2^14 : 1 : 2^16-1

    e1 = e1 +1;
    %[x(e1), sq_rt(e1), forg(e1), db(e1), r(e1),  error_final(e1), y1(
     e1), y2(e1), y3(e1)] = squared_root_ps(e,intervals);
    val(e1) = e1;
    [x(e1), sq_rt(e1), forg(e1), db(e1), r(e1),  error_final(e1), y1(
     e1), y2(e1), y3(e1)] = rounding(e,intervals);
    %[x(e1), sq_rt(e1), forg(e1), db(e1), r(e1),  error_final(e1), y1(
     e1), y2(e1), y3(e1)] = full_rounding(e,intervals);
    if abs(error_final(e1)) > max_error
        max_error = abs(error_final(e1));
        index= e1;
        input_value = e;
    end
end

% Statistics
err_min = min(y3);
err_max = max(y3);
bin = err_max - err_min;
bin = bin/intervals;
xx = err_min:bin:err_max;

standard_deviation = std(y3) % standard deviation
average = mean(y3) % Average or mean value
sqr = y3.^2;
```

```
sqr_sum = sum(sqr);
N = size(y3);
sqr_mean = sqr_sum/N(1,2);
x_rms = sqrt(sqr_mean) % rms value
bits = 20*log10(abs(average))/(-20*log10(2))

% Visualization

figure(1)
plot(x,y2)
xlabel('Normalized Input Value (x)')
ylabel('f_o_r_g(x) - sqrt(x)')
title(['Error Behaviour for ', num2str(intervals),' intervals'])
%hleg1 = legend('f_o_r_g - sqrt(x)')
grid on

figure(2)
plot(x,y1)
xlabel('Normalized Input Value')
ylabel('Absolute Error (dB)')
title(['Absolute Error in dB for ',num2str(intervals(1,1)),' intervals
    '])
grid on

figure(3)
hist(y2,xx)
xlabel('Error')
ylabel(['Number of Elements out of ',num2str(N(1,2)),' values'])
title('Probability Distribution')
grid on

% Bit True Simulation Verification

orig = dec2binvec(sqrt(input_value)*2^8,16);
ours = dec2binvec(r(index)*2^8,16);
resulting_in(:,1) = orig';
resulting_in(:,2) = ours'
accuracy = abs(orig - ours)
max_error

figure(4)
plot(accuracy)

% Hardware Implementation Verification

importfile('results.txt');
r1=(r*2^8)';
for i = 1:65535
    verificacion(i) = r1(i) - results(i);
end
figure(5)
plot(verificacion)
find(verificacion, 1, 'first')
```

LISTING A.3: Bit True Simulation Matlab Script

```
function [result] = bit_true(number, lengt, t);

% Shifting

shift = abs(number) * 2^(lengt);

% Rounding/Truncation

if t == 't'
integ = floor(shift);
else
integ = uint64(shift);
```

```
end

% Shifting Back
    if number > 0
        result = 2^(-lengt)*(double(integ));
    else
        result = -(2^(-lengt)*(double(integ)));
    end
end
```

LISTING A.4: Highest Hierarchy VHDL file

```
--
     ------------------------------------------------------------------------
-- Title      : Top Level of Abstraction (Hiercharchy)
-- Project    :
--
     ------------------------------------------------------------------------

-- File       : top.vhd
-- Author     :
-- Company    :
-- Created    : 2013-02-22
-- Last update: 2013-04-23
-- Platform   :
-- Standard   : VHDL'87
--
     ------------------------------------------------------------------------

-- Description: All the components are bound together
--
     ------------------------------------------------------------------------
-- Copyright (c) 2013
--
     ------------------------------------------------------------------------

-- Revisions  :
-- Date        Version  Author  Description
-- 2013-02-22  1.0      2011    Created
--
     ------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.comp_pack.all;
use work.constants.all;


entity top_level is

  generic (
    width_i : integer := WIDTH_OF_INPUT_SIGNAL;
    width_o : integer := WIDTH_GLOBAL_OUTPUT);

  port (
    v_mt : in  std_logic_vector(width_i - 1 downto 0);
    z_mt : out std_logic_vector(width_o - 1 downto 0));

end top_level;

architecture structural of top_level is

```

```
signal divid            : std_logic_vector(WIDTH_OF_INPUT_ROUNDING - 1 downto 0)                                    := (others => '0');
signal z_mt_tmp         : std_logic_vector(width_o - 1 downto 0)                                                    := (others => '0');
signal exp              : std_logic_vector(WIDTH_OF_THE_EXPONENT - 1 downto 0)                                      := (others => '0');
signal x_p_2            : std_logic_vector(WIDTH_OF_SQUARED_TRUNC_SIGNAL-1 downto 0)                                := (others => '0');
signal x_mt_2           : std_logic_vector(WIDTH_OF_SQUARED_INPUT_SIGNAL-1 downto 0)                                := (others => '0');
signal fp4_output       : std_logic_vector(WIDTH_OF_THE_OUPUT_FP4 - 1 downto 0)                                     := (others => '0');
signal x_mt_top_tmp     : std_logic_vector(WIDTH_OF_INPUT_ROUNDING - 1 downto 0)                                    := (others => '0');
signal x_mt             : std_logic_vector(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - 1 downto 0)                       := (others => '0');
signal s1_adder1        : std_logic_vector(WIDTH_ADDER_S1_1 - 1 downto 0)                                           := (others => '0');
signal s1_adder2        : std_logic_vector(WIDTH_ADDER_S1_2 - 1 downto 0)                                           := (others => '0');
signal s1_tmp           : std_logic_vector(WIDTH_OF_INPUT_ROUNDING - 1 downto 0)                                    := (others => '0');
signal s1               : std_logic_vector(WIDTH_OF_ROUNDED_S1 - 1 downto 0)                                        := (others => '0');
signal x_mt_over_flow   : std_logic_vector(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT + 1 downto 0)                       := (others => '0');
signal x_mt_shifting    : std_logic_vector(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT + 1 downto 0)                       := (others => '0');
signal s1_adder1_concat : std_logic_vector(WIDTH_ADDER_S1_2 - 1 downto 0)                                           := (others => '0');
signal x_mt_2_shifitng  : std_logic_vector(WIDTH_ADDER_S1_2 - 1 downto 0)                                           := (others => '0');
signal index            : std_logic_vector(BITS_OF_TRUNCATION - 1 downto 0)                                         := (others => '0');
signal mult1_s2_tmp     : std_logic_vector(WIDTH_TEMP_ADDER_S2_2 - 1 downto 0)                                      := (others => '0');
signal x_p              : std_logic_vector(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - BITS_OF_TRUNCATION - 1 downto 0) := (others => '0');
signal mult2_s2_tmp     : std_logic_vector(WIDTH_TEMP_ADDER_S2_1 - 1 downto 0)                                      := (others => '0');
signal mult2_s2         : std_logic_vector(WIDTH_ADDER_S2_1 - 1 downto 0)                                           := (others => '0');
signal adder1_s2_tmp    : std_logic_vector(WIDTH_J_I_COEFF + WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - BITS_OF_TRUNCATION downto 0);
```

