Master's Thesis

Hardware Architectures for the Inverse Square Root and the Inverse Functions

Niclas Thuning
Leo Bärring

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

LUND
UNIVERSITY

Department of Electrical and Information Technology,
Faculty of Engineering, LTH

Master of Science Thesis

Hardware Architectures for the Inverse Square Root and the Inverse Functions

using Harmonized Parabolic Synthesis

Authors:
Niclas Thuning
Leo Bärring

Supervisors:
Peter Nilsson
Erik Hertz
Rakesh Gangarajaiah

Examiner:
Fredrik Rusek

Lund, 17 Mar, 2016

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN
This thesis is set in Computer Modern 11pt, with the LaTeX Documentation System

Abstract

This thesis presents a comparison between implementations of the inverse square root function, using two approximation algorithms; Harmonized Parabolic Synthesis (HPS) and the Newton-Raphson Method (NR). The input is a 15 bit fixed-point number, the range of which is selected so that the implementation is suitable for use as a block implementing the inverse square root for floating-point numbers, and the designs are constrained by the error, which must be $< 2^{-15}$. Multiple implementations of both algorithms have been investigated and simulated as Application-Specific Integrated Circuits (ASIC) using ST Microelectronics 65.0nm Complementary Metal-Oxide Seminconductor (CMOS) technology libraries for Low Power (LP) and General Purpose (GP), V_{DD} levels of 1.00V and 1.10V, and for various clock speeds. Error distribution, area, speed, power, and energy consumption are analyzed for variants of the implementations of the two algorithms. Depending on how the properties rank in desirability, when choosing an implementation, the recommended choice will vary. The thesis finds that if mean error, and error distribution are important, the implementations of Harmonized Parabolic Synthesis show superiority regarding implementable clock speed, area requirements, power and energy consumption. If power and energy consumption is the most prioritised property, an implementation of the Newton-Raphson algorithm is promising, although at the cost of a worse error distribution.

Acknowledgements

This thesis was possible thanks to our supervisors, Professor Peter Nilsson and Lic. Eng. Erik Hertz. We would like to express our deepest gratitude and respect to Peter and Erik for the guidance and help they have given us throughout the completion of this master thesis. We are glad to have had them as our supervisors and could not have asked for any better. Furthermore, we like to thank Rakesh Gangarajaiah for his help in getting us started with the synthesis tools, whose interfaces are anything but intuitive. The authors would also like to thank the Departement of Electrical and Information Technology, Lund University, for providing access to the software tools necessary for conducting the research presented in this thesis.

We like to thank Lars Bärring and Alana York for their contribution of proof reading the thesis and giving valuable inputs and corrections.

Niclas Thuning wants as well to thank his family and friends for their love and support while working on this thesis. He wants to express a very special appreciation to his beloved girlfriend Alana York, who loved and supported him throughout this work.

Leo Bärring thanks his family for all the love and support, both during times of doubt, as well as when everything mysteriously worked. C. Arabica deserves mention for being a good friend and a constant source of much needed energy.

Peter Nilsson
in memoriam
For being our supervisor through the ups and downs during this project, and teaching in courses we attended before that, we feel deep sorrow that he is not here to share our joy and pride of seeing this thesis finally being brought to completion.

Contents

Preface

This thesis is written by Niclas Thuning and Leo Bärring for the Department of Electrical and Information Technology, Faculty of Engineering, Lund University. Niclas and Leo are both students in the Master of Science in Electrical Engineering Program with the specialization Design of Processors and Embedded Systems. This thesis is an admission for the degree of Master of Science.

For this thesis paper, most of the work has been divided between the two authors. The individual contributions and collaborations between the authors are as follows.

Niclas Thuning has written the chapter on the Newton-Raphson Method and all sections concerning the Newton-Raphson Method. Furthermore, Niclas Thuning has written the Introduction, Error Analysis Theory (apart from Skewness and Bit Precision) and Implementation: Hardware Design and Simulation (apart from Semi-Generic Multiplier and Main Process Control Script).

Leo Bärring has written the chapters on Harmonized Parabolic Synthesis and all sections concerning Harmonized Parabolic Synthesis. Leo has also written the Skewness part of Error Analysis Theory, Number Representation, Calculating the Inverse Square Root, and Main Process Control Script. Furthermore, Leo has done the final versions of the graphics and the generated plots for this thesis.

The two authors have through collaboration written the chapters/sections, Abstract, Preface, Acknowledgments, Scope, Bit Precision, Results, Semi-Generic Multiplier, Conclusions and Appendix, as well as discussed and collaborated on improving sentence structure, phrasing, and word choice, as needed throughout the thesis, regardless of the original author of the part.

List of Figures

List of Tables

List of Acronyms

ASIC Application-Specific Integrated Circuit

CMOS Complementary Metal-Oxide Seminconductor

dB Decibel

DRC Design Rule Check

DSP Digital Signal Processing

FA Full Adder

GPHVT General Purpose High Threshold Voltage

GPLVT General Purpose Low Threshold Voltage

GPSVT General Purpose Standard Threshold Voltage

GPU Graphic Processing Unit

HA Half Adder

HPS Harmonized Parabolic Synthesis

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

LPHVT Low Power High Threshold Voltage

LPLVT Low Power Low Threshold Voltage

LPSVT Low Power Standard Threshold Voltage

LSB Least Significant Bit

LUT Look-up Table

MSB Most Significant Bit

NR the Newton-Raphson Method

PNR Placement and Routing

RMS Root Mean Square

RTL Resistor Transistor Logic

SD Standard Deviation

SDC Synopsys Design Constraints

SDF Standard Delay Format

SNR Signal-to-Noise Ratio

SPEF Standard Parasitic Exchange Format

STA Static Timing Analysis

VCD Value Change Dump

VHDL Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

Unary functions, like logarithm, exponential, trigonometric functions, and the square root, are examples of widely used building blocks in more complex algorithms. These unary functions can, for example, be found in applications in the fields of wireless communication, Digital Signal Processing (DSP), Graphic Processing Units (GPU), etc [1].

Performance improvement for algorithms using software solutions is not always sufficient. Although software can compute with extreme accuracy, implementing algorithms in terms of sequences of simpler processor instructions is sometimes not fast enough for high speed and/or numerically intensive applications. In order to improve the performance, the algorithms can be implemented directly into hardware [1].

This thesis will investigate an algorithm called Harmonized Parabolic Synthesis to compute the inverse square root, proposed by Erik Hertz and Peter Nilsson [2]. As comparison, the Newton-Raphson Method algorithm is used.

Inverse square root is essential for normalizing a vector in linear algebra. A normalized vector, or unit vector $\hat{\boldsymbol{v}}$, is calculated as shown in (1.1) [3, pp. 131–192].

$$\hat{\boldsymbol{v}} = \frac{1}{|\boldsymbol{v}|}\boldsymbol{v} \tag{1.1}$$

where $|\boldsymbol{v}|$ is the norm calculated as $\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$ (with v_i being vector component in each dimension).

Unit vectors are crucial e.g. when calculating reflections, used e.g. in 3D graphics rendering code used by the gaming industry and when constructing graphic design tools. An increased speed of computation can mean either faster rendering, or rendering with higher precision [4].

Chapter 2

Scope

The objective of this project is to design and implement an inverse square root function and an inverse function using Harmonized Parabolic Synthesis and compare it with the Newton-Raphson Method with iterations unrolled in hardware. The implementations are required to be able to process input inputs $v \in [1, 4)$ represented with 15 bits precision in fixed-point format. The output $z = 1/\sqrt{v}$ is required to have 15 bits precision or better, as well as the output being *exactly* 1 when the input equals 1. The second output, the inverse function $z^2 = 1/v$ is not subject to any precision constraints. There is also a goal to have a balanced error probability distribution. Figure 2.1 shows the basic implementation architecture for the two designs.

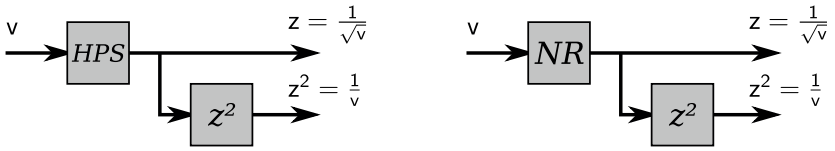
The designs are simulated and compared for accuracy, error behavior, power consumption, and performance. For power consumption, both static and dynamic power consumption are estimated. The core area of the implementations are estimated as well. Physical layouts of the designs are synthesized.

The hardware implementation is written in Very High Speed Integrated Circuit Hardware Description Language (VHDL) and synthesized with different clock frequencies, library technologies, supply voltage and threshold voltage. The library technologies used are ST Microelectronics 65.0nm CMOS and the following variants are examined.

- Low Power Low V_t (LPLVT)
- Low Power Standard V_t (LPSVT)
- Low Power High V_t (LPHVT)
- General Purpose Low V_t (GPLVT)

- General Purpose Standard V_t (GPSVT)
- General Purpose High V_t (GPHVT)

The results of this report will focus on General Purpose Standard V_t and General Purpose High V_t, with additional results in the appendix section. For each library, two different supply voltages are used: 1.00V and 1.10V.



(a) *Harmonized Parabolic Synthesis* (b) *The Newton-Raphson Method*

Figure 2.1: *Overview of the architecture implementing the inverse square root and inverse function with Harmonized Parabolic Synthesis* (a) *and the Newton-Raphson Method* (b).

Chapter 3

Harmonized Parabolic Synthesis

3.1 General Description

The Harmonized Parabolic Synthesis (HPS)[5] algorithm approximates a function $y = f_{org}(x)$. To facilitate development, the range of x, as well as the allowed range of y is constrained to $[0, 1]$. If there is a need to approximate a different range, the input and/or output will need conditioning, with corresponding changes to the $f_{org}(x)$ function.

The approximation of $f_{org}(x)$ is calculated as the product of two sub-functions $s_1(x)$ and $s_2(x)$, (see (3.1)) where the first sub-function, $s_1(x)$, is a second order polynomial and the second sub-function, $s_2(x)$, is a second order interpolation.

$$f_{org}(x) \approx s_1(x)s_2(x) \tag{3.1}$$

3.1.1 The First Sub-Function

The first sub-function $s_1(x)$, shown in (3.2) and coefficients are chosen to roughly approximate $f_{org}(x)$.

$$s_1(x) = l_1 + k_1x + c_1(x - x^2) \tag{3.2}$$

If possible, $s_1(x)$ is constructed so that it intersects $(0; 0)$ and $(1; 1)$, or $(0; 1)$ and $(1; 0)$, which allows the expression to be simplified to one out of the ones shown in (3.3), by way of setting $l_1 = 0$ and $k_1 = 1$, or $l_1 = 1$ and $k_1 = -1$.

$$s_1(x) = x + c_1(x - x^2) \tag{3.3a}$$
$$s_1(x) = 1 - x + c_1(x - x^2) \tag{3.3b}$$

Which particular variant of s_1 to use depends on the shape of the function to be approximated. The constant c_1 remains unassigned so far and a suitable value has to be determined experimentally.

3.1.2 The Second Sub-Function

The second sub-function, $s_2(x)$ which improves on the approximation in $s_1(x)$, is a second order interpolation of $f_{help}(x)$, which is defined in (3.4).

$$f_{help}(x) = \frac{f_{org}(x)}{s_1(x)} \tag{3.4}$$

Splitting $f_{help}(x)$ into I intervals yields a set of equations shown in (3.5), where i is the interval index $0 \leq i < I$, and x_w linearly increases from 0 to 1 over each interval.

$$s_{2,i}(x) = l_{2,i} + k_{2,i}x_w + c_{2,i}(x_w - x_w^2) \tag{3.5}$$

Assuming that the intervals are chosen to be of equal size, the interval index i can be calculated by extracting the integral part of Ix, i.e. by truncating Ix towards zero. Conversely x_w can be calculated by taking the fractional part of Ix.

Calculation of the coefficients of the $s_2(x)$ polynomials in each interval is shown in (3.6), and is based on the fact that the polynomial is chosen to intersect $f_{help}(x)$ in the beginning, middle, and end of each interval, where x_w takes on the values 0, 0.5, and 1 respectively.

$$\begin{aligned}
x_{start,i} &= i/I \\
x_{mid,i} &= (i + 0.5)/I \\
x_{end,i} &= (i + 1)/I \\
l_{2,i} &= f_{help}(x_{start,i}) \\
k_{2,i} &= f_{help}(x_{end,i}) - l_{2,i} \\
c_{2,i} &= -4f_{help}(x_{mid,i}) - 4l_{2,i} - 2k_{2,i}
\end{aligned} \tag{3.6}$$

Due to its role as a divisor in $f_{help}(x)$, it is necessary to make sure that for any x in the interval where $s_1(x) = 0$ there exists a limit value in $f_{help}(x)$. If choosing (3.3a), the zero occurrs at $x = 0$, and directly affects how the constant $l_{2,0}$ is calculated. In case of (3.3b) being the chosen form of $s_1(x)$, the zero occurrs at $x = 1$, and special care has

to be taken when calculating the value of the constant $k_{2,I-1}$. The limits that have to be calculated in either case, and their associated constants, are shown in (3.7). It is imperative that the limit exists for the implementation to be feasible.

$$l_{2,0} = \lim_{x \to 0} f_{help}(x) \tag{3.7a}$$

$$k_{2,I-1} = \lim_{x \to 1} f_{help}(x) - l_{2,I-1} \tag{3.7b}$$

Furthermore, there will be a second zero in either $x = 1 + 1/c_1$ (using s_1 from 3.3a), or $x = -1/c_1$ (using 3.3b), for which $f_{help}(x)$ may have no limit, in which case care has to be taken when choosing a value for c_1, so that the zero does not appear in the range $x \in [0, 1]$.

At this point, the method to calculate all constants in the second sub-function has been shown. The number of intervals I remains as an implementation detail.

3.1.3 Pre-Processing and Post-Processing Functions

As mentioned in the introduction to this chapter, when implementing an approxmation of a function with an arbitrary input and output range, such as the one shown in (3.8), additional functions may be needed in order to bring input to the working domain of the algorithm, as well as bringing the algorithm output from the algorithm domain, to the output range of the approximated function [6].

$$z = f(v) \tag{3.8}$$

The pre-processing and post-processing, as declared in (3.9), are used to translate between whatever ranges are desired, and the $[0, 1]$ range required by the implementation of $f_{org}(x)$ that is to be approximated.

$$\begin{aligned} x &= f_{pre}(v) \\ z &= f_{post}(y) \end{aligned} \tag{3.9}$$

The overall expression then becomes as in (3.10), with the second variant written with $\circ$ denoting function composition (meaning that $(g \circ f)(x) = g(f(x))$).

$$\begin{aligned} z &= f_{post}(\, f_{org}(\, f_{pre}(v)\,)\,) \\ z &= (f_{post} \circ f_{org} \circ f_{pre})(x) \end{aligned} \tag{3.10}$$

This arrangement makes it necessary to choose $f_{org}(x)$ so that it compensates the effect that the pre- and post-processing functions have on the overall expression, i.e. so that the (ideal) approximation matches the function in (3.8). This can be done by including the inverses of the pre- and post-processing functions in the definition of $f_{org}(x)$, as shown in (3.11), where $z = f(v)$ is the unconstrained function in (3.8).

$$f_{org}(x) = \quad (f_{post}^{-1} \circ f \circ f_{pre}^{-1})(x) \tag{3.11}$$

Doing this, the overall expression will compose the pre- and post-processing functions with their corresponding inverses, which makes them equal to the identity function ($f_{id}(x) = x$), and makes it possible to eliminate them from the expression. This is shown in (3.12).

$$\begin{aligned} f(v) &= (f_{post} \circ f_{org} \circ f_{pre})(v) \\ &= (f_{post} \circ f_{post}^{-1} \circ f \circ f_{pre}^{-1} \circ f_{pre})(v) \\ &= (f_{id} \circ f \circ f_{id})(v) \\ &= f(v) \end{aligned} \tag{3.12}$$

At this stage, the definition of $f_{org}(x)$ in terms of the function to be approximated ($f(v)$), and the pre- and post-processing functions, is clear and the search for suitable sub-functions can begin.

Two unknowns remain: the constant c_1 from the first sub-function, and the number of intervals, I, to use in the second sub-function. Both affect the complexity of a hardware implementation directly, as well as indirectly, by influencing error behaviour, and thus influencing what further constraints can be placed on the data paths. Suitable values have to be determined by e.g. simulation on a case-by-case basis.

Chapter 4

The Newton-Raphson Method

4.1 General Description

One method to find a root of an arbitrary function $f(x) = 0$ numerically is the Newton-Raphson method (NR) [7], named after Isaac Newton and Joseph Raphson. The objective of this method is to successively find a more accurate approximation by iteration. It starts from a presupplied approximate answer (initial guess) and then iterates until a sufficiently close approximation is reached. The general form of an iteration of NR is given in the formula shown in (4.1) [8, pp. 393].

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})} \textbf{ for } i = 1, 2 \ldots \quad \text{where } x_0 \text{ is the initial guess.} \tag{4.1}$$

For each iteration, x_i closes in on the sought value x. For the derivative in (4.1) the condition of $f'(x) \neq 0$ must be met.

In order to use NR to calculate an arbitrary function $y = g(v)$, the expression for which a root has to be calculated can be written as $f(y) = g^{-1}(y) - v$. A concrete example for the function $g(v) = \frac{1}{v}$ is given in (4.2) and (4.3).

$$\begin{aligned} y = g(v) = \frac{1}{v} \quad \Rightarrow \quad v = g^{-1}(y) = \frac{1}{y} \\ f(y) = \frac{1}{y} - v \\ f'(y) = -\frac{1}{y^2} \end{aligned} \tag{4.2}$$

$$y_i = y_{i-1} - \frac{\frac{1}{y_{i-1}} - v}{-\frac{1}{y_{i-1}^2}}$$
$$y_i = 2y_{i-1} - vz_{i-1}^2 \tag{4.3}$$

The final expression in (4.3) uses no division and can be implemented effectively in hardware to use as an approximation for the division function together with a suitable initial guess z_0. The number of iterations required would depend on the error of the initial guess and the required precision of the answer. There are functions and cases where NR may not converge as well, although they are outside the scope of this report.

4.2 Calculation of Look-Up Table

In order to increase the speed of the NR convergence, the initial guess should be as close as possible to the actual result. If the guess is too far away from the result, either more iterations or more initial guesses will be needed. For many initial guesses, a Look-Up Table (LUT) is preferable.

To keep the area of the LUT at a minimum, the number of stored values and the width of the data paths of said values should be kept as low as possible. A good trade-off between the number of iterations and the number of stored values is important.

Fewer iterations will require a bigger LUT with more initial guesses, in order to keep the error small and the answer within a given bit precision, Inversely, a smaller LUT will require more iterations in order to keep the precision.

More iterations in an unrolled architecture require more hardware and will increase area and latency, however, the size of the LUT will also increase area and latency. It is not known beforehand if a smaller LUT will compensate for higher amount of iterations and vice versa.

For the inverse square root function, an algorithm for finding an adequate initial guess has been made. More initial guesses will be needed closer to the y-axis due to the greater down slope. By using an algorithm which calculates the same maximal distance from the actual result for each initial guess throughout the function, a more even approximation will be computed for NR. Figure 4.1 is an example of how the LUT will be represented over the input v. The LUT is the plot which resembles a staircase. Each step is an initial guess over a certain range in v.

The algorithm works by setting the number of values wanted in the LUT, afterwards the algorithm will calculate the placement for each

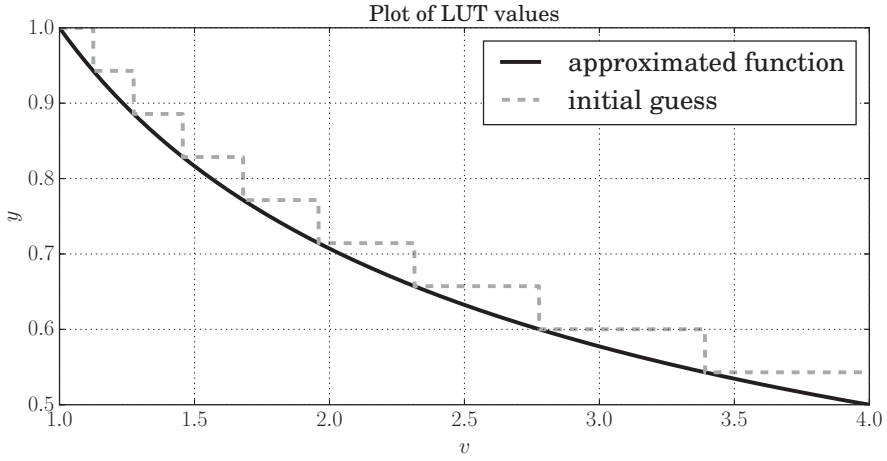


Figure 4.1: *Example of a look-up table with 9 entries for the Newton-Raphson Method.*

initial value so the maximum distance to the function is the same for all values. By doing this, the height between each initial value will be the same. The beginning of the input value range (x-axis) for the LUT is at the intersection of the guess and the approximated function, and the range ends at the beginning of the next LUT entry range, see (4.4).

$$\text{LUT length}\Delta x = x_1 - x_2 = \frac{1}{y_1^2} - \frac{1}{y_2^2} \tag{4.4}$$

Where y_1 is the start of the LUT in y-axes and y_2 is the start of the next LUT in y-axes minus one bit resolution.

Another method to decrease the errors originating from the initial values is to halve the maximum distance between the values in the LUT and the function. If the values of the LUT is lowering half its height in the y-axis, the interception of the inverse square function will be in the middle of the LUT and not in the beginning as in in Fig. 4.1. By doing this, the initial guess will be closer to the actual result as represented in Fig. 4.2.

By comparing Fig. 4.1 and Fig. 4.2, it can be seen that the gap between the initial guess and the actual value is smaller in Fig. 4.2. The longest distance between the initial guess and the actual value is cut in half in Fig. 4.2. Therefore, the method for choosing the initial values will be the one shown in Fig. 4.2.

There is another method when choosing the initial values in the LUT. Instead of first choosing initial values and then truncating the data-

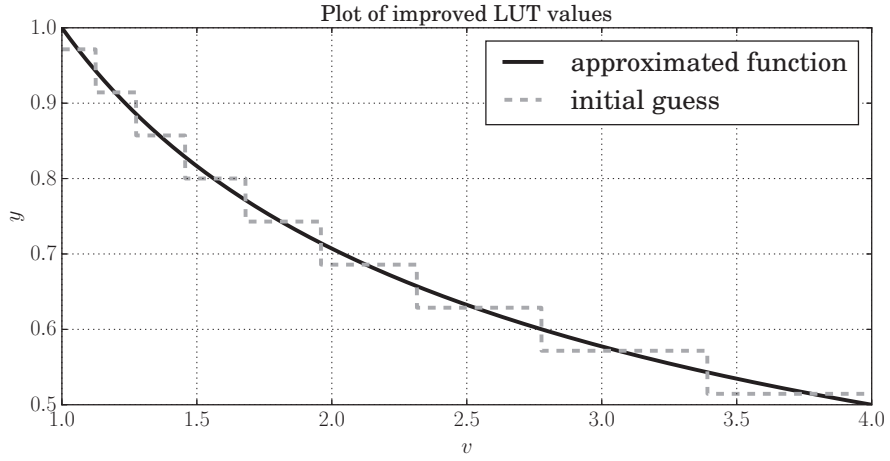


Figure 4.2: *Representation of an improved 9 entry Look-Up Table for the Newton-Raphson Method*

path widths, a reverse method should be possible, where the truncations are done first, and then the initial values are calculated. The idea is to choose a starting point below the function that is just inside the acceptable error range. As the function decreases, the absolute error goes toward zero, and then increases again. When the error goes outside the acceptable limit, a new interval with a new constant is created.

This method was found to be difficult to use. Input signals next (or near) to each other can give different bit precision, were the one closer to the real value can show a bit precision outside the maximum bit precision and the other one inside.

Chapter 5

Error Analysis Theory

To evaluate the error behavior, a number of five methods is used to analyze the two approximation models (the Newton-Raphson Method and Harmonized Parabolic Synthesis), maximum error, mean error, median error, Standard Deviation (SD) and Root Mean Square (RMS) [9].

5.1 Error Behavior Metrics

5.1.1 Maximum Error

The maximum error $e_{\max}$ is the absolute value of the biggest error between the approximation result $\hat{x_i}$ and the ideal result x_i.

$$e_{\max} \;=\; \max|e_i| \;=\; \max|\hat{x_i} - x_i| \tag{5.1}$$

5.1.2 Mean Error

The mean error $\bar{e}$ is the average error between the approximation value $\hat{x_i}$ and the real value x_i.

$$\bar{e} \;=\; \frac{1}{n}\sum_{i=1}^{n} e_i \;=\; \frac{1}{n}\sum_{i=1}^{n}(\hat{x_i} - x_i) \tag{5.2}$$

where n is the number of samples.

5.1.3 Median Error

When calculating the median error, a sorted list of all errors have to be created, which can be costly. The median error is the sample found in

the center of the sorted list. To calculate the median error there are two cases depending on if the numbers of samples is odd or even, see (5.3). If there is a large difference between the median and the mean error, then it is likely that there is a large number of errors placed closely to the mean on one side, and a large outlier on the other side of the mean, skewing the distribution.

$$e_{\text{median}} = \begin{cases} e_{(n+1)/2} & \text{if } n \text{ is } \textbf{odd} \\ \frac{1}{2}(e_{(n+1)/2} + e_{n/2}) & \text{if } n \text{ is } \textbf{even} \end{cases} \tag{5.3}$$

where e is a sorted list of errors,
and assuming 1-based indexing.

5.1.4 Standard Deviation

Standard Deviation σ is used to see the dispersion around the mean. A small standard deviation indicates that the samples tend to lie close to the mean. If the standard deviation is high, this indicates that the samples are spread over a wide range. Equation (5.4) shows how to calculate the standard deviation.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i - \bar{e})^2} \tag{5.4}$$

5.1.5 Root Mean Square Error

The root mean square error, shown in (5.5), gives a measure on how far the error is from 0 on average. The root mean square error differs from the mean error in that two values, of equal magnitude but different sign would have a zero mean, but nonzero root mean square. The main difference between the standard deviation of the error, and the root mean square error is that the size of the mean error has no effect on the standard deviation, but may show up on the root mean square error. If the mean error is small, the difference between the standard deviation and the root mean square error will be small as well.

$$e_{\text{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{x_i} - x_i)^2} \tag{5.5}$$

5.1.6 Skewness

Another way to asess whether or not a distribution is symmetrical, and to compare different distributions, is *skewness*. The definition of the term is somewhat vague and more than one way to measure it has been defined, the one used in this report is shown in (5.6) [10, pp. 183–184].

$$\gamma_1 = \frac{\mu_3}{\sigma^3} = \frac{\frac{1}{n}\sum_{i=1}^{n}(e_i - \bar{e})^3}{\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}|e_i - \bar{e}|^2}\right)^3} \tag{5.6}$$

5.2 Error Probability Distribution

To visualize the error probability distribution, a graph in form of a histogram is a good tool to use. A histogram can show a hint of both the mean error, as well as the standard deviation as described in the previous sections.

An error distribution histogram splits the error range into a number of smaller ranges, and plots a bar for each range, corresponding to how many errors occur in that range. Figure 5.1 shows an example of a histogram. The goal for the error distribution is to be centered around zero. The reason to have the error distribution around zero is that for a large number of calculations, the average error should converge towards zero. For this to happen, the error distributions needs to be centered around zero.

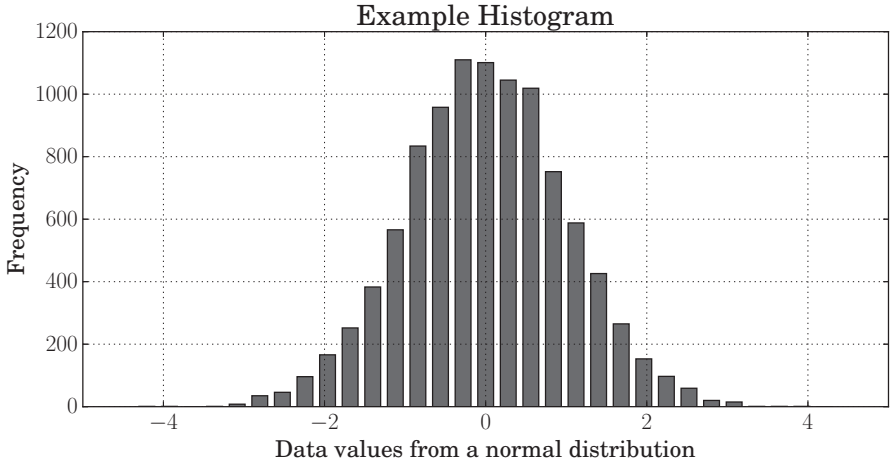


Figure 5.1: *Example of a histogram.*

5.3 Bit Precision

When looking at the error, it is common to do so on the logarithmic scale, a common approach is to use the decibel unit(dB), shown in (5.7), where x is the error [11].

$$x_{dB} = 20\log_{10}(x) \tag{5.7}$$

When using decibel as a measurement, the correlation between bits, and the decibel unit has to be established. This is done by using Signal-to-Noise Ratio(SNR), where noise will represent the error. First the bit resolution must be determent for one bit in fixed-point representation. Shifting the error by one bit to the left or right corresponds to a factor 2 multiplication or division.

