NTNU

Norwegian University of
Science and Technology

Construction of digital integer arithmetic

FPGA implementation of high throughput pipelined division circuit

Johan Arthur Øvergaard

Master of Science in Electronics
Submission date: June 2009
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Background
This assignment has background in challenges at Kongsberg Defence and Aerospace (KDA). KDA develops among other things digital arithmetic circuits for use in mobile radio, radio link and switching equipment. Those circuits include among other things modules for integer division of 16 and 64 bit operands. This master thesis will be based on results of earlier master theses in digital arithmetic at KDA.

Assignment
Preform an analysis and theoretical study of arithmetic circuits for integer division with respect to size, precision and number of iterations.

Preform a evaluation of module library versus module generator for division modules with different size, precision and speed properties.

Implement a set of modules or a module generator for 16 and 64 bits integer division. Then synthesise and test the results. Compare modules with subtractive and, or multiplicative solutions.

Assignment given: 20. January 2009
Supervisor: Kjetil Svarstad, IET

PREFACE

This master thesis concludes my education at Norwegian university of science and technology (NTNU). The assignment has been given by Kongsberg Defence and Aerospace (KDA). This assignment has been carried out in the spring of 2009, at the Department of Electronics and Telecommunications.

The subject for this master thesis has been to develop a 16 and 64 bit high throughput pipelined division circuit for implementation in FPGA. I selected this assignment because it seamed interesting to investigate how best to implement a division circuit, which is the most complicated arithmetic functions, in digital logic.

I would like to thank my supervisors' professor Kjetil Svarstad at NTNU and Simen Gimle Hansen from KDA for good supervision through this assignment.

Trondheim June 16, 2009

Arthur Øvergaard

Abstract

This assignment has been given by Defence Communication (DC) which is a division of Kongsberg Defence and Aerospace(KDA). KDA develops amongst other things military radio equipment for communication and data transfer. In this equipment there is use of digital logic that performes amongst other things integer and fixed point division.

Current systems developed at KDA uses both application specific integrated circuit (ASIC) and field programmable gate arrays (FPGA) to implement the digital logic. In both these technologies it is implemented circuit to performed integer and fixed point division. These are designed for low latency implementations. For future applications it is desire to investigate the possibility of implementing a high throughput pipelined division circuit for both 16 and 64 bit operands.

In this project several commonly implemented division methods and algorithms has been studied, amongst others digit recurrence and multiplicative algorithms. Of the studied methods, multiplicative methods early stood out as the best implementation. These methods include the Goldschmidt and Newton-Raphson method. Both these methods require and initial approximation towards the correct answer. Based on this, several methods for finding an initial approximation were investigated, amongst others bipartite and multipartite lookup tables.

Of the two multiplicative methods, Newton-Raphsons method proved to give the best implementation. This is due to the fact that it is possible with Newton-Raphsons method to implement each stage with the same bit widths as the precision out of that stage. This means that each stage is only halve the size of the succeeding stage. Also since the first stages were found to be small compared to the last stage, it was found that it is best to use a rough approximation towards the correct value and then use more stages to achieve the target precision.

To evaluate how different design choices will affect the speed, size and throughput of an implementation, several configurations were implemented in VHDL and synthesized to FPGAs. These implementations were optimized for high speed whit high pipeline depth and size, and low speed with low pipeline depth and size. This was done for both 16 and 64 bits implementations.

The synthesizes showed that there is possible to achieve great speed at the cost of increased size, or a small circuit while still achieving an acceptable speed. In addition it was found that it is optimal in a high throughput pipelined division circuit to use a less precise initial approximation and instead use more iterations stages.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
KDA	Kongsberg Defence and Aerospace
LSB	Least Significant Bit
LUT	Lookup Table
MSB	Most Significant bit
TIV	Table of Initial Values
TO	Table of Offset
ULP	Unit of Least Precision
VHDL	Very high speed integrated circuit Hardware Description Language

CHAPTER

1

INTRODUCTION

This assignment has been given by Defence Communication (DC) which is a division of Kongsberg Defence and Aerospace(KDA). KDA develops amongst other things military radio equipment for communication and data transfer. A feature of this equipment is frequency jumping to prevent jamming from enemy equipment. This is performed by radio frequency (RF) synthesizers that generate the output signal. For each frequency change, these synthesizers need configuration parameters calculated by a digital circuit. One of the operations involved in the calculation of these parameters is division. This puts high demand on the division circuit to calculate the parameters within acceptable time limits.

