IMPLEMENTATION AND EVALUATION OF A POLYNOMIAL-BASED DIVISION ALGORITHM

Examensarbete utfört vid Elektroniksystem,
Linköpings Tekniska Högskola
av

Stefan Pettersson

Reg nr: LiTH-ISY-EX-3455-2003
Linköping 2003-06-05
IMPLEMENTATION AND EVALUATION
OF A POLYNOMIAL-BASED DIVISION
ALGORITHM

Master’s thesis project at Electronics Systems,
Linköping University

Stefan Pettersson

Reg no: LiTH-ISY-EX-3455-2003

Supervisor: Per Löwenborg
Examiner: Per Löwenborg
Linköping, June 05, 2003
In comparison to other basic arithmetic operations, such as addition, subtraction and multiplication, division is far more complex and expensive. Many division algorithms, except for lookup tables, rely on recursion with usually complex operations in the loop. Even if the cost in terms of area and computational complexity sometimes can be made low, the latency is usually high anyway, due to the number of iterations required. Therefore, in order to find a faster method and a method that provides better precision, a non-recursive polynomial-based algorithm was developed by the Department of Electrical Engineering at Linköping University.

After having performed high-level modelling in Matlab, promising results were achieved for up to 32 bits of accuracy. However, since the cost model did not take in account other factors that are important when implementing in hardware, the question remained whether the division algorithm was also competitive in practice or not. Therefore, in order to investigate that, this thesis work was initiated.

This report describes the hardware implementation, the optimization and the evaluation of this division algorithm, regarding latency and hardware cost for numbers with different precisions. In addition to this algorithm, the common Newton-Raphson algorithm has also been implemented, to serve as a reference.
ABSTRACT

In comparison to other basic arithmetic operations, such as addition, subtraction and multiplication, division is far more complex and expensive. Many division algorithms, except for lookup tables, rely on recursion with usually complex operations in the loop. Even if the cost in terms of area and computational complexity sometimes can be made low, the latency is usually high anyway, due to the number of iterations required. Therefore, in order to find a faster method and a method that provides better precision, a non-recursive polynomial-based algorithm was developed by the Department of Electrical Engineering at Linköping University.

This report describes the hardware implementation, the optimization and the evaluation of this division algorithm, regarding latency and hardware cost for numbers with different precisions. In addition to this algorithm, the common Newton-Raphson algorithm has also been implemented, to serve as a reference.
ACKNOWLEDGEMENTS

First I would like to thank my supervisor and examiner Per Löwenborg for supporting me in different ways throughout the project. I would also like to thank Oscar Gustafsson for guiding me in the area of multiplication using multiple-constant techniques.

Finally I would like to thank the coffee break crew - Deborah, Jonas, Mikael, Nils and Terese - for a nice fellowship and many interesting discussions in the lunch room.
CONTENTS

1 Introduction ............................................................................. 1
  1.1 Background ......................................................................... 1
  1.2 Purpose and objectives ...................................................... 1
  1.3 Structure of the report ..................................................... 2

2 Arithmetic operations ............................................................. 3
  2.1 Introduction ......................................................................... 3
  2.2 Number representation ...................................................... 3
    2.2.1 Binary numbers ......................................................... 4
    2.2.2 Binary point ............................................................. 4
  2.3 Error analysis and quantization effects .............................. 5
  2.4 Addition ............................................................................ 7
    2.4.1 Carry-propagate addition .......................................... 7
    2.4.2 Carry-save addition ............................................... 8
    2.4.3 Wallace adder tree .................................................. 9
  2.5 Subtraction ........................................................................ 11
  2.6 Multiplication ................................................................... 12
    2.6.1 Shift-and-add multiplier ......................................... 12
    2.6.2 Parallel multiplier .................................................. 13
    2.6.3 Array multiplier ..................................................... 13
    2.6.4 Multiple constant multiplication ............................ 13

3 The polynomial-based algorithm .......................................... 17
  3.1 Introduction ....................................................................... 17
  3.2 The division algorithm ..................................................... 18
  3.3 Architecture ....................................................................... 21
  3.4 Implementation ............................................................... 23
    3.4.1 Implementation strategy ......................................... 23
    3.4.2 Implementation of precalculation block .................... 25
    3.4.3 Implementation of polynomial calculation block ...... 27
    3.4.4 Pipelining .............................................................. 29
3.5  Design and optimization of architecture ...................... 30
   3.5.1  Error analysis and optimization .......................... 30
   3.5.2  Optimization technique ..................................... 31
   3.5.3  Optimization problems ....................................... 32
   3.5.4  Optimization procedures ................................... 33
   3.5.5  Optimal solution choice ..................................... 40
3.6  Generation of architecture ......................................... 41

4  The Newton-Raphson algorithm ................................... 45
   4.1  Introduction .......................................................... 45
   4.2  The division algorithm ......................................... 47
   4.3  Architecture .......................................................... 49
   4.4  Implementation ...................................................... 52
      4.4.1  Implementation strategy ................................. 52
      4.4.2  Problems due to normalized numbers ............... 53
      4.4.3  Implementation of lookup table ....................... 55
      4.4.4  Implementation of divider ............................. 56

5  Generation, synthesis and evaluation ............................. 57
   5.1  Test and evaluation strategy ................................ 57
   5.2  Newton-Raphson divider test ............................... 58
   5.3  Polynomial-based divider test .............................. 60
      5.3.1  Optimization and generation ......................... 60
      5.3.2  Synthesis results ........................................ 63
   5.4  Comparison and analysis ..................................... 66
   5.5  Further tests ...................................................... 67

6  Conclusions and future work ....................................... 71

7  Bibliography .............................................................. 75
INTRODUCTION

1.1 BACKGROUND

In comparison to other basic calculations, such as addition, subtraction and multiplication, division is a complex and expensive operation. Many division algorithms, except for lookup tables, which are practical for low accuracy division, rely on recursion with usually complex operations in the loop. Even if the cost in terms of area and computational complexity could be made low, the latency is usually high anyway, due to the number of iterations required. Therefore, in order to find a faster method and a method that provides better precision, a non-recursive polynomial-based algorithm was developed in [1].

After having performed high-level modelling in Matlab, which included calculating the number of basic multiplications, promising results were achieved. However, since the cost model did not take in account other factors that are important when implementing in hardware, the question remained whether the division algorithm was also competitive in practice or not. Therefore, in order to investigate that, this thesis work was initiated.

1.2 PURPOSE AND OBJECTIVES

The purpose of this thesis work is to implement the polynomial-based algorithm in hardware as well as implementing the common Newton-Raphson algorithm to serve as a reference. The models shall be generic and after having defined parameters such as for instance precision, input wordlength and polynomial order for the polynomial-based algorithm, appropriate VHDL files shall be automatically generated. Then both algorithms shall be synthesized in the standard cell library AMS CSX 0.35 μm for some wordlengths. Finally, both the algorithms shall be compared.
The main objective is to prove whether the actual polynomial-based algorithm is competitive or not, regarding latency and chip area. Other objectives are that a VHDL generator and an optimizer shall be finished, so that this divider can be used at later occasions. The implementation shall be parameterizable and support pipelining.

The objective to get a fast divider is more important than to get a cheap one, therefore that is also the priority order when choosing between different ways of implementation.

1.3 STRUCTURE OF THE REPORT

- *Chapter 2:* This chapter explains the theory of different arithmetic operations and number representations. In addition to that, error analysis and quantization effects are described.

- *Chapter 3:* This chapter explains the theory of the polynomial-based division algorithm, a proposed implementation of a divider using it as well as the optimization technique used for designing an appropriate structure according to the desired precision.

- *Chapter 4:* This chapter explains the theory of the Newton-Raphson algorithm and a proposed implementation of a divider using it.

- *Chapter 5:* This chapter presents the results of the optimization, generation and synthesis of dividers with different precisions. Finally, both divider types are analyzed and compared.

- *Chapter 6:* This chapter summarizes the results and presents the conclusions of this thesis work.
This chapter explains the theory of different arithmetic operations and number representations. In addition to that, error analysis and quantization effects are described.

2.1 INTRODUCTION

To carry out division of a number there is a need for other basic arithmetic operations, such as addition, subtraction and multiplication - in theory as well as in hardware implementations.

The complexity of the three kinds of operations is different. The first two are relatively simple, whereas the third requires some more resources, even if multiplication still can be done in a relatively straightforward way using combinations of adders.

However, division is a little bit trickier to perform, since the operation is non-linear and cannot be calculated directly by adding differently right-shifted terms of the input data.

To be able to investigate the division operations and algorithms further, we must first have a look on the fundamental operations and the basic theory on how numbers are represented.

2.2 NUMBER REPRESENTATION

One important issue to consider, when dealing with arithmetic operations, is the number representation. The choice of which one to use stands in close relationship to the hardware architecture of the whole system and its interconnections, as well as the desired input and output data.
2.2.1 BINARY NUMBERS

In this thesis both ordinary binary number and 2’s-complement numbers are used, depending on whether the numbers are defined for negative numbers or not. The reason for using 2’s-complement representation instead of signed numbers is that it is convenient - subtraction can for example be carried out just by adding positive and negative numbers, without regard to any sign bit. However, when the numbers are only defined for positive numbers, the unsigned data format has been used. The reason for that is that it requires one data bit less and therefore implicates less hardware. Occasionally, when one operand is always positive and one is signed, a mix of the data types has been used, where the interconnections still have the original type and the positive one has been converted and widened first within the actual calculation block.

2.2.2 BINARY POINT

Another way to classify number representation is according to how the numbers are represented in hardware due to the binary point. In this thesis work one finds two types - the fixed-point representation and the floating-point representation [3, 8].

Fixed-point numbers has, as it sounds, the binary point fixed to a specified position. When for instance multiplying 0.75 with 0.125, one would make the following calculation:

\[ 0.11 \times 0.00110 = 00.001110 \]

The number of bits in the integer part is equal to the sum of the bits in the corresponding positions of the operands. After having calculated the solution, the answer’s wordlength must in turn be considered and if needed truncated to an appropriate length.

The second type is the floating-point numbers, which consists of three parts - a sign bit, a mantissa and an exponent. The main advantage of floating-point numbers in comparison to fixed-point numbers is that they are more suited for very small and very large numbers, since the wordlengths can be held shorter. The reason for that is that a number does not have to contain bits that cover the number’s whole range, but fewer decimals that are shifted according to an exponent.
In this thesis, floating-point numbers are only used for positive numbers. Therefore the sign bit is further on neglected. Hence the form of the number is

\[ p = m \cdot 2^E \quad (2.1) \]

where \( m \) is the mantissa and \( E \) is the exponent.

Distinguishing for the mantissa is that it is always written with a one-valued digit on the left side of the binary point - a digit which is many times considered as obvious and expected and therefore not always explicitly written in implementations where it is not needed. The mantissa is defined between one and two, or between 1.00000000 and 1.11111111 assuming that the mantissa has the length nine bits.

When calculating with floating-point numbers, operations are carried out separately on the exponent and the mantissa. In case two numbers are added or subtracted, the exponents of the terms must first be shifted until their exponents are equal. Then both the mantissas are added or subtracted.

For multiplication and division, operations on the mantissa can be made directly without any consideration of the exponents. The exponents are in this case added or subtracted in parallel, respectively.

In the division case, the following operation is performed

\[ Q = \frac{p_a}{p_b} = \frac{m_a \cdot 2^{E_a}}{m_b \cdot 2^{E_b}} = \frac{m_a}{m_b} \cdot 2^{E_a - E_b} \quad (2.2) \]

where \( p_a \) and \( p_b \) represent floating-point numbers.

Finally, the result must be normalized to the same form as previous, in other words so that the integer bit covers the leftmost non-zero bit of the mantissa [3, 8].

**2.3 ERROR ANALYSIS AND QUANTIZATION EFFECTS**

One problem that might occur, when realizing numerical computations in hardware, is errors due to the fact that hardware is non-ideal and can only use finite wordlengths.
Chapter 2 - Arithmetic operations

When talking about errors, there are two types of errors that could arise during arithmetic computations. The first kind is called *generated error*. This error is generated by the hardware itself, because of for instance finite wordlengths. The second type is called *propagated error*. This error is not generated by the hardware, but propagated through it. Example on this error are the error of initial approximations and errors in coefficients that are fed to the arithmetic [7].