1 bit error for fixed-point representation in the fractional part is equivalent to 0.5 (0.1_2). Equation (5.8) shows the corresponding decibel for one bit resolution.

$$x_{dB} = 20\log_{10}(0.5) \approx -6dB \tag{5.8}$$

From (5.8) it is given that one bit resolution in the fractional part is approximately -6 dB. This measurement is not precise but gives a relatively accurate approximation for one bit. For a more accurate equation for bit precision the formula for calculating one bit should be included instead of using the approximation 6 dB. The equation for bit precision using decibel will be as shown in (5.9).

$$bit\,precision = \frac{20\log_{10}(x)}{20\log_{10}(2^{-1})} \tag{5.9}$$

Instead of using division, logarithm law of change of the base can be use to simplify the expression. The logarithm law of change of the base is as followed [12, pp. 78-79].

$$\frac{\log_k(x)}{\log_k(b)} = \log_b(x), \qquad \log_k(x^t) = t\log_k(x) \tag{5.10}$$

From (5.10) a direct equation for bit errors in fractional part will be given without division of the resolution, see (5.11) [13, pp. 40].

$$bit\,error = -\log_2(x) \tag{5.11}$$

The assignment is to calculate the bit precision for the fractional part, were 1 bit is equivalent to 0.5. Using (5.11) this will give -1 bit ($\log_2(2^{-1}) = -1$).

Note that an error of 0.5 (0.1_2) will actually be 0 bit precision in the fractional part due for bit precision in fractional part to be one or higher the error have to be less then 0.1_2.

For the bit precision to be 1, the binary representation of the error has to be $0.1 > x \geq 0.01$, which is $0.5 > x \geq 0.25$ in decimal and $1 > x \geq 2$ from (5.11). This example gives the general condition for bit precision shown in (5.12), where x is the error and bit precision is e.

$$-\log_2(x) = e \implies bit\,precision = \begin{cases} (e-1) & \text{if e = integer} \\ \lfloor e \rfloor & \text{if e} \neq \text{integer} \end{cases} \tag{5.12}$$

For this project, one of the goals is that the output values should have 15 bits of precision to the right of the radix point. Having m significant bits to right of the radix point translates to that the largest absolute error $e_{\max}$ can have no bit set signifying a larger value than 2^{-m-1}. For infinite precision, this means that the largest absolute error allowable can be constructed by the infinite series $\sum_{i=m+1}^{\infty} 2^{-i}$, which converges towards the value 2^{-m}. Given that the representation in simulations and hardware will have limited precision, infinite series and convergence does not have to be considered, and the inequality to consider becomes $e_{\max} < 2^{-m}$. As this project requires 15 bits of precision to the right of the radix point, every implementation will have to pass a trial to make sure that $e_{\max} < 2^{-15}$ for all valid inputs.

Alternatively, one could express the goal as that the project requires a signal to noise ratio (SNR) of slightly above $90.3dB$, given that the peak output value of the function is 1 and the error limit is 2^{-15}, see (5.13). This can be related to the fact that each extra significant bit represents a power of two increase in dynamic range, corresponding to about $6.02dB$, as shown in (5.14)

$$SNR_{dB} = 20\log_{10}\left(\frac{signal}{noise}\right) = 20\log_{10}\left(\frac{1}{2^{-15}}\right) \approx 90.3 \tag{5.13}$$

$$x_{dB} = 20\log_{10}(2) \approx 6.02dB \tag{5.14}$$

Chapter 6

Number Representation

6.1 Fixed-Point Numbers

Fixed-point notation is a simple way to extend regular integral representation of a number into that of the rationals. For integers, a number expressed in base b with n digits, d_{n-1} down to d_0, has the value $\sum_{i=0}^{i-1} d_i b^i$. In the case of fixed-point numbers, the difference can be visualized by introducing an additional constant, the number of fractional digits k, and the value of the number becomes $\sum_{i=0}^{i-1} d_i b^{i-k}$.

When writing out the digits, a radix point[1] is usually used to help differentiate fixed-point numbers from integers, as well as showing which digits are integral, and which are fractional. With integers, the smallest representable number is $b^0 = 1$, whereas for fixed-point numbers it is b^{-k}. Arithmetic operations on fixed-point numbers are similar to integer arithmetic. Negative numbers can be represented through two's complement notation in the same manner as for integers.

6.2 Floating-Point Numbers

A common method to represent (or at least approximate) real numbers in computers and digital electronics, is binary floating-point representation, (e.g. as standardized by the Institute of Electrical and Electronics Engineers (IEEE) 754[14]), where a real number x is stored as a significand a paired with an exponent b so that $x = a2^b$.

[1] The radix point is more commonly known the decimal point when the base is ten, and in a more general sense demarcates the integral and fractional parts of the number

The representation is also normalized, meaning that the significand is made to be greater than one and smaller than the base, and the exponent scaled accordingly.[2] In the case of base two representation, this means that a significand of length n will have the Most Significant Bit (MSB) and Least Significant Bit (LSB) representing 2^0 and 2^{n-1} respectively.

6.2.1 Representation and the Inverse Square Root

The inverse square root operation on a floating-point number $v = a2^b$ can be simplified as shown in (6.1).

$$\frac{1}{\sqrt{a2^b}} = (a2^b)^{-0.5} = a^{-0.5}2^{-0.5b} = \frac{1}{\sqrt{a}}2^{-0.5b} \tag{6.1}$$

If b is odd, the exponent in the final expression $-0.5b$ will not be an integer (which is required due to the datatype representation), which has to be dealt with, e.g. by subtracting 0.5 from the exponent and multiplying the whole expression with the square root of the base, encumbering the equation with an extra multiplication that can not be converted to shifts or other simple operations. If instead base four is used for the representation of the input data, further simplifications can be made, shown in (6.2).

$$\frac{1}{\sqrt{c4^d}} = (c4^d)^{-0.5} = c^{-0.5}4^{-0.5d} = \frac{1}{\sqrt{c}}2^{-d} \tag{6.2}$$

Fortunately, converting integers from base two to base four is simple, since sequential pairings of base two digits (so that the most significant digit in the pair corresponds to an odd power of two) can be transparently interpreted as base four digits. In the case of floating-point represented numbers however, some additional work is necessary, given that the base of the exponential is changed. This conversion (6.3) is however lighter on computing resources than the extra multiplication needed if two was kept as the base.

$$a2^b = a4^{\frac{b}{2}} = \left\{ \begin{array}{lll} a4^{\frac{b}{2}} & \text{if } b \text{ is even} & (d = \frac{b}{2},\ c = a) \\ 2a4^{\frac{b-1}{2}} & \text{if } b \text{ is odd} & (d = \frac{b-1}{2},\ c = 2a) \end{array} \right\} = c4^d \tag{6.3}$$

The normalization of the significand is kept through the conversion to base four, in the case of b being even, the significand c will be in the range

[2] This is not always possible in a finite precision implementation, but subnormal numbers are considered outside the scope of this project

$[1, 2)$, and if b is odd, the range will be $[2, 4)$. From this, together with the final expression in (6.2), it can be concluded that by implementing the inverse square root function $\frac{1}{\sqrt{v}}$ for fixed-point numbers in the range $v \in [1, 4)$, the function can be implemented for the whole range of the floating-point data type with little extra cost.

Chapter 7

Calculating the Inverse Square Root

7.1 Harmonized Parabolic Synthesis

Section 6.2.1 showed the relative ease with which it is possible to extend a fixed-point implementation of the inverse square root, to a fully working floating-point implementation. The requirement is that the fixed-point implementation can operate on an input range $v \in [1, 4)$. However, the input working range for the HPS algorithm is $x \in [0, 1]$, as discussed in Chapter 3, and for this reason a pre-processing function is necessary. Such an equation and its inverse is shown in (7.1).

$$\begin{aligned} f_{pre}(v) &= \frac{v-1}{3} \\ f_{pre}^{-1}(x) &= 3x+1 \end{aligned} \tag{7.1}$$

A post-processing function is required as well, since the inverse square root function output varies between 1 and 0.5 for the desired input range, and the output working range of the HPS algorithm is $y \in [0, 1]$. The post-processing function and its inverse is shown in (7.2).

$$\begin{aligned} f_{post}(y) &= \frac{y+1}{2} \\ f_{post}^{-1}(z) &= 2z-1 \end{aligned} \tag{7.2}$$

Other pre-processing and post-processing functions exist, e.g. $f_{pre2}(v) = (4-v)/3$, however the subtraction of a variable is a more expensive

operation than the subtraction of a constant, and even more so if the addition of the constant only affects the integral part of a number in which most bits signify fractional values.

With both pre-processing and post-processing functions defined, the $f_{org}(x)$ function can be defined, as shown in (7.3).

$$\begin{aligned} f_{org} &= f_{post}^{-1} \circ f \circ f_{pre}^{-1} \\ f_{org}(x) &= \frac{2}{\sqrt{3x+1}} - 1 \end{aligned} \tag{7.3}$$

From the shape of $f_{org}(x)$, i.e. it intersects $(0;1)$ and $(1;0)$, the implementation of $s_1(x)$ is chosen to be that of (3.3b): $s_1(x) = 1 - x + c_1(x - x^2)$.

In order to check if $f_{org}(x)$ is viable for approximation with HPS, the limit of $f_{help}(x)$ as x approaches 1 needs to be checked for existence and calculated. The definition of $f_{help}(x)$ is shown in (7.4) and the limit value is shown in (7.5). It can be concluded that approximating the inverse square root with HPS should be possible as long as $c_1 \geq -1$, and as long as $-1/c_1 \notin [0, 1]$ (From the second zero discussed in Section 3.1.2, for which $f_{org}(x)$ has no defined limit.).

$$f_{help}(x) = \frac{f_{org}(x)}{s_1(x)} = \frac{\frac{2}{\sqrt{3x+1}} - 1}{1 - x + c_1(x - x^2)} \tag{7.4}$$

$$k_{2,I-1} = \lim_{x \to 1} f_{help}(x) = \frac{3}{8c_1 + 8} \tag{7.5}$$

7.2 The Newton-Raphson Method

In Chapter 4 it was shown with an example of how to use NR for calculating $g(v) = 1/v$. With the same principle, $g(v) = 1/\sqrt{v}$ will be calculated, shown in (7.6) and (7.7).

$$\begin{aligned} y = g(v) = \frac{1}{\sqrt{v}} \quad \Rightarrow \quad v = g^{-1}(y) = \frac{1}{\sqrt{y}} \\ f(y) = \frac{1}{y^2} - v \\ f'(y) = -\frac{2}{y^3} \end{aligned} \tag{7.6}$$

$$
\begin{aligned}
y_i &= y_{i-1} - \frac{\frac{1}{y_{i-1}^2} - v}{-\frac{2}{y_{i-1}^3}} \\
y_i &= y_{i-1} \times \tfrac{1}{2}(3 - v \times y_{i-1}^2)
\end{aligned}
\tag{7.7}
$$

As in (4.3) the algorithm for calculating the inverse square root can be done without any division (division by 2 can be done by shift). A good approximate algorithm is given for the inverse square root for a given amount of iteration. The bit precision will depend on the number of iterations and the proximity of the initial guess to the actual result.

Chapter 8

Hardware Architecture

8.1 Harmonized Parabolic Synthesis

The hardware architecture of the HPS algorithm follows the equations described earlier, as shown in Fig. 8.1 pre-processing and post-processing blocks that transform the input and output of the HPS algorithm block to and from the necessary range, and the HPS block itself, where the two subfunctions $s_1(x)$ and $s_2(x)$ reside.

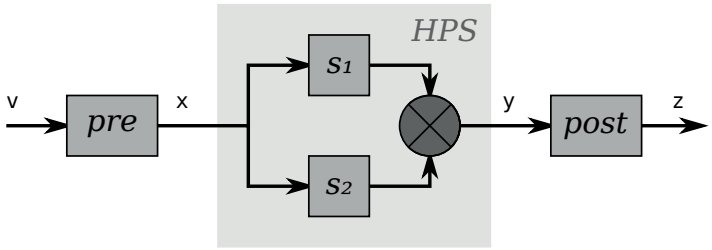


Figure 8.1: *A bird's eye view of the HPS hardware architecture*

8.1.1 Pre-Processing

The pre-processing block, shown in Fig. 8.2 is quite simple, one subtraction, and multiplication by a constant. The subtraction can be made very simple due to the fact that the input v is defined to be in the range $[1, 4)$, and the subtrahend is integral, consequently, the subtraction will

have no effect on the fractional bits of v, and only two integral bits will ever need to be modified.

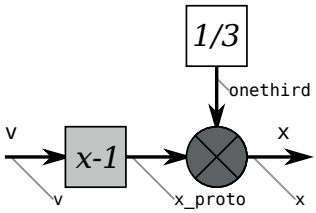


Figure 8.2: *The pre-processing Function*

8.1.2 Processing (HPS)

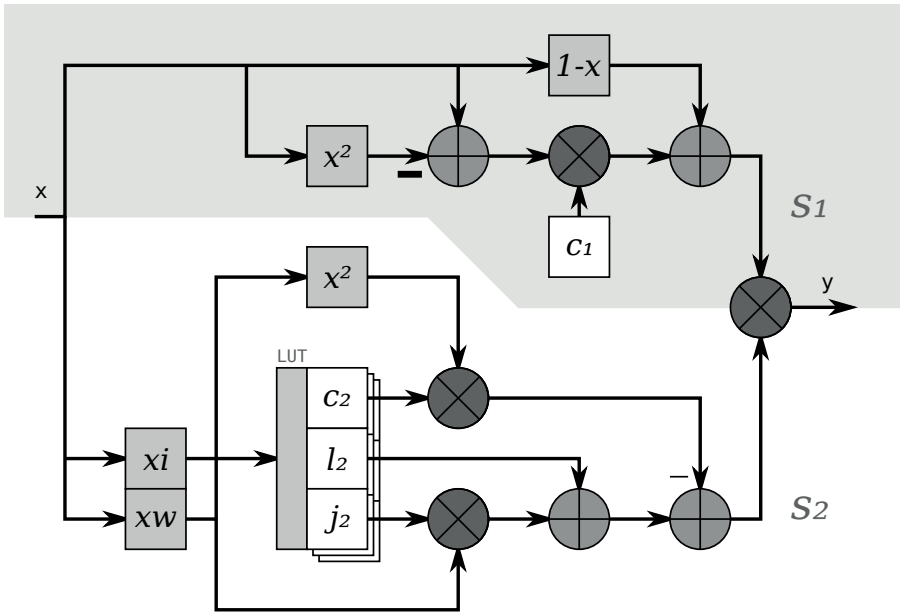


Figure 8.3: *The processing/HPS function*

The processing block contains the two subfunctions, $s_1(x)$ and $s_2(x)$. Choosing to restrict the potential interval counts I to integral powers of two simplifies the generation of x_i and x_w. The value x_i, used to index the look-up table, is calculated as $x_i = \lfloor I\,x \rfloor$, and x_w, is calculated as

$x_w = I\,x - \lfloor I\,x \rfloor$. In both cases the multiplication $I\,x$ becomes a shift, and the equations can be simplified as extracting the correct bits from x and reinterpreting their value. The expression of $s_2(x)$ in Fig. 8.3 is also a bit different than in (3.5), introducing the constant $j_{2,i} = k_{2,i} + c_{2,i}$, which allows expressing $s_2(x)$ with fewer operations, shown in (8.1). It should also be noted that in the actual hardware implementation, the subtraction of $c_{2,i}x_w^2$ is made into an addition, by changing the sign of the $c_{2,i}$ values stored in the look-up table. The motivation is that the hardware becomes simpler.

$$\begin{array}{rcccccc} s_2 & = l_{2,i} & + & k_{2,i}\,x_w & + & c_{2,i}\,(x_w - x_w^2) \\ & = l_{2,i} & + & (k_{2,i} + c_{2,i})\,x_w & - & c_{2,i}\,x_w^2 \\ & = l_{2,i} & + & j_{2,i}\,x_w & - & c_{2,i}\,x_w^2 \end{array} \tag{8.1}$$

8.1.3 Post-Processing

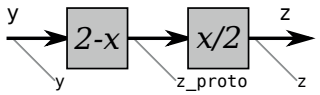


Figure 8.4: *The post-processing Function*

The post-processing block in Fig. 8.4 can be optimized as well, y is defined to be on the range $(0, 1]$ and, like in the pre-processing case, it interacts with an integer, so the actual change of the data when added together with a constant 1 does only affect the integral bits. The last block, dividing the result of the addition by 2 can be implemented as a simple bit shift, or in the case of a hardware implementation, as a reinterpretation of the significance of the bits in the data path.

8.2 The Newton-Raphson Method

From (7.7) the hardware architecture for the inverse square root can be designed. The simplified algorithm will be as follows:

$$y_i = y_{i-1} \times \frac{1}{2}(3 - v \times y_{i-1}^2) \tag{8.2}$$

The components needed for this algorithm are a subtractor, two multipliers and a shift. The initial value (y_0) based on the input v, will be placed in a look-up-table (LUT). Figure 8.5 shows an unrolled

NR architecture. The iteration count is preferable to keep low, in order to keep a narrow data path (enabling shorter clock periods) and area requirements.

The actual number of iterations and the values placed in the LUT will be determined by the results from the simulations in Matlab. These simulations do also decide the necessary data-path widths of the data paths, so a schematic can be made.

A number of different combination of iterations, LUT and data-path widths will be tested during the simulation process.

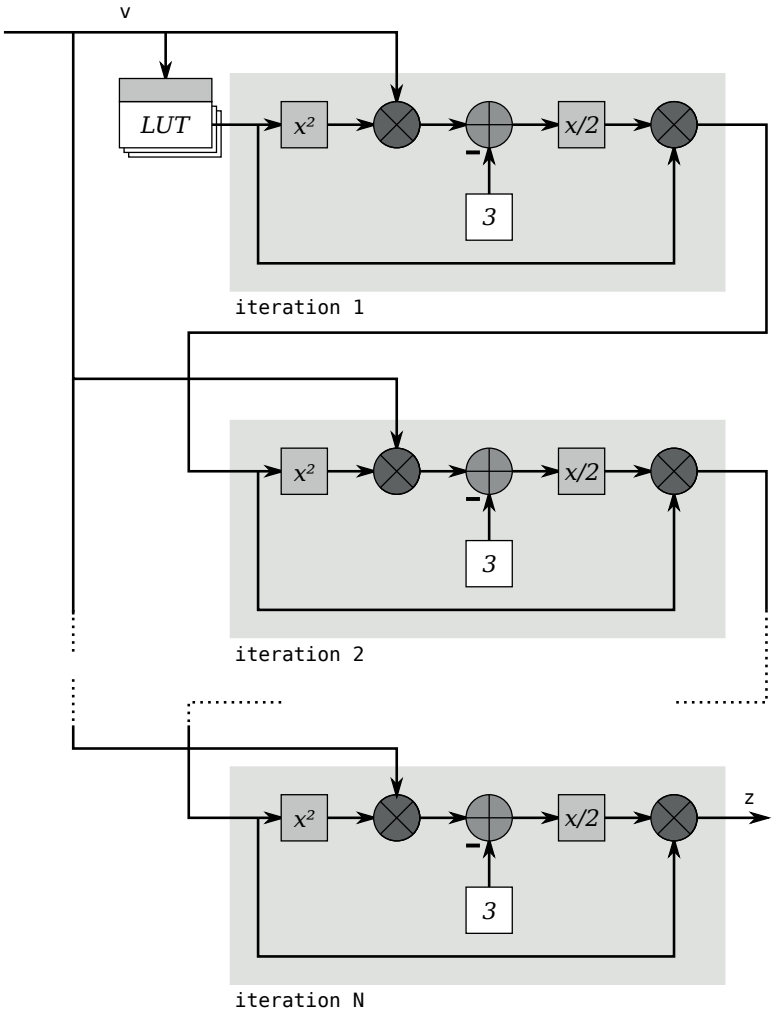


Figure 8.5: *Unrolled Newton-Raphson architecture*

Chapter 9

Implementation: Data Flow Design and Simulation

From the theory in Chapter 3 and Chapter 4, the algorithms for Harmonic Parabolic Synthesis and the Newton-Raphson Method are given. To be able to implement these algorithms in hardware, each algorithm needs to have their corresponding constants calculated and the data paths constrained. The effects of combining different constant values and constraints are tested and verified in software based simulations, written in Matlab and C.

For Harmonic Parabolic Synthesis, the value of the constant c_1 needs to be chosen, as well as the number of intervals to use in the subfunction $s_2(x)$. The values for the constants (c_2, l_2, j_2) in each interval is calculated as well.

In the Newton-Raphson Method, the LUT for the initial guesses needs to be calculated and the number of iterations chosen.

For both algorithms, the dataflow of a complete circuit with constrained widths for each data path will be simulated. The final result, z, is going to have at bit precision of at least 15 bits in fractional part, except for $v = 1$ for which the requirement is that $z = 1$ *exactly*, as described in Chapter 2 Scope. These are the primary design objectives.

This chapter will show how these constants are calculated or chosen with the help of the error metrics, probability distribution, and bit precision found in the error analysis theory of Chapter 5.

9.1 Harmonized Parabolic Synthesis

In the theory in Chapter 3, methods for calculating values for all the values are presented, except for the value of c_1, the remaining yet to be defined constant in the first sub-function $s_1(x)$, and the number of intervals I in which to split the second sub-function, $s_2(x)$. Additionally, the required precision of the various data paths in the implementation need to be determined.

There exists no known simple mathematical rule of thumb for choosing c_1 and I and they, as well as suitable constraints on the data path have to be determined by simulation.

9.1.1 Choosing c_1 and Interval Count I

In order to choose suitable values for the constant c_1 and the number of intervals in which to split $s_2(x)$, the maximum error for the approximation, over the entire input interval $[1, 4)$ was gathered and plotted for a number of intervals (integral powers of two), and values of c_1 (a high number of points selected from a continuous range). The results are shown in Fig. 9.1

Given the fact that c_1 is going to be fed into a multiplier, choosing a value that is an integral power of two is desirable. This since fixed-point multiplications and divisions by powers of two can be realized as shifts, which when it comes to hardware, is just a reinterpretation of what the bits present in the data path signify. From an implementation standpoint setting c_1 to zero should be even more beneficial, since multiplication by zero always equals zero, enabling removal of hardware that does not contribute to the final result.

From Fig. 9.1 it can be concluded that at least 32 intervals are required in order to get a small enough error, and that increasing the interval count decreases the error. For all interval counts for which there exists a point below the maximum error limit, choosing the constant $c_1 = 0$ is viable, and is selected as the value to be used in all implementations. Now the design can be simplified to the flow shown in Fig. 9.2.

Comparing the processing part in Fig. 8.3 with the corresponding part of Fig. 9.2 the immediate benefit of selecting $c_1 = 0$ becomes apparent, one multiplier and two adders can be removed.

Following the selection of zero as the value for c_1, software tools to simulate the data flow behaviour and to report and/or visualize it in various ways were developed. This in order to try multiple combinations of interval counts together with testing how much the data paths can be constrained without the maximum error becoming too big. Both of

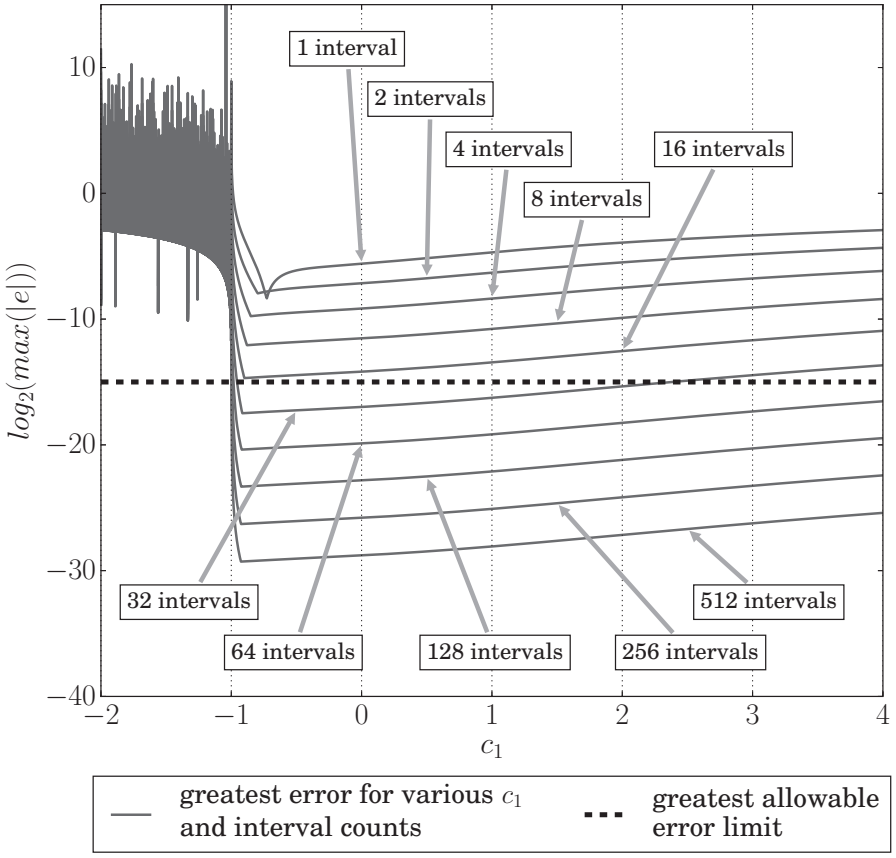


Figure 9.1: *Plot of the largest approximation errors for the whole input range ($v \in [1, 4)$), when choosing various numbers of intervals and values for the constant c_1*

these parameters give hard constraints for how the hardware can be implemented. The properties of the simulation software and development work is expanded on in Appendix A.2.1 HPS Data Flow Simulation.

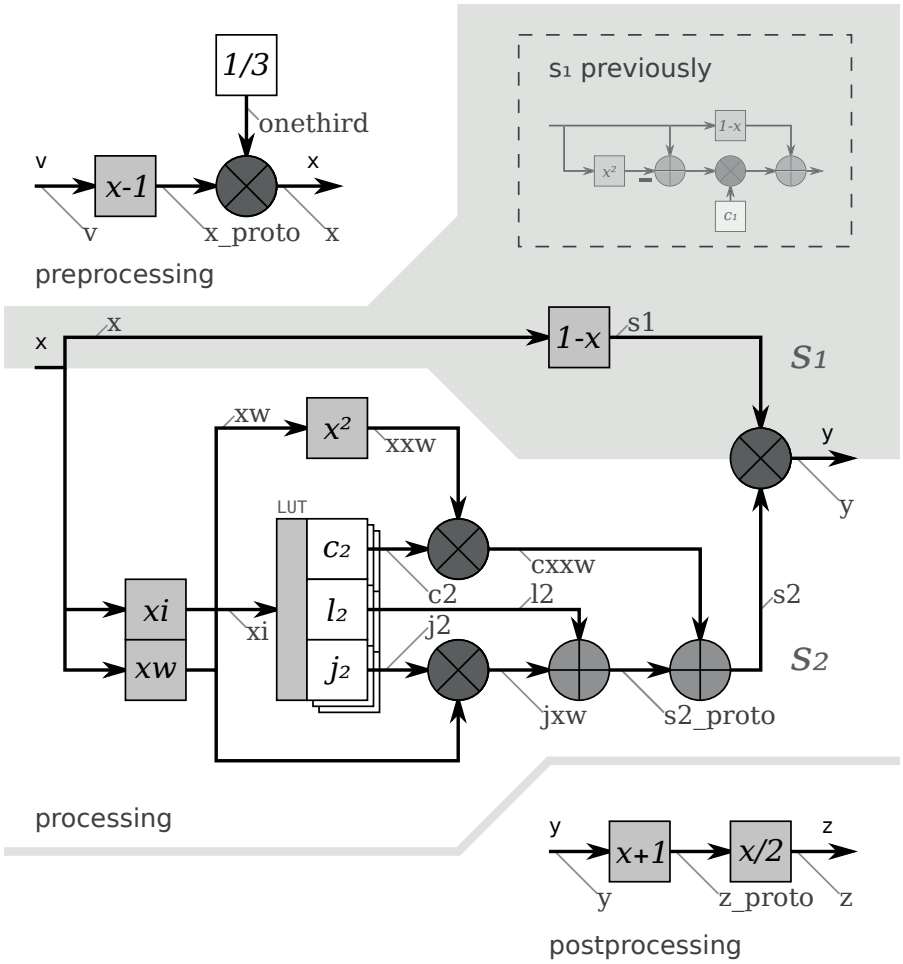


Figure 9.2: *The architecture of the implementation of the HPS algorithm; the data paths are annotated with the names they have been given in the simulation output. The implementation of s_1 has been simplified following the selection of $c_1 = 0$*

Apart from removing leading and trailing zeroes from the data paths, some of the paths are only ever representing negative values, making the most significant bit set at all times, making it possible to hard-wire it as a constant. In Table. 9.1 these paths are marked.

data path	data-path width	
	32 intervals	512 intervals
v	15	15
x_proto	15	15
onethird	14	14
x	16	16
xw	11	7
s1	17	17
xxw	12	5
l2	18	17
j2	13 (14)*	10
c2	9	9
jxw	14	9
cxxw	11	6
s2_proto	18	17
s2	18	18
y	16	16
z	17	17
zz	18	18

Notes:
* Always negative, meaning that the MSB is always 1 and does not have to be stored

Table 9.1: *Width of each data path in two different HPS implementations.*

9.2 The Newton-Raphson Method

9.2.1 Design Constraints

For the Newton-Raphson Method there are not many design constraints that will affect the approximation of the results. The two main constraints are the number of iterations and the count and values of the initial guesses, and the third is the width of the data paths in the design. By evaluating combinations of these three constraints, a design can be decided on.