Current systems developed at KDA uses both application specific integrated circuit (ASIC) and field programmable gate arrays (FPGA) to implement the digital logic. The ASIC circuit that is used in some of the equipment is the NOVA circuit, which is developed by KDA. This circuit has a 16 bit radix-2 implementation of the SRT-division method. To achieve the needed performance of this circuit, 16 stages of the SRT circuit is implemented in sequence. Operations performed by this division circuit is both integer division for calculation of the frequency synthesizer parameter as well as fixed point division for digital signal processing(DSP) of the output data. Other equipment developed at KDA uses FPGAs with implemented circuits for integer division.

For future applications it is desire to investigate the possibility of implementing a high throughput pipelined division circuit for both 16 and 64 bit operands. It is believed that the most challenging implementation will be the 64 bit division circuit. For this reason most attention has been given to the 64 bit implementation, and it is assumed most of the

findings for 64 bit implementations will be transferable to 16 bit implementations.

1.1 Previous work

This assignment continues the work of previous assignments form KDA on the subject. Hansen [1] investigated which possibilities that exist to perform division and square root calculation with multiplicative methods. Further on Rognerud [2] has, on the basis of Newton-Raphsons method of division, investigated methods to achieve an initial approximation towards the correct value. The precision of this initial approximation is of vital importance to how fast Newton-Raphson method will converge towards the final precision. Of the methods that were given most attention were bi- and multipartite lookup tables. Of those two methods, a simulation model of bipartite lookup table method was implemented in VHDL. No implementation code were synthesized to FPGA.

Stafto [3] continued on the work by Rognerud. Stafto studied several other methods of performing digital division, including subtractive division algorithms and the Goldschmidt division algorithm which is an other multiplicative division algorithm. Like Rognerud, Stafto focused most of his work on the problem with initial value on iterative division methods. In addition he sketched an implementation for a pipelined implementation, but there were not implemented any synthesizable code.

Other related work is found in [4] and [5] where bipartite and multipartit table methods has been investigated. The authors of these articles have developed methods to calculate optimal sizes and values for bi and multipartite lookup tables.

Of related work regarding division methods and algorithms are [6], [7] and [8]. In [6] a summary of different division methods is presented and positive and negative sides are evaluated. Further in [8] a pipelined implementation of the Goldsmith division algorithm has been implemented in FPGA. The major difference between that implementation and the implementation wanted in this assignment is that it is optimised for low latency rather than high throughput.

1.2 Objectives for this assignment

The objective with this assignment is to study theory on division methods, algorithms and its implementations. Then based on the theory study an evaluation of the different methods shall be performed. Based on this evaluation the method that is found to give the best implementation shall be implemented for 16 and 64 bit operands.

In this assignment the following shall be performed:

1. Study theory on division algorithms and methods.
2. Analyze the methods and find a method that is optimal for a high throughput implementation.
3. Implement a set of division circuits in VHDL for 16 and 64 bits operands, and synthesize these to FPGA.

1.3 Work on the assignment

The work on this assignment has been performed in three stages:

1. Study phase: Here different division methods and algorithms have been investigated. In addition some methods used to find an initial approximation for iterative methods has been evaluated.
2. Implementation phase: Here some circuits have been implemented for 16 and 64 bits operands, with optimization for both high speed/area, and low speed/area.
3. Documentation phase: Here the report has been finalized.

The work effort has approximately been distributed 40%, 40% and 20%.