There are three kinds of errors that can arise in this project due to quantization effects [11]

- **Roundoff errors** - occur because of rounding and truncation in the hardware.
- **Coefficient errors** - occur because coefficients can only be represented with finite precision.
- **Overflow of the number range** - occur on the output because the available range is exceeded. In the following implementations, such an error occurs when dividing with zero, where a singular point is found.

When doing error analysis, the quantization errors can be modelled by a simple linear model of a sum of the actual signal and a noise source [10, 11], see figure 2.1.

\[
\begin{align*}
   x_Q(n) &\xrightarrow{a} [Q] \rightarrow y_Q(n) \\
   x_Q(n) &\xrightarrow{\Sigma} y_Q(n)
\end{align*}
\]

*Figure 2.1:* Linear noise model for quantization.

The quantization of a product

\[
y_Q(n) = [a \cdot x_Q(n)]_Q
\]

(2.3)

can in that way be modelled by an additive error

\[
y_Q(n) = a \cdot x_Q(n) + e(n)
\]

(2.4)
The noise source $e(n)$ can normally be assumed to be additive white noise, which is independent of the signal itself. In other words, $e(n)$ is a stochastic process with a density function that can often be approximated by a rectangular function. The average value $m$ of the noise source is

$$m = \begin{cases} 
\frac{Q_c Q}{2} & \text{(rounding)} \\
\frac{Q_c - 1}{2} Q & \text{(truncation)} 
\end{cases}$$

where $Q$ is the quantization step [10, 11].

The most interesting thing to study in this thesis work is however the maximum error, since the implemented architectures must always present correct outputs, which means that the different parts are never allowed to give errors that are larger than the stipulated error limits.

The largest error when truncating after bit $n$ occurs when all bit-positions from $n+1$ to infinity are filled with ones, which corresponds to a number that asymptotically approaches the value $2^n$. When rounding, the maximum error is reduced with a factor two, since the rounding interval is half the size on both sides of the approximated number.

Therefore, the maximum value of the quantization error is

$$\varepsilon_{\text{max}} = \begin{cases} 
2^{-(n+1)} & \text{(rounding)} \\
2^{-n} & \text{(truncation)} 
\end{cases}$$

\[ \text{(2.6)} \]

### 2.4 ADDITION

#### 2.4.1 CARRY-PROPAGATE ADDITION

A carry-propagate adder (CPA) is an adder that calculates the sum bits sequentially, starting with the least significant term bits. The carry generated from one step is passed to the calculation of the next bit and so on [10].
The simplest form of this adder is the **ripple-carry adder**, which consists of a series of full adders, connected to each other with their carry lines.

![Figure 2.2: 4-bit carry-propagate adder.](image)

Each of the full adders is calculating its sum bit and carry bit simultaneously, by evaluating the functions

\[
S_i = U_i \oplus V_i \oplus C_i \quad (2.7)
\]

\[
C_{out,i} = U_i \cdot V_i + U_i \cdot C_i + V_i \cdot C_i \quad (2.8)
\]

However, since the output of a succeeding full adder is dependent on the carry output from the preceding full adder, the calculation of the sum is not finished until the carry signals have propagated through the whole CPA. Given the wordlength $W_d$ of the input terms, the speed of the adder will be determined by the carry-propagation time, which corresponds to $W_d$ times the calculation time of $S$ and $C_{out}$. Hence, the order of this adder is $O(W_d)$.

Looking at the hardware cost we see that the calculation will require $W_d$ fulladders. In other words, the area cost has also the order $O(W_d)$ [10].

### 2.4.2 CARRY-SAVE ADDITION

When adding many terms the CPA is inefficient seeing to the computation time. Therefore, the **carry-save adder** (CSA) has been introduced, which is more suitable when adding three or more operands [7, 10].

Instead of adding the words serially in pairs and propagating the carry at each step, all words are added in parallel, bit for bit in a number of stages, saving the carries until the last stage of the calculation.
The carry-save adder is basically a 3-to-2 reducer, which takes three operands and reduces them to two - to one partial sum and one partial carry. That means that by connecting a number of CSAs, the number of terms will successively decrease until two terms remain. Those are then added by a CPA.

An n-bit carry-save adder essentially consists of n full adders in a row, without any interconnections, see figure 2.3.

![Figure 2.3: 4-bit carry-save adder.]

Since the generated carry actually is supposed to control the left-hand bit in each partial addition, the carry output is shifted one step to the left. By doing that, they can be added as they are to any arbitrary adder step, without any further adjustments.

In other words, each CSA is computing the partial sums

\[ U + V + W = S + 2 \cdot C \]  \hspace{1cm} (2.9)

Finally, the two remaining outputs are added by a CPA or similar to one single output sum.

### 2.4.3 WALLACE ADDER TREE

As explained in the previous part, CSAs are connected to each other when building a complete adder. However, there are some different ways to connect them.

The simplest form is a *CSA-1 adder tree* [5, 7], in which one new term is added for every CSA block, as shown in figure 2.4.
For numerous terms, one sees that the latency is much shorter than for the CPA. The propagation time through the CPA is equivalent to that previous explained, but instead of having to add the time for a whole CPA for every added term, in other words that equal of $W_d$ full adders, the latency of only one single full adder can be added at the time.

Thinking a little bit deeper, one realizes that this adder structure is far from optimal when building a fast adder. By for instance adding two terms per CSA instead of one, thereby building a so called CSA-2 adder tree [5], the tree depth and the latency can be further reduced.

However, the most optimal adder is the so called Wallace tree, where the tree is initially made as wide as possible, see figure 2.4. This implicates the highest level of parallelism, the fastest term reducing rate and the lowest possible tree depth.

The latency is determined by the same parameters as for the CSA-1 adder, but in this case the tree depth becomes significantly lower when the number of terms increases.

If the number of terms is $k$, the tree depth can be approximated by [7]

$$h = \left\lceil \log_{\frac{3}{2}} k \right\rceil = \begin{cases} \frac{\log_2 k}{\log_2 \frac{3}{2}} \\ \frac{\log_3 k}{\log_3 \frac{3}{2}} \end{cases} \quad (2.10)$$

Then, by assuming that the terms’ length is $W_d$, the latency is

$$\tau_{Wal} = h \cdot \tau_{CSA} + \tau_{CPA} = (h + W_d) \cdot \tau_{FA} \quad (2.11)$$

Hence, the calculation time of the Wallace tree adder is of order $O(\log_{\frac{3}{2}} k)$.

The drawback with the Wallace tree adder, however, is that it might require large chip area due to the sometimes complex wiring needed. The order of the hardware cost for this adder is $O(W_d \cdot k)$ [7].
Figure 2.4: A 6-term CSA-1-adder (left). A 9-term Wallace tree adder (right). As can be seen in the figure their depths are equal. Each arrow represents a vector of bits.

2.5 SUBTRACTION

By using 2’s-complement’s representation positive and negative numbers can directly be added, using any of the above mentioned adder blocks or others. The subtraction term is negated by inverting the corresponding input [7, 10].

One important issue to consider when adding multiple terms in an adder tree or similar, where some of the numbers are negative, is that the ones at the most significant bits of a negative number cover the whole row to the most significant bit of the intermediate result’s width, so that the sign would not be wasted anywhere throughout the calculations, due to wrong interconnections.
2.6 MULTIPLICATION

2.6.1 SHIFT-AND-ADD MULTIPLIER

Multiplication with binary numbers can be carried out in the same way as for decimal numbers, by successively left-shifting a factor and adding it to an intermediate sum, in case the corresponding bitposition in the second factor is one. In other words, the sum is obtained by adding partial products to a successively accumulated sum, either in rows or in columns, depending on which factor that is used in which role.

For 2’s-complement numbers, however, there is a special feature to consider regarding the most significant bit of negative numbers, so that the sign will be kept and the product correct calculated. In this case, when reaching the most significant bit, the other factor is subtracted instead of added [8, 10].

Say we are to calculate

\[ z = y \cdot x \]  

Then the product can be written as

\[ z = y \cdot \left( -x_w + \sum_{i=0}^{W_d-1} x_i \cdot 2^{-i} \right) = -y \cdot x_w + \sum_{i=0}^{W_d-1} y \cdot x_i \cdot 2^{-i} \]  

Since this multiplier generates the bit-products sequentially and accumulates them successively, this is the slowest kind of multiplier. Therefore we should have a look on some other multiplier types. The latency of this shift-and-add multiplier is proportional to \( O(W_d^2) \) [10].

The shift-and-add multiplier can be realized either in bit-parallel or bit-serial arithmetic. In the first case, the adders are stacked serially on each other, each with a gate connected to the corresponding bitposition in the second factor, which controls whether the first factor or a zero should be added.

The bit-serial version consists of one adder and a register that holds the intermediate sum, which are always fed back to one of the adder’s inputs. This multiplier also requires some multiplexing logic to control which bit of the second factor that is controlling the addition. Due to the small amount of hardware, this multiplier can be made really small, but will on the other hand require several clock cycles to be completely run.
2.6.2 PARALLEL MULTIPLIER

There are three distinguishing features for the parallel multiplier - partial product generation, carry-free addition and a final carry-propagation addition. There are a number of ways to implement this kind of multiplier. One example of how to generate different partial results and connect the different blocks together is the Wallace multiplier [10].

This multiplier simply consists of a Wallace tree adder with a gate on each input, which controls whether the factor or a zero should be added. The factor connected to each term input must, however, be logically shifted to the right according to the corresponding bit position of the second factor that controls which terms that should be added.

The latency of the Wallace tree multiplier is proportional to O(\log_2(W_d)) and the chip area is proportional to O(W_d^2).

Because of its high degree of parallelism the Wallace tree multiplier is the fastest kind of multiplier, but also the most expensive one regarding chip area due to many blocks and irregular routing [5, 7, 10].

2.6.3 ARRAY MULTIPLIER

In addition to the two previously described types, there is a third kind of multiplier called array multiplier, which occurs if for instance a CSA-1 adder tree is used instead of a Wallace tree in section 2.6.2. This construction is similar to the multiplier called Baugh-Wooley’s multiplicator. Because of the structure, this is a slower multiplicator, though, with a latency proportional to O(W_d) [10].

However, further explanations on this subject are neglected, since this type is not used in this project. It is mentioned though, since it is closely related to the previously mentioned adder and multiplier types.

2.6.4 MULTIPLE CONSTANT MULTIPLICATION

The latency and area of multiplications with constants can be made smaller in comparison to multiplications with variables, since the constant’s zeros can be immediately and permanently removed from the multiplicator, manually or by optimizing in hardware synthesis.

When multiplying a number with multiple constants, it can be profitable to
have a look on the whole system of all multipliers, to see if the overall area can be significantly reduced by eliminating common subexpressions in the factors. When applying multiple constant techniques, the coefficients are factorized and a set of common subexpression are generated, which are only calculated once instead multiple times. Then they are added or subtracted to other subexpressions, either as they are or shifted a number of steps, in order to form new subexpressions [1, 10].

Some of the subexpressions are finally sent to the output lines, either directly or after having been shifted a number of times.

An example of how to perform multiplications with this technique is shown in figure 2.5, where the task is to multiply a number with the constants 7, 106 and 210.

![Figure 2.5: Graph for multiplication with 7, 106 and 210.](image)

In figure 2.5, the nodes represent addition or subtraction of two subexpressions. Each of the branches corresponds to a multiplication with a potens of two, which means either to use the number as it is or to right-shift it a number of steps.

When the tree is generated, the products are extracted either directly from a node or after having been shifted. In the above example, the products from the multiplications with the factors 7 and 106 are directly fetched from the nodes, whereas the multiplication with 210 is also shifted one time before the value reaches the output.

The complicated part with this method is to find out which nodes and subexpressions that should be used and how the interconnections should be drawn, or in other words, how to design the multiplier tree. However, there is a number of heuristic methods developed for finding the optimal solution and for
generating the corresponding interconnection tables.

Two examples on heuristics are the Bull’s and Horrock’s modified method (BHM) and the Reduced Adder Graph method (RAG-n). The difference between them is depending on what factor that is measured and which properties the included coefficients have. It has been shown that the RAG-n algorithm is better regarding the number of additions for small coefficient sets, but it is on the other hand slower than the BHM algorithm. However, for larger coefficients the RAG-n is faster. The optimal number of bits for the RAG-n algorithm is around eight. One drawback though, is that the included lookup tables are generated by an algorithm, which at present only covers the range up to 4096, in other words lengths up to 13 bits [1].
This chapter explains the theory of a polynomial-based division algorithm, developed by the Department of Electrical Engineering at Linköping university, as well as a proposed implementation of a divider using it.