The first of these constraints to be determined is the number of iterations versus the number of initial guesses in the Look-Up Table (LUT). In this early step of the design analysis from one iteration to four iterations is done in Matlab by using floating-point representation of the numbers. The method for calculating the initial guesses is explained in Chapter 4.2. When using Matlab to calculate it's crucial to use the

same value representation in float number as it would be in fixed-point representation. The solution is to divide v into 24576 values in the range of $1 \leq v < 4$ to simulate the real input from v with 15 bits in fixed-point representation. By starting at 1 and ending at value 3.998779296875 (11.1111111111111 in fixed-point representation). A representation of iteration vs. LUT is shown in Table 9.2.

1 iteration		2 iterations		3 iterations		4 iterations	
LUT	Max error	LUT	Max error	LUT	Max error	LUT	Max error
81	15.0915	9	16.0772	4	20.9267	2	19.6506
79	15.0403	8	15,3849	3	17,0273	1	17,2487
78	14.9747	7	14.5989	2	10,4489		

Table 9.2: *Bit precision for iteration count vs LUT size for floating-point numbers*

From Table 9.2 the number of LUT entries each iteration needs in order to have a bit precision of 15 bits or more. Table 9.2 gives that one and two iterations are enough for a precision of 15 bits. A good trade-off is done here, and three and four iteration are no longer candidates for further analysis. The decision is based on that more iterations results in a slower implementable minimum clock period and will also increase the area of the design.

9.2.2 Data-Path Width

The first analysis with floating-point representation is not optimized according to data-path width and an analysis with fixed-point representation for one and two iterations will be done. In fixed-point representation the design should have as small path width as possible to minimize the clock period and also decrease the area, since wider data requires bigger, slower components to implement operations on them. For this purpose Matlab will be used by using "Fixed-point numeric object" toolbox. The rounding method will be set to floor and overflow action to wrap to simulate truncation and overflow in a real Resistor Transistor Logic (RTL) design.

There are no good analytical methods to find the optimal data-path widths since a wider path can give a worse precision than a more narrow one. The method for this approach is simply trial and error, this project does it by decreasing paths in the beginning of the design.

Besides a precision of 15 bits, a good error distribution is to be achieved and needs to be accounted for.

Because of truncation, the LUT found in Table 9.2 will not be enough. The algorithm used in Table 9.2 uses the default rounding method in Matlab and no truncation for each operation will be done. In reality a truncation will be done after each operation of the algorithm. The main problem is not how many LUT entries the design will require but to find a good relation between bit precision and the error distribution which is to be centred around zero. Seen in Table 9.3 for one and two iterations, a number of LUT entries have been added to maintain a bit precision of at least 15 bits. For 1 iteration all paths have a width of 18 bits and for 2 iterations the width is 17 bits for all paths.

1 iteration		2 iterations	
LUT	Max error	LUT	Max error
94	15.4241	13	15.7297
92	15.2967	12	15.7708
88	15.2235	11	15.7897

Table 9.3: *Bit precision for iteration count vs LUT size for fixed-point numbers*

For these designs there is still room for improvement for the data paths, but the next step will be to improve the error distribution. It is beneficial to have some leeway, when modifying the design so that the error distribution comes closer to what is described in Chapter 5.2. The goal is to have SD and RMS as identical as possible. Changes in data-path widths can now make the bit precision go under the required 15 bits and/or make the error distribution worse.

After many simulations, LUT sizes were chosen to be 13 entries for 2 iterations and 94 entries for 1 iteration, due to the good error distribution with the given widths of the paths. The final schematics for 1 iteration and 2 iterations and can be seen in Fig. 9.3 and Fig. 9.4. The bit precision for the final schematic can be seen in Table 9.4.

When all the values in the LUT for 1 iteration and 2 iterations are chosen there is one more constraint that needs to be taken into account. When $v = 1$ the result needs to be $1/\sqrt{1} = 1$. If the first initial guess for when $v = 1$ is chosen to be 1, simulation of the final schematics of Fig. 9.3 and Fig. 9.4 gives the result 1, the constraint is satisfied.

This means that one more initial value will be put into the LUT; for the input value $v = 1$ the initial guess will be 1, ensuring a correct answer. This extra value will increase the amount of LUT entries to 95 for 1 iteration and 14 for 2 iterations.

1 iteration		2 iterations	
LUT	Max error	LUT	Max error
95	15.0376	14	15.7297

Table 9.4: *Final bit precision for fixed-point numbers, iteration vs. LUT.*

It is time consuming work to find the best error distribution by changing path widths for different LUT sizes. To not get stuck trying to find the best solution, the design is considered to have met its goal when a good enough error distribution is found with a bit precision of at least 15 bits.

Fig. 9.3 and Fig. 9.4 show the final schematics which are to be implemented in VHDL code and analyzed. The data-path widths for the architectures are shown in Table (9.5).

The 14 and 95 LUTs' values and the ranges of v the entries are responsible for can be seen in Appendix A.1.

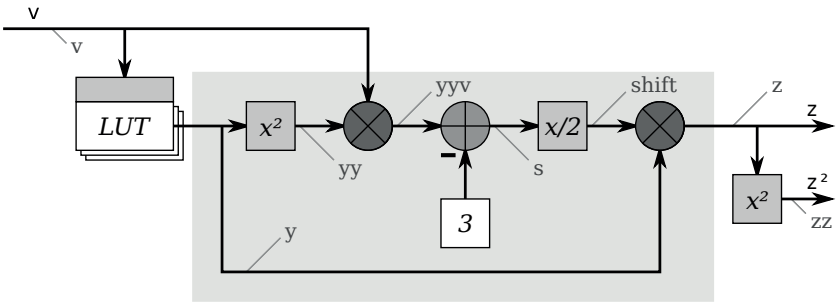


Figure 9.3: *The architecture for 1 iteration of the Newton Raphson method*

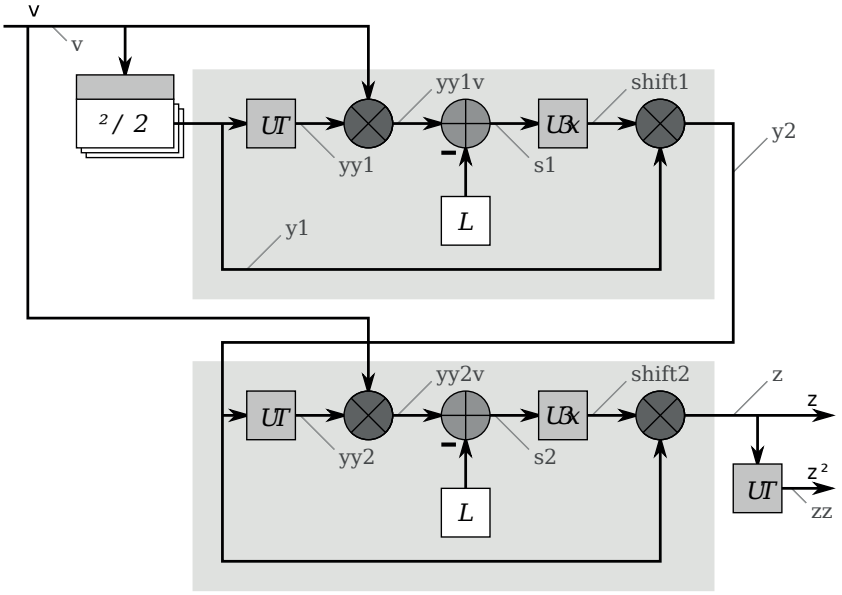


Figure 9.4: *The architecture for 2 iterations of the Newton Raphson method*

1 iteration		2 iterations			
		First		Second	
v	15	v	15	v	15
y	15	y1	16	y2	16
yy	16	yy1	16	yy2	17
yyv	19	yy1v	16	yy2v	17
s	19	s1	16	s2	18
shift	19	shift1	16	shift2	18
z	18			z	17
zz	19			zz	19

Table 9.5: *Width of each data-path in two different NR implementations.*

9.2.3 Optimization

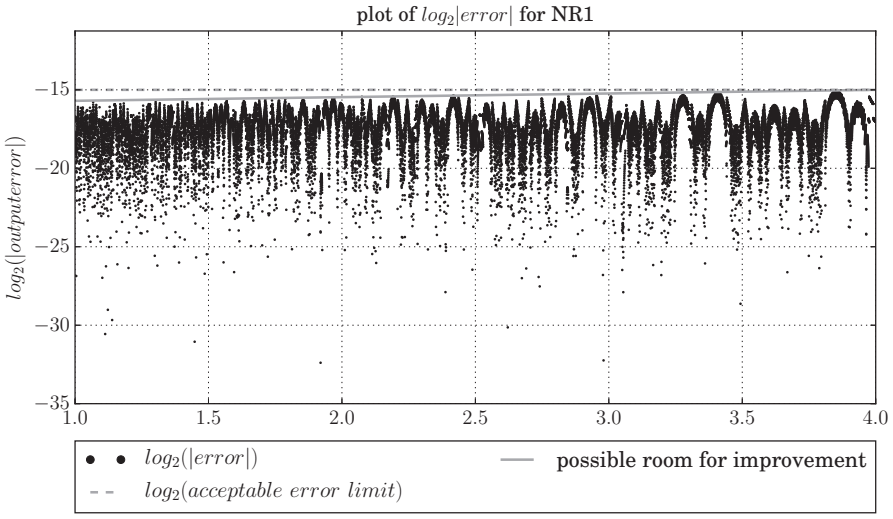


Figure 9.5: *Bit precision for 1 iteration with horizontal line.*

When analyzing the bit precision of z over the range of v for 1 iteration the bit precision slowly gets worse for an increasing v. Demonstrated by placing a straight line roughly atop the peak values of the calculated precision, as seen in Fig. 9.5. If more time could be spent on the algorithm for optimizing the LUT values, the extra leeway in precision for low v could possibly be used to increase the shape of the error distribution.

To find the best relation between error distribution and bit precision, a script to test different path widths would be the best solution. Because the simulation of the Newton Raphson method was done in Matlab every simulation took at least 30 min. to run, adding together all different combinations of wordlentgh, the time spent on data path simulations would be too long. To narrow it down, simulations were carried out for a set of numbers of LUT values meeting the bit precision and different combinations of path widths. For all these simulations, the bit precision and the error distribution was analyzed. A single change in one bit could be enough for making a good error distribution to change into a bad one, especially when the width of the paths already have been cut down.

Chapter 10

Implementation: Hardware Design and Simulation

To generate the circuit layout of the physical design, the algorithm to be implemented will go through three steps: expression as VHDL code, Synthesis, and Placement and Routing (PNR). Each of these steps include a number of verifications and tests before it can proceed to the next step. Figure 10.1 shows the workflow for the hardware part of this project.

After placement and routing, Simulation and Power Analysis are done to generate statistics concerning power consumption. These are the last steps for this project. Multiple runs of the steps from synthesis to power analysis, will be made for the physical design, using different libraries, voltages, and clock periods. The goal is to analyze the different characteristics for these three constraints. Workflow in Fig. 10.1 is a basic representation and is only showing the flow from a birds-eye view. In reality, each step is more complex and the chart would be too big if all were put into one flowchart. Section 10.1 - Section 10.4 will go through each step in more detail.

10.1 VHDL

The final simulations of the VHDL code have been done in QuestaSim - 64 10.0d from Mentor Graphics®. Initial behavioural simulations have been done to some extent in GHDL as well, accompanied by GTKWave.

Figure 10.2 shows the workflow for the creation of the VHDL code. The working process before writing VHDL is to split the algorithms

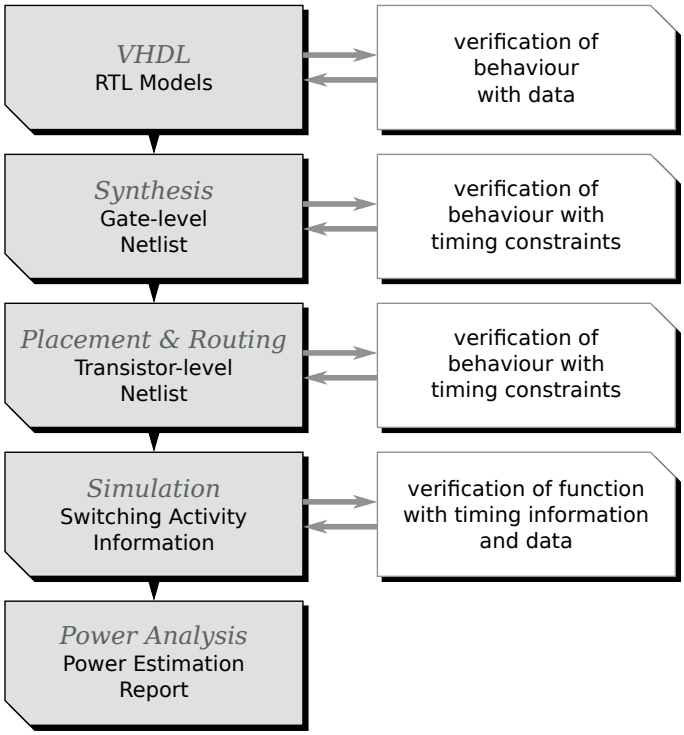


Figure 10.1: *Hardware design workflow*

into small individual components resulting in a top-down analysis of the design. Each component needed will be created and tested in a testbench. Components needed in more than one place are made so that width of the input and output signals can be set with VHDL generics.

When all components needed have been created and tested, they are put together in bigger blocks representing subfunctions in the algorithm. These blocks are tested and verified with test vectors generated from the earlier data-path simulations.

The final step is to connect all the blocks to create the fully functional algorithm. The fully functional algorithm is tested to verify the function with test vectors data-path simulations. The testing of all components and finally the whole algorithm is thus carried out in a bottom-up manner, where the bigger blocks are implemented once their internal components have been tested and their function verified.

To make the RTL design synchronized with a clock, flip-flops are added for the output signals. Figure 10.3 shows what the real RTL design look like, the first flip-flops will be only simulated by the test

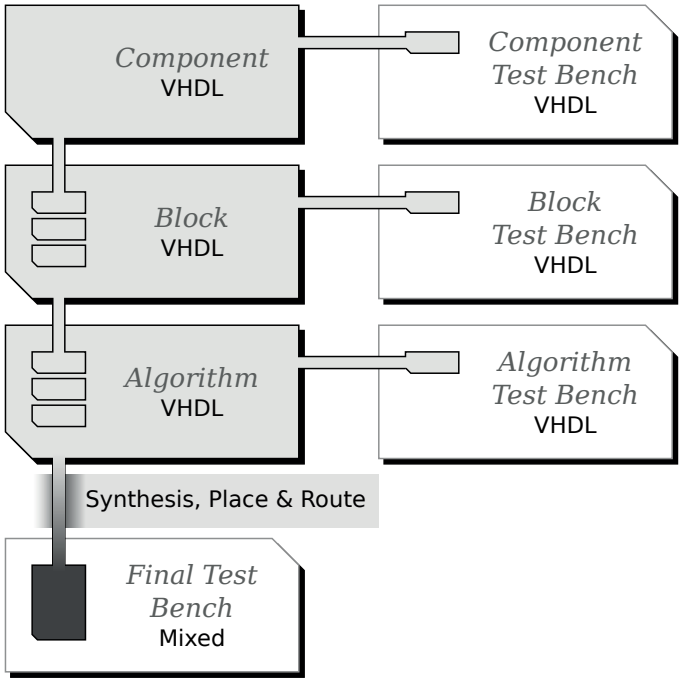


Figure 10.2: *VHDL development workflow*

bench. The content of the combinatorial logic in the figure depends on which algorithm to implement, other than that, the design will look the same for both algorithms.

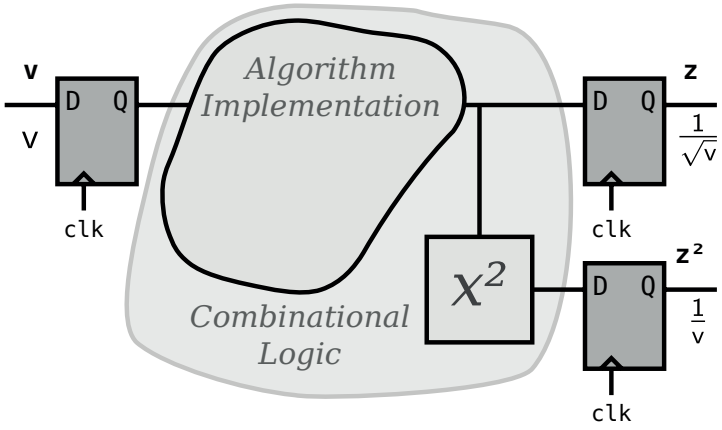


Figure 10.3: *The real RTL design with flip-flop.*

Before the creation of the components, analysis shows that numerous components can be specialized for their purpose. When examining the architecture, some standard components could be merged together in order to save latency and area. Some of these components can be used in both the architecture for HPS and the architecture for NR. Most of the specialized components are made specifically for only one of the two architectures. Chapter 10.1.1 - Chapter 10.1.5 describes how these components are optimized.

10.1.1 Semi-Generic Multiplier

When implementing a multiplier, it is useful if the properties of the numbers it is going to process can be known at implementation time. Some of the multipliers used in this project need to be able to multiply combinations of positive and negative numbers, but it so happens that for the cases in this project, only multiplications between two unsigned, or one unsigned and one signed, number is required. This is easier to implement than a multiplier that needs to be able to multiply two signed numbers.

Table 10.1 shows the operation of a simple multiplier, and also demonstrates that it is not sufficient for handling negative numbers.

	`010`	(2_{10})
$\times$	`110`	(6_{10} or in two's complement: -2_{10})
	`000`	$010_2 \times 0_2$
	`010`	$010_2 \times 10_2$
$+$	`010`	$010_2 \times 100_2$
	`001100`	(12_{10})

Table 10.1: *Simple binary multiplication*

The binary representation 110_2 will be -2_{10} in two's complement but 6_{10} otherwise, and since the multiplier is not built to handle negative numbers the result becomes 12_{10}. When exactly one of the numbers is signed some modifications can be done in the last step.

Given a binary two digit number with the digits a and b, the two's complement interpretation of the number would be $-2 \times a + 1 \times b$, multiplying it with a known to be unsigned number (xy), is relatively easy. The principle is shown in (10.1).

$$\begin{aligned} ab \times xy = \\ -2_{10} \times a \times xy + \quad 1_{10} \times b \times xy = \\ 2_{10} \times a \times -xy + \quad 1_{10} \times b \times xy = \\ 2_{10} \times a \times (\overline{xy} + 1) + \quad 1_{10} \times b \times xy \end{aligned} \tag{10.1}$$

By inverting the positive number xy and adding one (which corresponds to negation in two's) in the last step, the correct result will be given. Table 10.2 shows an example with the modified procedure. In this case 010 inverted becomes 101 and then by adding one on 101 the result will be 110.

	010	(2_{10})
$\times$	110	(-2_{10})
	000	$010_2 \times 0_2$
	010	$010_2 \times 10_2$
$+$	110	$(\overline{101_2} + 1) \times 100_2$
	11100	(-4_{10})

Table 10.2: *Binary multiplication modified for one negative input*

For hardware design this means that only slight modifications to the last step (the step where the unsigned input is multiplied by the most significant bit of the signed input) are needed. The 1 that has to be added in the negation can be implemented as routing the most significant bit of the signed input to the carry in of the last adder chain in the multiplier. Figure 10.4 is an example of a 3x3 multiplier of this kind, where x is the unsigned number, and y is signed. Note the inverter in the last step for x and the carry in set to y_2. This architecture only works in the cases where it is known that y is signed and x is always unsigned. This component will be used in the architecture HPS implementation, whereas in NR there will at no time be a multiplication with negative numbers.

The difference between this and a multiplier only handling unsigned numbers is the inverters for x on the lowermost row, and feeding the negative value bit (y_2 in the figure) as the carry-in signal to the adder producing z_2. Feeding y_2 and not a static one is done since the number should only be added when the most significant bit of y is set. This

multiplier would also work for two unsigned inputs, as long as y is extended with a zero on the most significant side, ensuring that input is always interpreted as a positive number, if it is knowable that both inputs are always positive (representable as unsigned numbers) it is a waste to build a more complex multiplier than necessary.

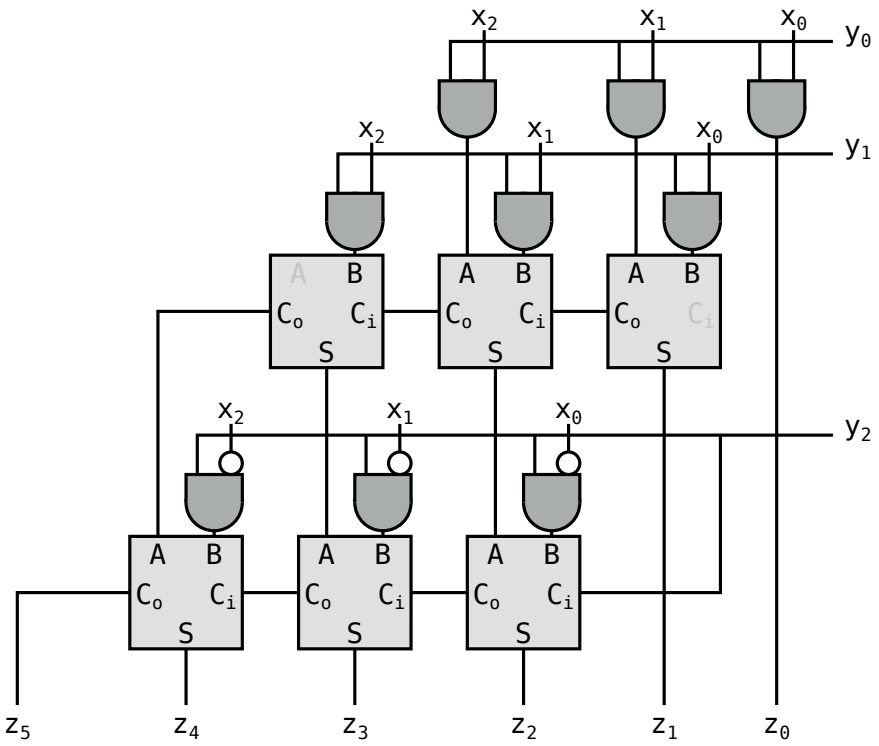


Figure 10.4: *3x3 multiplier for one negative and one positive number*

10.1.2 Algorithm for Squaring Component - Jingou Lai

The algorithm for the squaring component $y = x^2$ has been developed by Jingou Lai for his master thesis "Hardware Implementation of the Logarithm Function using Improved Parabolic Synthesis", 2013 [5].

10.1.3 Pre-Processing

The subtraction of 1 in the pre-processing component in Fig. 8.2 will, as previously stated, only affect the two integral bits in v, and an implementation can be designed from a truth table, containing only the integral bits, which are the two most significant bits, shown in Table 10.3.

in_{MSB} 2^1	in_{MSB-1} 2^0	out_{MSB} 2^1	out_{MSB-1} 2^0
0	0	–	–
0	1	0	0
1	0	0	1
1	1	1	0

Table 10.3: *Truth table for pre-processing subtraction*

From the truth table in Table 10.3 the boolean expression can be written for out_{MSB} and out_{MSB-1} as:

$$out_{MSB} = in_{MSB} \wedge in_{MSB-1}$$
$$out_{MSB-1} = \overline{in_{MSB-1}}$$

With the boolean expression, the final architecture of the subtraction in the integral part will be as Fig. 10.5

From this design the subtraction can be done by the use of one inverter and one AND gate, instead of the traditional subtraction which would use as many Full Adders (FA) and inverters as the input length.

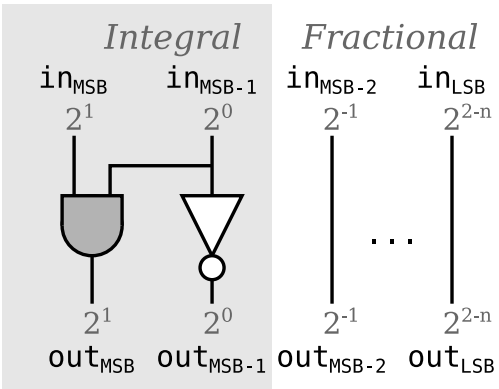


Figure 10.5: *Final design of the pre-processing subtraction only affecting the integral part of v*

For multiplying with one third a general multiplier will be used (described in Chapter 10.1.1) with one input as a fixed constant of one third. The representation will be a sequence of alternating '1's and '0's with two '0's in the beginning as 0.010101..., where the length will determined by the required precision.

10.1.4 Post-Processing

To bring the result of the HPS algorithm back to the range of the approximated function, the pre-processing function is implemented. The formula for this function is $z = (y + 1)/2$ as described in Section 8.1.1. The basic architecture is shown in Fig. 8.4.

This whole architecture can be merged together into one optimized component. This optimization can be done in one simple step.

We know from calculation the upper and lower boundary of y, (0,1]. Adding one will only effect the integral part and the truth table will be as in Table 10.4 (with y as the input and z as the output).

in_{MSB} 2^0	out_{MSB} 2^1	out_{MSB-1} 2^0
0	0	1
1	1	0

Table 10.4: *Truth table for post-processing*

The corresponding boolean expression given from Table 10.4 will be:

$$out_{MSB} = in_{MSB}$$
$$out_{MSB-1} = \overline{in_{MSB}}$$

The design of the addition of one in the post-processing function is shown in Fig. 10.6. Note that MSB for the output (z) is now interpreted as having the value 2^1. The division by two (shift) operation will be done in the next step.

The last step is to implement the shift operation. In the final design the most significant bit of the result will represent the value 2^0. There is an easy way to do a shift operation in fixed-point number representation, simply by changing the binary representation one step. When calculating with fixed-point represented numbers, the designer decides where the fractional part starts. Reinterpreting the value of the bits previously representing 2^n (where the value of n depends on the bit) to

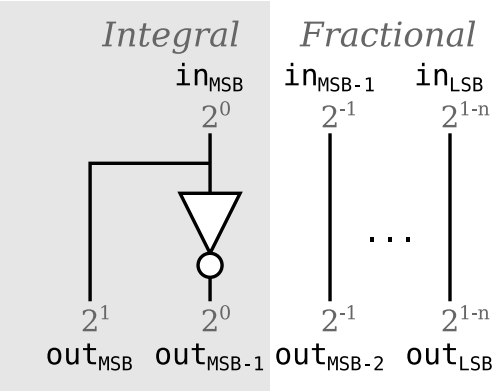


Figure 10.6: *Post-processing addition of one.*

instead represent 2^{n-1} a division by two is made, shown in (10.2). Of course, any multiplication by an integral power of two can be made this way.

$$\begin{aligned} x &= \sum_{i=a}^{b} 2^i \\ \frac{x}{2} &= \frac{1}{2}\sum_{i=a}^{b} 2^i = \sum_{i=a}^{b} 2^{-1}2^i = \sum_{i=a}^{b} 2^{i-1} \end{aligned} \tag{10.2}$$

Since the change is only in the interpretation of the data, no active component is required, the action is essentially free, and no information is added or lost, as shown in the example below, only the radix point is moved.

$3:$	$011.0_2 = 3.0_{10}$	$\text{MSB} = 2^2$	$\text{LSB} = 2^{-1}$
$3/2:$	$01.10_2 = 1.5_{10}$	$\text{MSB} = 2^1$	$\text{LSB} = 2^{-2}$
$3/4:$	$0.110_2 = 0.75_{10}$	$\text{MSB} = 2^0$	$\text{LSB} = 2^{-3}$

The position of the radix point does not matter to the hardware representation, although the designer has to keep track of it, and make sure to align the numbers for certain operations, such as addition.

The last step for the design of the post-processing component will be to change the representation of MSB to 2^0. Physically nothing is done but the fractional length will be increased by one (and one less bit will be needed on the integral side) and the representation of the number will therefore be divided by two, shown in Fig. 10.7

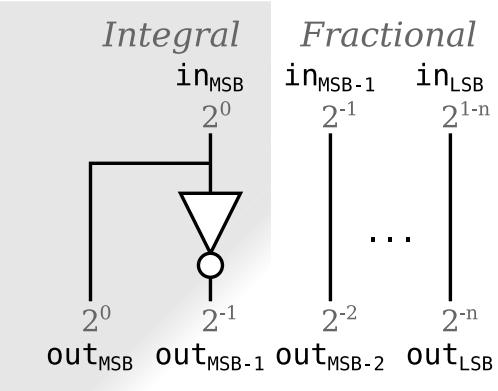


Figure 10.7: *Final design of post-processing.*

10.1.5 Subtraction and Shift in the Newton-Raphson Method Implementation

From (8.2) there is the subfunction $\frac{1}{2}(3 - v \times y_{i-1}^2)$. Given that the division is just a reinterpretation of the significance of the bits, the whole expression can be simply built as a single component. Figure 10.8 shows the basic design of the function using two's complement. For this reason, an extra adder is created on the left, to cater to the extension required for the bit with a negative sign.