```
65    signal adder1_s2        : std_logic_vector(WIDTH_ADDER_S2_2 - 1
         downto 0)                                          := (others =>
         '0');
66    signal mult1_s2         : std_logic_vector(WIDTH_ADDER_S2_2 - 1
         downto 0)                                          := (others =>
         '0');
67    signal s2_tmp           : std_logic_vector(WIDTH_ADDER_S2_2 - 1
         downto 0)                                          := (others =>
         '0');
68    signal s2_tmp_round     : std_logic_vector(WIDTH_ADDER_S2_2 - 1
         downto 0)                                          := (others =>
         '0');
69    signal s2               : std_logic_vector(WIDTH_OF_ROUNDED_S2 - 1
         downto 0)                                        := (others =>
         '0');
70    signal y_tmp            : std_logic_vector(WIDTH_OF_ROUNDED_S1 +
         WIDTH_OF_ROUNDED_S2 - 1 downto 0)                  := (others =>
         '0');
71    signal y_mt_1           : std_logic_vector(WIDTH_PROCESSING_OUTPUT -
          1 downto 0)                                     := (others => '0')
         ;
72
73  begin  -- structural
74
75    --
         ------------------------------------------------------------------
76    --                          FP4 CONVERSION
              --
77    --
         ------------------------------------------------------------------
78
79    FP4_1 : FP4
80      generic map (
81        width_i_f => WIDTH_OF_INPUT_SIGNAL,
82        width_e_f => WIDTH_OF_THE_EXPONENT,
83        width_o_f => WIDTH_OF_THE_OUPUT_FP4)
84      port map (
85        v_mt  => v_mt,
86        exp   => exp,
87        subst => fp4_output);
88
89    --
         ------------------------------------------------------------------
90    --                          DIVIDING BY 3
              --
91    --
         ------------------------------------------------------------------
92
93
94    divid <= std_logic_vector(unsigned(fp4_output) * unsigned(THIRD_COEF
         ));
95
96
97    --
         ------------------------------------------------------------------
98    --                          X_mt ROUNDING
              --
99    --
         ------------------------------------------------------------------
100
```

```
x_mt <= rounding1(to_integer(unsigned(divid)),
   WIDTH_OF_INPUT_ROUNDING - 2, WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT
   );

--
   ------------------------------------------------------------------------
--                          SQUARING
         --
--
   ------------------------------------------------------------------------


Square_Unit : squarer
  generic map (
    width_i_s => WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT,
    width_o_s => WIDTH_OF_SQUARED_INPUT_SIGNAL,
    width_p_s => WIDTH_OF_SQUARED_TRUNC_SIGNAL)
  port map (
    x_mt   => x_mt,
    x_mt_2 => x_mt_2,
    x_p_2  => x_p_2);

--
   ------------------------------------------------------------------------
--                          SUBFUNCTION S1
         --
--
   ------------------------------------------------------------------------


x_mt_over_flow   <= "00" & x_mt;

x_mt_shifting    <= '0' & x_mt & '0';
s1_adder1        <= std_logic_vector(unsigned(x_mt_shifting) +
   unsigned(x_mt_over_flow));
-- s1_adder overflows with certain values, therefore an extra 0
s1_adder1_concat <= s1_adder1 & (WIDTH_ADDER_S1_2 - WIDTH_ADDER_S1_1
    - 1 downto 0 => '0');
x_mt_2_shifitng  <= "00" & x_mt_2;
s1_adder2        <= std_logic_vector(unsigned(s1_adder1_concat) -
   unsigned(x_mt_2_shifitng));
-- The extra 0 is removed before calling "rounding1", to make sure
   the
-- rounding is done right
s1               <= rounding1(to_integer(unsigned(s1_adder2)),
   WIDTH_ADDER_S1_2 - 1, WIDTH_OF_ROUNDED_S1);

--
   ------------------------------------------------------------------------

--                          SUBFUNCTION S2
         --
--
   ------------------------------------------------------------------------


index        <= x_mt(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - 1 downto
    WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - BITS_OF_TRUNCATION);
mult1_s2_tmp <= std_logic_vector(unsigned(x_p_2) * unsigned(
   C2I_COEFF(to_integer(unsigned(index)))));
mult1_s2     <= (WIDTH_OPTIM_C2_I - 1 downto 0 => '0') &
   mult1_s2_tmp(mult1_s2_tmp'left downto WIDTH_TEMP_ADDER_S2_2 +
   WIDTH_OPTIM_C2_I - WIDTH_ADDER_S2_2 + 1) & '0';


x_p           <= x_mt(WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT -
   BITS_OF_TRUNCATION - 1 downto 0);
```