3.1 INTRODUCTION

Division is a complex and expensive operation in comparison to multiplication and addition. Many division algorithms, except for lookup tables, which are practical for low accuracy division, rely on recursion with usually complex operations in the loop. Even if the cost in terms of area and computational complexity can be made low, the latency is usually high anyway, due to the number of iterations required. Therefore, in order to find a faster method and a method that provides better precision, this polynomial-based algorithm was developed [4].

The function of the algorithm is to approximate the inverse $1/D$ to a number $D$. From this result a complete fraction can be easily calculated by multiplying with a numerator. In that case, however, considerations must also be taken to accomplish correct rounding, due to the wordlengths of the included numbers and the desired precision [6, 9].

Shortly described, the proposed algorithm produces the quotient in two steps, according to figure 3.1. The first step is essentially to sum a number of curves, which represent the inverse of a prescaled dividend, rounded to the closest value below that is a pothens of the base two. Finally, each of them are multiplied with some algorithm specific constants.
This first block produces an initial approximation $Y$ of the quotient, which is refined in the second block by polynomial interpolation. The required order of the polynomial is determined by the precision of the first approximation and the desired precision on the output.

![Diagram of the polynomial-based divider](image)

**Figure 3.1:** The two main blocks of the polynomial-based divider

According to [4] this algorithm is shown to be competitive to conventional iterative algorithms like Newton-Raphson for up to 32 bits of accuracy. For example, using Newton-Raphson with less than 12 bits of accuracy would require 33% more general multipliers than the proposed algorithm to achieve 24 bits of accuracy.

### 3.2 THE DIVISION ALGORITHM

Assume that we want to calculate the fixed-point number reciprocal

$$Q = \frac{1}{D} \quad 1 \leq D < 2^{W_d}$$

(3.1)

As previously mentioned the precalculation step basically sums a number of curves, which all represent some kind of scaled inverse to the prescaled dividend. The essential part of this step is the block producing the function

$$S_0(X) = 2^{\left\lfloor \log_2(X) \right\rfloor} = 2^{-i}$$

$$2^i \leq X < 2^{i+1}, \quad i \geq 0$$

(3.2)

This block extracts the largest power-of-two factor in $X$ and calculates the reciprocal of it, see example 3.1. This would not provide an approximation of $1/X$, but since the $X$ value is rounded downwards, the reciprocal would in most cases be larger than the correct solution. Therefore the number must be scaled to produce a reasonable solution.
EXAMPLE 3.1

How the \( S_0 \) function works is easy to illustrate by looking at some numbers.

\[
\begin{align*}
00000001 & \rightarrow 00000001 \rightarrow 1.0000000 \\
00000100 & \rightarrow 00000100 \rightarrow 0.0100000 \\
\end{align*}
\]

The first two dividends, \( \text{one} \) and \( \text{four} \) are both power-of-two factors, which means that the values remain unchanged after the logarithm. Using this value as the exponent in equation 3.2, the inbetween placed value above is just wrapped around. By inserting a binary point after the first digit, the correct reciprocal can be achieved, \( \text{one} \) and \( \text{one fourth} \) respectively.

Let us instead have a look on the case when the dividend is \( \text{seven} \), then

\[
00000111 \rightarrow 00000100 \rightarrow 0.0100000
\]

The rounded logarithm here gets the same value as for the value \( \text{four} \) above. In other words, one could say that this function keeps the leftmost non-zero digit in the number and sets all other non-zero bits to zero. In this case, the reciprocal gets the value \( \text{one fourth} \) instead of \( \text{one seventh} \), which as previously stated is larger than the correct value. In the same way, for 255 the result turns out to be

\[
11111111 \rightarrow 10000000 \rightarrow 0.0000001
\]

Since the function is non-linear and the values \( X \) can be so different, an appropriate scaling can be found for some \( X \), but not for its whole range. However, it has been shown by [4] that by adding multiple differently scaled reciprocal approximations of that kind, a more accurate solution can be achieved. The function of this precalculation block is simply to add an arbitrary number, say \( M \), of branches and performing the above mentioned function, according to figure 3.2.
Mathematically, the function of this block could be described as

\[
Y(D) = d \cdot \sum_{k=0}^{M-1} b_k \cdot S_0(b_k \cdot D)
\]  \hspace{1cm} (3.3)

where

\[
d = 2^{1/M} - 1
\]  \hspace{1cm} (3.4)

and

\[
b_k = 2^{k/M} \quad k = 0, 1, 2, \ldots, M - 1
\]  \hspace{1cm} (3.5)

To get a form that corresponds to figure 3.2, the constant

\[
c_k = d \cdot b_k = (2^{1/M} - 1) \cdot 2^{k/M}
\]  \hspace{1cm} (3.6)

can be extracted from (3.4) and (3.5).

The accuracy of this approximation increases as the number of branches increases, but so will also the chip area. To refine this approximation a polynomial interpolation block is used.

---

Figure 3.2: Function of the precalculation block.
Say we want to calculate the reciprocal $F(Z)$ of a number $Z$. Then it can be approximated by a polynomial

$$P(Z) = \sum_{j=0}^{N} p_j Z^j = F(Z) = \frac{1}{Z}$$

(3.7)

as long as correct polynomial coefficients $p_j$ are found. The error of this interpolating function depends on those coefficients, but also on the polynomial order $N$. The polynomial coefficients can for instance be found by minimizing the Chebychev norm error, in other words by solving the optimization problem

$$\min_{p_j} \varepsilon \quad \varepsilon = \max |P(Z) - F(Z)|$$

(3.8)

In this division algorithm, the dividend $D$ is not fed to the polynomial interpolator, but to the product of the dividend $D$ and the initial approximation $Y$. It has been shown by [4], that by calculating the function

$$G(D, Y) = Y(D) \cdot P(D \cdot Y(D))$$

(3.9)

the divider approximates the reciprocal $1/D$ on $1 \leq D < 2^{W_d}$ with $|G(D, Y) - 1/D| < \varepsilon$, if the polynomial is chosen so that $P(Z)$ approximates $F(Z)$ such that $|P(Z) - F(Z)| < \varepsilon$.

Hence, the accuracy of the precalculation block would also interfere with the error of the polynomial calculation block. Therefore, the precalculation block’s function as well as the chosen number of branches, must also be considered when optimizing the polynomial coefficients.

### 3.3 ARCHITECTURE

The architecture can be divided into two main parts, as shown in figure 3.1. One for calculating the first approximation $Y$ and one polynomial approximation unit for refining the previous result and produce the quotient output.
As illustrated in figure 3.2, the first part of the divider consists of \( M-1 \) branches that each calculates

\[
y_k = S_0(D \cdot b_k) \cdot c_k
\]  \hspace{1cm} (3.10)

As seen in equation (3.10), the architecture of one branch can be partitioned into three parts - a \( b_k \) multiplicator, a \( S_0 \) block and a \( c_k \) multiplicator. Due to the simplicity of the estimation, \( S_0 \) can easily be realized by a combinatorial network like the one illustrated in figure 3.3.

**Figure 3.3:** Architecture of the \( S_0 \) block (left). Example of a 4-bit \( S_0 \) block (right).

Assuming that

\[
\Phi = D \cdot b_k
\]  \hspace{1cm} (3.11)

the above network solves the equation

\[
z_{j-1} = \Phi_j + z_j
\]  \hspace{1cm} (3.12)

where

\[
z_{w_{d-1}} = 0
\]  \hspace{1cm} (3.13)

and

\[
s_{w_{d-j}} = \Phi_j \cdot \overline{z_j}
\]  \hspace{1cm} (3.14)

In the above equations, \( z_j \) are bits of the intermediate word \( Z \) and \( s_{w_{d-j}} \) are bits
of the output word $S_0$.

Since the output from the $S_0$ block is a $W_d + 1$ bit long number that can only have one bit set to one. Therefore, the $c_k$ multiplier can be simply realized as a $W_d+1:1$ multiplexer connected to each output bit of $S_0$, as illustrated in figure 3.4. This multiplexer takes the constant $c_k$ and multiplexes it to the positions beyond the position in the output word that corresponds to the non-zero bit in the $S_0$ word [4].

![Figure 3.4: Implementation of bit $j$ of the multiplexer realizing the $c_k$ multiplication (left). Example of the multiplexer realizing the $c_k$ multiplication of one branch (right).](image)

After having calculated the sum of all branches only multiplications remains, which can be implemented using general multipliers - inside the polynomial approximation block as well as when multiplying the polynomial approximation with the first block’s approximation. No closer discussion about appropriate multipliers to use at different positions will be given here, but will be further discussed in the parts covering the implementation.

### 3.4 IMPLEMENTATION

#### 3.4.1 IMPLEMENTATION STRATEGY

When discussing the implementation there are two separate questions that can immediately be found. The first issue to consider is how the hardware shall be generated for the desired architecture, and second, how to determine which architecture that is the desired.
Chapter 3 - The polynomial-based algorithm

The second question is however more difficult to answer, since the problem has multiple degrees of freedom and would require some kind of optimization procedure to be able to decide the values of all the included parameters, such as the number of branches, the polynomial order and the internal word-lengths. This subject will however be more closely discussed in section 3.5.3.

The implementation problem is, however, partitioned into those two parts. One question that immediately arises is how to connect the hardware generator and this mentioned parameter generator.

One solution is to build a matrix-based VHDL generator in Matlab, which takes parameter matrixes from the optimization block as inputs and generates the corresponding hardware.

The next question on this matter is how many blocks that shall be generated by the VHDL generator and which blocks that shall be made parameterizable. The basic rule that was decided early, is that all hardware that can be made parameterizable should also be made parameterizable, even if it sometimes can be tempting to generate it from Matlab due to the hardware’s complexity. The reason for that decision is the wish to maintain flexibility and exchangeability, to enable sharing of blocks between different parts and to keep the generator code as small as possible and increase the clearness.

A reason for not making all the implementations, except for the above mentioned interface to the optimizer, parameterizable is also that there are limitations in the VHDL language, which cannot handle the sizes of the variables that are needed to calculate various parameters within a parameterizable implementation.

Building blocks of different kinds can however be implemented in various ways and still employ the same functionality. The difference though, can be significant regarding the hardware’s complexity, the required chip area and the latency. Hence, there is a need to decide a policy for how to make trade-offs between size and latency. Both matters are of course important to consider, but since the main purpose is to implement a fast divider, the latency will always be given priority number one. However, when multiple implementations with similar latency are possible or when the fastest implementation will lead to an unreasonable amount of area, the smallest one shall be used.

According to the task, the divider shall be able to be used either as a pure combinatorial logic divider or as a synchronized pipelined divider. Therefore the strategy is chosen to first implement the combinatorial variant and then
introduce pipeline registers into possible positions in the divider, according to flags passed to the VHDL generator.

The order of work in the implementation phase of this thesis work was partitioned into the following parts:

1) Develop a VHDL package with arithmetic operations needed to support the VHDL code and its parameter handling.

2) Implement all single basic building blocks in VHDL.

3) Implement the precalculation block and the polynomial approximation block separately, either parameterizable or by writing a VHDL generator. Finally they are connected.

4) Develop a toolbox with Matlab functions to support the VHDL generator and the optimizing block.

5) Develop controlling and optimizing procedures used for designing the divider architecture and for setting its different parameters.

6) Introduce extra functionality like for instance pipelining.

For every step that has been taken, all implemented functions and all hardware blocks have been carefully tested and verified.

However, this was the general order of work during the project, even if the implementation work in practice has not always strictly followed this order.

3.4.2 IMPLEMENTATION OF PRECALCULATION BLOCK

This block mainly consists of a number of branches with constant multipliers and an $S_0$ calculation block, which are finally added to produce the initial quotient estimation $Y$, as shown in figure 3.5.
Figure 3.5: Implemented architecture of the precalculation block.

All blocks except for the $b_k$ multiplier are written in a parameterizable way and they are connected and supported with their parameters by the VHDL generator.

The $b_k$ multiplicators are implemented using multiple constant techniques and due to the complexity of generating such a multiplier tree, the whole $b_k$ multiplication block is generated by Matlab. The fastest solution to perform the multiplications would be to use a single Wallace tree multiplicator for each multiplication, but as the number of branches increases that would implicate a huge increase of area, since every branch would be equipped with an own multiplicator. The area of a Wallace tree is admittedly large, but the total area would still be very large, also if they were replaced with smaller stand-alone multiplicators.