1.4 Structure of the report

Chapter 1 *Introduction* gives an introduction to the project and the report, as well as a summary of previous work on the subject.

Chapter 2 *Division algorithms* gives an introduction to some of the most common division algorithms and methods that exists to perform digital division.

Chapter 3 *Implementation of table generator* presents a C++ implementation of a table generator that was developed to generate multipartite lookup tables.

Chapter 4 *Evaluation of division algorithms* presents an analysis of the different division methods presented in chapter 2, and the argument for choosing Newton-Raphson method for implementation.

Chapter 5 *Implementation of Newton-Raphson simulation model* presents calculations performed to determine the needed bit widths in Newton-Raphsons method, and an implementation of a simulation model to verify the calculations.

Chapter 6 *Implementation of synthesizable circuit* presents an implementation of a division circuit based on the Newton-Raphsons method.

Chapter 7 *Discussion* gives a discussion around the choices done in the project, and the results achieved.

Chapter 8 *Conclusion* gives a conclusion of the work performed in the project.

Appendix A *Simulation test bench* lists the test bench used to test the simulation model.

Remaining implemented code is submitted in attached zip file.

1.5 About reading the report

It is presumed that the reader has previous knowledge about digital systems design, in particular towards FPGA. Also it is presumed that the reader has knowledge about implementation of digital arithmetic in hardware.

Mathematical notation in the report It has been an effort to achieve consistency in the mathematical expressions through the report. Unless other is mentioned the following has been used:

- A Dividend
- B Divisor
- Q Quotient
- $\lfloor\ \rfloor$ means round down to nearest integer
- $\lceil\ \rceil$ means round up to nearest integer

1.6 Contribution

The main contribution in this report is that it has been found when implementing high throughput division circuit where latency is of less importance, that there it is more area efficient to use additional iteration stages rather than use large table methods to achieve precise initial approximation.

CHAPTER

2

DIVISION ALGORITHMS

Digital division may be performed in many different ways, and there have been several studies that have thoroughly investigated its performance and ways to improve on the speed, size and latency. Obermann and Flynn has in [6] compared different classes of division methods and evaluated their strengths and weaknesses. Obermann and Flynn have divided the division methods in to five different classes. These metodes were digit recurrence, functional iteration, very high radix, table lookup and variable latency.

The very high radix class of dividers is a special class of digit recurrence algorithms. These methods use complex circuits to calculate several bits of the final answer in one step. In systems where latency is critical this may be an ideal solution, in this system however, latency is not critical. Another effect of this method is that the complex quotient selection logic will presumably be difficult to efficiently implement in FPGAs, and therefore lead to a long critical path, low clock frequency and large area. Based on these assumptions no further study of this method has been performed.

The variable latency class of dividers exploits the fact that most input operands will converge towards the final answer faster than the worst case operands. This may be used to decrease the average latency in microprocessors where the code execution is halted until the division is completed. Such an implementation will not be suited in this implementation where throughput is most important and the latency has to be deterministic for all operations. For these reasons this method will not be discussed further in this assignment. The three other methods will be presented in the following sections.

2.1 Digit recurrence algorithms

The digit recurrence class of algorithms is often known as subtractive division algorithms as one of the dominant operations is a subtraction. A feature of the digit recurrence algorithms is that they have linear converges. This means that they find one or more bits of the final quotient for each iteration. One of the advantages of this class of division circuits is that they are possible to implement using very small circuits when latency is less important. Much research has been published around digit recurrence algorithms and how to improve their efficiency, in [9], [10], [11] and [12] the different authors have developed alternative ways to implement digit recurrence algorithms. While in [13] and [14] the authors has investigated ways to increase the speed of already known digit recurrence algorithms. Ibrahem, Elismry and Salama [15] has evaluated an implementation of a Radix 2 SRT division algorithm on FPGA.

Algorithm 2.1 illustrates how digit recurrence division is performed.