```
142   mult2_s2_tmp  <= std_logic_vector(unsigned(x_p) * unsigned(J_I_COEFF
          (to_integer(unsigned(index)))));
143   mult2_s2      <= (WIDTH_OPTIM_J_I - 1 downto 0 => '0') &
          mult2_s2_tmp(mult2_s2_tmp'left downto WIDTH_TEMP_ADDER_S2_1 +
          WIDTH_OPTIM_J_I - WIDTH_ADDER_S2_1);
144   adder1_s2_tmp <= std_logic_vector(unsigned(L_I_COEFF(to_integer(
          unsigned(index)))) - unsigned('0' & mult2_s2));
145   adder1_s2     <= adder1_s2_tmp(WIDTH_MULT_S2 - 1 downto
          WIDTH_TO_T_ADDER2);
146   s2_tmp        <= std_logic_vector(unsigned(adder1_s2) + unsigned(
          mult1_s2));
147   s2            <= rounding1(to_integer(unsigned(s2_tmp)),
          WIDTH_ADDER_S2_2, WIDTH_OF_ROUNDED_S2);
148   y_tmp         <= std_logic_vector(unsigned(s1) * unsigned(s2));
149   y_mt_1        <= y_tmp(WIDTH_OF_ROUNDED_S1 + WIDTH_OF_ROUNDED_S2 - 1
           downto WIDTH_OF_ROUNDED_S1 + WIDTH_OF_ROUNDED_S2 -
          WIDTH_PROCESSING_OUTPUT);
150
151   --
          ------------------------------------------------------------------------
152   --                      POST-PROCESSING UNIT
                --
153   --
          ------------------------------------------------------------------------
154
155   post_processing1 : post_processing
156     generic map (
157       width_y    => WIDTH_PROCESSING_OUTPUT,
158       width_exp  => WIDTH_OF_THE_EXPONENT,
159       width_z_mt => WIDTH_FINAL_RESULT)
160     port map (
161       y_mt => y_mt_1,
162       exp  => exp,
163       z_mt => z_mt_tmp);
164
165   --
          ------------------------------------------------------------------------
166   --                        MUX FOR INPUT = 0
                --
167   --
          ------------------------------------------------------------------------
168
169   z_mt <= (width_o - 1 downto 0 => '0') when v_mt = (width_i - 1
          downto 0 => '0') else
170           z_mt_tmp;
171
172
173
174 end structural;
```

LISTING A.5: Components Package (VHDL)

```
1   --
          ------------------------------------------------------------------------
2   -- Title      : sqrt_pack
3   -- Project    :
4   --
          ------------------------------------------------------------------------
5   -- File       : comp_pack.vhd
6   -- Author     : Alejandro Vazquez Bofill
7   -- Company    :
8   -- Created    : 2013-02-21
```

```
-- Last update: 2013-04-23
-- Platform   :
-- Standard   : VHDL'87
--
-------------------------------------------------------------------------------
-- Description: Square root VHDL Package
--
-------------------------------------------------------------------------------
-- Copyright (c) 2013
--
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2013-02-21  1.0      2011    Created
--
-------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants.all;

--
-------------------------------------------------------------------------------
--                      PACKAGE DECLARATION                                  --
-------------------------------------------------------------------------------

package comp_pack is

  -- Floating Point base 4 conversion unit

  component FP4
    generic (
      width_i_f : integer := WIDTH_OF_INPUT_SIGNAL;
      width_e_f : integer := WIDTH_OF_THE_EXPONENT;
      width_o_f : integer := WIDTH_OF_THE_OUPUT_FP4);  -- Input length
    port (
      v_mt  : in  std_logic_vector((width_i_f - 1) downto 0);
      exp   : out std_logic_vector((width_e_f - 1) downto 0);
      subst : out std_logic_vector((width_o_f - 1) downto 0));
  end component;

  -- Squarer Unit

  component squarer
    generic (
      width_i_s  : integer := WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT;
      width_o_s  : integer := WIDTH_OF_SQUARED_INPUT_SIGNAL;
      bits_trunc : integer := BITS_OF_TRUNCATION;
      width_p_s  : integer := WIDTH_OF_SQUARED_TRUNC_SIGNAL);
    port (
      x_mt   : in  std_logic_vector(width_i_s - 1 downto 0);
      x_mt_2 : out std_logic_vector(width_o_s - 1 downto 0);
      x_p_2  : out std_logic_vector(width_p_s - 1 downto 0));
  end component;

  -- Rounding1 Function

```

```
  function rounding1 (
    input_value : integer;
    bits_input  : integer;
    bits_output : integer)
    return std_logic_vector;

  -- Post-Processing Unit

  component post_processing
    generic (
      width_y    : integer := WIDTH_PROCESSING_OUTPUT;
      width_exp  : integer := WIDTH_OF_THE_EXPONENT;
      width_z_mt : integer := WIDTH_FINAL_RESULT);
    port (
      y_mt : in  std_logic_vector(width_y - 1 downto 0);
      exp  : in  std_logic_vector(width_exp - 1 downto 0);
      z_mt : out std_logic_vector(width_z_mt - 1 downto 0));
  end component;


end comp_pack;


--------------------------------------------------------------------------------
--                              PACKAGE BODY                                  --
--------------------------------------------------------------------------------

package body comp_pack is

  --
  ------------------------------------------------------------------------------
  --                          ROUNDING FUNCTION                               --
  ------------------------------------------------------------------------------
  -- Name: Rounding1
  -- Description: Rounds the input signals to the specified amouont of bits and
  -- returns the value in the specified number of bits
  -- Inputs: input_value --> Integer; Input value to be rounded
  --         bits_input --> Integer; Number of bits to be rounded to
  --         bits_output --> Integer;
  -- Outputs: rounded_value --> std_logic_vector(varied); Rounded value
  --
  ------------------------------------------------------------------------------


  function rounding1 (
    input_value : integer;
    bits_input  : integer;
    bits_output : integer)
    return std_logic_vector is

    variable result          : std_logic_vector(bits_output - 1 downto 0);
    variable bin_input_value : std_logic_vector(bits_input - 1 downto 0);

  begin
```

```
    bin_input_value := std_logic_vector(to_unsigned(input_value,
     bits_input));
    if bin_input_value(bits_input - bits_output - 1) = '0' then
      result := bin_input_value(bin_input_value'left downto bits_input
       - bits_output);
    else
      result := std_logic_vector(unsigned(bin_input_value(
     bin_input_value'left downto bits_input - bits_output)) + 1);
    end if;
    return result;
  end rounding1;



end comp_pack;