Therefore, in order to take advantage of common subexpressions in the factors and in that way be able to decrease the area, the multiple constant multiplier is used. The drawback is, however, that the latency of the multiplications increases, but due to the above presented facts, this trade-off seems to be reasonable unless the opposite is proven.

If the tree depth and the latency become too high, there is one way however, to speed up the arithmetic. That is to use carry-save arithmetic to add the different nodes instead of ordinary carry-propagate adders. That implicates that all additions of nodes are performed three and three, with separated carries and sums, which are finally added by a carry-propagate adder at the output.
This structure is far more complex and requires more area than the first one. If the tree depth is small, the version with carry-save adders will also not produce any significant improvement in latency. For those reasons, the decision was to first develop this multiplicator without any carry-save arithmetic and see whether it was needed or not.

Generation of the tree is complex, regarding the problems to know how to connect and shift different nodes in the tree, as well as to know how long the different numbers and adders should be. Hence, a number of Matlab functions were developed to calculate appropriate dimension matrixes for the VHDL generator, going out from the provided table generators, which are using the RAG-n and the BHM algorithms.

The $S_0$ block is constructed as illustrated in figure 3.3. Regarding the $c_k$ multiplication, there is no use of multiple constant techniques, since there is no common input factor to all the multipliers, like the dividend $D$ that was fed to all the $b_k$ multipliers. However, the multiplicator block is constructed in a simple and cheap way, since it is realized basically as a $W_d + 1:1$ multiplexer for each output bit of the $S_0$ block, which is illustrated in figure 3.4.

Finally, due to the numerous branches, a Wallace adder is used to sum all the branches and calculate the output of the precalculation block.

Beside the arithmetic blocks there are a number of quantizers present in the architecture, as can be seen in figure 3.5, which truncate the words to appropriate lengths. Regarding the $b_k$ multiplicator output, all decimals are truncated after the binary point, since the decimals will not have any influence on $S_0$’s value, due to the fact that all numbers are rounded to the closest integer below and that the product’s value cannot be smaller than one. Since $b_k$ is defined in the interval between one and two, the product’s wordlength is fixed to $W_d + 1$, assuming that the dividend $D$ has wordlength $W_d$. Because of $S_0$’s function, the length of its output is fixed and set to the equal length $W_d + 1$.

The rest of the quantizer lengths are determined externally by the optimizing function and forwarded to the VHDL generator. All numbers in this block are unsigned.

### 3.4.3 IMPLEMENTATION OF POLYNOMIAL CALCULATION BLOCK

This block mainly consists of one adder and a number of multipliers. Since the number of multipliers is limited, the area of the whole block is also lim-
ited, even if large multipliers are used. Therefore, in order to obtain the fastest possible hardware, Wallace trees are used in every block, in the multipliers as well as in the adder.

The different kind of VHDL blocks that are used in this implementation are shown in figure 3.6.

Figure 3.6: Implemented architecture of the polynomial approximation block.

The $p_k$ multiplicator is realized by the VHDL generator, by adding appropriately shifted $u_k$ vectors for every non-zero position in the $p_k$ vector. Since all zeros of the $p_k$ vector are removed and not brought into the multiplicator, the tree can be made as small as possible and its depth be minimized.

Since the coefficient $p_k$ can carry both a positive and a negative sign, measures have been taken for both the multiplications and the final addition to be performed correctly. Therefore, all vectors that are handled from the $u_k$ inputs of the $p_k$ multiplier blocks to the output of the adder are dimensioned as 2’s-complement carrying vectors. In practice that means that the vectors are equipped with an extra bit in comparison to the other unsigned vectors.

As can be seen in figure 3.6, there are a number of quantizers to control the wordlengths of all the interconnections. In order to control the position of the binary point at every position, the quantizer’s function also holds a generic function for widening the vector at the most significant bit, either logically or arithmetically, depending on whether the vector is aimed for negative numbers or not.

The wordlengths of the integer and decimal part of each internal vector, as well as the wordlengths of the input and output ports to the complete system, are externally determined and provided to the VHDL generator.
3.4.4 PIPELINING

Pipelining is a method to increase the throughput of a sequential algorithm. Since throughput is defined as the reciprocal of the time between successive outputs, the number of performed calculations per time unit increases when the throughput increases \([10, 11]\).

The idea of pipelining is to split the longest path of a system, the critical path, into smaller paths of equal lengths and introduce a register between them. Since the introduced registers store the intermediate results, a new computation can start before the calculation of an input data is finished. Since each path is shorter, the sample frequency can be increased and, consequently, so can the throughput. One drawback, however, is that the latency is also increased, due to delay of the introduced registers.

---

**Figure 3.7:** Pipelining of three processes in a sequential algorithm.

In figure 3.7, an example of a system with three processes and a total latency \(T_{cp1}\) is shown. By breaking the path into three parts, a latency \(T_{cp2}\) is achieved, which is about one third of \(T_{cp1}\).

Since the throughput is defined as the reciprocal of it, that means that the throughput and the sample frequency can be increased to three times the initial value. If the different paths are not of equal lengths, the sample frequency will be determined by the delay of the longest path.

To support pipelining in the implemented divider structure, a number of optional pipeline registers have been introduced. By passing flags to the VHDL generator, registers are implemented on the corresponding positions and a clock input port is added to the division block, when generating the hardware. No fixed positions have been determined, in order to maintain flexibility and due to the fact that different parameter configurations and divider designs would implicate different partitioning and more or less suitable positions for pipeline registers. Besides, it is not obvious that pipelining should always be introduced into the architecture.

The different positions for possible pipeline registers are shown in figure 3.8.
3.5 DESIGN AND OPTIMIZATION OF ARCHITECTURE

3.5.1 ERROR ANALYSIS AND OPTIMIZATION

In order to achieve solutions on the output that are accurate enough, there is a need for each hardware part to have dimensions that provide sufficient precision. Looking at the architecture in figure 3.5 and 3.6, there are numerous quantizers to be considered, which all add some noise due to their finite wordlengths. This generated error can easily be reduced by using larger wordlengths, but since every extra bit in the words will increase the chip area, there is on the other hand an interest to decrease the wordlengths as much as possible. To be able to make an acceptable trade-off between precision and area, some kind of optimization is required.

The generated error could easily be derived throughout the whole architecture, by adding noise sources as in figure 2.1 to each quantizer. However, because of the large hardware structure, the final expression would be far too complicated to be analytically solved and optimized, due to the numerous
bracketed sums of signal and noise that would be nested in each other throughout the algorithm, if such an analysing work were tried.

Another problem that occurs when considering the generated errors only, is that there are also constant errors that are propagated through the architecture, which will also interfere with the result on the output. In order to minimize the chip area, also the coefficients’ lengths must be reduced as far as possible. The problem though, is to know which parameters to adjust and in which order. By for instance having more accurate hardware with longer internal words, the constants might be made a little shorter, and the desired output precision would still be maintained and the other way around.

One possibility to solve both the problems is to look at the absolute error on the output and introduce an optimizing function that reduces wordlengths without any considerations on if they belong to a coefficient or an internal word.

3.5.2 OPTIMIZATION TECHNIQUE

The method of optimizing that is used is to successively reduce the wordlengths as far as possible, as long as the maximum error on the output is smaller than the maximum allowed. The idea is to reduce all included wordlengths with the value one, for each case determine the maximum error for all possible dividends, and finally examine which of the length configurations that generates the minimal maximum error on the output. Then this length configuration is set as input to the next iteration of the same procedure and so on. When the error on the output reaches the limit and no further reductions are possible, this optimizing step has reached its termination.

Occasionally, there are multiple lengths configurations in the same iteration that implicate equal errors on the output, especially in the beginning when there are a lot of redundant bits existing. In that case, the element to reduce must be randomly chosen to avoid a harder and more systematic reduction from one direction to the other in the corresponding length configuration matrix. Otherwise, there is a risk for the elements to be generally decreased one by one, from the first element to the last, which could lead to greater accuracy loss and destroy the possibility for later elements in the architecture to be optimally decreased.

Depending on in which order the elements are reduced, many different valid lengths configurations can be found, which all represent some kind of subop-
timal solution. Due to the nature of the optimization and the great number of parameters, a global optimum is not likely to be found. Instead, effort must be made to find a suboptimum that is as good as possible.

Hence, the suggestion is to run the optimization procedure several times to produce a bunch of solutions, from which the best one could be picked. Therefore, after having found a solution, the optimization procedure should be restarted a number of times with the recent lengths configuration as input, extended with added random numbers.

This is the general idea behind the two optimization procedures that have been implemented in this thesis work. A closer description on those will be presented later in section 3.5.4.

3.5.3 OPTIMIZATION PROBLEMS

One crucial aspect of the optimizing is that it is time consuming and that it requires a lot of computer resources, due to the size and complexity of the architecture. Regarding this division algorithm, the complexity is also amplified by the fact that the parameters have many degrees of freedom. In addition to the internal wordlengths and coefficient lengths, the polynomial order as well as the number of branches must be optimally chosen.

However, for different reasons this is a tricky part to include in the optimizer. Beside the fact that it would increase the calculation time, it would also require work with length configuration matrixes of different lengths, which are incompatible to use at the same time without making the optimizing procedure more complex.

Second, when synthesizing and evaluating different designs, it can sometimes be desired to be able to set those factors manually. For those reasons, the number of branches and the polynomial order are excluded from the optimization and regarded as fixed parameters. There is one small exception in the VQL optimizer below though, which removes a branch when its wordlength has been totally reduced in the optimization.

A third problem occurs when the input wordlength increases. If the output error is to be validated and verified for all values within the dividend’s interval, in all length reduction steps, the optimization would take a very long time.

Therefore, in order to speed up the optimization, the testvector $X_{\text{test}}$, which is
3.5 Design and optimization of architecture

used in the iterations, is limited to an adjustable number of randomly generated elements. During the iterations a number of successively decreasing length configurations are found, which might not all be valid for all values within the dividend’s interval. In order to find a generally valid configuration, some kind of backtracking procedure can be used to find the cheapest possible solution that is valid for all dividends.

In some cases with large wordlengths, it might be impossible to use a complete testvector in any case, due to the calculation time. According to the performed optimizations in this thesis work, it was found reasonable to use a complete testvector for up to 16 bits of wordlengths in the backtracking procedure.

A fourth problem is to find proper start lengths. Despite the fact that the optimizer successively reduces the wordlengths, the initial values can be crucial for the result. The reason for that is the existence of suboptimums that might occur also for longer wordlengths, which has also been seen during this project.

Related to that, there is an analogous problem to define the constraints of the lengths and to set the interval of the random numbers to be added at each iteration. By adding too large numbers to the next iteration, there is a risk for getting too far from the recent optimum and to find a new optimum with significantly larger wordlengths, which can lead to divergence in the search for smaller lengths.

3.5.4 OPTIMIZATION PROCEDURES

3.5.4.1 GQL OPTIMIZATION

When optimizing, the lengths can be decreased in several ways. One way is to group all constants of a specific type and decrease all of their members simultaneously. This optimization technique is in this report called GQL, an abbreviation for Grouped Quantizer Lengths.

A second way is to let all quantizer lengths be variable and allow each element to be separately reduced. This method is in this report called VQL, which stands for Variable Quantizer Lengths.

In other words, the length elements that the GQL procedure is operating on are those represented in the matrix described in (3.15), where \( N \) is the polyno-
mial order and the variables are those represented in figure 3.5 and 3.6.

\[ \text{Lengths} = \]
\[
\begin{bmatrix}
 b & c & y & p & u_1 & \cdots & u_N & v_1 & \cdots & v_N & w
\end{bmatrix}
\]

The GQL optimization is developed mainly for speeding up the search for valid, but not fully optimal solutions and to being able to quickly find proper start values for the constant lengths and for the different internal wordlengths. The result could either be used directly to generate a hardware architecture or be forwarded to the VQL optimizer.

The optimizer’s function is visualized by the flowchart in figure 3.9.

In comparison to the VQL optimizer, the GQL variant is relatively simple and enables only one single optimization. The testvector $X_{\text{test}}$ could either contain the complete range of dividends or be set to a random vector of arbitrary length. In the second case, a number of additional random numbers can be generated and used in each iteration, in order to increase the possibility to find a non-valid solution that terminates the optimization procedure.