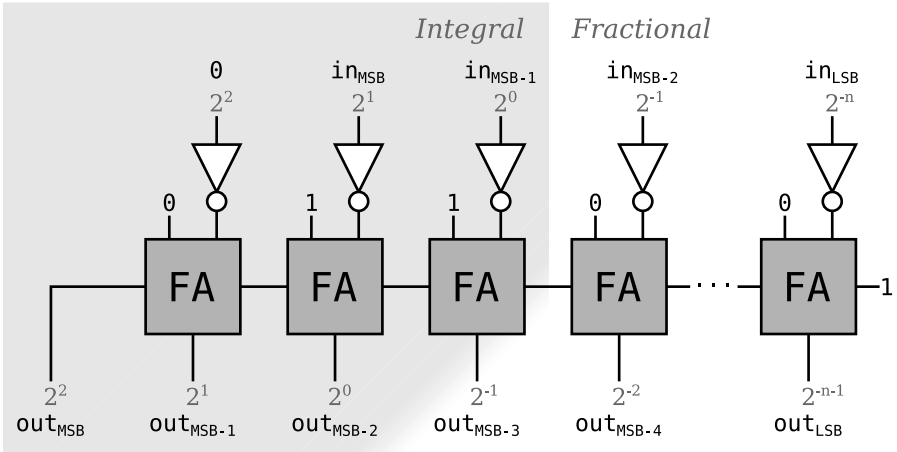


Figure 10.8: *Basic design of subtraction and shift for Newton Raphson without optimizations*

The first step for optimizing the hardware will be to replace the FA which have one input set to constant zero, with Half Adders (HA).

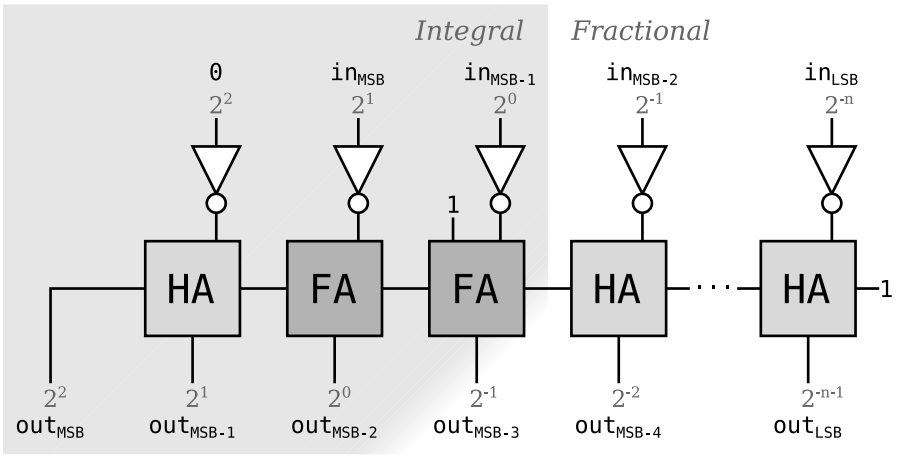


Figure 10.9: *Subtraction and shift for Newton Raphson with full adders replaced with half adders where possible*

The result from this component will always be less than 1 when using the range $v \in [1, 4)$. This means that the result will only depend on one of the integral bits in the input, and the left HA and FA can be removed together with supporting components and data paths. The shift operation is done in the same way as described in Section 10.1.4 for post-processing. The final design for the combined subtraction and shift component can be seen in Fig. 10.10.

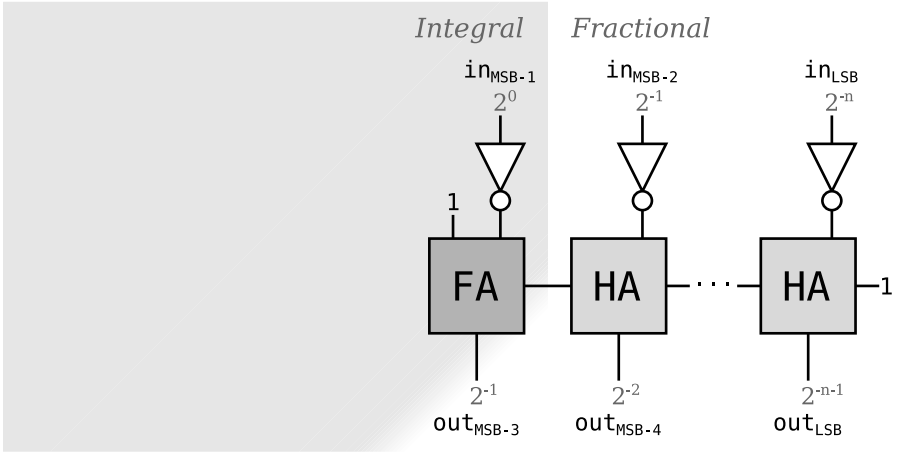


Figure 10.10: *Subtraction and shift for Newton Raphson with unnecessary components removed*

The final design to be implemented is not optimized to its full potential. There are still changes that can be made for the last adder to the left, given that one of the inputs are constant. However, the Design Compiler tool is optimizing the VHDL code as well as the selection of logic gates (standard cells) in order to meet timing and area constraints. Design Compiler will use algebraic and boolean techniques to optimize, the tool will also do local gate-level optimization [15, Chapters 1.4–1.9].

10.2 Synthesis

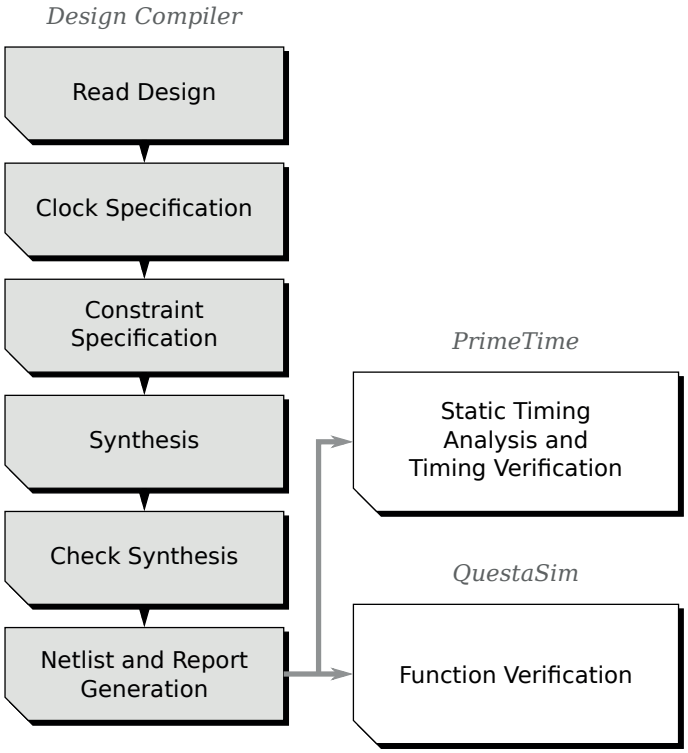


Figure 10.11: *Synthesis flow.*

All synthesis have been done in Design Compiler from Synopsys® inc. Verification of functionality has been done in QuestaSim, see Chapter 10.1, VHDL, as well as some initial simulation and verification on a behavioral level with GHDL and GTKWave during development. Verification of timing has been done in PrimeTime using Static Timing Analysis (STA). The synthesis instructions are written as a Tcl script and is executed automatically from a process control script written in Perl. The synthesis flow can be seen in Fig. 10.11.

In the beginning of synthesis, the library of the standard cell technology to be used is selected. Later on in the process, the VHDL files are read in the order of bottom-up for Design Compiler to analyze and elaborate. When this is done, Design Compiler has read the design.

The specification of the clock is done by setting clock period, clock uncertainty and clock transition. For this project a clock uncertainty of 2% has been chosen. The transition time was chosen after the result of

placement and routing depending of the constraint report. Transition time varies depending of library.

The last constraints to be set are input delay and output delay. These constraints are mainly used when the design is a part of a bigger design with delay from the clock for input and output signals. These constraints are set to a low value for this project.

When all the constraints are set, the synthesis can begin. The command *compile* in Design Compiler performs a gate-level synthesis and optimization of the design. The optimization depends on the *map_ effort* and *area_ effort* settings. The effort settings will be set depending of what is to be achieved. One goal is to find the fastest circuit and another is to see the area change over clock period, voltage and library. Given that the number of designs that are needed to be synthesized with different clocking, library, etc. combinations is big, only the combination of setting both map and area effort to *high* is used.

Another optimization method is to use the option *-ungroup all*, which allows the synthesis tool to break the interfaces between internal components, if it leads to a simplification. This could possibly both reduce the required area and increase the maximum implementable speed of the device.

From the synthesis step, a gate-level netlist file is generated as a Verilog file. A Synopsys Design Constraints file (SDC) and DDC file is also generated and will be used in other steps. For a more detailed view, the script for synthesis can be found in Appendix A.6.1.

PrimeTime is used to verify timing and to generate a Standard Delay Format file (SDF) from the netlist created in synthesis. Figure 10.12 show the flow of STA in PrimeTime.

To simplify the workflow of STA, specification of timing constraint and timing expectation can be neglected if the DDC file from synthesis is used. DDC is an internal database format by Synopsys and contains both gate-level netlist and design constraints. A timing report is created from reading the DDC. The timing report from PrimeTime will give a more accurate report than the timing report from Design Compiler. If the timing report from PrimeTime shows a violation in slack time the synthesis needs to be redone with a clock period high enough to pass time exception. To save time, the increase of clock period will be done before the next step of placement and routing instead of re-synthesis of the VHDL code. This was deemed a reasonable tradeoff, considering that re-synthesis for a slower clock did not guarantee passing the tests performed by PrimeTime.

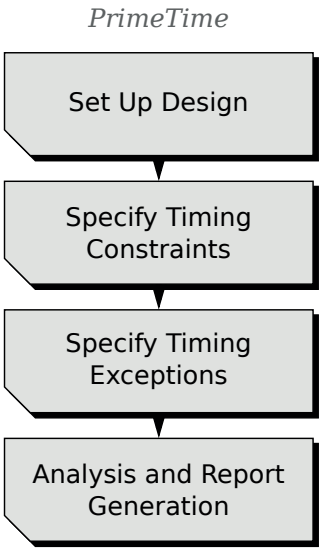


Figure 10.12: *STAs flow.*

The method of how to increase the clock period while retaining clock uncertainty of 2% will be explained in Chapter 10.3, Placement and Routing.

From the DDC file, an SDF file of the netlist can be generated for simulation and time verification in Questasim. If PrimeTime's timing report gives a slack time violation, it is important to add this to the clock period when simulating the netlist generated from synthesis with the SDF file. Simulation in QuestaSim with SDF will verify function and timing verification through this simulation. This kind of simulation was done in the beginning of the project, to verify that the synthesis was working. Later on in the project, this step is done only after placement and routing to verify function and timing.

The script for STA can be found in Appendix A.6.2.

10.3 Placement and Routing

Placement and routing is the final step towards a physical layout of the Integrated Circuit (IC) construction.

All placement and routing has been done in Encounter® Digital Implementation System v10-10 from Cadence®. Verification of functionality have been done i QuestaSim, see Section 10.1, VHDL. The workflow is described in a Tcl script and is executed automatically from

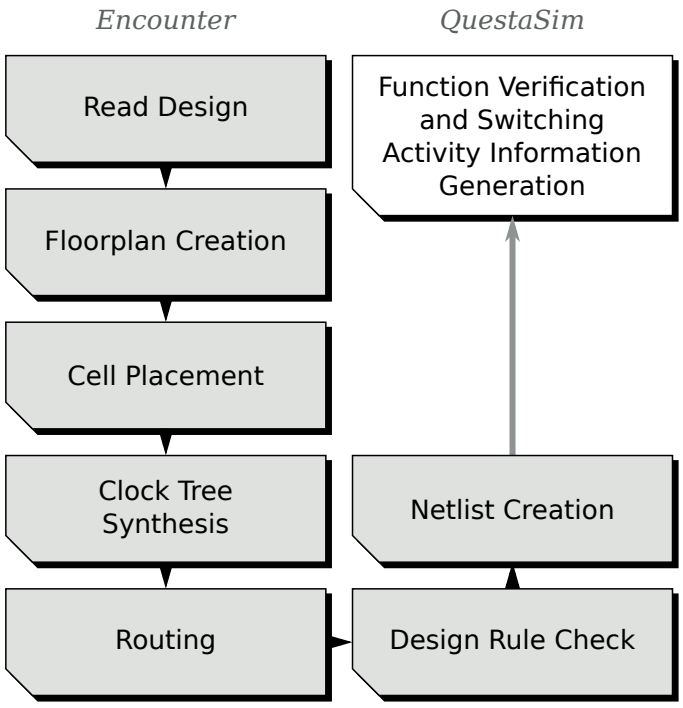


Figure 10.13: *Placement and routing flow.*

a process control script written in Perl. The placement and routing flow can be seen in Fig. 10.13.

At setup, Encounter needs to read the design netlist and the corresponding technology library. Encounter will read the Verilog gate-level netlist and the SDC file generated during the synthesis step. The SDC file contains information about clock and other constraints needed for placement and routing. The settings corresponding the actions during a manual project import of the design is loaded from a separated file.

After the design is read the floor-planning will be set. During floor-planning the area and cell density of the chip is set. Since the number of and type of gates are given by the netlist, and their area are described by the technology library, the overall area requirements can be calculated from this. Power rings are added as well as the power grid. In this project there is no need for I/O pads, since the circuit is realized and tested as an internal IP core to be included in a bigger chip, rather than as a standalone circuit.

Next step is to place the standard cells described by the netlist, which are placed on the chip automatically. So far no internal connection or clock network have been added.

To create the clock network a clock-tree synthesis is done. The clock tree is generated by using information from the SDC file from synthesis, and saves some of the information (timing information, what buffers are available, etc.) to a clock-tree synthesis technology file (CTSTCH). Clock-tree synthesis will place clock buffers and place the clock network in a way to meet timing constraints from STA.

The STA gives a more accurate timing report than Design Compiler and this mean that the clock period may need to be increased equal to the setup violations reported from PrimeTime. Before the SDF file is used it needs to be modified to increase the clock period but still allow for a clock uncertainty of 2%. The following equation is used for calculating the new clock period with a clock uncertainty of 2%, shown in (10.3).

$$t_{new} = t_{old} + \frac{t_{slack}}{0.98} \tag{10.3}$$

t_{new} is the new clock period with 2% clock uncertainty added, t_{old} is the old clock period with 2% clock uncertainty added, t_{slack} is a positive number of how much slack needed to be added. This is done in the main process control script between the STA step and the PNR step.

During the routing process, the cells get connected to the power grid. This process also makes sure the cells get their internal connections, as well as tries to find the optimal routing paths. This process is done more than one time with incremental improvements, in order to be able to meet the time constraint. When routing is finished, all locations on the die will be filled with filler cells.

When all standard cells are placed and routed a Design Rule Check (DRC) is done. The DRC will for instance check the specific rules for the semiconductors used in the chosen library [16, pp. 27, 208]. All reports generated by Encounter are read programmatically and parsed to check for pass or fail conditions.

A netlist in Verilog is created of the final design as well as an SDF file and SPEF file. SPEF stands for Standard Parasitic Exchange Format and contains all parasitic data of the IC construction [17, Chapter 9].

To verify the function and timing, the netlist from Encounter is simulated in QuestaSim. The script for placement and routing can be found in Appendix A.6.3.

10.4 Power Analysis

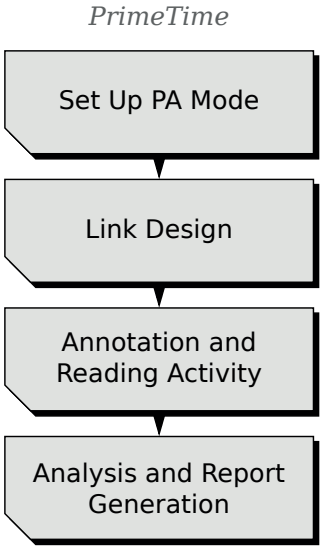


Figure 10.14: *Power analysis flow.*

All power analysis has been done in PrimeTime® PX from Synopsys®. The workflow is done in a Tcl script and is executed automatically from a process control script written in Perl. The power analysis workflow can be seen in Fig. 10.14.

In order to analyze power consumption of the design generated in the placement and routing step, a file containing signal switching activity (Value Change Dump (VCD)) is required. The VCD is generated in QuestaSim by simulating the verilog file created in Encounter in a testbench using the SDF back-annotated file. A simulation is done for all valid input values v and a switching activity is saved in the VCD file. During this simulation the function and time verification is simultaneously done, due to the use of the SDF.

Besides the VCD there are three other files needed for a power analysis of the IC construction. The files needed are generated in the previous step of placement and routing, except for the VCD file which is generated in QuestaSim. The files are as follows:

- Netlist: Verilog file, gate-level netlist

- Timing Constraint: SDC file

- Parasitics: SPEF file

- Switching activity: VCD file

To get a more accurate power consumption a time-based power analysis is performed. The time-based power analysis will report both peak power and average power compared to average-power analysis which only covers the average power consumption.

The full script describing the power analysis can be seen in Appendix A.6.4.

10.5 Main Process Control Script

The hardware workflow tool control scripts evolved from a desire to automate interaction with the hardware workflow tools, since producing results required days of computation, with input needed every few hours or so. The scripts are written in Perl, with a common set of functions that revolve around

- Analyzing already existing data and report files, to see if the last step for a particular constraint combination has failed or succeeded, and if any additional steps are needed.

- Setting up and cleaning up the work environment for the tools, creating and removing directories, instantiating templates.

- Starting the tools from shell script stubs (tcsh) and point them to what Tcl scripts to use, what files to read and write, etc.

During the work, three specialized scripts have emerged with different duties.

The first one has the task of trying to find the fastest clock speed that yields a successful synthesization. Initially this was done by hand, and iteratively, but has evolved to something more like a binary tree search, beginning with a large clock period, and for each synthesization either adding or removing an amount from the clock period, halving the amount to add or remove in each iteration.

The second script scans selected directories for signs of partially complete work, and continues e.g. with place and route operations if synthesis and STA were successful.

The third script makes sorted lists of which implementations have been successful, so that the only differentiating property in each list is the clock period. Then it calculates the average clock period of each pair of adjacent implementations and try to pull that configuration through the workflow as well. This in order to create more datapoints so that trends can be better observed.

Chapter 11

Results

11.1 Data Flow Behavior

11.1.1 Harmonized Parabolic Synthesis

Results from the data flow simulations of some of the HPS implementations are shown here.

The plot in Fig. 11.1 shows the distribution of errors and values for a 32 interval HPS implementation, where the data paths are not truncated. (Actually, due to the simulation software implementation, this means that every value is forced to fit into a 32 bit signed integer where the three most significant bits are treated as integral, and the other 29 as fractional.)

The ringing like phenomenon shown in the left subplot is due to the polynomials in the second sub-function being calculated to equal the $f_{help}(x)$ function at the beginning, middle, and end of the interval, which in the case of the inverse square root function leads to an overshoot and undershoot between these points.

Figure 11.2 displays the change introduced by truncating the data paths. The error distribution is visibly more bell shaped, but there is a significant mean error which is not desirable. To remedy this, the constants in the look-up table are modified, and chosen in order to minimize the absolute mean error per interval, and with the hard limit that the worst error must remain within the specified constraints. The result of this can be seen in Fig. 11.3. Even with the mean error brought down, there is a bit of skewness. It is possible to select constants not only based on mean error, but also based on skewness: Figure 11.4 shows the result of primarily selecting constants based on minimizing the absolute

mean error per interval, but in case of multiple sets of constants reaching an absolute mean error 5×10^{-8} (somewhat arbitrarily chosen) for a certain interval, the set that yields the least skew is selected.

The corresponding plot with constants selected for absolute mean error and skew, for the 512 interval implementation of the HPS algorithm, is shown in Fig. 11.5, the error distribution is flatter and more widespread. Plots for implementations with between 32 and 512 intervals show a gradual increase in flatness as the number of intervals grow. These can be viewed in Appendix A.2.2.

It is important to present the comparisions of the different implementations in a way that gives a fair view of the properties even at a cursory glance, for this reason, the error distribution graphs are all done to the same scale. This does however have the effect that the highest bar in the error histogram for the 32 interval implementation with no truncation. The tallest bar in Fig. 11.1, does not fit in the plot when scaled in a way that is suitable for comparing with other implementations, and a text label has been supplied to indicate the value. This was chosen as preferable to either not plot everything to the same scale, or changing the scale of every plot with resulting loss of detail in all other plots.

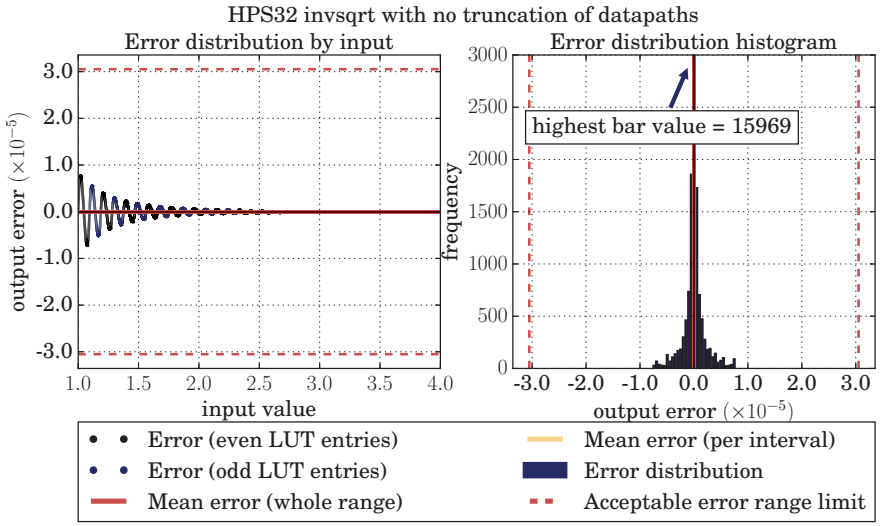


Figure 11.1: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 32 intervals and without truncation of data paths*

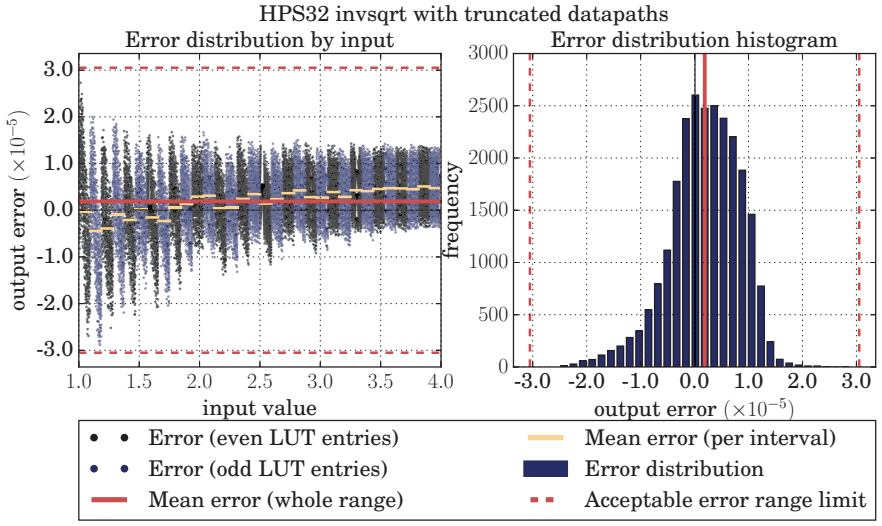


Figure 11.2: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 32 intervals and with truncated data paths.*

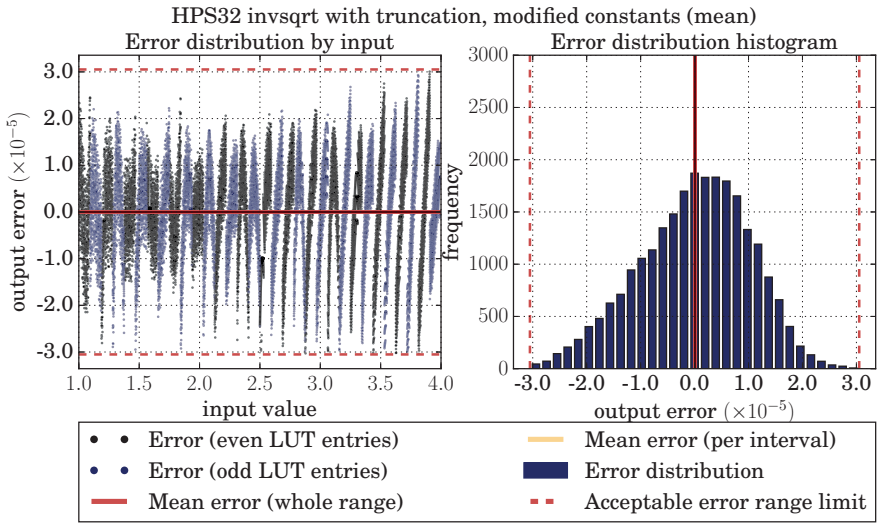


Figure 11.3: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 32 intervals, with truncated data paths, and constants modified to minimize the absolute mean error per interval.*

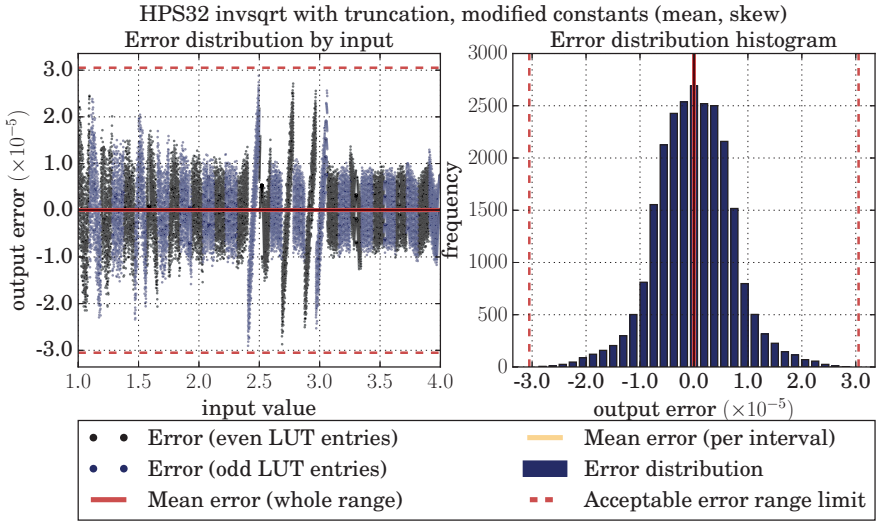


Figure 11.4: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 32 intervals, with truncated data paths, and constants modified to minimize per interval absolute mean error as well as per interval absolute skew.*

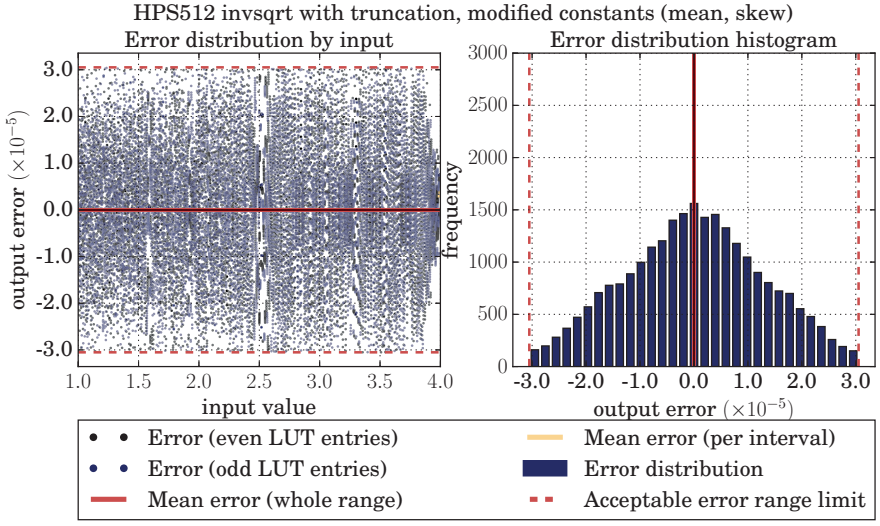


Figure 11.5: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 512 intervals, with truncated data paths, and constants modified to minimize per interval absolute mean error as well as per interval absolute skew.*

11.1.2 The Newton Raphson Method

The results from the simulation for NR 1 iteration can be seen in Fig. 11.6. Truncation used for the simulation can be seen in Table 9.5. The look-up table used for the simulation is visible in Appendix A.1.

In the leftmost plot in Fig. 11.6 the error for each value in v is shown, and in the plot to the right, the error distribution is represented in the form of a histogram, which gives a more condensed summary of the error behaviour. There is no apparent skewing, although there is a bit of a mean error, and especially looking at the per-interval mean error. Whether this is a problem or not would depend on the requirements of the application, they are however all within the constraints specified for the work presented in this report.