--
     ------------------------------------------------------------------
--                                FP4
          --
--
     ------------------------------------------------------------------
-- Name: FP4 (Floating Point base 4)
-- Description: Takes a 16 bits integer as input and outputs it
     floating point
-- base 4. Substrcts "1" to the integer part and concatenate the rest
     or zeros
-- Inputs: v_mt --> 16 bits
-- Outputs: subst --> 8 bits(2 of the integer and 6 of the fractional
     part)
--
     ------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants.all;

entity FP4 is

  generic (
    width_i_f : integer := WIDTH_OF_INPUT_SIGNAL;  -- Width of the
     inout signal
    width_e_f : integer := WIDTH_OF_THE_EXPONENT;  -- Width of the
     exponent
    width_o_f : integer := WIDTH_OF_THE_OUPUT_FP4);  -- Width of the
     output signal

  port (
    v_mt  : in  std_logic_vector((width_i_f - 1) downto 0);
    exp   : out std_logic_vector((width_e_f - 1) downto 0);
    subst : out std_logic_vector((width_o_f - 1) downto 0));

end FP4;

architecture behavioral of FP4 is

begin  -- behavioral

  process (v_mt)
  begin  -- process

    exp   <= (others => '0');
```

```
    subst <= (others => '0');

    for i in (width_i_f/2 - 1) downto 0 loop
      if v_mt(i*2+1 downto i*2) > "00" then
        exp <= std_logic_vector(to_unsigned(i, width_e_f));
        if i >= ((width_o_f/2)-1) then
          subst <= std_logic_vector(unsigned(v_mt(i*2+1 downto i*2)) -
     1) & std_logic_vector(unsigned(v_mt(i*2-1 downto (i-3)*2)));
        elsif i > 0 and i < ((width_o_f/2)-1) then
          subst <= std_logic_vector(unsigned(v_mt(i*2+1 downto i*2)) -
     1) & std_logic_vector(unsigned(v_mt(i*2-1 downto 0)) & (
     width_o_f-2*i-3 downto 0 => '0'));
        else
          subst <= std_logic_vector(unsigned(v_mt(1 downto 0)) - 1) &
     (width_o_f-3 downto 0 => '0');
        end if;
        exit;
      end if;
    end loop;  -- i

  end process;

end behavioral;


--------------------------------------------------------------------------------
--                                SQUARER                                     --
--------------------------------------------------------------------------------
-- Name: Squarer Unit
-- Description: Calculates the square of the input signal and the truncated
-- input signal, all in 1 clock cycle, using only shifting and additions
-- Inputs: x_mt --> 10 bits
-- Outputs: x_mt_2 --> 17 bits; Square of the input signal
--          x_p_2 --> 12 bits; Square of the trnucated input signal.
--
--------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants.all;

entity squarer is

  generic (
    width_i_s  : integer := WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT;  -- Width of the input signal
    width_o_s  : integer := WIDTH_OF_SQUARED_INPUT_SIGNAL;  -- Width of the square of the input signal
    bits_trunc : integer := BITS_OF_TRUNCATION;  -- Bits to be truncated from X to form Xp
    width_p_s  : integer := WIDTH_OF_SQUARED_TRUNC_SIGNAL);  -- Width of the square of the truncated input signal

  port (
    x_mt   : in  std_logic_vector(width_i_s - 1 downto 0);  -- input signal
    x_mt_2 : out std_logic_vector(width_o_s - 1 downto 0);  -- Square of the input signal
```

```
211     x_p_2  : out std_logic_vector(width_p_s - 1 downto 0));  -- Square
          of truncated input
212 -- signal width_p_s
213
214 end squarer;
215
216 architecture behavioral of squarer is
217
218   function andlookup (
219     x_mt_2_in : std_logic;
220     x_mt_in   : std_logic;
221     x_mt_in2  : std_logic)
222     return std_logic_vector is
223
224     variable tmp3 : std_logic_vector(1 downto 0) := "00";
225   begin
226     tmp3(1) := (((x_mt_in) and (x_mt_in2)) and x_mt_2_in);
227     if ((x_mt_in = x_mt_in2) and (x_mt_2_in = x_mt_in2)) then
228       tmp3(0) := '0';
229     elsif ((x_mt_in /= x_mt_in2) and (x_mt_2_in = '0')) then
230       tmp3(0) := '0';
231     else
232       tmp3(0) := '1';
233     end if;
234     return tmp3;
235   end andlookup;
236
237   function adding (
238     single_bit : std_logic;
239     two_bits   : std_logic_vector(1 downto 0))
240     return std_logic_vector is
241
242     variable tmp3 : std_logic_vector(1 downto 0) := "00";
243   begin
244     if two_bits = "00" then
245       tmp3(0) := single_bit;
246       tmp3(1) := '0';
247     elsif two_bits = "01" then
248       tmp3(0) := not single_bit;
249       tmp3(1) := single_bit;
250     elsif two_bits = "10" then
251       tmp3(0) := single_bit;
252       tmp3(1) := '1';
253     else
254       tmp3 := "00";
255     end if;
256     return tmp3;
257   end adding;
258
259   type   vector_array is array (width_i_s-1 downto 0) of
          std_logic_vector(1 downto 0);
260   type   vector_array1 is array (width_i_s - 3 downto 0) of
          std_logic_vector((width_i_s-1)*2 downto 0);
261   signal tmp2       : vector_array1;
262   signal tmpa       : vector_array;
263   signal tmp_final  : vector_array1;
264   signal x_mt_2_tmp : std_logic_vector((width_i_s*2)-1 downto 0);
265   signal x_p_2_tmp  : std_logic_vector(2*(width_i_s - bits_trunc) - 1
          downto width_i_s - bits_trunc);
266
267
268 begin  -- behavioral
269
270   process (x_mt, tmpa, tmp2, x_mt_2_tmp, tmp_final, x_p_2_tmp)
271   begin  -- process
272     for i in 0 to width_i_s - 1 loop
273       if i = 0 then
274         x_mt_2_tmp(0) <= x_mt(0);       -- Sets the LSB
```

```
 elsif i = 1 then
   x_mt_2_tmp(1) <= '0';            -- Sets the second bit from the right
   tmpa(0)       <= andlookup(x_mt(1), x_mt(1), x_mt(0));  -- Sets the first bit for further calculations
 elsif i = 2 then
   tmpa(1)                   <= andlookup(x_mt(2), x_mt(2), x_mt(1));
   tmp2(0)(1 downto 0)       <= andlookup(tmpa(0)(1), x_mt(2), x_mt(0));
   tmp_final(0)(1 downto 0) <= adding(tmp2(0)(1), tmpa(1));
   x_mt_2_tmp(2)             <= tmpa(0)(0);  -- Sets the third bit from the right
 else                       -- Sets the other bits up to half of wordlength
   tmpa(i-1) <= andlookup(x_mt(i), x_mt(i), x_mt(i-1));
   for j in 0 to i-2 loop
     if j < i-2 then
       tmp2(i-2)(2*j+1 downto 2*j) <= andlookup(tmp_final(i-3)(2*j), x_mt(i), x_mt(j));
     else
       tmp2(i-2)(2*j+1 downto 2*j) <= andlookup(tmp_final(i-3)(2*j-1), x_mt(i), x_mt(j));
     end if;