Due to the purpose of this procedure, no backtracking function has been implemented. Hence, when the precision limit is reached, so is the solution. This vector is finally converted to the VQL vector format shown in (3.16) and returned as the procedure’s solution.
3.5 Design and optimization of architecture

Figure 3.9: Flowchart illustrating the function of the GQL optimizer.
3.5.4.1 VQL OPTIMIZATION

This is the main optimization procedure for the implementation, which lets all included length elements be freely variable. Every length configuration are represented by a vector of the VQL format, see (3.16), which uses the same variable names that are illustrated in figure 3.5 and 3.6. The size of the vector depends on the number of branches $M$ and the polynomial order $N$.

\[
\text{Lengths} = \begin{bmatrix}
b_1 & \ldots & b_M & c_1 & \ldots & c_M & y & p_0 & p_1 & \ldots & p_N & u_1 & \ldots & u_N & v_1 & \ldots & v_N & w \\
\end{bmatrix} \tag{3.16}
\]

The VQL matrix is central throughout the whole divider implementation, when optimizing as well as when to pass lengths to the VHDL generator. This vector is used both for the integer part and for the decimal part, which are almost always separately dealt with. In the optimizer, only the decimal part is considered, since this part is the one that has influence on the precision.

During the reduction iterations, the decimal lengths can be decreased down to zero, with exception for the $b_k$ coefficients, which can be decreased down to -1. The integer part of the constants has the length one bit, and by reducing a $b_k$ element to -1, the high-level model would interpret that as a removal of the complete branch.

To be able to use the optimizer for large wordlengths and high precision numbers, both complete and random number testvectors can be generated and used differently at different positions in the optimizer. When a limited test vector is chosen, a number of random numbers are initially generated and added to a test vector that is used in the whole optimizing procedure, optionally together with a number of manually set arbitrary numbers. To increase the possibility to quickly find non-valid solutions and terminate the element reduction procedure, a number of new random numbers can be generated and exchanged in every reduction iteration step, which could speed up the iteration as well as the subsequent backtracking search, due to the generation of fewer possible length configurations that have to be verified.

To maintain flexibility and experimentability, the backtracking procedure is optional to use. For the same reasons as above, its testvector can in the same way be set to use either the complete number space of the dividend or a number of generated random numbers within its range. The reduction iteration and the backtracking function are also illustrated in figure 3.11.
In addition to the block including the reduction iteration and its succeeding backtracking procedure, a second equal block is subsequently added. The reasons for that are to give the possibility to check if the first solution could be further reduced and to give a possibility to experiment with different reduction orders, for instance to reduce a specific group of variables at each time. In the experiments, however, it has been shown that if the first solution is generated by the above mentioned optimization technique, it will not likely be further reduced by using the second optimization block on it. Hence, this block has been disconnected during the final generation of the divider architecture.

In order to find a solution that is as optimal as possible, the optimizer shall be run several times to generate numerous solutions, from which the fastest or the cheapest one can be chosen. The number of iterations is initially set by the user, but the procedure can also optionally be set to terminate when a more expensive solution is found than the preceding one, according to an implemented generic cost function.

Between the optimizer's iterations, the lengths of the elements in the previously generated VQL solution vector are randomly increased, either by giving all VQL elements a random number or by giving a few elements a random number and increase the rest of the elements by a weight of those. Since the code is modular, this function can easily be adapted to the user's preferences, but the default is to choose all numbers randomly every third time and weighted combinations the rest of the times. The random numbers are chosen from a user-specified interval and internal checks guarantee that the length of any element will never exceed the given constraints.

The procedure returns both a complete matrix with the solution in every iteration and a vector with the suggested choice, according to the implemented generic cost function. Returned matrixes and vectors include the length configurations, the corresponding maximum error and the estimated cost.

The function of this optimizer is visualized by the flowchart in figure 3.10 and 3.11.
Start with parameters, startvalues and length boundaries set from outside

Generate VQL startvector if not provided from outside

\[ \text{Lengths} = [b_1 \ldots b_M \ldots v_1 \ldots v_N w] \]

Generate testvector \( X_{\text{test}} \) of random numbers and predefined expected worst-case values, as well as a solution vector.

First reduction iteration step

Use second step?

Yes

Second reduction iteration step

Add random numbers to elements in the \( \text{Lengths} \) vector

Valid solution found?

No

Calculate cost weight

Store solution to file

Yes

Generate testvector \( X_{\text{test}} \) of random numbers and predefined expected worst-case values, as well as a solution vector.

Lengths valid for \( X_{\text{test}} \)?:

No

Iteration limit reached?

Yes

Return the matrixes \( \text{Validlengths} \), \( \text{Validevalds} \) and \( \text{Validcosts} \), with the solutions from all iterations

Figure 3.10: Flowchart illustrating the function of the VQL optimizer.
Figure 3.11: Flowchart illustrating the reduction iteration of the VQL optimizer.
Chapter 3 - The polynomial-based algorithm

3.5.5 OPTIMAL SOLUTION CHOICE

After having finished the optimization run, a number of different length combinations are likely to be available. The question that now arises is how to pick the best one.

The problem is that there are many different elements in the VQL vector, which can all implicate different area for equal lengths, due to the corresponding components’ characteristics. An example on that is the $c_k$ value, which represents a multiplexer with an area linear to the actual length, whereas the $u_k$ value represents a Wallace tree multiplier with an area proportional to the square of the length. Hence, that must be held in mind when choosing.

In order to quickly be able to make the decision on which solution that is the best, the earlier mentioned estimation procedure has been developed. The function is relatively simple; it takes the elements and multiplies the linear ones with coefficients that approximate the gate depth of the actual circuits. In case the hardware’s area is of higher order, a combination of the included lengths are multiplied or added to achieve the appropriate order. This approximation can on one hand be seen to be rough and arbitrary made, but on the other hand it is hard to perform any better estimations without doing any synthesis.

Since the estimation procedure is placed in a single Matlab file, it can easily be modified and refined, for instance by synthesizing each single hardware block separately and creating some linear functions of the area as function of the length. Such an estimation is also inaccurate though, since the synthesis, in practice, works better on the global hardware system. Therefore, the values of these single blocks might not be representative anyway.

However, the cost weight estimation procedure has been shown to perform well under the carried out optimizations. Mostly the proposed solution can also be visually verified to be the best, or a candidate of many similar vectors that have significantly smaller values at every position or at many positions in the vector.

Beside the area, different length configurations also result in different latency. The discussion is analogous with the above area discussion, but generally a vector with smaller lengths could also be expected to be faster. One crucial block in the implementation, however, is the multiple constant multiplier tree, which can have different depths and different number of nodes for different...
length configurations. Therefore, before choosing which solution to use, it is recommended to find out the number of nodes and the tree depth for the possible length combinations. During the tests, however, it has been seen that the suggested solution from the above mentioned cost estimator mostly also is the best solutions regarding this issue. The explanation is that small coefficient lengths usually also implicate small sizes of the tree, but due to the nature of this kind of trees, this cannot be seen as generally true.

The default solution provided to the VHDL generator is the one recommended by the cost estimator, but the vector can also be manually chosen. Regarding the errors, no more considerations have to be made, since all of them fulfil the defined precision, with smaller or greater margin.

### 3.6 Generation of architecture

In the implementation, the optimizer and the VHDL generator are fully integrated with each other. Generation of an architecture can be accomplished by sending a VQL decimal vector directly to the input or by sending a matrix of such vectors and let the generator choose the cheapest one according to the cost function. The first one is useful when experimenting or when using another vector than the cheapest one proposed, for instance when choosing manually according to the depth of the $b_k$ multiplier trees. A matrix or a single vector is also used when reading previously optimized solutions from a file. When no matrix or vector is provided, the optimizer will automatically start and perform as many iterations as specified.

The default when running the optimizer or when supplying the generator with a length matrix or a length vector is also to investigate and document the $b_k$ multiplicator tree’s characteristics, according to the provided flag, which controls whether to use the RAG-n algorithm or the BHM-algorithm. In case the RAG-n algorithm is chosen and the numbers are too large for using it, the flag will be overridden though and a message sent to the user.

When generating the divider architecture, the first step to perform with the chosen VQL decimal vector is to remove redundant branches, in other words branches with $b_k = -1$. Then a new VQL vector must be generated, according to that result and the number of branches must be adjusted.

To be able to completely define the architecture’s wordlengths, a corresponding VQL vector must be generated for the integer part. This is carried out by a
high-level modelling function, which calculates the maximum value at every quantizer, for all values defined in the dividend’s testvector.

By using the VQL vectors for the integers and the decimals, the divider’s hardware can be generated. The blocks that are generated are the $b_1$ multiplier tree, the precalculation block, the polynomial calculation block and a global constant file, as well as the global main file that connects those blocks to each other. Pipelining is introduced by sending vectors with non-zero elements to the generators of the precalculation block and the polynomial block respectively. The non-zero positions in those vectors correspond to the register positions represented in figure 3.8.

The function of the main generator procedure is illustrated in figure 3.12.

When generating an architecture, the different occurring events are logged and the final results as well as the included parameters are stored in a file. To be able to extract parts of the solution during a time consuming optimization and to be able to recover already finished iterations after a manual break or a computer break down, all intermediate solutions are also saved in a file during the optimization.
3.6 Generation of architecture

Figure 3.12: Flowchart illustrating the function of the divider generator.
THE NEWTON-RAPHSON ALGORITHM

This chapter explains the theory of the Newton-Raphson algorithm and a proposed implementation of a divider using it.

4.1 INTRODUCTION

One way to achieve numerical solutions for non-linear functions is to use iterative algorithms such as the Newton-Raphson algorithm, Goldschmidt’s algorithm or using other methods like e.g. the secant method [2].

Significative for the iterative methods is the start with an initial approximated seed value, which can be obtained from for instance a lookup table or from functions generating acceptable approximations. One example of such a function is using polynomial interpolation. The simplest variant is approximating the function with a single linear function, \( y(x) = a + bx \), where \( a \) and \( b \) are constants [7].

After using the algorithm on the more or less crude start approximation a more accurate approximation will be achieved. By repeated use of the algorithm, successively better approximations will be achieved, which will asymptotically converge to the function’s root as the number of iterations grows [6, 7].

In mathematical terms, say we are looking for the root to the function

\[
    f(x) = 0
\]

using the Newton-Raphson formula

\[
    x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)}
\]
What the algorithm is doing is approximating the function with its tangent at the actual $x$ value and estimating the function’s zero point with its tangent’s, see figure 4.1. For every iteration the estimate is moving closer to the exact solution, provided that its convergence criterias are fulfilled [2, 3, 7].

![Figure 4.1: Approximation of the function with its tangent.](image)

Another approach to explaining the algorithm’s function is to look at the first two terms in the Taylor series around $x$. By setting the right side of the equation

$$f(x_i + h) = f(x_i) + h f'(x_i)$$  \hspace{1cm} (4.3)

to zero the variable $h$ can be extracted, which is the correction term to the actual approximation, corresponding to the quotient in equation (4.3) [2].

If the analytical value of the root is $x$, the absolute error of the approximation at the $n$th iteration is

$$\varepsilon_n = |x_n - x|$$  \hspace{1cm} (4.4)

If the initial approximation is accurate enough, the iteration function is convergent and will provide a solution for $f(x)$ so that

$$\lim_{n \to \infty} x_n = x \quad \lim_{n \to \infty} \varepsilon_n = 0$$  \hspace{1cm} (4.5)

However, for practical reasons, only a finite number of iterations is possible to perform. Therefore, an error analysis must be done to find out how many iterations $n$ that must be carried out to achieve solutions such that the
4.2 The division algorithm

requested accuracy will be fulfilled.

The cost for achieving a specific accuracy is however a trade-off between the
number of iterations and the accuracy of the initial value. By passing an ini-
tial value with higher accuracy to the algorithm, fewer iterations must be
done and the other way around.

4.2 THE DIVISION ALGORITHM

Suppose we are to evaluate the quotient $Q = N / D$. First, rewrite the fraction
as a product of the numerator and the reciprocal of the denominator $D$, so that

$$Q = \frac{N}{D} = N \cdot \frac{1}{D} \quad (4.6)$$

Since the multiplication is easy to perform we have now simplified the prob-
lem to solving the reciprocal. After having found a valid solution to this prob-
lem the multiplication with the dividend can be easily performed [6].