Algorithm 2.1 Digit recurrence algorithm for calculating $Q = \frac{A}{B}$, where Q, A and B are n bit integers

Require: $B \neq 0$
1: Initilize $B \leftarrow B \cdot 2^n$, $Q \leftarrow 0$
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $B \leftarrow \mathrm{LSR}(B)$ *Logic shift right*
4: $Q \leftarrow \mathrm{LSL}(Q)$ *Logic shift left*
5: **if** $A \geq B$ **then**
6: $A \leftarrow A - B$
7: $Q \leftarrow Q + 1$
8: **end if**
9: **end for**
10: **return** Q

The efficiency of digit recurrence algorithms is determined by its maximum frequency and the number of bits calculated in each iteration. By increasing the radix r used by the algorithm, the number bits calculated in each iteration will be increased, and the number of iterations i needed will be reduced. With n bits operands the number of iterations is given by equation (2.1). Such modification will lead to a more complex implementation and therefore also a reduced clock frequency.

$$i = \left(\frac{n}{\log_2 r} \right). \tag{2.1}$$

One of the best known and most used digit recurrence algorithm implementation is the

SRT algorithm. This was developed independently by three people Sweeney, Robertson and Tocher, around the same time [16]. In [17] radix 2, 4 and 8 implementation of the SRT algorithm has been implemented on a Virtex FPGA. They have testes their implementations on integers but have done no mention of the bit width used. This makes it difficult to evaluate how an implementation of 64 bit pipelined integer division circuit will perform, based on their results.

2.2 Newton-Raphson method

Functional iteration or multiplicative division algorithms are one of the most used methods to perform division in digital circuits. This class of division circuits uses some mathematical function that converges quadratic closer towards the correct answer for each iteration that is performed. The dominant operation involved in these division methods is multiplication, and they are therefore often referred to as multiplicative division methods. In this class of division circuits there are two methods that are commonly implemented, the Newton-Raphson method and the Goldschmidt method.

One of the most common and best known implementation of multiplicative division algorithms is the Newton-Raphson method [7]. The Newton-Raphson method is an iterative method developed by Isaac Newton and Joseph Raphson to calculate roots of equations. The method starts by some approximate value and then, if this value satisfies the convergence interval, a more precise approximation will be achieved for each iteration. Figure 2.1 shows a representation of how the Newton-Raphson method will converge closer to the exact value.

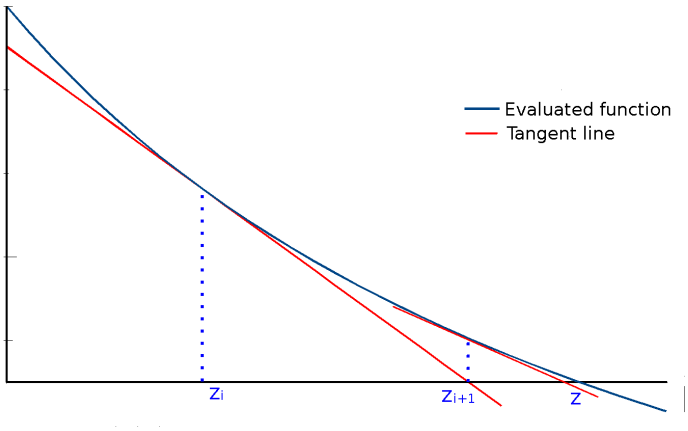

Figure 2.1: Newton-Raphsons method will for some initial values of z converge toward a correct value for z

Mathematically Newton-Raphson method is expressed by equation (2.2). By choosing a

start value z_0 that satisfies some start conditions, the next value z_{i+1} will be a better approximation towards the correct value. One problem with the Newton-Raphson method is that it is not guarantied to converge towards the correct value if the initial value is outside the convergence region.