     if j = 0 then
       tmp_final(i-2)(1 downto 0) <= adding(tmp2(i-2)(1), tmp2(i-2)(3 downto 2));
     elsif j < i-2 then
       tmp_final(i-2)(2*j+1 downto 2*j) <= adding(tmp_final(i-2)(2*j-1), tmp2(i-2)(2*(j+1)+1 downto 2*(j+1)));
     elsif j = i - 2 then
       tmp_final(i-2)(2*j+1 downto 2*j) <= adding(tmp_final(i-2)(2*j-1), tmpa(i-1));
         end if;
       end loop;  -- j
       x_mt_2_tmp(i) <= tmp2(i-3)(0);
     end if;
   end loop;  -- i
   for k in width_i_s to width_i_s*2-1 loop  -- Sets the bits of the upper half(X_mt^2)
     if k = width_i_s*2-1 then
       x_mt_2_tmp(k) <= tmp_final(width_i_s - 3)(2*(width_i_s-3) + 1);
     elsif k = width_i_s then
       x_mt_2_tmp(k) <= tmp2(width_i_s - 3)(0);
     else
       x_mt_2_tmp(k) <= tmp_final(width_i_s - 3)((k-1 - width_i_s)*2);
     end if;
   end loop;  -- k
   for l in width_i_s - bits_trunc to 2*(width_i_s - bits_trunc) - 1 loop  --For(X_p^2)
     if l = 2*(width_i_s - bits_trunc) - 1 then
       x_p_2_tmp(l) <= tmp_final(width_i_s - bits_trunc - 3)(2*(width_i_s - bits_trunc -3) + 1);
     elsif l = width_i_s - bits_trunc then
       x_p_2_tmp(l) <= tmp2(width_i_s - bits_trunc - 3)(0);
     else
       x_p_2_tmp(l) <= tmp_final(width_i_s - bits_trunc - 3)((l-1 - (width_i_s - bits_trunc))*2);
     end if;
   end loop;  -- l
end process;

x_mt_2 <= x_mt_2_tmp(2*width_i_s - 1 downto 2*width_i_s - width_o_s);
```

```
  x_p_2  <= x_p_2_tmp & x_mt_2_tmp(width_i_s - bits_trunc - 1 downto
      2*(width_i_s-bits_trunc)-width_p_s);


end behavioral;

--
      ------------------------------------------------------------------

--                       POST-PROCESSING UNIT
            --
--
      ------------------------------------------------------------------

-- Name: Post_Processing
-- Description: Do the post-processing or the oposite action of the
-- normalization. The goals is to transform the input into the final
      result.
-- Inputs: y --> 16 bits; Output of the processing stage
--         exp --> 3 bits; Exponent
-- Outputs: z_mt --> 16 bits; Final result
--
      ------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants.all;

entity post_processing is

  generic (
    width_y    : integer := WIDTH_PROCESSING_OUTPUT;
    width_exp  : integer := WIDTH_OF_THE_EXPONENT;
    width_z_mt : integer := WIDTH_FINAL_RESULT);

  port (
    y_mt : in  std_logic_vector(width_y - 1 downto 0);
    exp  : in  std_logic_vector(width_exp - 1 downto 0);
    z_mt : out std_logic_vector(width_z_mt - 1 downto 0));

end post_processing;

architecture behavioral of post_processing is

begin  -- behavioral

  process (exp, y_mt)
    variable tmp       : integer                           := 0;
    variable z_mt_tmp1 : unsigned(width_z_mt - 1 downto 0) := (others
     => '0');
  begin  -- process
    tmp       := to_integer(unsigned(exp));
    z_mt_tmp1 := unsigned('1' & unsigned(y_mt( width_y - 1 downto 1)))
     ;
    z_mt      <= std_logic_vector(z_mt_tmp1 srl (INTERVALS - 1 - tmp))
     ;
  end process;

end behavioral;
```

LISTING A.6: Constants Package (VHDL)

```
--
      ------------------------------------------------------------------

-- Title      : Constants Declaration
```

```
-- Project    :
--
-- File       : constants.vhd
-- Author     :
-- Company    :
-- Created    : 2013-03-11
-- Last update: 2013-04-11
-- Platform   :
-- Standard   : VHDL'87
--
-- Description: All the constants used for the genetic values are declared
-- here. If changes to the width of any signal shall be done, it is to be here
-- where the change should occur.
--
-- Copyright (c) 2013
--
-- Revisions  :
-- Date        Version  Author  Description
-- 2013-03-11  1.0      2011    Created
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package constants is

  --
    ------------------------------------------------------------------------------
  --                              FP4 CONSTANTS
        --
  --
    ------------------------------------------------------------------------------

  constant WIDTH_OF_INPUT_SIGNAL  : integer := 16;  -- Width of V_mt
  constant WIDTH_OF_THE_EXPONENT  : integer := 3;  -- Width of exp
  constant WIDTH_OF_THE_OUPUT_FP4 : integer := 8;  -- Width of the
    output signal of the Floating Point 4 unit

  --
    ------------------------------------------------------------------------------
  --                          SQUARER UNIT CONSTANTS
        --
  --
    ------------------------------------------------------------------------------

  constant WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT : integer := 10;  --
    X_mt
  constant WIDTH_OF_SQUARED_INPUT_SIGNAL        : integer := 17;  --
    X_mt^2
  constant INTERVALS                            : integer := 8;  --
    Amount of Intervals for S2
```

```
constant BITS_OF_TRUNCATION                    : integer := 3;  -- log2(INTERVALS)
constant WIDTH_OF_SQUARED_TRUNC_SIGNAL         : integer := 12;

--------------------------------------------------------------------
--                        ROUNDING CONSTANTS                      --
--------------------------------------------------------------------

constant WIDTH_OF_INPUT_ROUNDING : integer := 21;  -- Width of the all the signals that have to be rounded
constant WIDTH_OF_ROUNDED_X_MT   : integer := 10;  -- Width of the X_mt after rounding
constant WIDTH_OF_ROUNDED_S1     : integer := 13;