However, that last step is not so interesting in this thesis work, seeing that this
operation would not interfere significantly on the efficiency comparison of
the different division algorithms and that this multiplication must take place
in both compared algorithms. Therefore, further on in this report, only the
reciprocal of the divisor will be considered.

As explained in the previous part, the algorithm is used for finding roots to
equations. That means that for using it to find the value of the reciprocal we
have to find a priming function that has its root value corresponding to the
quotient [7].

A function that has its root value $x = 1 / D$ is

$$f(x) = \frac{1}{x} - D = 0 \quad (4.7)$$

By inserting it and its derivative in the equation (4.2) the following Newton-
Raphson formula will be yielded.
The equation (4.8) makes it clear that the operations required for executing one iteration of the algorithm are two multiplications and one subtraction.

The question is now how many iterations that must be done to achieve a specific accuracy. By deriving the error function, the convergence rate can be determined.

In this case, the error can be written as

$$
\epsilon_{i+1} = \frac{1}{D} - x_{i+1}
$$

or by inserting the error of the initial approximation as

$$
\epsilon_i = D^{i-1} \epsilon_0^2
$$

Looking at equation (4.9) above it is clear that the algorithm in this division case converges quadratically, starting from the accuracy of the initial approximation.

In other words, saying that the initial accuracy is one bit, the accuracy for every iteration will be improved like

$$
1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64
$$

By for instance starting with four bits of accuracy, three iterations would be required to achieve 32 bits of accuracy.

The error could of course have both a positive and a negative sign, but that
does not matter for the convergence rate itself. The only crucial thing to consider here is the magnitude of the error, which under no circumstances is allowed to be larger than one [7].

In addition to the previously mentioned error of the algorithm there are some errors generated by the hardware due to the quantization effects. Therefore, a few additional bits must be used in the implementation to keep the error beyond the bit representing the desired final accuracy. Since the algorithm is self-correcting and the occurring errors are beyond the error limit of each iteration, no more attention will be given to this matter, other than adding two extra bits in the implementation as in [3].

4.3 ARCHITECTURE

As previously stated, two multiplications and one subtraction are required to carry out one iteration step of the algorithm. Except for those, there is a need for some additional blocks like for example registers and a generator for the initial seed value.

The design of the divider could be made in several ways, depending on the desired amount of hardware sharing and pipelining. The most straightforward way is just to put the pieces together in the way that equation (4.8) tells. Assuming that the initial approximation is generated through a lookup table that would implicate an architecture like in figure 4.2 below.

![Figure 4.2: A straightforward architecture of the Newton-Raphson divider.](image-url)
The divisor is first loaded into an input register and then forwarded to the address bits of the ROM and the first multiplicator. In the first step the estimate of the quotient is generated by the lookup table resided in the ROM block. After having finished the first iteration step, the result is stored in a register and fed back to the input of the block to be used as the start approximation in the next iteration.

To control the multiplexer and to control when to load the input register, an additional finite state machine is used.

This architecture is, however, far from optimal regarding the amount of hardware and the required area. To decrease the size of the divider only one single multiplicator could be used. That would on the other hand require some more logic like multiplexers, registers and more stages in the controlling finite state machine, but since a multiplication block is relatively expensive in comparison to those blocks the area can still be reduced.

Therefore, an architecture is proposed by [5], which takes advantage of this kind of sharing, see figure 4.3 below. In the figure the dotted arrow shows an example of how the multiplication with the numerator $N$ could be introduced into the architecture.

**Figure 4.3:** A Newton-Raphson divider with one shared multiplier.
The division algorithm is also employed in a similar way in Texas Instrument’s TI 8847 processor [5].

This architecture also reminds of the IBM 360/91 architecture, with that exception that the one in figure 4.3 uses a complete CSA multiplicator. In the other approach, the multiplicator is replaced by a single CSA block and a shift operation, in other words a shift-and-add multiplier. This will further reduce the area, but will on the other hand require more clock cycles, one for every bit in the multiplicand [7].

Further on, by introducing more multiplexers in the structure, sharing can be utilized so that also other calculation operations, e.g. multiplications, can use the same hardware and in that way save area in a larger implementation [7].

Regarding pipelining, the Newton-Raphson divider is not suitable or optimal though. Since the algorithm is iterative and temporary values are used in combination with the initial dividend of the division it is hard to introduce pipelining.

However, by unfolding the structure in figure 4.3 with two multipliers and making it non-iterative, there is a possibility to introduce pipelining though. That would implicate a hardware block for each corresponding iteration step, which are serially connected. In other words, instead of feeding the intermediate result back to the next iteration, it is clocked and forwarded to the next cascade step.

The initial dividend must in this case be stored in a shift register and the value by this way fed to the correct cascade stage at the right moment. An example on such a structure is illustrated in figure 4.4
4.4 IMPLEMENTATION

4.4.1 IMPLEMENTATION STRATEGY

Since the main purpose of this thesis work is to prove whether the polynomial divider is better or not, the first strategy was to develop a less complex structure and first see if it would be beaten by the polynomial variant. If the Newton-Raphson divider was still better, but slightly too large in comparison to the polynomial-based one, there would be an idea to also refine the Newton-Raphson divider and see if it still could be made competitive.
One advantage with the polynomial-based divider is that it is highly suitable for pipelining, which in case of longer latency still might be competitive regarding the throughput. In that case it would be convenient to be able to quickly transform the iterative Newton-Raphson divider to a non-iterative one. Hence, the dual multiplicator variant was the first one to be developed.

4.4.2 PROBLEMS DUE TO NORMALIZED NUMBERS

There is one important problem regarding the comparability between the different algorithms - the different number representations that they are suited for. As previously stated, the polynomial-based algorithm is developed for fixed-point numbers, whereas the Newton-Raphson algorithm is used for normalized floating-point numbers. It is not at all possible to use the Newton-Raphson algorithm for fixed-point numbers, which is illustrated in example 4.1.

EXAMPLE 4.1

The need for the Newton-Raphson’s number to be of floating-point type can easily be proven by looking at for instance an eight bit number with a four-bit initial approximation. The four bits of the initial approximation are all zero due to the large dividend $D$. The fifth non-zero decimal represents half the value of the maximal possible quantization error.

$$D = 10000000$$
$$x_0 = 0.00001$$
$$x_0D = 100.00000000$$
$$2 - x_0D = 101.00000000 < 0$$
$$x_1 = x_0(2 - x_0D) = 0.00101000 > x_0$$

In the above example, we find an intermediate result that becomes less than zero, that $x_1$ is larger than $x_0$ and that it because of this growth will not converge, even if negative intermediate results were allowed.

The table 4.1 shows an example of inputs and outputs of the both algorithms, for a precision of eight bits. In this example as well as in the practical imple-
mentation, the most significant one-valued digit of the mantissa is placed at the first decimal’s place in the number instead of in the integer position. The reason for that is that the normalization, in order to achieve correct solutions, is done before the quotient calculation instead of afterwards.

Table 4.1: Example on inputs and outputs of dividers using fixed-point numbers and floating-point numbers respectively.

<table>
<thead>
<tr>
<th>$D_{\text{Radix 10}}$</th>
<th>$D_{\text{fixp}}$</th>
<th>$Q_{\text{fixp}}$</th>
<th>$D_{\text{float, mant}}$</th>
<th>$Q_{\text{float, mant}}$</th>
<th>$Q_{\text{exp}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000001</td>
<td>1.00000000</td>
<td>0.10000000</td>
<td>1.00000000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>000000010</td>
<td>0.10000000</td>
<td>0.10000000</td>
<td>1.00000000</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>000000011</td>
<td>0.01010101</td>
<td>0.11000000</td>
<td>1.01010101</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>000000100</td>
<td>0.01000000</td>
<td>0.10000000</td>
<td>1.00000000</td>
<td>-2</td>
</tr>
<tr>
<td>7</td>
<td>000000111</td>
<td>0.00100100</td>
<td>0.11100000</td>
<td>1.01001010</td>
<td>-3</td>
</tr>
<tr>
<td>8</td>
<td>000001000</td>
<td>0.00100000</td>
<td>0.10000000</td>
<td>1.00000000</td>
<td>-3</td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
<td>0.00000001</td>
<td>0.10000000</td>
<td>1.00000000</td>
<td>-7</td>
</tr>
<tr>
<td>255</td>
<td>111111111</td>
<td>0.000000001</td>
<td>0.11111111</td>
<td>1.00000001</td>
<td>-8</td>
</tr>
</tbody>
</table>

By studying the above numbers, it is clear that it is not possible to directly compare the two algorithms. The question that arises is if the inputs and outputs really matter or if any of the implementations could give the same output as the other, if the input or output were appropriately modified.

In the first case, say we define the accuracy as eight bits. By applying that desired accuracy to the polynomial-based divider, we see that the double wordlength 16 is required in comparison to the opposite case, which would require 8 bits. To illustrate this, one can study the output when the dividend has the greatest number 255 set.

To get all eight decimals of the Newton-Raphson divider, the polynomial-based implementation must generate the result $0.0000000100000001$, which implicates a great increase of hardware and area. Besides, when converting it to floating-point number, some additional multiplexing logic must be used, since the most significant bits cannot just be truncated. The reason for that is that all bit-positions are required, which is easily proved by setting the dividend to one. The conclusion is that this is far from feasible. Hence, it is not recommended to use the polynomial-based divider for floating-point numbers.
What about defining the accuracy according to the polynomial based algorithm?

In this case we can extract the mantissa and the exponent from the fixed-point number, calculate the quotient and then shift the mantissa logically right to the correct position according to the initial dividend’s exponent. In other words some adapters must be introduced, see figure 4.5.

**Figure 4.5**: Floating-point divider with attached number converting adapters.

This would sometimes also implicate calculation of redundant bits, which are finally wasted. One example is seen when using the dividend 255, when only the first bit of the result is taken care of and the rest is thrown away. However, the maximum wordlength in the implementation never exceeds the desired wordlength, in opposition to the previous case.

The result of using adaptors in the implementation will unfortunately be more area as well as longer delay. However this is required for being able to set up fair comparisons between the two algorithms. Seeing to all the facts presented, this implementation approach is more suitable to use than the first one.

### 4.4.3 IMPLEMENTATION OF LOOKUP TABLE

As previously mentioned there are several ways to generate the initial approximation. Many approximation functions have a specific precision though. In order to be flexible and to be able to generate arbitrary accurate approximations, a lookup table is used in this thesis work. This is also a widely used approach in practice and in the processor examples that are given in this report.

The lookup table is implemented in a ROM, simply like a constant list in a VHDL file, generated by Matlab. The contents of each row are determined by the desired accuracy and half-way reduction of possibly occurring errors [7].
For generating a specific number of bits of accuracy, the ROM must be addressed using the same number of bits. Suppose the desired accuracy is $n$, then the ROM is addressed with the $n$ most significant bits beyond the first digit that is always one. To reduce the truncation error, a one is added to the right-hand position of the least significant bit [7], when feeding the value from the table to the output port. By adding a one-valued digit at that bit-position, half the possible value that might be generated by all the truncated bits from the actual position to the infinity is added, and in that way the possible truncation error is reduced with a factor two.

### 4.4.4 IMPLEMENTATION OF DIVIDER

The implementation is written as parameterizable VHDL code with exception for the ROM block that is generated by Matlab. To be able to carry out the multiplication as quickly as possible and to make the comparison tests fair, the same Wallace tree multiplier block is used as in the polynomial-based algorithm implementation.

The dividend is loaded into the division block by a load signal, which leads the finite state machine to the first state and loads the number into the input register. However, this input register is neglected in the tests, since in reality, an input register as well as an output register would not probably be realized in the same block. The reason for that is that this block would likely be a part of a chain of blocks that each has a register, either in the front or at the end. Therefore, the input block is not used in the synthesis of this implementation.

The function as well as the implementation of the finite state machine is very simple. Therefore, it is not further described in this report.
This chapter presents the results of the optimization, generation and synthesis of dividers with different precisions. Finally, both divider types are analyzed and compared, in order to be able to make conclusions on which one that is the best.

5.1 TEST AND EVALUATION STRATEGY

The main purpose of this thesis is to examine if the polynomial-based algorithm is faster than a general method using the Newton-Raphson algorithm and to investigate the difference in terms of hardware cost.