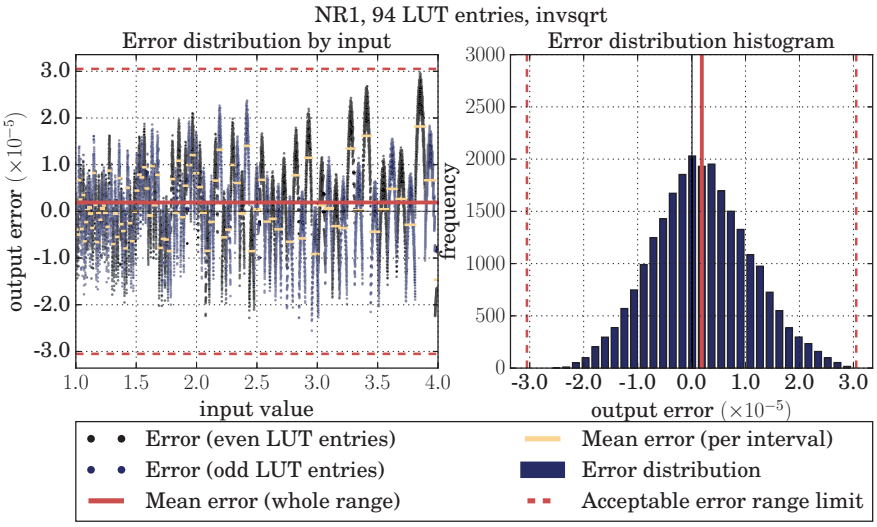


Figure 11.6: *Input to output error plot and error distribution histogram for the inverse square root implemented with NR using 1 iteration and with 94 intervals.*

The results from the simulation for NR 2 iterations can be seen in Fig. 11.7. Truncation used for the simulation can be seen in Chapter 9.2 in Table 9.5. The look-up table used for the simulation is visible in Appendix A.1.

In the leftmost plot in Fig. 11.7 the error for each value in v is shown, and in the plot to the right, the error distribution is represented in the form of a histogram, which gives a more condensed summary of the error behaviour.

Compared with the implementation of Newton-Raphson using only one iteration, the error distribution is improved in just about every way including the standard deviation and error range.

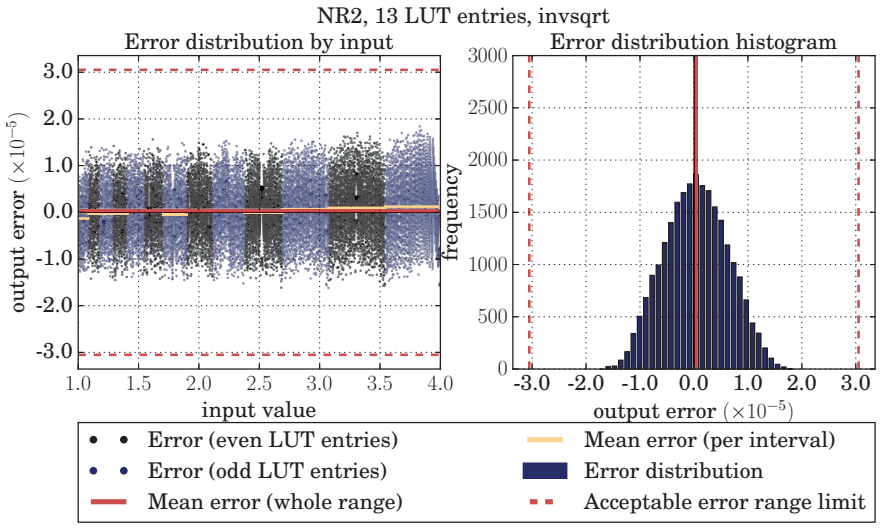


Figure 11.7: *Input to output error plot and error distribution histogram for the inverse square root implemented with NR using 2 iterations and with 14 intervals.*

11.1.3 Comparision of Statistics

The result from the error behavior analysis explained in Chapter 5 is shown in Table 11.1 and Table 11.2.

For the Newton-Raphson Method, the results are shown for Newton-Raphson with one iteration(NR1) and for Newton-Raphson with two iterations(NR2).

For Harmonized Parabolic Synthesis, the results are shown for Harmonized Parabolic Synthesis with 32 intervals(HPS32) and for Harmonized Parabolic Synthesis with 512 intervals(HPS512). The numbers presented for HPS here are all for the implementation with truncation and constants that have been modified to keep both mean error and skewness low.

Implementation	Max error	Mean error	Median error	SD	RMS	Skew
HPS32	2.9032e-05	1.2857e-08	2.9354e-08	6.9841e-06	6.9841e-06	−1.9069e-02
HPS512	3.0508e-05	1.5962e-08	9.2308e-08	1.2863e-05	1.2863e-05	−1.6186e-02
NR1	2.9733e-05	1.9028e-06	1.6665e-06	8.9800e-06	9.1794e-06	1.1874e-01
NR2	1.8403e-05	3.8414e-07	3.6850e-07	5.8901e-06	5.9026e-06	1.8627e-02

Table 11.1: *Error metrics from simulations.*

Implementation	Max error	Mean error	Median error
HPS32	15.07	26.21	25.02
HPS512	15.00	25.90	23.37
NR1	15.04	19.00	19.19
NR2	15.73	21.31	21.37

Table 11.2: *Error metrics from simulations in bits.*

11.2 Hardware Behavior

The hardware design and simulation from which the results for hardware behavior come, is described in Chapter 10. The main CMOS technology libraries for which data is presented in this work are ST Microelectronics 65nm General Purpose Standard V_T (GPSVT) and Low Power High V_T (LPHVT). Additional libraries for which hardware simulation work has been done are General Purpose Low V_T (GPLVT), General Purpose High V_T (GPHVT), Low Power Low V_T (LPLVT) and Low Power Standard V_T (LPSVT). Presenting *all* variants of the algorithms, implemented using *all* the technology libraries, would take up too much space, or make the plots too cluttered, for little or no added benefit. The sections in this chapter will will discuss the main implementations, while additional plots and tabular data can be found in Appendix A.3 and A.4.

For each technology library, two supply voltage variants have been investigated, and hardware layout and simulation have been done for numerous clock periods. However it is not possible to realize hardware designs for all combinations of implementations, technology libraries, clock periods etc. as each process step introduces more and more real-world imperfections and parasitics (as compared to dealing with ideal components), which may or may not make a particular combination pass or fail that particular step.

The chosen implementations to be presented for Harmonized Parabolic Synthesis are 32 and 512 intervals, see Fig. 9.2 and Table 9.1. The LUT constants for the implementations are the ones that give the error behaviours described in Fig. 11.4 and Fig. 11.5 respectively. For the Newton-Raphson Method, the implementations with 1 and 2 iterations are chosen for presentation, see Fig. 9.3, Fig. 9.4 and Table 9.5. These implementations are the same for which the data flow behavior have been described in Section 11.1. Additionally, data for the best results for each implementation, regardless of library, is presented.

The goal is to examine clock speed, area and power consumption. Section 11.2.1 will discuss the results for clock speed and area. Section 11.2.2 will present results for power consumption and energy consumption, simulated from the physical layout.

11.2.1 Clock Speed and Area

For all successfully synthesized implementations of HPS32, HPS512, NR1, and NR2, Fig. 11.8 (synthesis with the LPHVT technology library) and Fig. 11.9 (synthesis using the GPSVT technology library) show how the clock period affect the cell area required for a successful synthesis.

Furthermore, Fig. 11.10 (LPHVT) and Fig. 11.11 (GPSVT) plot the same properties, but only for implementations that have successfully completed all steps in the process, from synthesis, through PNR and finally simulation and power estimation. The clock period and area represented in these figures are the values reported at the end of the workflow, meaning that the clock period for a given implementation attempt will likely be slightly higher in the plots and tables presenting post-PNR data, compared to synthesis data.

In the figures it can be seen that all implementations have a point where decreasing clock period greatly increases the area requirement. The location of this "knee" seems to depend partially upon the implementation, as well as the V_{DD} of the library, but overwhelmingly on the selected technology library.

There is also a lower bound to the required area that is about the same for both technology libraries used. In this respect the HPS32 perform a little bit better than the NR1 implementation for lowest area requirements, NR2 requires the most area, and the area requirement for HPS512 is slightly higher than NR1, although the knee seems to possibly appear a little earlier, or introduce a little steeper change for NR1.

Comparing the clock speeds after synthesis, it seems that HPS512 has a slight upper hand compared to the other implementations, although when doing the same comparision for implementations that have passed all steps in the workflow, it is instead HPS32 that has the most favourable clock speed.

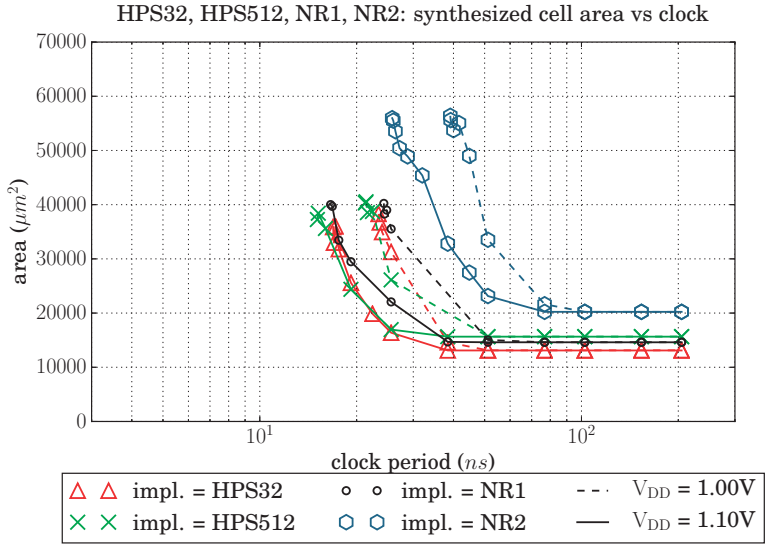


Figure 11.8: *Clock period vs area for implementations that have been successfully synthesised using the LPHVT technology library.*

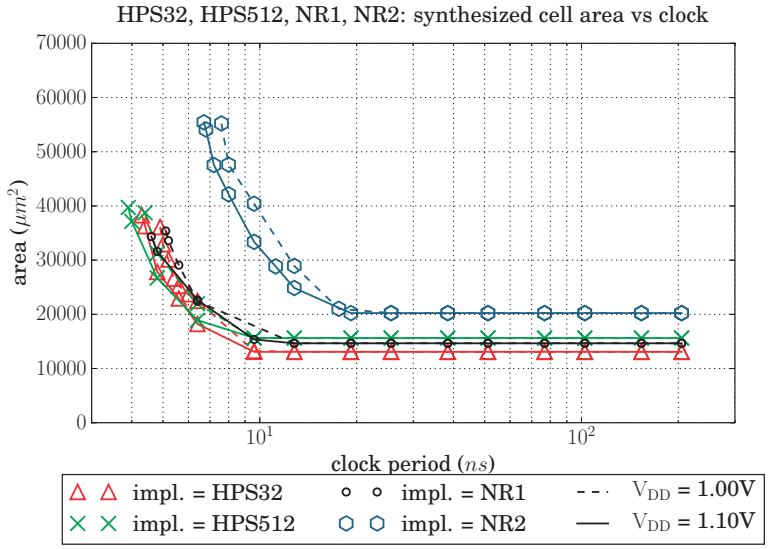


Figure 11.9: *Clock period vs area for implementations that have been successfully synthesised using the GPSVT technology library.*

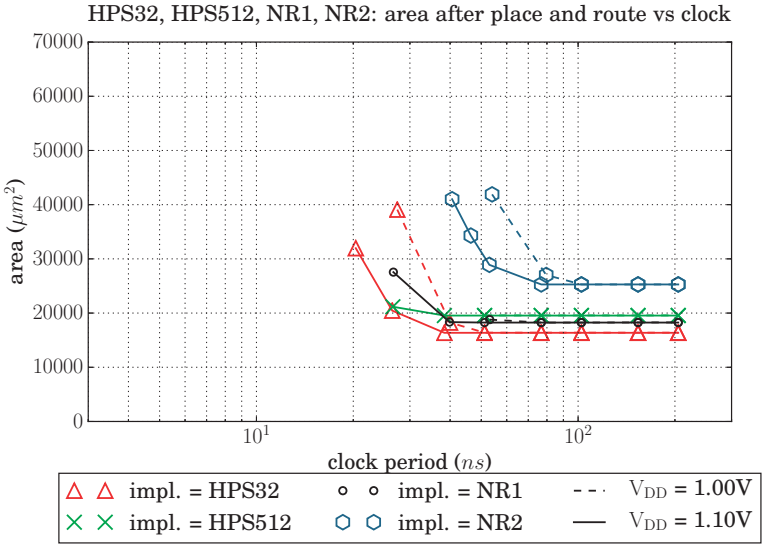


Figure 11.10: *Clock period vs area for implementations using the LPHVT technology library and where the implementation have successfully completed all simulation steps.*

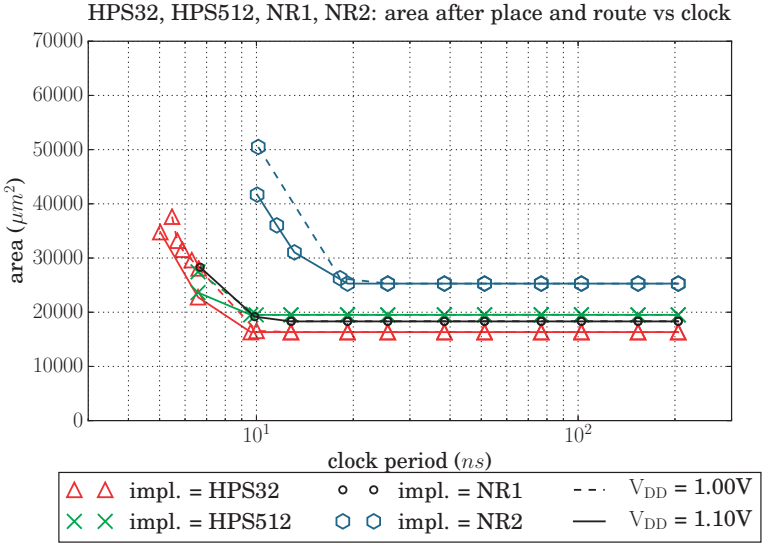


Figure 11.11: *Clock period vs area for implementations using the GPSVT technology library and where the implementation have successfully completed all simulation steps.*

Table 11.3 and Table 11.4 list data for the fastest clock speeds (synthesized only, and successfully simulated after PNR respectively), and Table 11.5 lists the implementations with the smallest area requirements. In the tables, the implementation that has the fastest clock or smallest area (depending on table) among the listed, is distinguished with bold text.

More implementations of HPS have been examined than is being plotted here. Implementations with 64, 128 and 256 intervals have also been taken through the full set of steps in the workflow, from synthesis to power estimation. Additional technology libraries have been used as well for both HPS and NR, and for all implementation variants. Thus, in order to try to find the fastest, the smallest, the most energy conservative instance of each algorithm, and to investigate if and how the algorithms would differ in what benefits and drawbacks the properties of the libraries would offer.

For a more exhaustive listing of timing and area requirements, with all implementations and libraries compared, see Appendix A.4.1. Table A.3 lists the best synthesised clocks for each implementation and library and Table A.4 lists the properties of the fastest implementations that have successfully completed all simulation steps. For the corresponding listing of smallest area requirement after PNR, see Table A.5.

Impl.	Library	V_{DD}	Synth Clock (ns)
HPS32	GPSVT	1.10V	4.3
HPS512	GPSVT	1.10V	3.9
NR1	GPSVT	1.10V	4.6
NR2	GPSVT	1.10V	6.7
HPS32	LPHVT	1.10V	16.9
HPS512	LPHVT	1.10V	15.1
NR1	LPHVT	1.10V	16.6
NR2	LPHVT	1.10V	25.8

Table 11.3: *Listing of the lowest synthesised clock period for the implementations plotted in Chapter 11 Results.*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPSVT	1.10V	5.024	34779	21.9	110.026
HPS512	GPSVT	1.10V	6.584	23624	8.97	59.058
NR1	GPSVT	1.10V	6.686	28288	7.21	48.206
NR2	GPSVT	1.10V	10.049	41696	19.2	192.941
HPS32	LPHVT	1.10V	20.363	31965	3.26	66.383
HPS512	LPHVT	1.10V	26.559	21164	1.16	30.808
NR1	LPHVT	1.10V	26.651	27551	0.827	22.040
NR2	LPHVT	1.10V	40.584	41001	2.56	103.895

Table 11.4: *Listing of the lowest clock period after PNR for the implementations plotted in Chapter 11 Results.*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPSVT	1.10V	9.6	16333	3.81	36.576
HPS512	GPSVT	1.10V	9.6	19512	3.73	35.808
NR1	GPSVT	1.10V	12.8	18306	1.6	20.48
NR2	GPSVT	1.10V	19.2	25281	4.34	83.328
HPS32	LPHVT	1.10V	38.4	16358	0.942	36.173
HPS512	LPHVT	1.10V	38.4	19540	0.784	30.106
NR1	LPHVT	1.10V	51.2	18246	0.303	15.514
NR2	LPHVT	1.10V	76.8	25281	0.741	56.909

Table 11.5: *Listing of the lowest area requirement after PNR for the implementations plotted in Chapter 11 Results.*

11.2.2 Power and Energy

The power consumption in a CMOS component consists of two parts, static power (P_{static}) and dynamic power ($P_{dynamic}$). The static power depends of the leakage current and supply voltage, while dynamic power depends on the switching power (P_{switch}) and short-circuit power (P_{short}). This section will detail the relations. Total CMOS power consumption is as shown in (11.1) [18, pp. 4]:

$$P_{CMOS} = P_{static} + P_{dynamic} \tag{11.1}$$

The static power is the leakage power when the circuit is powered on but inactive. Ideally the leakage current should be zero, however as PMOS and NMOS devices in a complementary MOS transistor pair are never on simultaneously in the steady-state operation there is always a small leakage current between source or drain and the substrate. The equation for static power is shown in (11.2) [19, pp. 223].

$$P_{static} = V_{DD}\ I_{leak} \quad \text{where } V_{DD} \text{ is the supply voltage and } I_{leak} \text{ is leakage current.} \tag{11.2}$$

As mentioned in the beginning of this section, the dynamic power consists of switching power and short-circuit power shown in (11.3)[20].

$$P_{dynamic} = P_{switch} + P_{short} \tag{11.3}$$

Every time a switch cycle occurs (0-1-0) in the circuit the CMOS will charge and discharge its load capacitances. The amount of energy that will be lost depends on frequency, supply voltage and load capacitance. Switching power is shown in (11.4) [19, pp. 214–216].

$$P_{switch} = \alpha\, C_L\, V_{DD}^2\, f \quad \text{where } \alpha \text{ is the switching activity, } C_L \text{ is load capacitance and } f \text{ is the frequency} \tag{11.4}$$

During switching there is a short period of time when both the PMOS side and the NMOS side are open. The reason for that is due to the transition between on and off or vice versa is not instantaneous, the rise- and fall times are not zero. This condition will create a direct path between V_{DD} and ground, causing current to leak as shown in (11.5). [20][19, pp. 220].

$$P_{short} = V_{DD}\, I_{SC} \quad \text{where } I_{SC} \text{ is the short-circuit current.} \tag{11.5}$$

For electrical portable units it is more relevant to calculate the energy consumption instead of power consumption. Electrical mobile units use battery which means that it runs on limited available energy stored in the battery. Low energy consumption will give a longer operating time on a given limited battery charge [21, pp. 271].

Energy consumption is measured as the amount of energy (E) consumed by the circuit compared to power consumption that is measured as the amount of energy consumed during a certain amount of time.

Energy consumption per clock period can be calculated if the power consumption and clock period are already known, the calculation will be as shown in (11.6) [21].

$$E = P\,t \quad \text{were } P \text{ is average power consumption during a clock cycle, and } t \text{ is the clock period.} \tag{11.6}$$

The total power consumption is shown in Fig. 11.13 for the GPSVT technology library, and Fig. 11.12 for the LPHVT technology library. Looking at the power consumption of the algorithm implementations, the ranking, compared to area requirements, have changed somewhat. NR2 has a higher power consumption, likely from having a bigger circuit with more active components, while NR1 consumes the least power. The two implementations of HPS have similar power consumption, almost overlapping when implemented with the GPSVT technology library, and with the 512 interval implementation slightly lower when using the LPHVT technology library. In general, a lower V_{DD} can be seen to result in a lower power consumption, which is to be expected, as shown in (11.2), (11.4) and (11.5). The validity of this comparison does however diminish at higher clock speeds after the "knee".

LPHVT, being a technology library targeted towards low power applications generally outperforms the GPSVT library regarding power consumption, when comparing implementations with otherwise equal features.

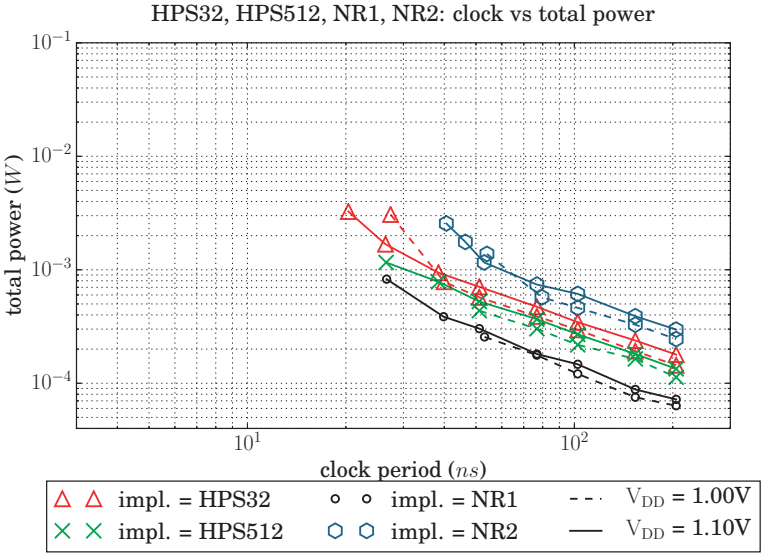


Figure 11.12: *Power consumption for implementations using the LPHVT technology library.*

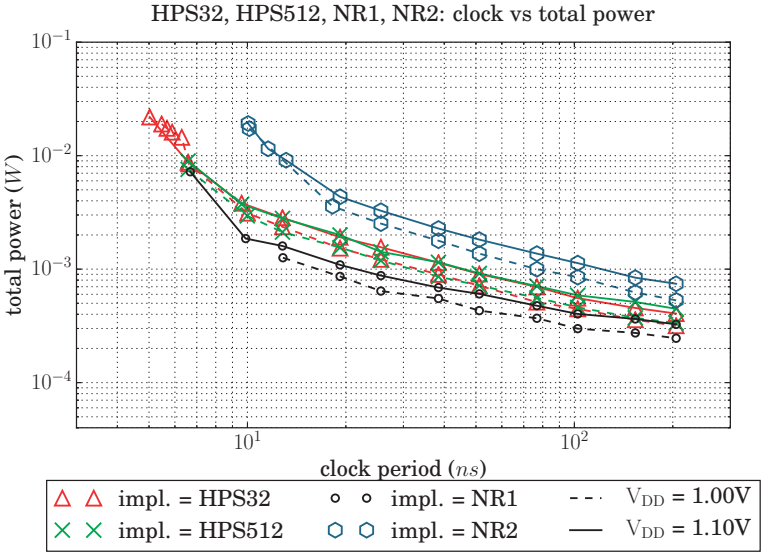


Figure 11.13: *Power consumption for implementations using the GPSVT technology library.*

It is also interesting to calculate the energy consumed during a single calculation, since high power consumption during a short time may use the same amount of energy as low power consumption during a longer duration. Referencing (11.4), the amount of energy consumed as switching losses should be independent from the clock frequency, since $E = Pt$ and $f = 1/t$. On the other hand, static power consumption is independent from timing, which means that the energy consumed as static losses in the device should increase as the clock period increases. At least if the assumption can be made that the device is completely powered off when not actively calculating anything, and that the change between active and powered off is reasonably instantaneous.

The total energy consumption calculated this way is shown by Fig. 11.15 (GPSVT) and Fig. 11.14 (LPHVT) which show the energy required for the calculation of one single value. In the plot for the GPSVT technology library, there exist approximate local minimums for all combinations of algorithm, V_{DD}, etc. regarding energy consumption. The LPHVT library plot does not have as pronounced minimums, the effect could possibly be shown better if additional implementations are tried with even slower clocks. To the left of these minimums, the energy consumption starts to increase at the same points where area starts to increase in the previous figures, and the tool chain struggles to meet the timing requirements. To the right of the minimum, the increased energy usage is a direct result of a fairly constant static power consumption integrated over longer and longer time intervals as clock period increases.

Comparing the libraries, it seems that for a lot of LPHVT technology implementations, there exists a GPSVT technology implementation with the same energy requirement, but a faster clock speed. The reason for this can be found when examining static and dynamic power consumption separately. Graphs for these properties are presented in Appendix A.5 and show that the main difference between the implementations using the two libraries, lie in their *static* power consumption, while the *dynamic* power consumption were similar for similar clock speeds. This has the effect that the energy consumption caused by dynamic energy losses is relatively equal between the libraries and independent of implementation clock speed (except for when the tool chain has trouble getting implementations to pass the constraint checks), whereas the energy consumption from static losses decrease as clock speed increases.

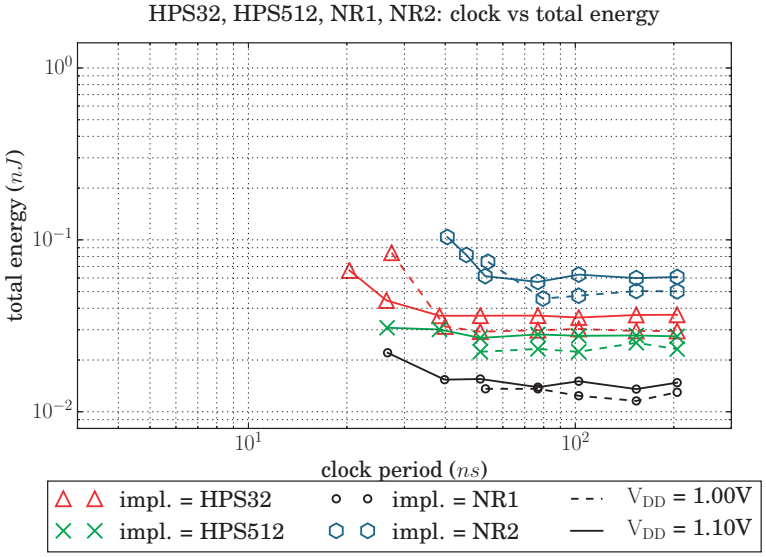


Figure 11.14: *Total energy used for one calculation for implementations using the LPHVT technology library.*

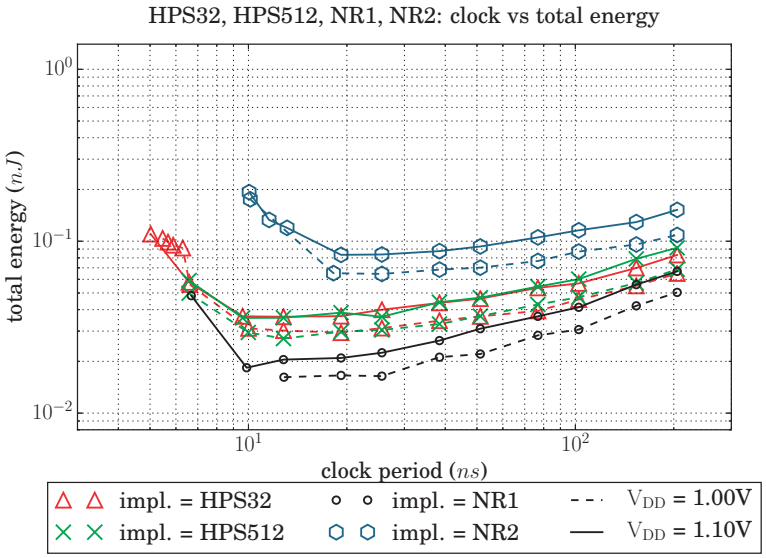


Figure 11.15: *Total energy used for one calculation for implementations using the GPSVT technology library.*

From Fig. 11.12 and Fig. 11.13 a table with the lowest total power consumption for each implementation is shown in Table 11.6, for the lowest energy consumption per clock period, see Table 11.7. The implementation that has lowest power consumption or lowest energy consumption depending on table, is distinguished with bold text. As with the clock and area results, more comprehensive results are presented in Appendix A.4.2, where Table A.6 lists the lowest total power consumption, and Table A.7 lists the lowest total energy consumption for one calculation.

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPSVT	1.00V	204.8	16333	0.317	64.922
HPS512	GPSVT	1.00V	204.8	19512	0.335	68.608
NR1	GPSVT	1.00V	204.8	18333	0.246	50.381
NR2	GPSVT	1.00V	204.8	25281	0.531	108.749
HPS32	LPHVT	1.00V	204.8	16384	0.144	29.491
HPS512	LPHVT	1.00V	204.8	19568	0.113	23.142
NR1	LPHVT	1.00V	204.8	18272	0.063	12.984
NR2	LPHVT	1.00V	204.8	25281	0.245	50.176

Table 11.6: *Listing of lowest total power consumption for the implementations plotted in Chapter 11 Results.*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPSVT	1.00V	19.2	16333	1.53	29.376
HPS512	GPSVT	1.00V	12.8	19512	2.12	27.136
NR1	GPSVT	1.00V	12.841	18333	1.26	16.180
NR2	GPSVT	1.00V	25.6	25281	2.52	64.512
HPS32	LPHVT	1.00V	51.2	16384	0.571	29.235
HPS512	LPHVT	1.00V	51.2	19568	0.436	22.323
NR1	LPHVT	1.00V	153.6	18272	0.075	11.566
NR2	LPHVT	1.00V	79.586	27027	0.572	45.523

Table 11.7: *Listing of the lowest total energy consumption per calculation for the implementations plotted in Chapter 11 Results.*

Chapter 12

Conclusions

Choosing an implementation to use depends on how the various properties rank in desirability, there is no clear candidate that is superior in all respects. We have shown in Section 11.1, that the constants for HPS can be selected to target specific desired properties of the error distribution. For this reason HPS shows very good results in this regard. It does seem that a lower interval count for HPS keeps the error standard deviation down. The standard deviation was not an optimization goal when selecting constants however, the results say very little about how interval count would effect it, were standard deviation to be specifically targeted in the selection of constants.