--------------------------------------------------------------------
--                          TOP LEVEL CONSTANTS                   --
--------------------------------------------------------------------

constant WIDTH_GLOBAL_OUTPUT     : integer          := 16;  -- Width of V_mt
constant WIDTH_THIRD_COEF        : integer          := 13;
constant THIRD_COEF              : std_logic_vector := "0101010101011";
constant WIDTH_ADDER_S1_1        : integer          := 12;
constant WIDTH_ADDER_S1_2        : integer          := 19;
constant WIDTH_ADDER_S2_1        : integer          := 15;
constant WIDTH_ADDER_S2_2        : integer          := 13;
constant WIDTH_OF_ROUNDED_S2     : integer          := 12;
constant WIDTH_PROCESSING_OUTPUT : integer          := 16;
constant WIDTH_FINAL_RESULT      : integer          := 16;


--------------------------------------------------------------------
--                        CONSTANTS COEFFICIENTS                  --
--------------------------------------------------------------------

constant WIDTH_C2I_COEFF  : integer := 7;   -- with 6 ceros in front
constant WIDTH_J_I_COEFF  : integer := 8;   -- with 4 ceros in front
constant WIDTH_L_I_COEFF  : integer := 16;  -- Could not be optimized
constant WIDTH_OPTIM_C2_I : integer := 6;
constant WIDTH_OPTIM_J_I  : integer := 4;

type C2I_COEFF_TYPE is array (0 to INTERVALS - 1) of std_logic_vector(WIDTH_C2I_COEFF - 1 downto 0);
constant C2I_COEFF : C2I_COEFF_TYPE := ("1011100",
                                        "0111010",
                                        "0101010",
                                        "0100001",
                                        "0011101",
                                        "0011011",
                                        "0011011",
                                        "0011101");
```

```
91  type J_I_COEFF_TYPE is array (0 to INTERVALS - 1) of
        std_logic_vector(WIDTH_J_I_COEFF - 1 downto 0);
92  constant J_I_COEFF : J_I_COEFF_TYPE := ("11001111",
93                                          "01110100",
94                                          "00111011",
95                                          "00010001",
96                                          "00010000",
97                                          "00101101",
98                                          "01001000",
99                                          "01100011");
100 type L_I_COEFF_TYPE is array (0 to INTERVALS - 1) of
        std_logic_vector(WIDTH_L_I_COEFF - 1 downto 0);
101 constant L_I_COEFF : L_I_COEFF_TYPE := ("1000000000000000",
102                                         "0111101011110100",
103                                         "0111100000111010",
104                                         "0111011100001010",
105                                         "0111011100000100",
106                                         "0111011111110110",
107                                         "0111100111001001",
108                                         "0111110001110101");
109
110 --
        ----------------------------------------------------------------------
111 --                          TOP LEVEL CONSTANTS
            --
112 --
        ----------------------------------------------------------------------
113
114 constant WIDTH_TEMP_ADDER_S2_1 : integer := WIDTH_J_I_COEFF + (
        WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - BITS_OF_TRUNCATION);
115 constant WIDTH_TEMP_ADDER_S2_2 : integer := WIDTH_C2I_COEFF +
        WIDTH_OF_SQUARED_TRUNC_SIGNAL;
116
117
118 --
        ----------------------------------------------------------------------
119 --                      ADDER/SUBTRACTOR CONSTANTS
                --
120 --
        ----------------------------------------------------------------------
121
122 constant WIDTH_MULT_S2 : integer := WIDTH_J_I_COEFF +
        WIDHT_OF_INPUT_SIGNAL_OF_SQUARE_UNIT - BITS_OF_TRUNCATION;
123 constant WIDTH_TO_T_ADDER2 : integer := WIDTH_MULT_S2 -
        WIDTH_ADDER_S2_2;
124
125
126
127 end constants;
```

Listing A.7: Test Bench (VHDL)

```
1   ----------------------------------------------------------------------
2   -- Title      : TESTBENCH
3   -- Project    :
4   ----------------------------------------------------------------------
5   -- File       : testbench.vhd
6   -- Author     :
7   -- Company    :
8   -- Created    : 2013-03-14
```

```
-- Last update: 2013-04-18
-- Platform   :
-- Standard   : VHDL'87
--
-------------------------------------------------------------------------------
-- Description: Testing the whole design for proper functionallity
--
-------------------------------------------------------------------------------
-- Copyright (c) 2013
--
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2013-03-14  1.0      2011    Created
--
-------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity testbench is

end testbench;

architecture structural of testbench is

  component TOP_LEVEL
    --generic (
      --width_i : integer := 16;
      --width_o : integer := 16);
    port (
      v_mt : in  std_logic_vector(15 downto 0);
      z_mt : out std_logic_vector(15 downto 0));
  end component;

  component Pads
    generic (
      width_i_P : integer;
      width_o_P : integer);
    port (
      v_mt : inout std_logic_vector(width_i_P - 1 downto 0);
      z_mt : inout std_logic_vector(width_o_P - 1 downto 0));
  end component;

  signal v_mt : std_logic_vector(15 downto 0) := (others => '0');
  signal z_mt : std_logic_vector(15 downto 0) := (others => '0');

begin  -- structural

      top_level_1 : TOP_LEVEL
        --generic map (
          --width_i => 16,
          --width_o => 16)
        port map (
          v_mt => v_mt,
          z_mt => z_mt);

      --Pads_1 : Pads
      --  generic map (
      --    width_i_P => 16,
      --    width_o_P => 16)
      --  port map (
```

```
69        --     v_mt => v_mt,
70        --     z_mt => z_mt);
71
72
73        process
74          file file_pointer : text;
75          variable line_num : line;
76        begin  -- process
77          file_open(file_pointer, "results.txt", write_mode);
78          for i in 0 to 65536 loop
79            v_mt <= std_logic_vector(to_unsigned(i, 16));
80            wait for 10 ns;
81            write(line_num, to_integer(unsigned(z_mt)));
82            writeline (file_pointer, line_num);
83          end loop;  -- i
84        end process;
85
86    end structural;
```

LISTING A.8: Simulation Tcl Script (Modelsim)

```
1  vlib work
2  #vcom ./vhdl/constants.vhd
3  #vcom ./vhdl/comp_pack.vhd
4  #vcom ./vhdl/top.vhd
5  #vcom ./vhdl/Pads.vhd
6  vcom ./netlists/netlist.v
7  vcom ./vhdl/testbench.vhd
8  vsim -c -do 'run 655360ns;quit' testbench
```

LISTING A.9: Library Compilation Tcl Script (Modelsim)