According to performed high-level tests by [4] the polynomial-based divider has been shown to be competitive for up to about 32 bits of precision. Saying that the desired accuracy is 24 bits, the Newton-Raphson method would require $2^{12}$ addresses and one iteration or $2^6$ addresses and two iterations. In other words that means two or four general multiplications respectively. In case the polynomial order is two, the method would require one general multiplication less than the Newton-Raphson method with two iterations. To achieve 16 bits of accuracy the Newton-Raphson method would require $2^8$ addresses and one iteration. The crucial thing with this method is the size of the lookup table, which must be large for high precision calculations with few iterations or smaller with more iterations and higher latency.

Therefore, the polynomial method is expected to be especially advantageous for long words in terms of for instance latency [4]. For that reason, so is the main hypothesis in this investigation. Hence, the first approach to prove this is to compare its implementation with a simple Newton-Raphson architecture, like the one in figure 4.2, and see if this is enough to reject the hypothe-
sis. If the area of the Newton-Raphson divider would be slightly too large, a refined version, similar to the multiplier sharing divider in figure 4.3 would also be implemented and compared with the others. This divider requires the double number of iterations, but only one general multiplier and consequently a smaller size. One advantage with the first mentioned architecture is also that it is possible to unfold and pipeline, according to the discussion in section 4.3. Finally, a decision will be made on whether to introduce pipelining or not, according to the previously achieved synthesis results.

### 5.2 NEWTON-RAPHSON DIVIDER TEST

The first Newton-Raphson divider that was implemented was the two-multiplier architecture shown in figure 4.2. To avoid double registers at either the input or the output, when cascading multiple hardware blocks after each other, the input register has been removed.

To be able to compare the divider with the polynomial-based one, adapters have been introduced that convert fixed-point numbers to floating-point numbers and vice versa. In the synthesis, however, tests have been made both on dividers equipped with such adapters and without them.

The different values that were achieved in the synthesis without any adapters are shown in table 5.1, where one *mil* corresponds to 25 µm. In other words, one *sqmil* corresponds to 625 µm².

<table>
<thead>
<tr>
<th>$W_d$ [bits]</th>
<th>$W_{d0}$ [bits]</th>
<th>Area [sqmil]</th>
<th>$f$ [Hz]</th>
<th>$T_{rp}$ [ns]</th>
<th>Clock periods</th>
<th>$T_{tot}$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>6</td>
<td>266</td>
<td>27.8</td>
<td>36.0</td>
<td>1</td>
<td>36.0</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>437</td>
<td>24.9</td>
<td>40.2</td>
<td>2</td>
<td>80.4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>468</td>
<td>23.0</td>
<td>43.5</td>
<td>1</td>
<td>43.5</td>
</tr>
<tr>
<td>24</td>
<td>6</td>
<td>940</td>
<td>16.7</td>
<td>59.9</td>
<td>2</td>
<td>59.9</td>
</tr>
<tr>
<td>12</td>
<td>1398</td>
<td>1398</td>
<td>13.2</td>
<td>75.8</td>
<td>1</td>
<td>75.8</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>1655</td>
<td>13.1</td>
<td>76.3</td>
<td>3</td>
<td>228.9</td>
</tr>
<tr>
<td>8</td>
<td>1684</td>
<td>1684</td>
<td>13.2</td>
<td>75.8</td>
<td>2</td>
<td>151.6</td>
</tr>
</tbody>
</table>

Table 5.1: Synthesis results of the Newton-Raphson divider without number converting adapters.
The corresponding values for the architecture with adapters are shown in table 5.2.

**Table 5.2:** Synthesis results of the Newton-Raphson divider with number converting adapters.

<table>
<thead>
<tr>
<th>$W_d$ [bits]</th>
<th>$W_{d0}$ [bits]</th>
<th>Area [sqmil]</th>
<th>f [Hz]</th>
<th>$T_{cp}$ [ns]</th>
<th>Clock periods</th>
<th>$T_{tot}$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>6</td>
<td>300</td>
<td>24.4</td>
<td>41.0</td>
<td>1</td>
<td>41.0</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>491</td>
<td>21.4</td>
<td>46.7</td>
<td>2</td>
<td>93.4</td>
</tr>
<tr>
<td>24</td>
<td>6</td>
<td>1027</td>
<td>14.7</td>
<td>68.0</td>
<td>2</td>
<td>136.0</td>
</tr>
<tr>
<td>12</td>
<td>1487</td>
<td>11.6</td>
<td>86.2</td>
<td>1</td>
<td></td>
<td>86.2</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>1784</td>
<td>12.1</td>
<td>82.6</td>
<td>3</td>
<td>247.8</td>
</tr>
<tr>
<td>8</td>
<td>1813</td>
<td>11.8</td>
<td>84.8</td>
<td>2</td>
<td></td>
<td>169.6</td>
</tr>
</tbody>
</table>

To figure out how large impact the adapters have on the total latency we can simply subtract the delay of one clock cycle of the architecture using the adapter with the corresponding value from the architecture without any adapters. Important though, is to only look at the case when the divider only performs one iteration. The reason for that is that the adapters also affect the clock frequency due to the simplified architecture, even though they are only used once in the beginning and once at the end respectively. Thus, the results in table 5.3 were calculated.

**Table 5.3:** Estimated latency for the two number converting adapters for different precision and different accuracy $W_{d0}$ of the initial approximation.

<table>
<thead>
<tr>
<th>$W_d$ [bits]</th>
<th>$W_{d0}$ [bits]</th>
<th>$T_{adapter}$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>6</td>
<td>5.0</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>8.6</td>
</tr>
<tr>
<td>24</td>
<td>12</td>
<td>10.4</td>
</tr>
</tbody>
</table>

Consequently, we see that the adapters increase the delay with approximately five to eleven nanoseconds, depending on which wordlengths that are used. This corresponds to an increase of the latency by between 14 and 20 percent.
However, the delay of the adapters decreases relative to the whole calculation time, when more iterations are used, due to the fact that they are only used in the interface to the surrounding system and not within the iteration loop.

5.3 POLYNOMIAL-BASED DIVIDER TEST

5.3.1 OPTIMIZATION AND GENERATION

The first step when generating different divider designs is to find out how many branches and which polynomial orders that are required for different values of accuracy. However, the problem is a bit complex, since the two measures are depending on each other as well as on the defined boundaries and the initial values of the coefficients and the internal wordlengths. It has for instance been seen during the tests, that for different sets of boundaries and start values, solutions can be found for smaller number of branches and smaller polynomial order, but not for larger. However, if those values are set too small no solutions will be found at all, independent of how large the wordlengths are.

Due to the multiple degrees of freedom in this problem, no automation in the parameter generation has been introduced. Instead, a manual but systematic hands on method is used, which successively increases the two parameters for different wordlengths until a proven valid solution can be found.

These solutions could be seen as very arbitrary due to the above method, but also because of the fact that it is nearly impossible to find a global optimum for the wordlengths in the optimization. However, since the optimization problem is so complex and therefore unreasonably time consuming when aiming for more accurate solutions, this method is used and the somewhat suboptimal solutions are accepted.

The required parameter lengths that have been found during the tests are shown in table 5.4.
5.3 Polynomial-based divider test

Table 5.4: Required number of branches for different polynomial order and precision.

<table>
<thead>
<tr>
<th>Wordlength $W_d$</th>
<th>Polynomial order $N$</th>
<th>Branches $M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Considering the values in table 5.4, it can be shown that the first calculation step in some cases can be almost removed, for instance when the desired accuracy is twelve and the polynomial order is three. Then this precalculation step would only require two branches and the approximation would mainly be carried out in the polynomial interpolation part.

Trials to find valid solutions have also been made for 32 bits of accuracy, but without any success. Since no solution within reasonable length interval and parameter interval have been found, no further tests on this precision have been done, even though there might possibly be some solution that cannot be found within reasonable time.

Another problem that has occurred is the problem to test whether a solution is valid for all possible values within the dividends defined interval or not, since such tests for large wordlengths would require unreasonably long calculation time. During the tests, it has been found that it is reasonable to perform complete tests for up to 16 bits of wordlength. By increasing the wordlength with one bit, the calculation time would not only be increased with a factor two, but more, due to the multiple times the test vector is used in each optimization step.

Therefore, for all tests above 16 bits a test vector of $2^{16}$ randomly chosen elements within the dividends interval have been chosen instead of the complete test vector. In addition to the testvector, approximately $2^9$ random numbers plus $2^8$ alternating random numbers have been used during the length reduction phase of the optimization.
Using these restrictions, the optimization procedure will unfortunately not provide an optimal solution, but due to the above method, the procedure would probably generate an acceptable solution regarding the order of the size and the delay.

When generating the multiple constant multiplication trees, the BHM algorithm has been used. One reason for that is that it has been seen during the optimizations that the $b_k$ constant lengths are usually within the interval up to ten bits, and since this algorithm is more optimal regarding the latency for small lengths than the RAG-n algorithm, this sounds like a proper choice.

Another reason for that choice is the RAG-n’s limitation of possible wordlengths. Since some of the cases might include some large wordlengths, this algorithm cannot always be used. Therefore, it would be unfair to compare different lengths sets with each other after having used different algorithms. Most fair to the whole divider however, would be to test both algorithms, but due to the above discussion and the fact that the tree depth has been shown to be small in most cases and the fact that the synthesis results show that this part of the architecture represents a relatively small share of the total delay, such operations are considered as unnecessary time consuming work in this thesis.

The tree depth and the number of nodes for each performed generation are shown in table 5.5

**Table 5.5:** Depth and number of nodes in the multiple constant multiplier trees that have been generated for different precisions and parameter sets.

<table>
<thead>
<tr>
<th>$W_d$ [bits]</th>
<th>$N$</th>
<th>$M$</th>
<th>Depth</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>22</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>16</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

As can be seen in table 5.5, the case with 50 branches for 24 bits accuracy has been removed, due to the long optimization time that would be required to evaluate a set with so many variables. In addition to that, it is uncertain that this lower order solution would be able to implicate a better result, since the
results presented in the section 5.3.2 shows that for the tested cases, fewer branches and a higher order would probably be faster.

In every optimization, ten iterations have been performed. By increasing the number of iterations better solutions might be found, but since every iteration is costly in terms of calculation time no more iterations have been performed in this thesis work.

5.3.2 SYNTHESIS RESULTS

After having generated architectures for different parameter sets, those have been synthesized using Leonardo. In order to achieve as good and fast solutions as possible, the preferences are set to maximum effort of optimization and that no hierarchy shall be preserved. Therefore, it can in some cases be difficult to extract the exact delay of a specific hardware block, since multiple blocks have been grouped and globally optimized. Example on that is shown in table 5.6 below, for the case when a precision of twelve bits is desired and the polynomial order is set to three. In that case, no delay value can be found for a single branch. Mostly, however, the blocks stay relatively intact, due to the fact that many blocks already are optimally designed regarding the latency.

All dividers are synthesized using the standard cell library for a 0.35 µm CMOS technology, but since the implementations are independent of technology they would work for e.g. the 0.18 µm technology as well. In this thesis, that does not matter, since the only interesting thing is to compare the different implementations with each other. Some differences can however occur between the different technologies, but not of a size that significantly would interfere with the results and the conclusions.

The result of the synthesis is shown in table 5.6.
Table 5.6: Synthesis results for the polynomial-based divider.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>16</td>
<td>33.4</td>
<td>55.6</td>
<td>89.0</td>
<td>1071</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>15.0</td>
<td>56.4</td>
<td>71.4</td>
<td>623</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>22</td>
<td>28.3</td>
<td>75.0</td>
<td>103.3</td>
<td>2092</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>26.4</td>
<td>85.2</td>
<td>111.7</td>
<td>1495</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>16</td>
<td>44.99</td>
<td>110.0</td>
<td>154.9</td>
<td>3789</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>32.9</td>
<td>116.8</td>
<td>149.7</td>
<td>2862</td>
</tr>
</tbody>
</table>

To figure out which part that requires the longest delay and to be able to find out whether advantage can be taken from pipelining or not, a more extensive study has been made. The different blocks that have been measured as well as the critical path are illustrated in figure 5.1.

![Critical path and the measured subblocks of the divider.](image)

Figure 5.1: Critical path and the measured subblocks of the divider.

The synthesis results for the different parts are shown in table 5.7.