NR2 has a smaller range over which the error is spread. There is, in other words, room for modifications that would worsen the standard deviation and maximum error, but provide improvements in other respects. On the other hand, NR1 has the largest mean error and skew of the algorithm implementations and is from an error behaviour point of view probably the least desirable implementation.

Both implementations of HPS outperform NR2 in all properties of speed, area, power and energy consumption. The "knee" for NR2 begins at higher clock periods compared to the implementation of HPS, both when looking at synthesis only, and for variants that have passed PNR. If a good error behavior is critical, the implementations of the Harmonized Parabolic Synthesis is a good choice compared to the Newton-Raphson with two unrolled iterations, due to the properties of the hardware.

If the mean error shown by NR1 is considered good enough, the tables are somewhat turned. NR1 has a lot to offer regarding power and energy consumption. The area requirements of NR1 is comparable to the HPS implementations, although HPS32 holds a constant slight edge.

The difference in total power consumption, and total energy consumption is more significant, especially when using the LPHVT technology library. Considering these factors, a recommendation would depend on how much weight is put on each of energy consumption, area requirements, and error behaviour.

When it comes to which implementation is fastest, it is difficult to make an absolute statement, different runs of the synthesis and place and route workflows have yielded slightly different results at different times. It is however clear and consistent that NR2 is the slower out of the implementations. It is more difficult comparing NR1 to the HPS implementations.

The difference between using technology libraries with a V_{DD} of 1.00V or 1.10V is increasingly pronounced as clock periods decrease. The increase in area requirements as clock periods decrease always occurs earlier for 1.00V than for 1.10V. This means that a voltage of 1.10V is to prefer to receive lower area requirements for low clock periods. For higher clock periods there is no notable differences in area requirements. The downside of choosing 1.10V as supply voltage is the increased power consumption and required energy per clock cycle, which shows in the power and energy plots in Section 11.2.2. However, same plots show that for lower clock periods the 1.00V technology libraries will surpass their 1.10V equivalents in power and energy consumption for low clock periods. The clock periods when this effect starts to show correlate to the clock period for when the 'knee' in the area requirement plots start. For higher clock periods the supply voltage of 1.00V will be preferred because of its lower power and energy consumption, but for lower clock periods 1.10V may perform better.

12.1 Future Work

There are a number of things that can be improved upon. Neither the search space for truncation options nor the space for nudging the constants were exhausted. It is possible that there exist even better values than those presented in this report, although increasing the examined search spaces and thus doubling or more the necessary run time, seems to run into diminishing returns.

When deciding on a value for c_1 and the interval count I for the implementation of HPS, only the maximum error and simplicity of the hardware was considered, a more exhaustive examination could e.g. compare mean error, standard deviation, etc. With access to sufficient computational power c_1 could be made a design parameter to iterate

over, on par with the number of intervals, selection of technology libraries, clock speeds, etc. This would however make it more difficult to write specialized VHDL components, since they would possibly need to handle the general case.

A large difference between the implementations of HPS and NR, is that a lot of the calculations in the former are happening in parallel, while NR is largely operating serially, especially if implementing multiple iterations. This could mean that implementing both algorithms with a faster (but perhaps larger and/or more power hungry) multiplier type would benefit NR more in speed performance. On the other hand, HPS would then have more room for selecting a possibly slower multiplier implementation in pursuit of lower power consumption and/or lower area requirements.

For this report, a cell density of 80% was chosen during the placement and routing step. Different densities may have an effect on whether a given implementation at a given clock speed passes DRC checks, as well as power and energy consumption. Area is directly coupled to the density of the chip. Some experiments using different densities were done, but ultimately canceled. The time consumption was too big as well as not giving enough conclusive results at the time. With more computing power it may however be a worthwhile endeavor.

In all the implementations tested in this thesis work, the inverse function is generated as well by squaring the result of the inverse square root function, this extra block at the end will increase area and latency. It could be interesting to see the increase in performance when only the implementation of inverse square root function without inverse function is implemented. Although it is unlikely to shift the ranking of the implementations by their absolute performance numbers.

Bibliography

[1] P. Pouyan, E. Hertz and P. Nilsson, "A VLSI Implementation of Logarithmic and Exponential Functions Using a Novel Parabolic Synthesis Methodology Compared to the CORDIC Algorithm," *in the proc. of 20th European Conference on Circuit Theory and Design (ECCTD),* pp. 709–712, Linköping, Sweden, August, 2011.

[2] E. Hertz, *"Harmonized Parabolic Synthesis,"* Private communication, Lund, October, 2014.

[3] H. Anton and C. Rorres, *"Elementary Linear Algebra, Applications version,"* Hoboken, USA, John Wilay & Sons Inc, tenth edition, ISBN 9780470458211, 2010.

[4] E. Lengyel, *"Mathematics for 3D Game Programming and Computer Graphics,"* Boston, USA, Cengage Learning PTR, third edition, ISBN 9781435458864, 2012.

[5] J. Lai, *"Hardware Implementation of the Logarithm Function using Improved Parabolic Synthesis,"* Masters of Science Thesis, Department of Electrical and Information Technology, Lund University, Lund, Sweden, September 2, 2013.

[6] P.T.P. Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis," *in the proc. of 10th IEEE Symposium on Computer Arithmetic,* pp. 232–236, Grenoble, France, June, 1991.

[7] T.J. Ypma, "Historical Development of the Newton–Raphson Method," *SIAM Review,* vol. 37, no. 4, pp. 531–551, December, 1995.

[8] L. Råde and B. Westergren, *"Mathematics Handbook for Science and Engineering,"* Lund, Sweden, Studentlitteratur, fifth edition, ISBN 9144031092, 2004.

[9] A.A. Giunta and L.T. Watson, "A Comparison of Approximation Modeling Techniques: Polynomial Versus Interpolating Models," *American Institute of Aeronautics and Astronautics,* September, 1998.

[10] H. Cramér, *"Mathematical Methods of Statistics,"* Princeton, USA, Princeton University Press, vol. 9, ISBN 9780691005478, 1945.

[11] B.L. Danielson, "Optical Time-Domain Reflectometer Specifications and Performance Testing," *Appl. Opt.,* vol. 24, no. 15, pp. 2313–2322, August, 1985.

[12] A. Persson and L.-C. Böiers, *"Analys i en Variabel,"* Lund, Sweden, Studentlitteratur, second edition, ISBN 9144020562, 2005.

[13] J.-M. Muller, *"Elementary Functions: Algorithm Implementation,"* Birkhäuser, Boston, USA, Springer Science & Business Media Inc., second edition, ISBN 0817643729, 2006.

[14] D. Zuras, M. Cowlishaw, et al., "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008,* pp. 1–70, August, 2008.

[15] "Design Compiler® Optimization Reference Manual," Version I-2013,12, December, 2013.

[16] M.D. Birnbaum, *"Essential Electronic Design Automation (EDA),"* Upper Saddle River, USA, Pearson Education, Inc., ISBN 0131828290, 2004.

[17] H.J. Beatty III, T.J. Ehrler, et al., "IEEE Standard for Integrated Circuit (IC) Open Library Architecture (OLA)," *IEEE STD 1481-2009,* March, 2010.

[18] G.K. Yeap, *"Practical Low Power Digital VLSI Design,"* New York, USA, Springer Science & Business Media inc, ISBN 9781461560654, 2012.

[19] J. Rabeay, A. Chandrakansan and B. Nikolić, *"Digital Integrated Circuits A Design Perspective,"* Beijing, China, Qinghua University Press, second edition, ISBN 7302079684, 2004.

[20] H.J.M. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits,* vol. 19, no. 4, pp. 468–473, August, 1984.

[21] W.-K. Chen, *"The Electrical Engineering Handbook,"* San Diego, USA, Academic Press, ISBN 0121709604, 2004.

Appendix A

Appendix

A.1 Look-up Tables for the NR Method

The following look-up tables in this section Table A.1 and Table. A.2 show the values and range of which the iteration 1 and iterations 2 are using in this thesis. X Represent the range in v of its corresponding value.

A graphical representation of the look-up table is presented after its corresponding table and is a representation of the LUTs over $1/\sqrt{v}$. The graphical representation is shown in Fig. A.1 and Fig. A.2.

No.	Value	Range
0	1	X = 1
1	0.997314453125000	$1 < X < 1.0108642578125$
2	0.991912841796875	$1.0108642578125 \leq X < 1.0218505859375$
3	0.986572265625000	$1.0218505859375 \leq X < 1.0330810546875$
4	0.981170654296875	$1.0330810546875 \leq X < 1.0444335937500$
5	0.975799560546875	$1.0444335937500 \leq X < 1.0560302734375$
6	0.970428466796875	$1.0560302734375 \leq X < 1.0677490234375$
7	0.965057373046875	$1.0677490234375 \leq X < 1.0797119140625$
8	0.959686279296875	$1.0797119140625 \leq X < 1.0919189453125$
9	0.954284667968750	$1.0919189453125 \leq X < 1.1042480468750$
10	0.948944091796875	$1.1042480468750 \leq X < 1.1168212890625$
11	0.943572998046875	$1.1168212890625 \leq X < 1.1296386718750$
12	0.938171386718750	$1.1296386718750 \leq X < 1.1427001953125$
13	0.932800292968750	$1.1427001953125 \leq X < 1.1558837890625$

No.	Value	Range
14	0.927429199218750	$1.1558837890625 \leq X < 1.1693115234375$
15	0.922088623046875	$1.1693115234375 \leq X < 1.1829833984375$
16	0.916717529296875	$1.1829833984375 \leq X < 1.1968994140625$
17	0.911376953125000	$1.1968994140625 \leq X < 1.2110595703125$
18	0.906005859375000	$1.2110595703125 \leq X < 1.2254638671875$
19	0.900634765625000	$1.2254638671875 \leq X < 1.2401123046875$
20	0.895294189453125	$1.2401123046875 \leq X < 1.2551269531250$
21	0.889923095703125	$1.2551269531250 \leq X < 1.2703857421875$
22	0.884521484375000	$1.2703857421875 \leq X < 1.2858886718750$
23	0.879180908203125	$1.2858886718750 \leq X < 1.3016357421875$
24	0.873809814453125	$1.3016357421875 \leq X < 1.3177490234375$
25	0.868438720703125	$1.3177490234375 \leq X < 1.3341064453125$
26	0.863098144531250	$1.3341064453125 \leq X < 1.3508300781250$
27	0.857696533203125	$1.3508300781250 \leq X < 1.3677978515625$
28	0.852355957031250	$1.3677978515625 \leq X < 1.3851318359375$
29	0.846984863281250	$1.3851318359375 \leq X < 1.4028320312500$
30	0.841613769531250	$1.4028320312500 \leq X < 1.4207763671875$
31	0.836273193359375	$1.4207763671875 \leq X < 1.4390869140625$
32	0.830902099609375	$1.4390869140625 \leq X < 1.4577636718750$
33	0.825561523437500	$1.4577636718750 \leq X < 1.4768066406250$
34	0.820190429687500	$1.4768066406250 \leq X < 1.4962158203125$
35	0.814849853515625	$1.4962158203125 \leq X < 1.5159912109375$
36	0.809478759765625	$1.5159912109375 \leq X < 1.5362548828125$
37	0.804107666015625	$1.5362548828125 \leq X < 1.5568847656250$
38	0.798767089843750	$1.5568847656250 \leq X < 1.5778808593750$
39	0.793395996093750	$1.5778808593750 \leq X < 1.5993652343750$
40	0.788024902343750	$1.5993652343750 \leq X < 1.6212158203125$
41	0.782684326171875	$1.6212158203125 \leq X < 1.6435546875000$
42	0.777343750000000	$1.6435546875000 \leq X < 1.6663818359375$
43	0.771972656250000	$1.6663818359375 \leq X < 1.6896972656250$
44	0.766601562500000	$1.6896972656250 \leq X < 1.7135009765625$
45	0.761260986328125	$1.7135009765625 \leq X < 1.7377929687500$
46	0.755889892578125	$1.7377929687500 \leq X < 1.7625732421875$
47	0.750549316406250	$1.7625732421875 \leq X < 1.7879638671875$
48	0.745178222656250	$1.7879638671875 \leq X < 1.8138427734375$
49	0.739807128906250	$1.8138427734375 \leq X < 1.8403320312500$
50	0.734466552734375	$1.8403320312500 \leq X < 1.8674316406250$
51	0.729095458984375	$1.8674316406250 \leq X < 1.8951416015625$
52	0.723724365234375	$1.8951416015625 \leq X < 1.9234619140625$

No.	Value	Range
53	0.718353271484375	$1.9234619140625 \leq X < 1.9523925781250$
54	0.712982177734375	$1.9523925781250 \leq X < 1.9819335937500$
55	0.707641601562500	$1.9819335937500 \leq X < 2.0122070312500$
56	0.702270507812500	$2.0122070312500 \leq X < 2.0430908203125$
57	0.696929931640625	$2.0430908203125 \leq X < 2.0747070312500$
58	0.691558837890625	$2.0747070312500 \leq X < 2.1070556640625$
59	0.686218261718750	$2.1070556640625 \leq X < 2.1402587890625$
60	0.680847167968750	$2.1402587890625 \leq X < 2.1741943359375$
61	0.675506591796875	$2.1741943359375 \leq X < 2.2089843750000$
62	0.670135498046875	$2.2089843750000 \leq X < 2.2446289062500$
63	0.664764404296875	$2.2446289062500 \leq X < 2.2811279296875$
64	0.659423828125000	$2.2811279296875 \leq X < 2.3184814453125$
65	0.654052734375000	$2.3184814453125 \leq X < 2.3568115234375$
66	0.648681640625000	$2.3568115234375 \leq X < 2.3959960937500$
67	0.643341064453125	$2.3959960937500 \leq X < 2.4361572265625$
68	0.638000488281250	$2.4361572265625 \leq X < 2.4774169921875$
69	0.632629394531250	$2.4774169921875 \leq X < 2.5196533203125$
70	0.627288818359375	$2.5196533203125 \leq X < 2.5629882812500$
71	0.621948242187500	$2.5629882812500 \leq X < 2.6075439453125$
72	0.616577148437500	$2.6075439453125 \leq X < 2.6531982421875$
73	0.611236572265625	$2.6531982421875 \leq X < 2.7000732421875$
74	0.605895996093750	$2.7000732421875 \leq X < 2.7481689453125$
75	0.600524902343750	$2.7481689453125 \leq X < 2.7976074218750$
76	0.595184326171875	$2.7976074218750 \leq X < 2.8483886718750$
77	0.589813232421875	$2.8483886718750 \leq X < 2.9005126953125$
78	0.584472656250000	$2.9005126953125 \leq X < 2.9541015625000$
79	0.579132080078125	$2.9541015625000 \leq X < 3.0091552734375$
80	0.573791503906250	$3.0091552734375 \leq X < 3.0657958984375$
81	0.568420410156250	$3.0657958984375 \leq X < 3.1240234375000$
82	0.563079833984375	$3.1240234375000 \leq X < 3.1839599609375$
83	0.557739257812500	$3.1839599609375 \leq X < 3.2456054687500$
84	0.552398681640625	$3.2456054687500 \leq X < 3.3090820312500$
85	0.547027587890625	$3.3090820312500 \leq X < 3.3745117187500$
86	0.541687011718750	$3.3745117187500 \leq X < 3.4418945312500$
87	0.536315917968750	$3.4418945312500 \leq X < 3.5112304687500$
88	0.530975341796875	$3.5112304687500 \leq X < 3.5827636718750$
89	0.525634765625000	$3.5827636718750 \leq X < 3.6564941406250$
90	0.520263671875000	$3.6564941406250 \leq X < 3.7325439453125$
91	0.514923095703125	$3.7325439453125 \leq X < 3.8109130859375$

No.	Value	Range
92	0.509552001953125	$3.8109130859375 \leq X < 3.8918457031250$
93	0.504211425781250	$3.8918457031250 \leq X < 3.9753417968750$
94	0.498870849609375	$3.9753417968750 \leq X \leq 3.9998779296875$

Table A.1: *Look-up table for the Newton-Raphson method using 1 iteration*

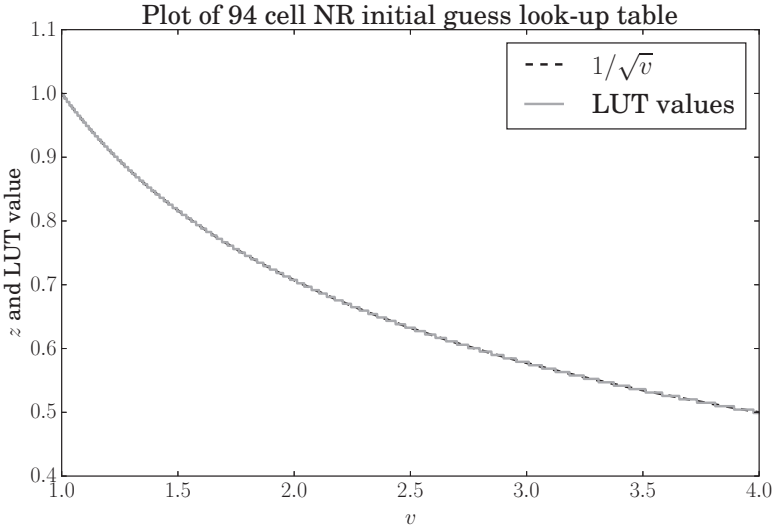


Figure A.1: *Representation of the look-up table for 1 iteration over the function* $1/\sqrt{v}$

No.	Value	Range
0	1	$X = 1$
1	0.980468750000000	$1 < X < 1.0830078125000$
2	0.941375732421875	$1.0830078125000 \leq X < 1.1767578125000$
3	0.902313232421875	$1.1767578125000 \leq X < 1.2832031250000$
4	0.863250732421875	$1.2832031250000 \leq X < 1.4047851562500$
5	0.824188232421875	$1.4047851562500 \leq X < 1.5444335937500$
6	0.785125732421875	$1.5444335937500 \leq X < 1.7060546875000$
7	0.746063232421875	$1.7060546875000 \leq X < 1.8944091796875$
8	0.707000732421875	$1.8944091796875 \leq X < 2.1157226562500$

No.	Value	Range
9	0.667968750000000	$2.1157226562500 \leq X < 2.3781738281250$
10	0.628906250000000	$2.3781738281250 \leq X < 2.6926269531250$
11	0.589874267578125	$2.6926269531250 \leq X < 3.0738525390625$
12	0.550842285156250	$3.0738525390625 \leq X < 3.5422363281250$
13	0.511779785156250	$3.5422363281250 \leq X \leq 3.9998779296875$

Table A.2: *Look-up table for the Newton-Raphson method using 2 iterations*

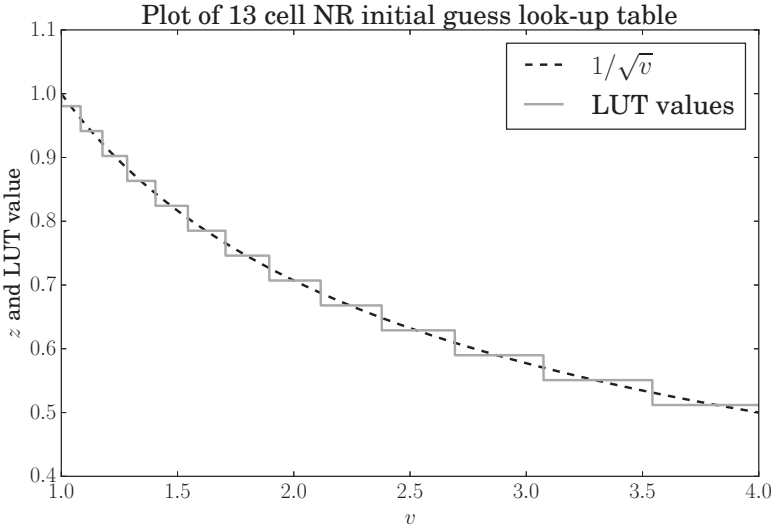


Figure A.2: *Representation of the look-up table for 2 iterations over the function* $1/\sqrt{v}$

A.2 HPS Dataflow Simulation

A.2.1 Elaboration of Software Development

Following the selection of a c_1 value, software to simulate the dataflow behaviour and to visualize the results was written. Matlab and its fixed-point math toolbox was chosen at first, but was dropped in favour of writing the simulations in C due to performance issues. The C simulation code was initially written making as few assumptions as possible, which quickly lead to complex code that proved difficult to debug and maintain. Among the features/problems were:

- Data paths implemented with fixed-point math supporting arbitrarily wide data paths.
- Trying to predict and rank relative area requirements by keeping track of required sizes of the multipliers in the design.
- Experimenting with replacing interval pairs with a single interval, reducing the number of constants, at the cost of slightly more complex addressing.
- Trying all truncation combinations, given an unwillingness to risk passing over a "silver bullet" combination, should one exist.

The last two points increased the processing time a lot, since the number of simulations needed would be the product of the number of possible choices for each constraint. After a while, lower limits for the various data paths, for which below, the simulation would always fail, started to emerge. Still, the number of simulations was huge, and would need rerunning at each and every change of this or that parameter, in the end necessitating adapting the code to run on multiple threads.

During this work the existence of a few misconceptions came to light, slightly changing the requirements of the implementation, but in a way that invalidated data produced so far. Given that the simulation code had been continuously developed and grown organically, to take advantage of what could be gleaned from the results as they were generated, at this point it was decided that a complete rewrite was preferable to trying to adapt it to the new requirements.

Lessons learned from the first iteration of simulations, that were included in the second version include:

- Data paths are implemented with 32 bit signed integers, interpreted as Q3.29 signed fixed-point numbers, enabling the simulations to run on both 32 and 64 bit hardware without resorting to possibly nonstandard, or compound datatypes.

- Keeping to a single interval size over the whole input range.
- Greedy truncation, going from output to input, truncating as much as possible at each step.

With the new software the simulation workflow is as follows:

1. For each tested interval count, a set of constants (c_2,l_2,j_2) with which to populate the lookup table is generated.
2. The dataflow behaviour is simulated with the generated lookup table for all the possible input values.
3. Going from z towards v, each data path is increasingly truncated until the error of z becomes greater than acceptable.
4. When no more truncation is possible without crossing the error limit, save the details of how much each data path could be truncated.
5. For each set of constants in each interval, iteratively try to make small changes to the constants and see if the average error can be made smaller, while at the same time staying below the error limit for all points in the interval. Save the new set of constants.
6. (optionally) Check if the truncation of any data path can be increased after the modification of constants.

Additionally there are tools written to convert the look-up table values to VHDL code, identify if any of the data paths have any redundant bits that can be hardwired to '1' or '0', generate files suitable to use with VHDL testbenches and simulation, and to help visualize the results.

A.2.2 Error Behaviour of Additional HPS Implementations

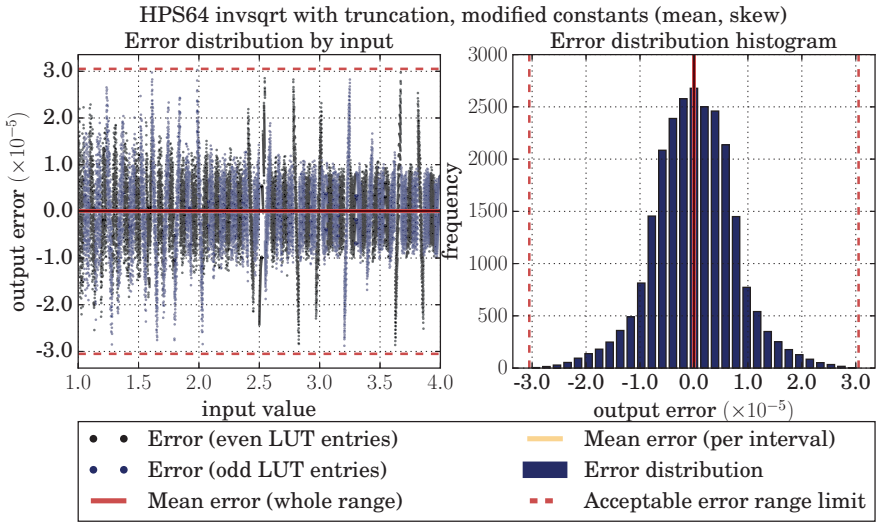


Figure A.3: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 64 intervals, with truncated data paths, and constants modified to minimize per interval absolute mean error as well as per interval absolute skew.*

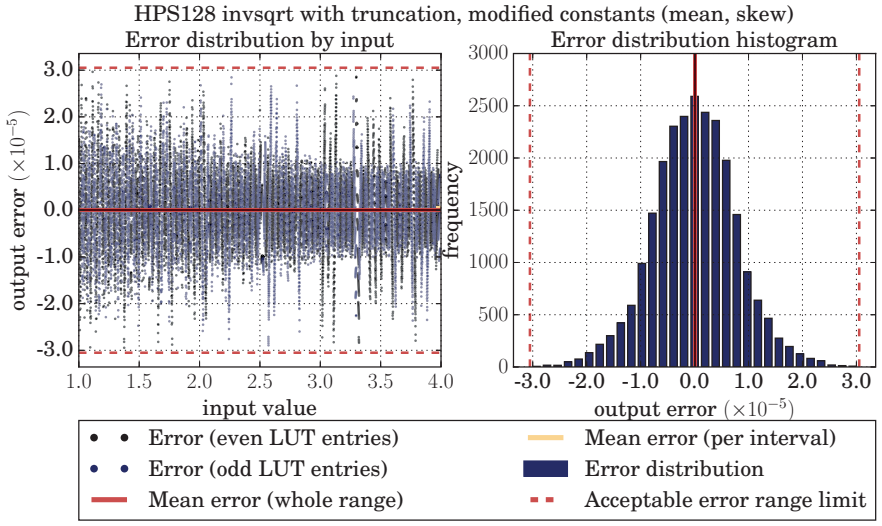


Figure A.4: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 128 intervals, with truncated data paths, and constants modified to minimize per interval absolute mean error as well as per interval absolute skew.*

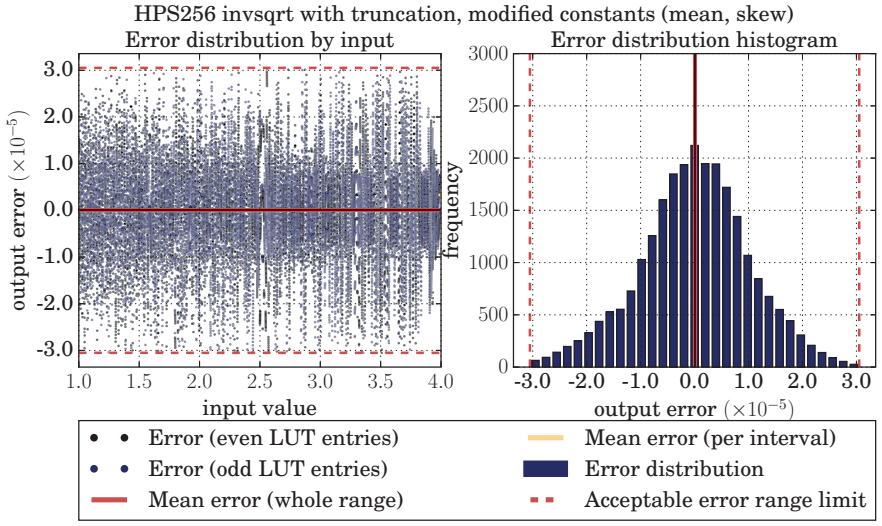


Figure A.5: *Input to output error plot and error distribution histogram for the inverse square root implemented with HPS with 256 intervals, with truncated data paths, and constants modified to minimize per interval absolute mean error as well as per interval absolute skew.*