```
1  ln -s /usr/local-eit/cad2/cmpstm/stm065v536/CORE65LPHVT_5.1/behaviour/
       verilog/setup.modelsim.verilog
2  source /usr/local-eit/cad2/cmpstm/stm065v536/newdig
3  ./setup.modelsim.verilog
```

LISTING A.10: Synthesis Tcl Script (Design Vision)

```
1  remove_design -designs
2  # set CORE "CORE65";
3  # set VOLTAGE(1) "LPHVT";
4  # set VOLTAGE(2) "LPLVT";
5  # set VOLTAGE(3) "GPSVT"
6  # set LIBTRAIL "_5.1";
7  # set TRAIL2(1) "_nom_1.20V_25C.db";
8  # set TRAIL2(2) "_nom_1.00V_25C.db";
9  # set TRAIL2(3) "_nom_0.80V_25C.db";
10 # set TRAIL2(4) "_nom_0.82V_25C.db";
11
12 # eval file delete [glob -nocomplain results/*.dat]
13
14 # set i 1;
15 # set j 1;
16
17 # while {$i < 4} {
18
19 #     while {$j < 5} {
20
21 #   set path_value $CORE$VOLTAGE($i)$LIBTRAIL;
22
23 #   set lib_value $CORE$VOLTAGE($i)$TRAIL2($j);
24
25 #   set search_path "$env(STM065_DIR)/
       IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_7.0/libs \
```

```
#        $env(STM065_DIR)/$path_value/libs \
#        $search_path"

#   set link_library "
     IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db \
#                  $lib_value \
#         standard.sldb dw_foundation.sldb"

#   set target_library "
     IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db \
#                    $lib_value"

# read_lib /export/space/soc11ava/rechar/libs/core_lib/
    CORE65LPHVT_RECHAR_0v8.lib
# write_lib CORE65LPHVT_RECHAR_0v8 -out /export/space/soc11ava/rechar/
    libs/CORE65LPHVT_RECHAR_0v8.db


# set search_path "$env(STM065_DIR)/IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_7.0/
    libs \
#        /export/space/soc11ava/rechar/libs \
#        $search_path"

# set link_library "
    IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db  \
#                 CORE65LPHVT_RECHAR_0v8.db \
#                standard.sldb dw_foundation.sldb"


# #"IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db \
# #                 CORE65LPHVT_RECHAR_0v8.db\
# #           standard.sldb dw_foundation.sldb"

# set target_library "
     IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db \
#                    CORE65LPHVT_RECHAR_0v8.db"

# #"IO65LPHVT_SF_1V8_50A_7M4X0Y2Z_nom_1.00V_1.80V_25C.db \
#
#CORE65LPHVT_RECHAR_0v8.db"

        analyze -library WORK -format vhdl {/home/piraten/soc11ava/
    Desktop/test/vhdl/top.vhd \
                        /home/piraten/soc11ava/Desktop/test/vhdl/
    constants.vhd \
                        /home/piraten/soc11ava/Desktop/test/vhdl/
    comp_pack.vhd}

    elaborate TOP_LEVEL -architecture structural -library WORK -update

    set_max_delay 10 -to {z_mt}

    check_design

    compile -map_effort medium -area_effort medium

    remove_unconnected_ports -blast_buses [get_cells "*" -hier]
    remove_unconnected_ports [get_cells "*" -hier]

    change_names -rules verilog -hierarchy
    write -format verilog -hierarchy -output netlists/netlist.v
    write_sdf netlists/timing_info.sdf
    write_sdc netlists/PlaceRoute_info.sdc

    report_constraint -all_violators
    report_design
    report_area -hierarchy
```

```
# >> ./results/area$lib_value.dat
    report_timing
# >>  ./results/time$lib_value.dat
    report_power
#>>  ./results/power$lib_value.dat


#   set j [expr {$j + 1}];
#     }

#     set i [expr {$i + 1}];
#     set j 1;
# }
```

LISTING A.11: Place-and-Route Tcl Script (SOC Encounter)

```
freeDesign

#- Setup some basic parameters

setPreference ConstraintUserXGrid 0.2
setPreference ConstraintUserXOffset 0.0
setPreference ConstraintUserYGrid 0.2
setPreference ConstraintUserYOffset 0.0


#- Import the design.
#-
#- ( Menu: Design > Design Import )

loadConfig sqrt_PR.conf
commitConfig


#- Define the global power nets.
#-
#- ( Menu: Floorplan > Global Net Connections )

clearGlobalNets
globalNetConnect vdd -type pgpin -pin VDD -inst PVCCc
globalNetConnect vdd -type pgpin -pin vdd -inst *
globalNetConnect vdd -type tiehi
globalNetConnect gnd -type tielo
globalNetConnect gnd -type pgpin -pin GND -inst PGNDc
globalNetConnect gnd -type pgpin -pin gnd -inst *

#- Floorplanning
#-
#- ( Menu: Floorplan > Specify Floorplan )
#-
#- (x-width y-width core-left core-down core-right core-up) keep width
      on .28 grid.
#-
#- For some reason the snapping does not work. Use 0.28 grid for width
      & height.
#-

floorPlan -site CORE -r 0.95 0.7 9.0 9.0 9.0 9.0
# floorPlan -site CORE -s 130 110 9.0 9.0 9.0 9.0


#- Power rings and stripes.
#-
#- ( Menu: Power > Power Planning > Add Rings / Add Stripes )

```

```
addRing -spacing_bottom 1 -width_left 3 -width_bottom 3 -width_top 3 \
           -spacing_top 1 -layer_bottom M3 -stacked_via_top_layer AP \
           -width_right 3 -around core -jog_distance 2.5 -layer_top M3 \
           -threshold 2.5 -spacing_right 1 -spacing_left 1 \
     -layer_right M4 \
           -nets {gnd vdd } -stacked_via_bottom_layer M1 -layer_left M4

addStripe -block_ring_top_layer_limit M5 -max_same_layer_jog_length 6 \
          -padcore_ring_bottom_layer_limit M3 -number_of_sets 1 \
          -stacked_via_top_layer AP -padcore_ring_top_layer_limit M5 \
          -spacing 1 -xleft_offset 60 -merge_stripes_value 2.5 -layer M4 \
          -block_ring_bottom_layer_limit M3 -width 3 -nets {gnd vdd } \
          -stacked_via_bottom_layer M1

addStripe -block_ring_top_layer_limit M4 -max_same_layer_jog_length 6 \
          -padcore_ring_bottom_layer_limit M2 -number_of_sets 1 \
          -ybottom_offset {0} -stacked_via_top_layer AP \
          -padcore_ring_top_layer_limit M4 -start_from top -spacing 1 \
          -merge_stripes_value 2.5 -direction horizontal -layer M3 \
          -block_ring_bottom_layer_limit M2 -ytop_offset 50 -width 3 \
          -nets {gnd vdd } -stacked_via_bottom_layer M1