Table 5.7: More extensive results of the delay values.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>16</td>
<td>27.5</td>
<td>6.1</td>
<td>13.0</td>
<td>10.7</td>
<td>7.5</td>
<td>24.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>15.0</td>
<td>8.5</td>
<td>21.2</td>
<td>12.5</td>
<td>14.2</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>22</td>
<td>13.1</td>
<td>15.2</td>
<td>15.7</td>
<td>16.2</td>
<td>11.9</td>
<td>31.2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>14.2</td>
<td>12.2</td>
<td>14.1</td>
<td>31.5</td>
<td>11.2</td>
<td>28.6</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>16</td>
<td>26.0</td>
<td>19.0</td>
<td>18.1</td>
<td>38.8</td>
<td>13.8</td>
<td>38.8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>14.8</td>
<td>18.1</td>
<td>14.6</td>
<td>52.7</td>
<td>23.0</td>
<td>26.4</td>
</tr>
</tbody>
</table>

64
By studying the above tables, it is obvious that a higher polynomial order does not automatically implicate a higher latency for the whole system. That is shown in table 5.6 for the 12 bits case as well as for the 24 bits case, but not for the 16 bits case. For the first two, we can distinguish a significant decrease of calculation time for the precalculation block. See for example $T_Y$ in table 5.6, when the order is increased with one. That is also shown in the $T_A$ column of table 5.7, where higher numbers of branches in the lower-order cases implicates higher propagation times through the multiple constant multiplier.

However, due to the complex nature of the multiplier and the different characteristics of every single coefficient and each decimal length set, it is not always evident that numerous constants or long words would implicate slower multiplications. Example on that is shown in the 16 bits case of table 5.7. Regarding the area, an analogous discussion as above could be held.

According to the $T_B$ column in table 5.7, the branch adder seems to work efficiently. It can be seen that the calculation time slightly increases, when the wordlengths increase. However, the time is not significantly affected by an increase of the number of terms, which is one of the great advantages with the Wallace tree adder.

The $T_C$ column represents one single multiplier in the polynomial calculation block and its delay is therefore relatively constant, in the same way as for the Wallace tree adder.

The $T_D$ column represents the multipliers from the second one to the last one in the exponentiation part of the polynomial calculation block. Naturally, the propagation time through this block is proportional to the number of multipliers that are cascaded after each other, in other word proportional to the polynomial order, which can easily be verified by looking at the values in table 5.7.

A similar adder as in the precalculation block is represented by $T_E$, which sums all exponential products in the block, all multiplied with the polynomial coefficients.

Finally, the $T_F$ column represents the multiplication of the polynomial interpolation output and the precalculation step output. Since this is the last multiplication, more bits are required than in previous multiplications to maintain the desired accuracy. Hence, the propagation time of this block is relatively long in comparison to other multiplications.
5.4 COMPARISON AND ANALYSIS

The remaining work after having achieved the synthesis results presented in 5.3 and 5.4, is to compare the both divider implementations and try to answer the question whether the polynomial-based divider is competitive or not.

By having a fast look on the tables 5.2 and 5.6 it is clear that the polynomial-based divider is not competitive at all regarding the combinatorial implementation. For the present precisions, the total latency is approximately twice as long as the corresponding latency for the Newton-Raphson divider.

For each desired precision, the Newton-Raphson divider can have different initial approximations and consequently different sizes of the lookup table, which influences both the area and the propagation time through the used ROM. However, independent of the initial approximation’s accuracy and the number of required iterations, the Newton-Raphson divider is always faster.

Important to keep in mind is that this divider is also not optimally constructed, since it contains some extra adapters and that it has registers included, even in those cases when only one iteration is supposed to be performed. Despite those facts the Newton-Raphson divider is still better.

One drawback that has been suspected with the Newton-Raphson divider is the hardware cost of the required ROM. After having studied the synthesis results in the tables 5.2 and 5.6, it is clear that the Newton-Raphson divider is superior also in this respect. By for instance looking at the accuracy of 16 bits and 24 bits, the required area is approximately between a third and a fourth of the corresponding area for the polynomial-based divider, even though the divider is twice as fast.

The advantages with the Newton-Raphson divider are also obvious for high precision numbers. First of all it always calculates correct quotients, due to its simple structure, in opposition to the polynomial-based one, which for larger wordlengths can only be verified for a limited amount of numbers. Second, the polynomial-based divider requires time consuming optimizations, which do not guarantee that a feasible solution can be found within reasonable time. For instance, for 32-bit accuracy no solutions have been found, whereas the Newton-Raphson divider has the same structure as for smaller precision numbers. Third, it has been shown that the cost in terms of area increases rapidly for the polynomial-based divider.

A good example on the differences is that the Newton-Raphson divider for an accuracy of 32 bits and an initial approximation of 8 bits, calculates the cor-
rect quotient only 20 ns slower than the corresponding polynomial-based divider calculates a non-verified quotient with 24 bits of accuracy. In addition to that, the Newton-Raphson divider would require approximately one third less area than the polynomial-based one.

Considering that, how come that the conclusion of [4] was that the polynomial-based divider would be especially competitive regarding larger word-lengths between 24 and 32 bits? The explanation is possibly as simple as that it depends on which measures one is comparing. The initial hypothesis was based on a high level model, which counted only the number of general multiplications, but did not take in account any internal wordlengths or how large different building blocks would be according to their architecture in the real implementation. However, to be able to make a correct and fair statement in this issue, such tests that have been performed in this thesis must be done.

5.5 FURTHER TESTS

Considering the previously accomplished results, the question arises whether at least some advantages could be granted the polynomial-based divider or not. As earlier mentioned, the polynomial-based divider is easy to pipeline and maybe that would be an alternative to use in the trial to beat the Newton-Raphson divider.

The first step when deciding whether to use pipelining or not is to have a look on table 5.7, in order to figure out which blocks that are suitable to group and try to approximate the delay of the slowest group. In the real implementation, however, the delays would slightly increase in comparison to those values, due to the extra delay of the pipeline registers that are introduced. Since the wish is to find a faster solution than the Newton-Raphson divider, this extra delay would not matter if it is shown that the polynomial-based divider could not beat it anyway. In that case, the extra delay would only amplify the conclusion that the Newton-Raphson divider is the better one.

In case a Newton-Raphson divider with more than one iteration is considered, the architecture must be modified and unfolded according to the discussion in section 4.3. However, that would not decrease the performance of the divider, since the first dividers in that case could be made smaller due to the smaller precision needed in the first divider sets. This would in the same way as above further convince us that the Newton-Raphson divider is the best one, in
case the results go in that way. One drawback though, is that the area of such unfolded multiple-iteration dividers would increase.

For 12 bits accuracy the delay of the Newton-Raphson divider is 41 ns. The longest delay of the polynomial-based divider is 21.2 ns and by grouping blocks in the groups \((T_A, T_B, T_C), (T_D), (T_E, T_F)\), the longest group delay would be at least 26.7 ns and the calculation would require three clock cycles. In other words, the total delay would be 80.1 ns. Considering the above circumstances as well as the fact that the Newton-Raphson divider’s delay also is negatively affected by the number converting adapters, it is uncertain if this divider could be competitive.

For 16 bits of accuracy the fastest possible block requires at least 31.2 ns in comparison to 46.7 ns of the Newton-Raphson divider, which also gives a small margin in time saving potential. Even if the first divider must be unfolded, its area would in all cases be smaller than the double area of the original one, since this area would also include two ROM blocks. That would implicate an area of approximately 980 sqmil in comparison to the 2092 sqmil for the polynomial-based divider.

When studying 24-bit dividers, the slowest block of the polynomial-based divider requires 38.8 ns in comparison to 68.8 ns of the Newton-Raphson divider. However, since pipelining is only possible to introduce at some specific positions, a suitable grouping might be \((T_A), (T_B, T_C), (T_D), (T_E)\) and \((T_F)\). This would probably give a better result in terms of throughput, but would on the other hand significantly increase the latency. This design would require five clock cycles of at least 38.8 ns, which corresponds to a total propagation time of 194 ns. The corresponding time for the Newton-Raphson divider would be 136 ns or as low as 86.2 ns when no pipelining or unfolding is introduced. Regarding the area, the both dividers differs with approximately a factor two, from about 2000 sqmil to about 4000 sqmil, respectively.

With these figures on our minds, it does not seem like the polynomial-based divider could be advantageous in this way of use either. Hence, since no significant improvements are likely to be found if pipelining were introduced, the decision was made not to make any further tests on this.

Finally, about the refinement of the Newton-Raphson divider in terms of area it seems to be no idea to carry out further tests on synthesizing the TI divider architecture in figure 4.3. The reason for that is that the previously tested model already is superior to the polynomial-based divider in terms of area. Due to its architecture with sharing of only one multiplier, the latency would
certainly increase with such an architecture, and in that case the relative result of the polynomial-based divider might consequently be improved. However, if someone is willing to implement a specific stand-alone divider like the polynomial-based one, one could also expect that the implemented Newton-Raphson divider with better performance and less cost would also be accepted, even though it is not suited in the same way as the TI divider in terms of sharing hardware with other kinds of arithmetic operations. In any case the already implemented architecture is enough to reject the hypothesis about the polynomial-based divider’s high performance.
CONCLUSIONS AND FUTURE WORK

To summarize this thesis work some conclusions will be drawn and presented. The generally concluding result is that the polynomial-based divider does not seem to be competitive at all in comparison to a general Newton-Raphson divider.

First of all, both types of dividers that are proposed in [4] are not directly comparable, since the polynomial-based divider is intended for fixed-point numbers, whereas the iterative Newton-Raphson divider uses floating-point numbers. Therefore, number converting adapters must be introduced, in order to be able make fair comparisons, which of course also implicate some extra delay to be added to the total latency.

The optimization and the generation of the polynomial-based divider is very time consuming. Due to the multiple degrees of freedom and the numerous variables it is difficult to define suitable boundaries and it is almost impossible to find a global optimum. In this investigation, architectures have been found for 12, 16 and 24 bits of precision, but for 32 bits no solution at all could be found.

Because of that complexity, the polynomial-based architecture cannot be verified for all possible values within the dividends range, when the desired precision is larger than 16 bits. The Newton-Rapson divider however, does not suffer from this problem.

When the polynomial order increases, the number of required branches decreases. If the polynomial order is set high enough, the precalculation step almost vanishes and almost all of the estimation work is carried out by the polynomial interpolation block.

It has been seen though, that higher order and fewer branches can implicate a lower latency and smaller hardware cost than the opposite case. However, the hardware cost for the polynomial-based divider increases rapidly as the spec-
ified accuracy increases.

In all cases it has been shown that the Newton-Raphson divider is superior to the polynomial-based one, in terms of both latency and hardware cost, for low precision numbers as well as for high precision numbers. A pipelined polynomial-based divider might for some cases possibly be competitive in terms of throughput, but it is uncertain if it is worth the disadvantages in terms of hardware cost and the increased total latency.

Due to the above conclusions the recommendation is not to work further on this divider. If the result was closer to the Newton-Raphson variant, refinements could have been done. For instance could more iterations in the optimizations have been done and more length configurations for each set of branches and polynomial order have been synthesized, in order to find out the best possible solution in each single case. Another example could be to test both the BHM algorithm and the RAG-n algorithm. Since the differences between the both division algorithms are so large, the kind of work that now has been mentioned would not probably generate any significant improvement of the actual divider.

If the polynomial-based divider shall be used, one idea could be to try modifying the optimization algorithm. However, since the results in the performed optimizations seemed to converge and the second optimization step did not have any influence on the results, that would probably not result in any significant improvements, especially since the differences between the both algorithms are so large, regarding latency as well as area.

If the divider would have been competitive, it might also have been interesting to investigate the power consumption of the both dividers, but since this was not the purpose of this thesis to examine and since the performance of the divider is not so good in terms of the other measures, this has not been further investigated. However, since both dividers are almost always unclocked and therefore combinatorial, a fair expectation would be that the larger architecture would also consume more energy, in other words the polynomial-based divider.

Finally, if any suggestion at all on future work should be presented, one idea is to optimize and test the precalculation block only. Perhaps, could a solution be found that can serve as for instance a generator of an initial approximation to for instance another iterative divider [4]. The problem though, is that it still suffers from the problem with different number representations and that it therefore would still need the adapters. A second problem is that it in the test
of the Newton-Raphson divider has been seen that initial approximations still can be generated to a rather small cost in a ROM, at least for numbers up to between eight and twelve bits of accuracy. With these facts on our minds though, the question is whether the actual block could be competitive at all, in any kind of architecture.
BIBLIOGRAPHY


In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Stefan Pettersson