$$z_{i+1} = z_i - \frac{f(z_i)}{f'(z_i)}. \tag{2.2}$$

2.2.1 Newton-Raphson division

To solve division in digital circuits by using Newton-Raphson method, an expression for the division has to be found that is possible to solve using equation (2.2), in addition this expression has to be easily implemented in digital logic. These problems has been studied several times with different focus, amongst others [1], [6] and [7]. One solution to this problem is given by the following equation:

$$Q = A \cdot \frac{1}{B}. \tag{2.3}$$

By separating the division operation in to two parts by first calculating the reciprocal and then multiply this with the dividend, equation (2.3), it is possible to construct an expression that may be used in equation (2.2). The task is then to find a function $f(z)$ that can be used to calculate the reciprocal. Mathematically several equations may lead to the correct result and some has been presented in [7]. The function that has been showed to give the simplest implementation is given in the following equation.

$$f(z) = \frac{1}{z} - B. \tag{2.4}$$

By inserting equation (2.4) and its derivative in to equation (2.2) an expression for calculating the reciprocal will be achieved, equation (2.5). This may be implemented in hardware using two multiplications and one subtraction.

$$\begin{aligned} z_{i+1} &= z_i - \left(\frac{(1/z_i) - B}{-(1/z_i^2)} \right) \\ &= 2z_i - Bz_i^2 \\ &= z_i(2 - Bz_i). \end{aligned} \tag{2.5}$$

The process for calculating $Q = \frac{A}{B}$ with Newton-Raphson method can be summarized in algorithm 2.2. As shown this method requires two multiplications that are dependent on

each other, meaning that they can not be done in parallel. In addition one final multiplication is needed between the reciprocal and the dividend to calculate the final answer. By using high precision on the initial approximation of the reciprocal the number of iterations k needed to achieve the target precision on the reciprocal will be reduced.

Algorithm 2.2 Newton-Raphson method for calculating $Q = \frac{A}{B}$, using k iterations

Require: $B \neq 0$
1: Initilize $z \leftarrow \frac{1}{B} + e_0$
2: **for** $i = 1$ **to** k **do**
3: $\quad z \leftarrow z(2 - B \cdot z)$ *Two dependent multiplications*
4: **end for**
5: $Q \leftarrow A \cdot z$ *One final multiplication*
6: **return** Q

2.2.2 Error analysis of Newton-Raphson method

As previously mentioned the Newton-Raphson method is not guaranteed to converge towards the correct value for all initial values [18]. For instance if $z_0 = 0$ then the algorithm will converge towards 0 which in incorrect. To ensure that this will not occur a bound for the initial value of z_0 has to be calculated. This bound may be calculated by analyzing the error E_i in each iteration. For the algorithm to converge towards the correct reciprocal the error term E_i must go towards zero as i goes towards infinity. These calculations may also be used to determine how many iterations that is necessary for the algorithm to reach desired precision.

The error in the reciprocal E_{i+1} at each iteration of the method is given by the difference between the true value and the calculated:

$$\begin{aligned} E_{i+1} =& \frac{1}{B} - z_{i+1} \\ =& \frac{1}{B} - \left(2z_i - Bz_i^2\right) \\ =& B\left(\frac{1}{B} - z_i\right)^2 \\ =& BE_i^2, \end{aligned} \tag{2.6}$$

Here we have used that $\left(\frac{1}{B} - z_i\right) = E_i$. An expression for E_i may then be found by expanding equation (2.6):

$$\begin{aligned} E_{i+1} =& BE_i^2 \\ E_i =& BE_{i-1}^2 \\ E_{i-1} =& BE_{i-2}^2 \\ E_{i-1}^2 =& B^2E_{i-2}^4. \end{aligned} \tag{2.7}$$

From equation (2.7) it is possible to express E_i as:

$$\begin{aligned} E_i =& B\left(B^2\left(B^4\left(\dots\left(B^{2^{i-2}}\left(B^{2^{i-1}}E_0^{2^i}\right)\right)\dots\right)\right)\right) \\ =& B^{\sum_{n=1}^{i}(2^{n-1})}E_0^{2^i}. \end{aligned} \tag{2.8}$$