#- Start placing cells. No cell under existing metal.
#-
#- ( Menu: Place > Specify > Placement Blockage )
#- ( Menu: Place > Place )

setPrerouteAsObs {1 2 3 4 5 6 7 8}

placeDesign -prePlaceOpt

#- Connect some power.
#-
#- ( Menu: Route > SRoute )

# sroute -noBlockPins -noPadRings -jogControl { preferWithChanges
     differentLayer }

#- Route what is left.
#-
#- ( Menu: Route > NanoRoute )

 # globalDetailRoute

 setNanoRouteMode -quiet -drouteStartIteration default
 setNanoRouteMode -quiet -routeTopRoutingLayer default
 setNanoRouteMode -quiet -routeBottomRoutingLayer default
 setNanoRouteMode -quiet -drouteEndIteration default
 setNanoRouteMode -quiet -routeWithTimingDriven false
 setNanoRouteMode -quiet -routeWithSiDriven false
 routeDesign -globalDetail

 optDesign -postRoute


#- Fill empty space between standard cells.

 addFiller -minHoleCheck -cell HS65_LH_FILLERCELL4 HS65_LH_FILLERCELL3 \
       HS65_LH_FILLERCELL2 HS65_LH_FILLERCELL1 -prefix fico \
```

```
            -fitGap -fixDRC

# Save Design

saveDesign sqrt.enc

rcOut -spef top_level.spef

saveNetlist top_level.v

delayCal -sdf top_level.sdf

fit
```

LISTING A.12: Power Analysis Tcl Script (Prime Time)

```
##-------------------------------------------------------
## Design       : medianfilter_pt_power
## File Name    : medianfilter_pt_power.tcl
## Purpose      : PrimeTime PX power template for sqrt
## Limitation   : none
## Errors       : none known
## Include Files: none
## Author       : Dalia Iurascu/Alejandro Vazquez Bofill
## Software     : Synopsys PrimeTime
##-------------------------------------------------------
## Revision List:
##+-----------+--------------------+-----------------+
    ----------------------------------+
##| Version   |Author              | Date            | Changes
                        |
##+-----------+--------------------+-----------------+
    ----------------------------------+
##|1.0        |   Dalia/Alejandro  | 2013/02/15      | original
    created                  |
##+-----------+--------------------+-----------------+
    ----------------------------------+



#################### set up power analysis mode ######################
# step 1: enalbe power analysis and set analysis mode

remove_design -all
set power_enable_analysis TRUE
set power_analysis_mode time_based

######################## set up libaries ############################
# step 2: link to your design libary

set search_path "/usr/local-eit/cad2/cmpstm/stm065v536/CORE65LPHVT_5.1
    /libs"
set link_library "* CORE65LPHVT_nom_1.20V_25C.db"
set target_library "CORE65LPHVT_nom_1.20V_25C.db"

######################## design input    ############################
# step 3: read your design (netlist) & link design

read_verilog ./soc/top_level.v
current_design top_level
link_design -keep_sub_designs

########################  timing information  ######################
# step 4: read SDC and set transition time (post-syn) or annotate
    parasitics (post-layout)

```

```
read_parasitics ./soc/top_level.spef

################## read switching activity file ######################
# step 5: read vcd file obtained from post-layout (syn) simulation

read_vcd -strip_path testbench/top_level_1 ./results_vcd.vcd

################## check or update or report power #################
# step 6: analysis and output analysis report

check_power
update_power
report_power -verbose -hierarchy > ./results/power.rpt
```

# Bibliography

[1] R.Andraka, *A survey of CORDIC algorithms for FPGA based computers*. ACM, 1998, vol. FPGA '98 Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, no. 0-89791-978-5.

[2] M. O.Esteban, R.Cumplido, "Pipelined cordic design on fpga for a digital sine and cosine waves generator," *Electrical and Electronics Engineering, 2006 3rd International Conference on*, no. 2006.251917, pp. 1,4, September 2006.

[3] Y. Hu, "Cordic-based vlsi architectures for digital signal processing," *Signal Processing Magazine, IEEE*, vol. 9, no. 79.143467, pp. 16–35, July 1992.

[4] A.Boudabous, "Implementation of hyperbolic functions using cordic algorithm," *Microelectronics, 2004. ICM 2004 Proceedings. The 16th International Conference on*, no. 2004.1434772, pp. 738–741, 2004.

[5] P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, no. 1991.145565, pp. 232–236, June 1991.

[6] J. Muller, *Elementary Functions: Algorithms and Implementation*, 2nd ed. ACM, 2005, no. 0817643729.

[7] B. E.Hertz, P.Nilsson, "Parabolic synthesis combined with second-degree interpolation."

[8] E.Hertz, "Parabolic synthesis," Master's thesis, Lund Universitet, 2011.

[9] P. E.Hertz., "Parabolic synthesis methodology implemented on the sine function," *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, no. 2009.5117733, pp. 253 –256, May 2009.

[10] ——, "A methodology for parabolic synthesis of unary functions for hardware implementation," *Signals Circuits and Systems, 2008. SCS 2008. 2nd International Conference*, no. 2008.4746866, pp. 1–6, November 2008.

[11] P. Pouyan, "A vlsi implementation of logarithmic and exponential functions using a parabolic synthesis methodology, which is compared to the cordic algorithm," Master's thesis, Lund University, 2010.

[12] L. A.A.Giunta, "A comparison of aproximation modeling techniques," *American Institute of Aeronautics and Astronautics*, 1998.

[13] B.Parhami, *Computer Arithmetic-Algorithms and Hardware Designs*. Oxford University, 2000, vol. 2nd Edition, no. 0-19-512583-5.

[14] Synopsys, "Design compiler using design vision," Tech. Rep.

[15] STMicroelectronics, "Core65lphvt for 1.00v standard cell library user manual," Tech. Rep.

[16] ——, "Core65lplvt standard cell library user manual," Tech. Rep.

[17] ——, "Core65gpsvt for 1.00v standard cell library user manual," Tech. Rep.

LUND
UNIVERSITY

http://www.eit.lth.se