A.3 Extra Hardware Behaviour Plots

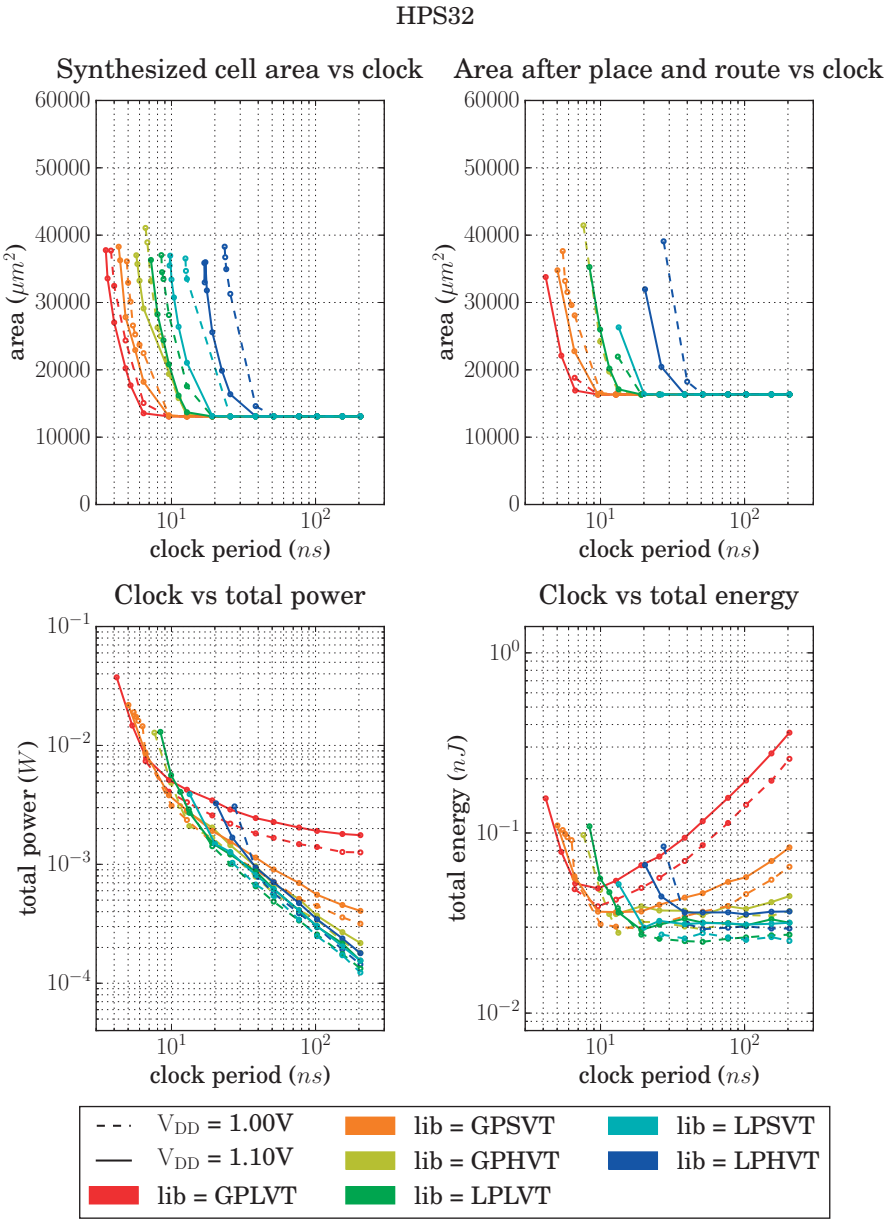


Figure A.6: *Hardware behaviour data for HPS with 32 intervals, implemented with all technology libraries.*

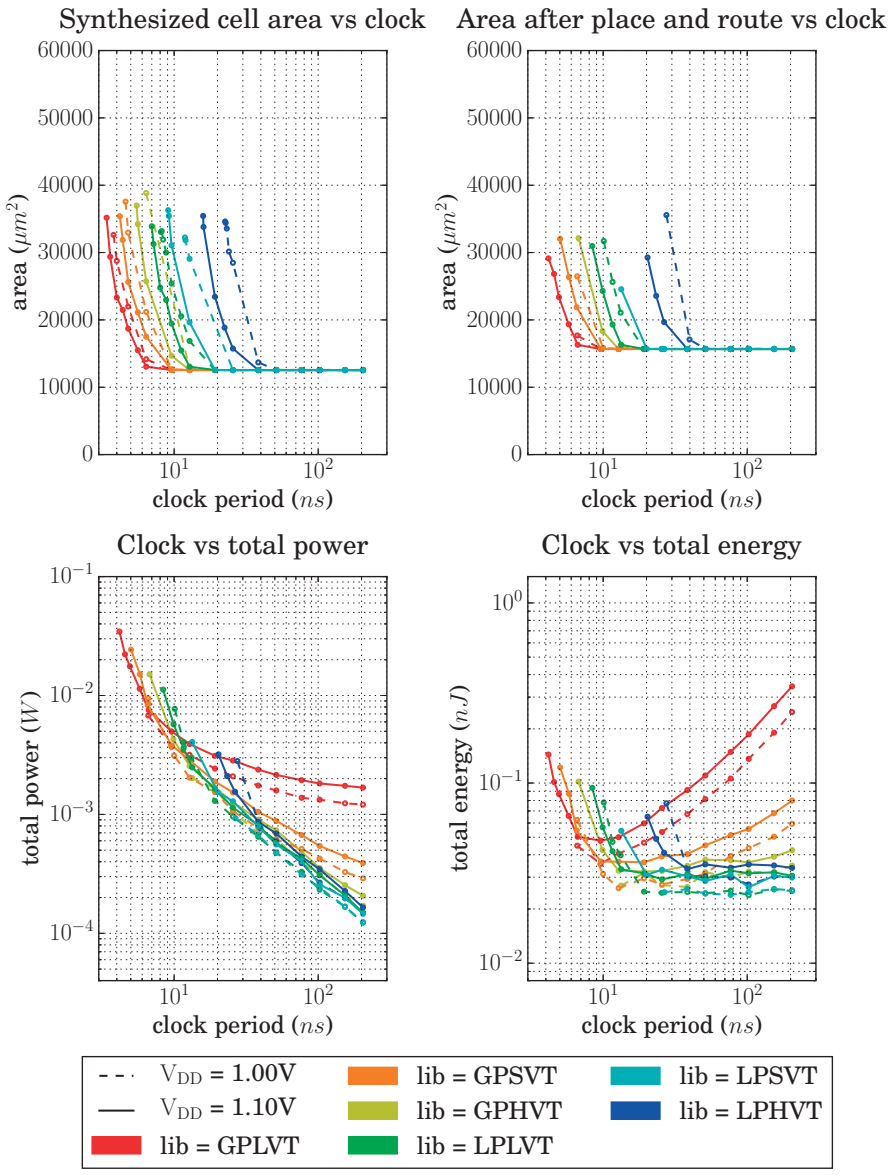


Figure A.7: *Hardware behaviour data for HPS with 64 intervals, implemented with all technology libraries.*

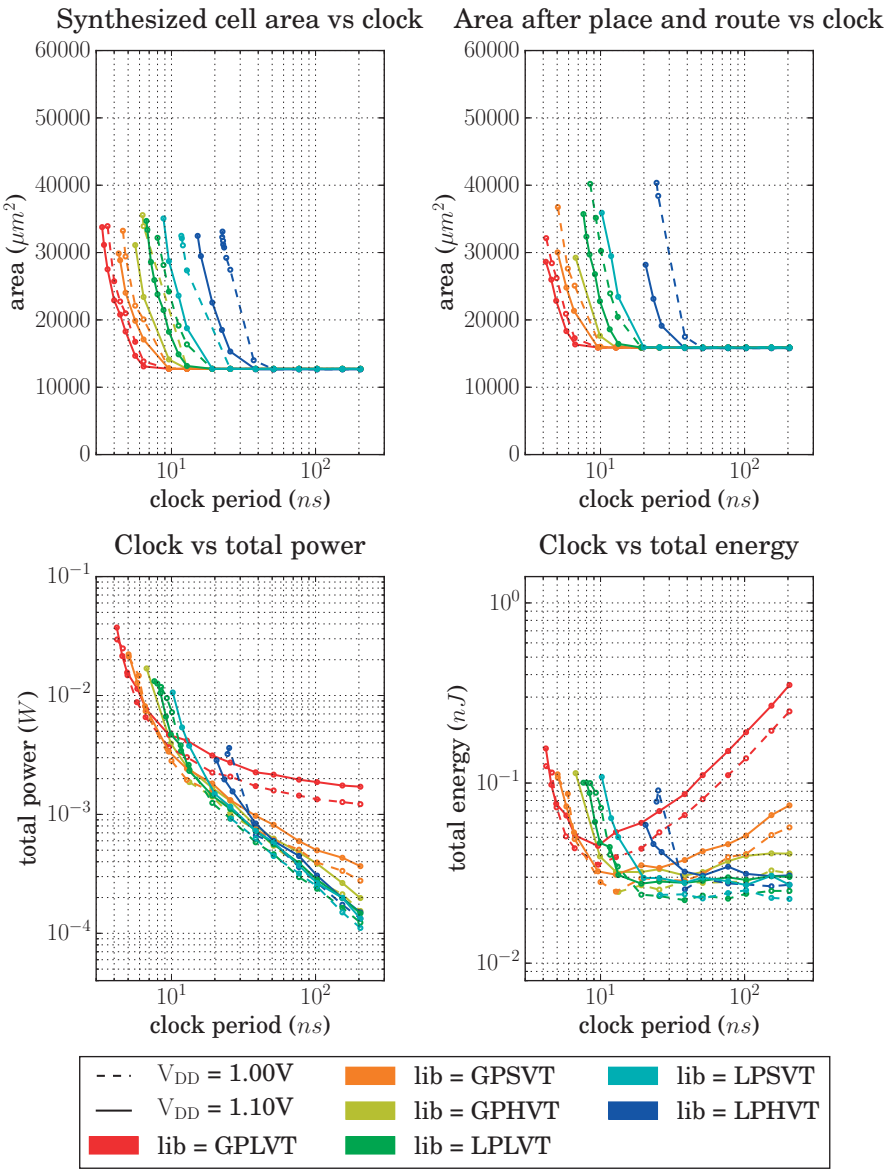


Figure A.8: *Hardware behaviour data for HPS with 128 intervals, implemented with all technology libraries.*

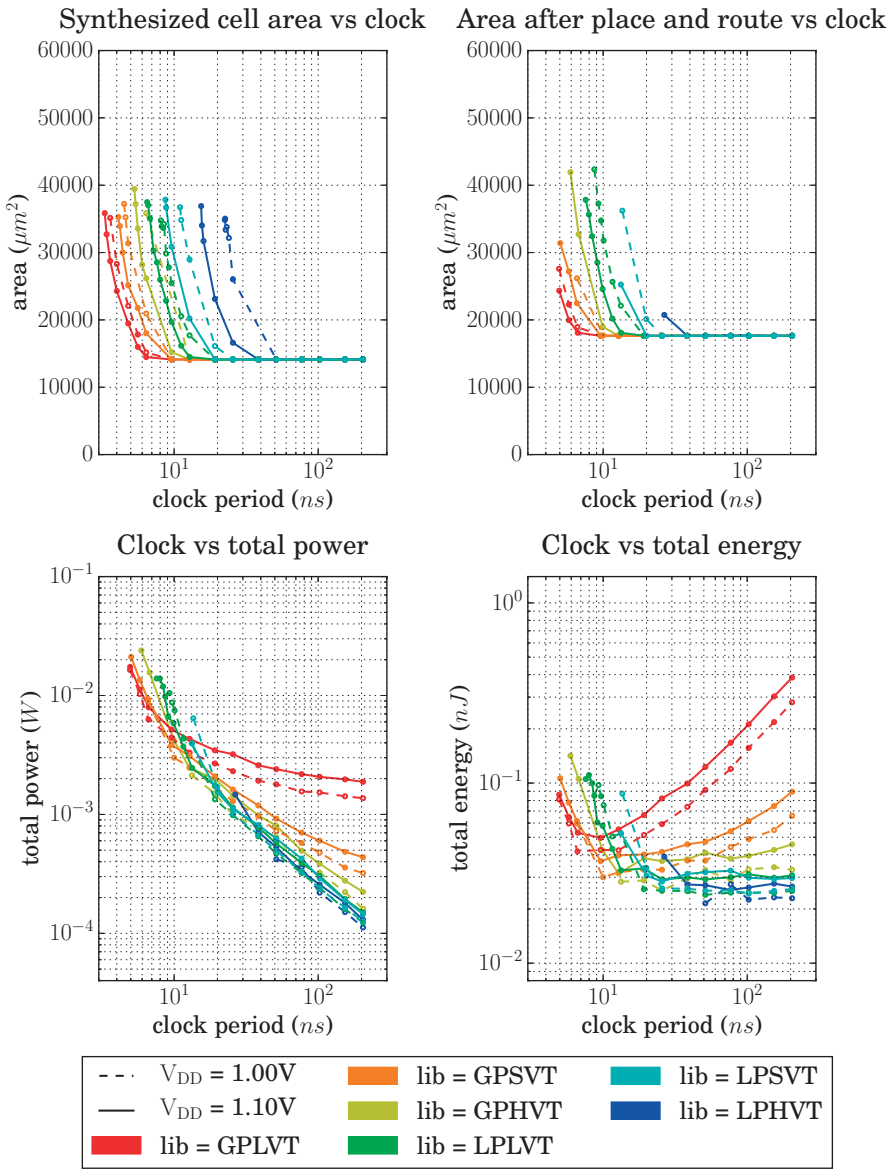


Figure A.9: *Hardware behaviour data for HPS with 256 intervals, implemented with all technology libraries.*

HPS512

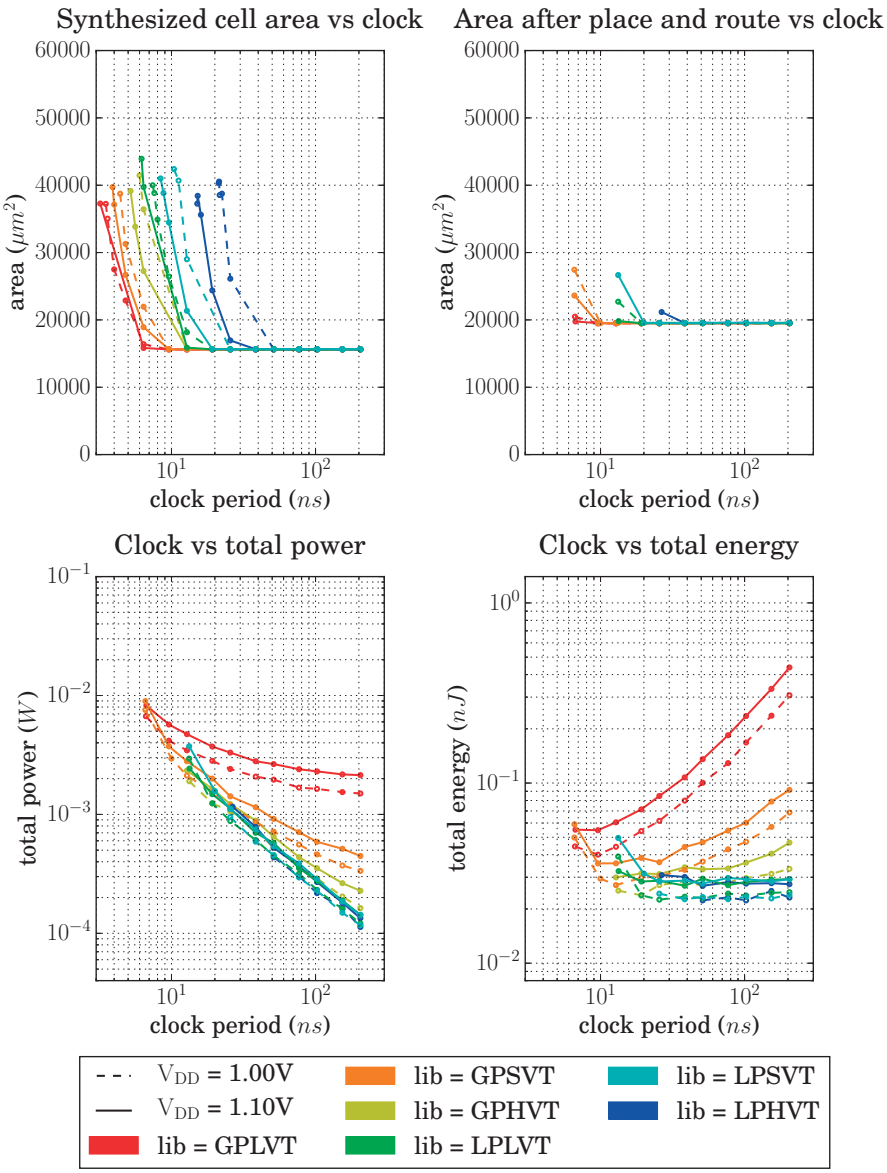


Figure A.10: *Hardware behaviour data for HPS with 512 intervals, implemented with all technology libraries.*

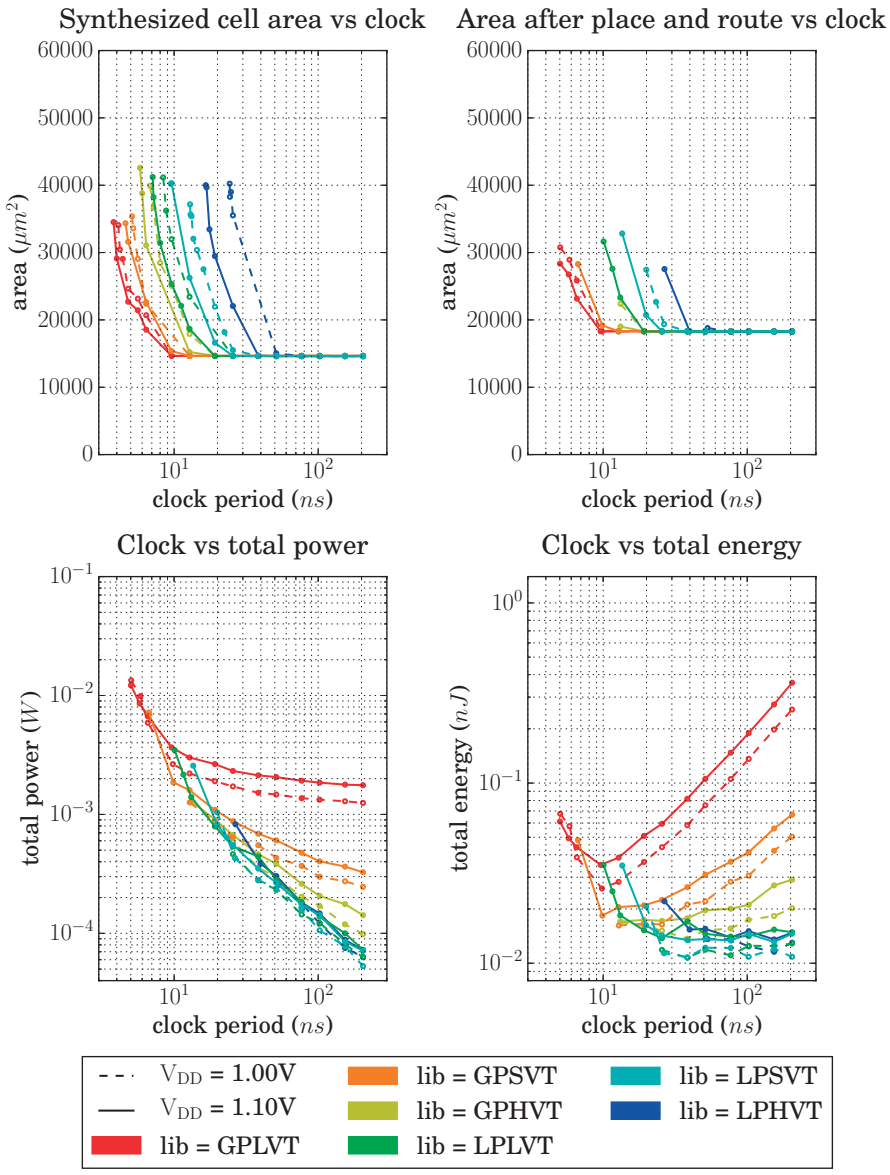


Figure A.11: *Hardware behaviour data for NR with 1 iteration, implemented with all technology libraries.*

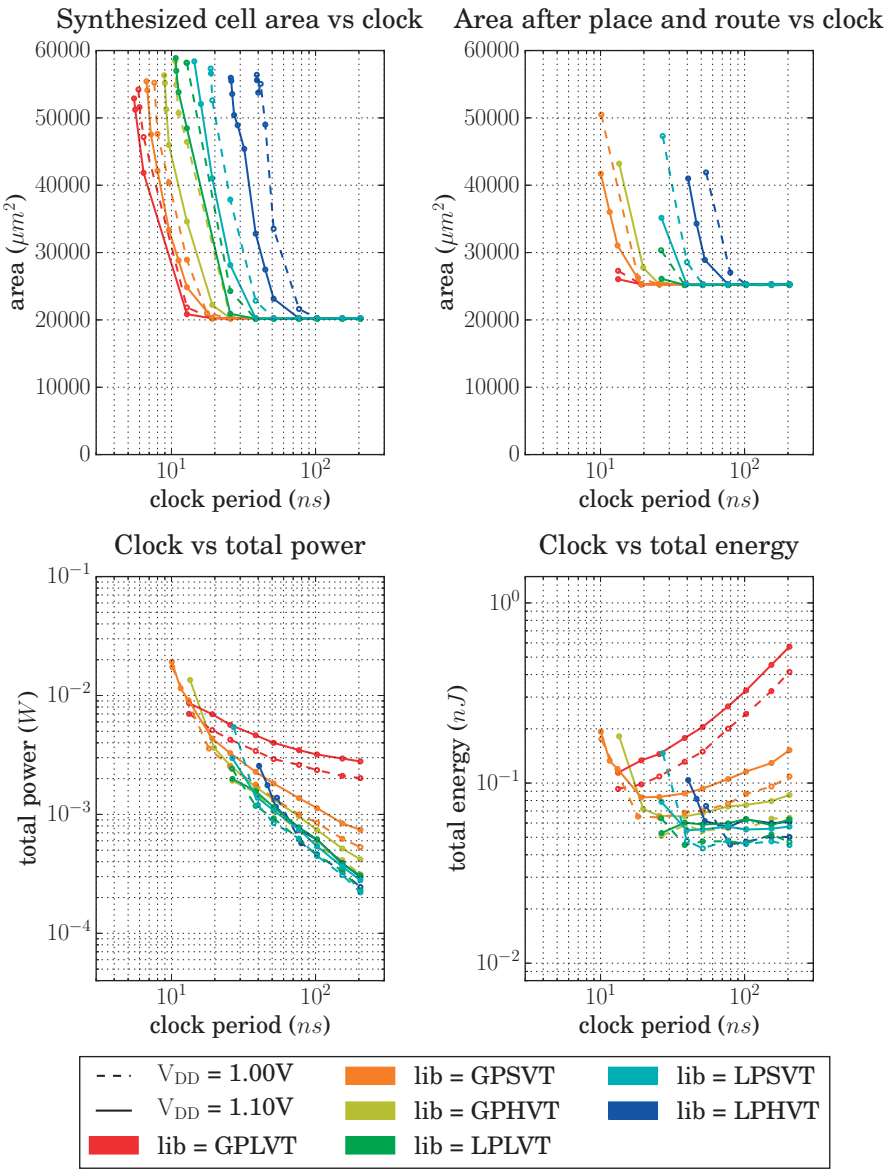


Figure A.12: *Hardware behaviour data for NR with 2 iteration, implemented with all technology libraries.*

A.4 Extra Hardware Behaviour Tables

A.4.1 Clock and Area

Impl.	Library	V_{DD}	Synth Clock (ns)
HPS32	GPLVT	1.10V	3.5
HPS64	GPLVT	1.10V	3.4
HPS128	GPLVT	1.10V	3.3
HPS256	GPLVT	1.10V	3.3
HPS512	GPLVT	1.10V	3.2
NR1	GPLVT	1.10V	3.8
NR2	GPLVT	1.10V	5.5
HPS32	GPSVT	1.10V	4.3
HPS64	GPSVT	1.10V	4.2
HPS128	GPSVT	1.10V	4.3
HPS256	GPSVT	1.10V	4.1
HPS512	GPSVT	1.10V	3.9
NR1	GPSVT	1.10V	4.6
NR2	GPSVT	1.10V	6.7
HPS32	GPHVT	1.10V	5.7
HPS64	GPHVT	1.10V	5.5
HPS128	GPHVT	1.10V	5.6
HPS256	GPHVT	1.10V	5.3
HPS512	GPHVT	1.10V	5.2
NR1	GPHVT	1.10V	5.8
NR2	GPHVT	1.10V	8.9
HPS32	LPLVT	1.10V	7.2
HPS64	LPLVT	1.10V	7
HPS128	LPLVT	1.10V	6.7
HPS256	LPLVT	1.10V	6.5
HPS512	LPLVT	1.10V	6.2
NR1	LPLVT	1.10V	7.1
NR2	LPLVT	1.10V	10.7
HPS32	LPSVT	1.10V	9.7
HPS64	LPSVT	1.10V	9.1
HPS128	LPSVT	1.10V	8.8
HPS256	LPSVT	1.10V	8.7
HPS512	LPSVT	1.10V	8.4
NR1	LPSVT	1.10V	9.5
NR2	LPSVT	1.10V	14.4
HPS32	LPHVT	1.10V	16.9
HPS64	LPHVT	1.10V	15.9
HPS128	LPHVT	1.10V	15.2
HPS256	LPHVT	1.10V	15.4
HPS512	LPHVT	1.10V	15.1
NR1	LPHVT	1.10V	16.6
NR2	LPHVT	1.10V	25.8

Table A.3: *Listing of the lowest synthesized clock periods for all implementations (selected from all technology libraries etc.)*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPLVT	1.10V	4.163	33779	37.4	155.696
HPS64	GPLVT	1.10V	4.173	29136	34.5	143.968
HPS128	GPLVT	1.10V	4.173	28629	37.3	155.653
HPS256	GPLVT	1.00V	4.943	27584	16.4	81.065
HPS512	GPLVT	1.00V	6.645	20478	6.69	44.455
NR1	GPLVT	1.10V	5.014	28355	12.2	61.171
NR2	GPLVT	1.10V	13.239	26050	8.62	114.120
HPS32	GPSVT	1.10V	5.024	34779	21.9	110.026
HPS64	GPSVT	1.10V	5.024	32036	24.3	122.083
HPS128	GPSVT	1.10V	5.035	30030	22.2	111.777
HPS256	GPSVT	1.10V	5.035	31400	21.1	106.239
HPS512	GPSVT	1.10V	6.584	23624	8.97	59.058
NR1	GPSVT	1.10V	6.686	28288	7.21	48.206
NR2	GPSVT	1.10V	10.049	41696	19.2	192.941
HPS32	GPHVT	1.00V	7.608	41452	12.8	97.382
HPS64	GPHVT	1.10V	6.747	32142	15.1	101.880
HPS128	GPHVT	1.10V	6.716	29237	16.9	113.500
HPS256	GPHVT	1.10V	5.927	41939	23.9	141.655
HPS512	GPHVT	1.10V	12.8	19540	2.34	29.952
NR1	GPHVT	1.10V	13.188	19016	1.29	17.013
NR2	GPHVT	1.10V	13.463	43216	13.5	181.750
HPS32	LPLVT	1.10V	8.378	35268	13	108.914
HPS64	LPLVT	1.10V	8.398	30973	11.2	94.058
HPS128	LPLVT	1.10V	7.608	35718	13.2	100.426
HPS256	LPLVT	1.10V	7.567	37826	13.9	105.181
HPS512	LPLVT	1.00V	13.259	22704	2.95	39.114
NR1	LPLVT	1.10V	10.069	31647	3.48	35.040
NR2	LPLVT	1.00V	26.345	30346	2.42	63.755
HPS32	LPSVT	1.10V	13.361	26308	3.89	51.974
HPS64	LPSVT	1.10V	13.351	24554	4.07	54.339
HPS128	LPSVT	1.10V	10.202	35905	10.6	108.141
HPS256	LPSVT	1.10V	13.28	25249	3.96	52.589
HPS512	LPSVT	1.10V	13.259	26662	3.74	49.589
NR1	LPSVT	1.10V	13.565	32830	2.57	34.862
NR2	LPSVT	1.10V	26.478	35156	2.97	78.640
HPS32	LPHVT	1.10V	20.363	31965	3.26	66.383
HPS64	LPHVT	1.10V	20.333	29271	3.2	65.066
HPS128	LPHVT	1.10V	20.557	28188	2.85	58.587
HPS256	LPHVT	1.10V	26.569	20735	1.47	39.056
HPS512	LPHVT	1.10V	26.559	21164	1.16	30.808
NR1	LPHVT	1.10V	26.651	27551	0.827	22.040
NR2	LPHVT	1.10V	40.584	41001	2.56	103.895

Table A.4: *Listing of the lowest clock periods after PNR for all implementations (selected from all technology libraries etc.)*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPLVT	1.00V	9.6	16333	4.08	39.168
HPS64	GPLVT	1.10V	9.6	15675	4.99	47.904
HPS128	GPLVT	1.00V	153.6	15875	1.27	195.072
HPS256	GPLVT	1.00V	102.4	17609	1.53	156.672
HPS512	GPLVT	1.00V	9.6	19485	4.17	40.032
NR1	GPLVT	1.10V	9.6	18306	3.66	35.136
NR2	GPLVT	1.00V	19.2	25281	5.12	98.304
HPS32	GPSVT	1.10V	9.6	16333	3.81	36.576
HPS64	GPSVT	1.10V	9.651	15675	3.8	36.674
HPS128	GPSVT	1.10V	9.6	15900	3.37	32.352
HPS256	GPSVT	1.10V	9.6	17636	3.84	36.864
HPS512	GPSVT	1.10V	9.6	19512	3.73	35.808
NR1	GPSVT	1.10V	12.8	18306	1.6	20.48
NR2	GPSVT	1.10V	19.2	25281	4.34	83.328
HPS32	GPHVT	1.10V	12.8	16333	2.77	35.456
HPS64	GPHVT	1.10V	12.8	15675	2.55	32.64
HPS128	GPHVT	1.10V	12.8	15900	2.51	32.128
HPS256	GPHVT	1.10V	12.8	17636	2.46	31.488
HPS512	GPHVT	1.00V	19.2	19512	1.24	23.808
NR1	GPHVT	1.10V	19.2	18306	0.906	17.395
NR2	GPHVT	1.10V	25.6	25281	2.57	65.792
HPS32	LPLVT	1.10V	19.2	16333	1.51	28.992
HPS64	LPLVT	1.10V	19.2	15675	1.64	31.488
HPS128	LPLVT	1.00V	19.2	15900	1.25	24
HPS256	LPLVT	1.00V	19.2	17636	1.34	25.728
HPS512	LPLVT	1.00V	19.2	19512	1.24	23.808
NR1	LPLVT	1.00V	76.8	18246	0.144	11.059
NR2	LPLVT	1.00V	38.4	25154	1.18	45.312
HPS32	LPSVT	1.10V	25.6	16333	1.27	32.512
HPS64	LPSVT	1.10V	25.6	15675	1.29	33.024
HPS128	LPSVT	1.00V	25.906	15900	0.924	23.937
HPS256	LPSVT	1.10V	20.016	17636	1.53	30.624
HPS512	LPSVT	1.00V	25.6	19512	0.95	24.32
NR1	LPSVT	1.00V	38.4	18279	0.281	10.790
NR2	LPSVT	1.00V	51.292	25249	0.846	43.393
HPS32	LPHVT	1.10V	38.4	16358	0.942	36.173
HPS64	LPHVT	1.10V	38.4	15675	0.867	33.293
HPS128	LPHVT	1.10V	38.4	15800	0.839	32.218
HPS256	LPHVT	1.10V	38.4	17636	0.712	27.341
HPS512	LPHVT	1.10V	38.4	19540	0.784	30.106
NR1	LPHVT	1.10V	51.2	18246	0.303	15.514
NR2	LPHVT	1.10V	76.8	25281	0.741	56.909