The power of B in equation (2.8) is a geometric series, and the n'th sum of a geometric series is given by the following equation:

$$s_n = \frac{a(1-r^n)}{1-r}. \tag{2.9}$$

In this case $a = 1$ and $r = 2$. This gives that B is in power of $s_i = 2^i - 1$, and the total error at i'th iteration is then given by:

$$\begin{aligned} E_i =& B^{2^i-1}E_0^{2^i} \\ E_i =& B^{2^i-1}\left(\frac{1}{B} - z_0\right)^{2^i} \\ E_i =& B^{2^i-1}\frac{(1-Bz_0)^{2^i}}{B^{2^i}} \\ E_i =& \frac{(1-Bz_0)^{2^i}}{B}. \end{aligned} \tag{2.10}$$

For the Newton-Rapson method to converge the error term E_i must go towards zero as i goes towards infinity. From equation (2.10) we can see that this will be true if $|1 - Bz_0| < 1$. This gives that the range of z_0 is bounded by $0 < z_0 < \frac{2}{B}$.

2.2.3 Summing up Newton-Raphson method

Newton-Raphson method is an iterative method that may be used to calculate division. The method calculates a value for the reciprocal based on an initial approximation. For

each iteration performed with the method, a more precise reciprocal will be achieved. To ensure that the method will converge correctly it was shown that the initial approximation must lay in the range $z_o \in \langle 0, \frac{2}{B} \rangle$.

2.3 Goldschmidt method

As was shown for the Newton-Raphson method it involved two dependent multiplications. This makes it impossible to parallelize the multiplications and thereby decrease the latency. Another method that solves this problem is the Goldschmidt method. This method was first described by Robert E. Goldschmidt [19]. As with the Newton-Raphson method this also has two multiplications in each stage, but these multiplications are independent of each other and may therefore be executed in parallel.

The fundamental mathematical method involved in Goldschmidt method is Taylor series expansion of a function. Taylor series is a method to calculate function values based on an approximation of the real function. The general form of a Taylor series is expressed as following:

$$f(z) = f(p) + (z-p)f'(p) + \frac{(z-p)^2}{2!}f''(p) + \cdots + \frac{(z-p)^n}{n!}f^{(n)}(p) + \cdots . \tag{2.11}$$

As with Newton-Raphson method this method requires an expression for the reciprocal that will lead to an implementation that will be small and efficient.

$$Q = \frac{A}{B} = A \cdot f(z). \tag{2.12}$$

An expression for $f(z)$ that has been found to give a good implementation is $f(z) = \frac{1}{1+z}$ [6]. By inserting this in equation (2.11) and setting $p = 0$, the following Maclaurin series, which is a special case of Taylor series where $p = 0$, is achieved.

$$f(z) = \frac{1}{1+z} = \sum_{n=0}^{\infty}(-z)^n = 1 - z + z^2 - z^3 + z^4 - \cdots . \tag{2.13}$$

By setting $z = B - 1$ in equation (2.13), and normalize B to the range $0.5 \leq B < 1$ it is

possible to express the division by the following equation.

$$\begin{aligned} Q =& A \cdot \frac{1}{1+(b-1)} = A \cdot \frac{1}{1+z} \\ =& A \cdot \left(1 - z + z^2 - z^3 + z^4 - \cdots\right) \\ =& A \left[(1-z)(1+z^2)(1+z^4)(1+z^8)\cdots(1+z^{2^i})\right]. \end{aligned} \tag{2.14}$$

An implementation of equation (2.14) in hardware will regard each part in the series $\left[(1-z)(1+z^2)(1+z^4)(1+z^8)\cdots\right]$ as a product in the sequence $r_0 \cdot r_1 \cdot \ldots \cdot r_n$ that makes the divisor B converge towards one as the dividend converges towards the quotient Q. Mathematically this is expressed as:

$$Q = \frac{A}{B} = \frac{A \cdot r_0 \cdot r_1 \cdot \ldots \cdot r_n}{B \cdot r_0 \cdot r_1 \cdot \ldots \cdot r_n} = \frac{A \prod_{i=0}^{n} r_i}{B \prod_{i=0}^{n} r_i}. \tag{2.15}$$