Table A.5: *Listing of the lowest area requirements after PNR for all implementations (selected from all technology libraries etc.)*

A.4.2 Power and Energy

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPLVT	1.00V	204.8	16333	1.26	258.048
HPS64	GPLVT	1.00V	204.8	15700	1.21	247.808
HPS128	GPLVT	1.00V	204.8	15875	1.22	249.856
HPS256	GPLVT	1.00V	204.8	17636	1.37	280.576
HPS512	GPLVT	1.00V	204.8	19485	1.5	307.2
NR1	GPLVT	1.00V	204.8	18306	1.25	256
NR2	GPLVT	1.00V	204.8	25281	2.02	413.696
HPS32	GPSVT	1.00V	204.8	16333	0.317	64.922
HPS64	GPSVT	1.00V	204.8	15675	0.29	59.392
HPS128	GPSVT	1.00V	204.8	15900	0.277	56.730
HPS256	GPSVT	1.00V	204.8	17636	0.321	65.741
HPS512	GPSVT	1.00V	204.8	19512	0.335	68.608
NR1	GPSVT	1.00V	204.8	18333	0.246	50.381
NR2	GPSVT	1.00V	204.8	25281	0.531	108.749
HPS32	GPHVT	1.00V	204.8	16333	0.18	36.864
HPS64	GPHVT	1.00V	204.8	15675	0.17	34.816
HPS128	GPHVT	1.00V	204.8	15900	0.154	31.539
HPS256	GPHVT	1.00V	204.8	17636	0.161	32.973
HPS512	GPHVT	1.00V	204.8	19512	0.163	33.382
NR1	GPHVT	1.00V	204.8	18306	0.099	20.193
NR2	GPHVT	1.00V	204.8	25281	0.313	64.102
HPS32	LPLVT	1.00V	204.8	16333	0.133	27.238
HPS64	LPLVT	1.00V	204.8	15675	0.123	25.190
HPS128	LPLVT	1.00V	204.8	15900	0.123	25.190
HPS256	LPLVT	1.00V	204.8	17636	0.123	25.190
HPS512	LPLVT	1.00V	204.8	19512	0.121	24.781
NR1	LPLVT	1.00V	204.8	18246	0.063	12.861
NR2	LPLVT	1.00V	204.8	25154	0.23	47.104
HPS32	LPSVT	1.00V	204.8	16333	0.123	25.190
HPS64	LPSVT	1.00V	204.8	15675	0.124	25.395
HPS128	LPSVT	1.00V	204.8	15900	0.111	22.733
HPS256	LPSVT	1.00V	204.8	17636	0.126	25.805
HPS512	LPSVT	1.00V	204.8	19512	0.117	23.962
NR1	LPSVT	1.00V	204.8	18279	0.053	10.854
NR2	LPSVT	1.00V	204.8	25249	0.221	45.261
HPS32	LPHVT	1.00V	204.8	16384	0.144	29.491
HPS64	LPHVT	1.00V	204.8	15700	0.149	30.515
HPS128	LPHVT	1.00V	204.8	15800	0.133	27.238
HPS256	LPHVT	1.00V	204.8	17662	0.112	22.938
HPS512	LPHVT	1.00V	204.8	19568	0.113	23.142
NR1	LPHVT	1.00V	204.8	18272	0.063	12.984
NR2	LPHVT	1.00V	204.8	25281	0.245	50.176

Table A.6: *Listing of the total power consumption for all implementations (selected from all technology libraries etc.)*

Impl.	Library	V_{DD}	PNR Clock (ns)	PNR Area (μm^2)	Power (mW)	Energy (pJ)
HPS32	GPLVT	1.00V	9.6	16333	4.08	39.168
HPS64	GPLVT	1.00V	9.6	15700	3.71	35.616
HPS128	GPLVT	1.00V	9.6	15900	3.67	35.232
HPS256	GPLVT	1.00V	6.635	18955	6.29	41.734
HPS512	GPLVT	1.00V	9.6	19485	4.17	40.032
NR1	GPLVT	1.00V	9.804	18306	2.65	25.981
NR2	GPLVT	1.00V	13.239	27289	7.01	92.805
HPS32	GPSVT	1.00V	19.2	16333	1.53	29.376
HPS64	GPSVT	1.00V	12.8	15675	2.04	26.112
HPS128	GPSVT	1.00V	12.8	15900	1.95	24.96
HPS256	GPSVT	1.00V	9.998	17689	3	29.994
HPS512	GPSVT	1.00V	12.8	19512	2.12	27.136
NR1	GPSVT	1.00V	12.841	18333	1.26	16.180
NR2	GPSVT	1.00V	25.6	25281	2.52	64.512
HPS32	GPHVT	1.00V	13.3	16740	2.1	27.93
HPS64	GPHVT	1.00V	38.4	15675	0.69	26.496
HPS128	GPHVT	1.00V	13.3	16099	1.87	24.871
HPS256	GPHVT	1.00V	25.6	17636	0.989	25.318
HPS512	GPHVT	1.00V	19.2	19512	1.24	23.808
NR1	GPHVT	1.00V	38.4	18306	0.356	13.670
NR2	GPHVT	1.00V	26.478	25566	1.92	50.838
HPS32	LPLVT	1.00V	51.2	16333	0.486	24.883
HPS64	LPLVT	1.00V	102.4	15675	0.233	23.859
HPS128	LPLVT	1.00V	38.4	15900	0.583	22.387
HPS256	LPLVT	1.00V	51.2	17636	0.468	23.962
HPS512	LPLVT	1.00V	25.6	19512	0.88	22.528
NR1	LPLVT	1.00V	38.4	18279	0.279	10.714
NR2	LPLVT	1.00V	38.4	25154	1.18	45.312
HPS32	LPSVT	1.00V	204.8	16333	0.123	25.190
HPS64	LPSVT	1.00V	76.8	15675	0.312	23.962
HPS128	LPSVT	1.00V	204.8	15900	0.111	22.733
HPS256	LPSVT	1.00V	102.4	17636	0.238	24.371
HPS512	LPSVT	1.00V	38.4	19512	0.59	22.656
NR1	LPSVT	1.00V	38.4	18279	0.281	10.790
NR2	LPSVT	1.00V	51.292	25249	0.846	43.393
HPS32	LPHVT	1.00V	51.2	16384	0.571	29.235
HPS64	LPHVT	1.00V	102.4	15700	0.267	27.341
HPS128	LPHVT	1.00V	38.527	17498	0.667	25.698
HPS256	LPHVT	1.00V	51.2	17662	0.42	21.504
HPS512	LPHVT	1.00V	51.2	19568	0.436	22.323
NR1	LPHVT	1.00V	153.6	18272	0.075	11.566
NR2	LPHVT	1.00V	79.586	27027	0.572	45.523

Table A.7: *Listing of the total energy consumption for one calculation for all implementations (selected from all technology libraries etc.)*

A.5 Static and Dynamic Power Plots

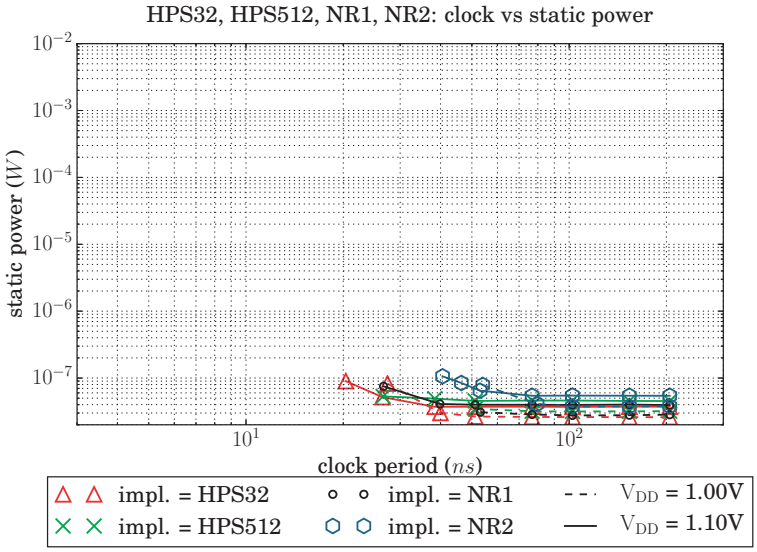


Figure A.13: *Static power consumption for implementations using the LPHVT technology library.*

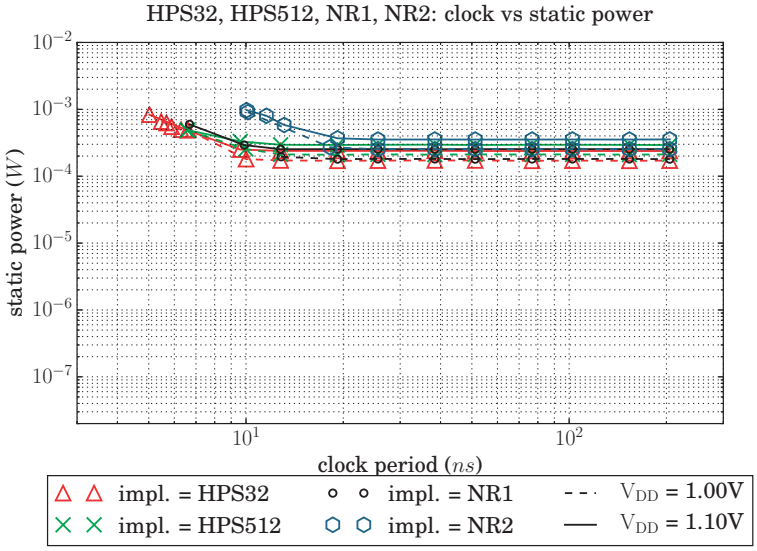


Figure A.14: *Static power consumption for implementations using the GPSVT technology library.*

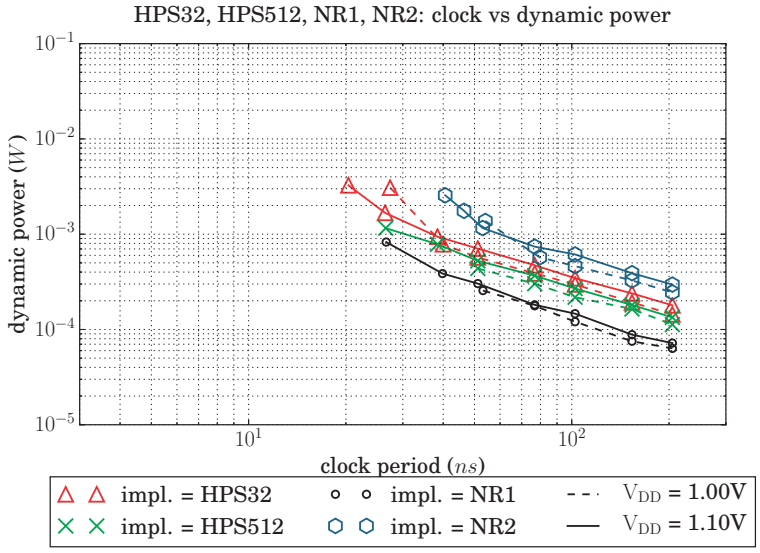


Figure A.15: *Dynamic power consumption for implementations using the LPHVT technology library.*

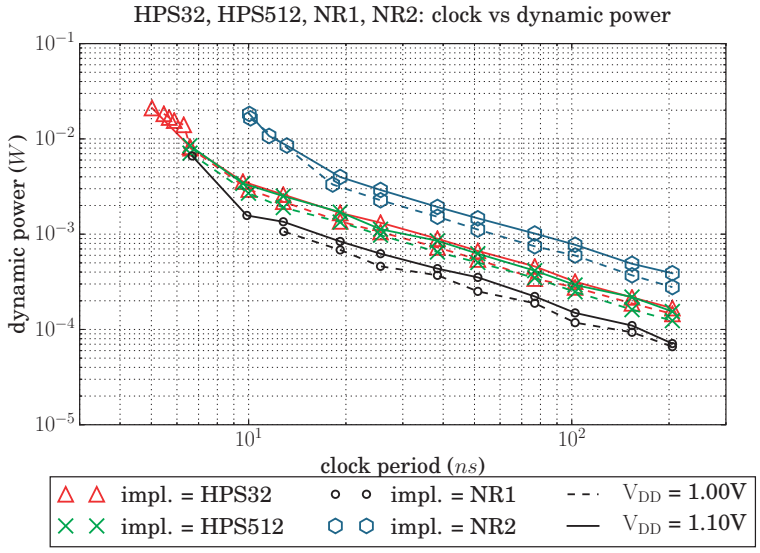


Figure A.16: *Dynamic power consumption for implementations using the GPSVT technology library.*

A.6 Tool Control Scripts

A.6.1 Synthesis Script

01_synthesis_dv.tcl

```
remove_design -all
set company  "Electrical and Information Technology"
set cache_read ./;
set cache_write ./;

define_design_lib DEFAULT -path "$::env(work_dir)/WORK"
define_design_lib WORK -path "$::env(work_dir)/WORK"
define_design_lib IO65LPHVT_SF1V8_50A_7M4X0Y2Z \
    -path "no_such_lib"
set search_path "\
    $::env(STM065_DIR)/CORE65$::env(library)_5.1/libs \
    $::env(STM065_DIR)/CLOCK65$::env(library)_3.1/libs \
    /usr/local-eit/cad2/synopsys/syn2011/libraries/syn/ \
    ."
set synthetic_library "standard.sldb dw_foundation.sldb"
set target_library "\
    CORE65$::env(library)_nom_$::env(vdd)_25C.db \
    CLOCK65$::env(library)_nom_$::env(vdd)_25C.db \
    "

set link_library [concat $target_library $synthetic_library]

set symbol_library "\
    CORE65$::env(library).sdb \
    CLOCK65$::env(library).sdb \
    "

puts $search_path
puts $target_library
puts $link_library
puts $symbol_library

set vhdlout_use_packages "\
    IEEE.std_logic_1164 \
    IEEE.std_logic_arith \
    CORE65$::env(library).all \
    CLOCK65$::env(library).all
    "


set vhdlout_dont_create_dummy_nets "false"
set vhdlout_bit_type  "std_logic"
set vhdlout_bit_vector_type "std_logic_vector"
set vhdlout_single_bit "VECTOR"
set vhdlout_preserve_hierarchical_types "VECTOR"
set vhdlout_dont_write_types "true"
set vhdlout_write_components "true"
set verilog_no_tri "true"
set compile_fix_multiple_port_nets "true"

# this is where the vhdl files are read in and analyzed
source $::env(algorithm_specifics_script)

check_design > \
    $::env(report_dir)/$::env(synth_file_base).check_design
link > \
    $::env(report_dir)/$::env(synth_file_base).link

set clock [get_ports clk]
set uncertainty [ expr $::env(period) * 0.02 ]

create_clock $clock -period $::env(period)
set_clock_uncertainty $uncertainty $clock
set_fix_hold $clock

if {$::env(use_transition) == "1"} {
  set_clock_transition $::env(transition) $clock
  set_dont_touch_network $clock
}

if {$::env(use_io_delay) == "1"} {
    set_input_delay \
        -max $::env(input_delay) \
        -clock clk [get_ports data_in*]
    set_output_delay \
        -max $::env(input_delay) \
        -clock clk [get_ports data_out*]
}

set_drive 0 $clock

uniquify

compile \
    -boundary_optimization -ungroup_all \
    -map_effort high -area_effort high
```

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

change_names -rules verilog -hierarchy

report_constraint -all_violators > \
    $::env(report_dir)/$::env(synth_file_base)
        .report_constraint
report_area -hierarchy > \
    $::env(report_dir)/$::env(synth_file_base).report_area
report_clock -attributes > \
    $::env(report_dir)/$::env(synth_file_base).report_clock
report_timing > \
    $::env(report_dir)/$::env(synth_file_base).report_timing

write -hierarchy -format ddc \
    -output $::env(data_dir)/$::env(synth_file_base).ddc
write -hierarchy -format verilog \
    -output $::env(data_dir)/$::env(synth_file_base).v

write_sdf $::env(data_dir)/$::env(synth_file_base).sdf
write_sdc $::env(data_dir)/$::env(synth_file_base).sdc

quit
```

A.6.2 Static Timing Analysis Script

02_postsynth_pt.tcl

```
remove_design -all
set power_enable_analysis true

set stmpath "/usr/local-eit/cad2/cmpstm/stm065v536"
set search_path "\
    $::env(STM065_DIR)/CORE65$::env(library)_5.1/libs \
    $::env(STM065_DIR)/CLOCK65$::env(library)_3.1/libs \
    ."

set target_library "\
    CORE65$::env(library)_nom_$::env(vdd)_25C.db \
    CLOCK65$::env(library)_nom_$::env(vdd)_25C.db \
    "

set link_library $target_library

set symbol_library "\
    CORE65$::env(library).sdb \
    CLOCK65$::env(library).sdb \
    "

read_ddc $::env(data_dir)/$::env(synth_file_base).ddc

write_sdf -map "\
    $stmpath/CORE65$::env(library)_5.1/behaviour/verilog/
        CORE65$::env(library).verilog.map \
    $stmpath/CLOCK65$::env(library)_3.1/behaviour/verilog/
        CLOCK65$::env(library).verilog.map \
    " \
    -context Verilog -significant_digits 10 \
    -output $::env(data_dir)/$::env(postsynth_file_base).sdf

report_timing > $::env(report_dir)/$::env(postsynth_file_base)
    .report_timing

quit
```

A.6.3 Placement and Routing Script

03_pnr_en.tcl

```
setDesignMode -process 65

# load config
loadConfig $::env(pnr_config_file) 1

# set some variables
set horizontal_stripe_groups 5
set vertical_stripe_groups 5
set power_stripe_width 1
set power_stripe_spacing 1
set power_ring_offset 1
set stripe_group_width \
    [ expr (2 * $power_stripe_width) + $power_stripe_spacing ]
set margins \
    [ expr 2*$power_ring_offset + $stripe_group_width ]

# floorplan
floorPlan -site CORE \
    -s $::env(pnr_width) $::env(pnr_height) \
```

```
    $margins $margins $margins $margins

# power rings and stripes
set vstripe_distance \
    [ expr ($::env(pnr_width)+2*$stripe_group_width)/(1+
        $vertical_stripe_groups) ]
set hstripe_distance \
    [ expr ($::env(pnr_height)+2*$stripe_group_width)/(1+
        $horizontal_stripe_groups) ]
set vstripe_offset \
    [ expr $vstripe_distance - $stripe_group_width ]
set hstripe_offset \
    [ expr $hstripe_distance - $stripe_group_width ]
set vstripe_layer M6
set hstripe_layer M5

set common_stripe_settings {
  -nets {commongnd vddfilc} \
  -width $power_stripe_width \
  -max_same_layer_jog_length 6 \
  -padcore_ring_bottom_layer_limit M2 \
  -padcore_ring_top_layer_limit M4 \
  -merge_stripes_value 2.5 \
  -block_ring_bottom_layer_limit M2 \
  -block_ring_top_layer_limit M4 \
  -stacked_via_bottom_layer M1 \
  -stacked_via_top_layer AP \
  }

# POWER RING
addRing \
        -nets {vddfilc commongnd} \
        -around core \
        -width_top $power_stripe_width \
        -width_left $power_stripe_width \
        -width_right $power_stripe_width \
        -width_bottom $power_stripe_width \
        -spacing_top $power_stripe_spacing \
        -spacing_left $power_stripe_spacing \
        -spacing_right $power_stripe_spacing \
        -spacing_bottom $power_stripe_spacing \
        -layer_top $hstripe_layer \
        -layer_left $vstripe_layer \
        -layer_right $vstripe_layer \
        -layer_bottom $hstripe_layer \
        -offset_left $power_ring_offset \
        -offset_right $power_ring_offset \
        -offset_top $power_ring_offset \
        -jog_distance $power_ring_offset \
        -offset_bottom $power_ring_offset \
        -stacked_via_bottom_layer M1 \
        -stacked_via_top_layer AP

# POWER STRIPES
if { $horizontal_stripe_groups > 0 } {
  eval addStripe \
    $common_stripe_settings \
    -direction horizontal \
    -spacing \$power_stripe_spacing \
    -ytop_offset \$hstripe_offset \
    -set_to_set_distance \$hstripe_distance \
    -start_from top \
    -layer \$hstripe_layer
}
if { $vertical_stripe_groups > 0 } {
  eval addStripe \
    $common_stripe_settings  \
    -direction vertical \
    -spacing \$power_stripe_spacing \
    -xleft_offset \$vstripe_offset \
    -start_from left \
    -set_to_set_distance \$vstripe_distance \
    -layer \$vstripe_layer
}

# cell placement
setMultiCpuUsage -localCpu 4 -cpuPerRemoteHost 1 \
    -remoteHost 0 -keepLicense true
setPlaceMode -fp false

# prePlaceOpt not mentioned in man pages, but no warnings given
      either
placeDesign -prePlaceOpt
redraw
fit

# clock tree synthesis
clockDesign
    -genSpecOnly $::env(data_dir)/$::env(pnr_file_base).ctstch

setCTSMode -traceDPinAsLeaf false -traceIoPinAsLeaf false \
    -routeClkNet false -routeGuide true\
    -routeTopPreferredLayer M4 \
    -routeBottomPreferredLayer M3 \
```

```
    -routeNonDefaultRule {} -routeLeafTopPreferredLayer M4 \
    -routeLeafBottomPreferredLayer M3 \
    -routeLeafNonDefaultRule {} \
    -useLefACLimit false \
    -routePreferredExtraSpace 1 \
    -routeLeafPreferredExtraSpace 1 \
    -opt true -optAddBuffer true -moveGate true \
    -useHVRC true -fixLeafInst true \
    -fixNonLeafInst true -verbose false -reportHTML false \
    -addClockRootProp false -nameSingleDelim false \
    -honorFence false -useLibMaxFanout true \
    -useLibMaxCap false

clockDesign \
    -specFile $::env(data_dir)/$::env(pnr_file_base).ctstch \
    -outDir clock_report -unfixedInstBeforeCTS \
    -postCTSsdcFile \
    $::env(data_dir)/$::env(postsynth_file_base).sdc

deleteTrialRoute
setOptMode -fixCap true -fixTran true -fixFanoutLoad true
optDesign -postCTS
optDesign -postCTS -hold

# 040 route
# route clock nets with high priority
# and some extra space to reduce coupling
setAttribute -net @clock -weight 5 -preferred_extra_space 1

selectNet -clock
setNanoRouteMode -quiet -routeWithTimingDriven false
setNanoRouteMode -quiet -envNumberProcessor 1
setNanoRouteMode -quiet -routeSelectedNetOnly true
setNanoRouteMode -quiet -routeWithViaInPin true
setNanoRouteMode -quiet \
    -routeWithViaOnlyForStandardCellPin false
setNanoRouteMode -quiet -drouteUseMultiCutViaEffort high

routeDesign -globalDetail

# 050 route
setNanoRouteMode -quiet -routeSelectedNetOnly false
setNanoRouteMode -quiet -routeWithTimingDriven true
setNanoRouteMode -quiet -routeTdrEffort 10
setNanoRouteMode -quiet -drouteFixAntenna true
setNanoRouteMode -quiet -routeWithSiDriven true
setNanoRouteMode -quiet -routeSiLengthLimit 200
setNanoRouteMode -quiet -routeSiEffort normal
setNanoRouteMode -quiet -routeWithViaInPin true
setNanoRouteMode -quiet \
    -routeWithViaOnlyForStandardCellPin false
setNanoRouteMode -quiet -drouteUseMultiCutViaEffort high

routeDesign
# optimization
puts "optimization"
foreach xyzzy {0 1 2} {
    setOptMode -fixFanoutLoad true \
        -holdFixingEffort high \
        -addInst true \
        -addInstancePrefix postrouteoptInst \
        -effort high -fixDrc true\
        -congOpt true

    setAnalysisMode -clockPropagation sdcControl
    optDesign -postRoute -drv

    setAnalysisMode -checkType hold \
        -propSlew true -timingEngine static
    optDesign -postRoute -hold

    setAnalysisMode -checkType setup \
        -timingEngine static
    optDesign -postRoute -si
}
# incremental optimization
foreach yzxxz {0 1 2} {
  optDesign -postRoute -incr
}
# write reports
setAnalysisMode -checkType hold
report_timing -check_type hold -max_paths 1
report_timing -check_type hold -max_paths 10 > \
    $::env(report_dir)/$::env(pnr_file_base).hold.report_timing

setAnalysisMode -checkType setup
report_timing -check_type setup -max_paths 1
report_timing -check_type setup -max_paths 10 > \
    $::env(report_dir)/$::env(pnr_file_base)
        .setup.report_timing

report_constraint -all_violators
report_constraint -all_violators > \
    $::env(report_dir)/$::env(pnr_file_base).report_constraint
```

```
204
205 # add filler cells
206 addFiller -cell \
207     HS65_LH_FILLERCELL4 HS65_LH_FILLERCELL3 \
208     HS65_LH_FILLERCELL2 HS65_LH_FILLERCELL1 \
209     -prefix fico
210
211 # verify and output
212 puts "verification and report generation"
213 clearDrc
214 checkDrc
215 checkRoute
216 verifyGeometry -allowDiffCellViol
217 verifyConnectivity -type all -error 1000 -warning 50
218 verifyProcessAntenna
219 report_power -leakage -outfile $::env(report_dir)/$::env(
        pnr_file_base).report_power
220
221 saveNetlist $::env(data_dir)/$::env(pnr_file_base).v
222 setExtractRCMode -engine postRoute -effortLevel low
223 extractRC
224 write_sdf -precision 5 $::env(data_dir)/$::env(pnr_file_base)
        .sdf
225 rcOut -spef $::env(data_dir)/$::env(pnr_file_base).spef
226
227 # save encounter project
228 saveDesign $::env(data_dir)/$::env(pnr_file_base).enc
229
230 exit
```

A.6.4 Power Analysis Script

05_power_pt.tcl

```
1  #-----------------------------------
2  #---------- power analysis ---------
3  #-----------------------------------
4
5  #Set the Power Analysis Mode
6
7  set power_enable_analysis TRUE
8  #set power_analysis_mode averaged
9  set power_analysis_mode time_based
10 set timing_use_zero_slew_for_annotated_arcs never
11 #time_based/averaged
12
13 #Set the search path and link path
14
15 set search_path "\
16   $::env(STM065_DIR)/CORE65$::env(library)_5.1/libs \
17   $::env(STM065_DIR)/CLOCK65$::env(library)_3.1/libs \
18   $search_path"
19
20 set link_library "\
21   CORE65$::env(library)_nom_$::env(vdd)_25C.db \
22   CLOCK65$::env(library)_nom_$::env(vdd)_25C.db \
23   $link_library"
24
25 set target_library "\
26   CORE65$::env(library)_nom_$::env(vdd)_25C.db \
27   CLOCK65$::env(library)_nom_$::env(vdd)_25C.db"
28
29 set symbol_library "CORE65$::env(library).sdb"
30
31
32 read_db $target_library
33
34 #Read the design and the libraries
35
36 read_verilog $::env(data_dir)/$::env(pnr_file_base).v
37 current_design $::env(top)
38
39 #Link the top design
40
41 link_design
42
43 read_sdc $::env(data_dir)/$::env(postsynth_file_base).sdc
44
45 #Annotate parasitict
46
47 read_parasitics $::env(data_dir)/$::env(pnr_file_base).spef
48
49 #Read switching activities (only for time_based)
50
51 #read_vcd -strip_path tb_iteration_2/iteration_2_test /tmp/
       NR_pnr651.vcd
52 read_vcd -strip_path \
53     $::env(test_bench)/dut $::env(data_dir)/$::env(
          sim_file_base).vcd
54
55 #Power analysis
56
```

```
check_power
update_power

#Report power

report_power -verbose -hierarchy > \
    $::env(report_dir)/$::env(power_file_base).report_power
quit
```

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-491

http://www.eit.lth.se