As the method iterates, the following can be observed,

$$\prod_{i=0}^{n} r_i \rightarrow \frac{1}{B}. \tag{2.16}$$

This will then lead to,

$$B \cdot \prod_{i=0}^{n} r_i \rightarrow 1, \quad \text{and} \quad A \cdot \prod_{i=0}^{n} r_i \rightarrow \frac{A}{B} = Q. \tag{2.17}$$

From equation (2.17) it is possible to create a iterative product series for the divisor:

$$\begin{aligned} B_i &= B \cdot r_0 \cdot r_1 \cdot \ldots \cdot r_i \\ &= B_{i-1} \cdot r_i. \end{aligned} \tag{2.18}$$

From equation (2.14) we know that $r_i = (1 + z^{2^i})$. By inserting this in to equation (2.18) it is possible to make a general expression for the i'th iteration:

$$\begin{aligned} B_i &= B_{i-1} \cdot r_i \\ &= B_{i-1} \cdot (1 + z^{2^i}) \\ &= (1 - z^{2^i}) \cdot (1 + z^{2^i}) = (1 + z^{2^{i+1}}). \end{aligned} \tag{2.19}$$

From equation (2.19) we can see that since the divisor is normalized to the range $B \in [0.5, 1.0\rangle$, the divisor B will converge towards 1. The part r_i in equation (2.19) may be calculated using a two's complement of $B_i - 1$:

$$r_i = 2 - B_{i-1} = 2 - (1 - z^{2^i}) = 1 + z^{2^i}. \tag{2.20}$$

Since r_i will converge towards $\frac{1}{B}$ following expression may be created for the quotient:

$$A_i = A_{i-1} \cdot r_i. \tag{2.21}$$

As the method iterates the dividend A_i will converge towards the quotient.

Algorithm 2.3 summarizes the steps involved in the Goldschmidts method for iterative division.

Algorithm 2.3 Goldschmidt method for calculating $Q = \frac{A}{B}$, using k iterations

Require: $B \neq 0$, $|e_0| < 1$
1: Initialise $N \leftarrow A$, $D \leftarrow B$, $R \leftarrow \frac{1-e_0}{B}$
2: **for** $i = 0$ **to** k **do**
3: $N \leftarrow N \cdot R$
4: $D \leftarrow D \cdot R$ *Two independent multiplication*
5: $R \leftarrow 2 - D$
6: **end for**
7: $Q \leftarrow N$
8: **return** Q

As seen from algorithm 2.3 the two multiplications performed in each iteration are independent of each other and possible to execute in parallel, and thereby reduce the latency through the system. One problem with the Goldschmidt method is that, as opposed to the Newton-Raphson method that will correct it self for each iteration, this method will accumulate truncation error for each iteration [20]. This will result that extra guard bits is needed to be calculated in each stage to maintain the accumulated error bellow the target error limit.

2.4 Normalization of input operands

Both the Newton-Raphson and Goldschmidt method are only valid for input parameters in a limited range, $B \in [1, 2\rangle$ for Newton-Raphson method and $B \in [0.5, 1.0\rangle$ for the Goldschmidt method. This constraint is not satisfied by the range of the n bit integer

input which will be in the range of $[0, 2^n - 1]$. To solve this problem the input parameters must be normalized. Normalization of operands is performed by left shifting the operand until the most significant bit (MSB) is '1', and then redefining the value for the MSB by inserting a comma at the appropriate position, see figure 2.2. This operation is equal to multiplication with 2 for each left shift this is then compensated for by right shifting the output from the division circuit.

00000001101001 initial input operand
11010010000000 left shift until MSB is '1'
1.1010010000000 insertion of decimal point

Figure 2.2: Example of normalization of input operands

2.5 Initial values

As mentioned in sections 2.2 and 2.3 iterative methods need an initial approximation of the resiprocal for the first iteration. In section 2.2.2 it was showed that the initial value z_0 for Newton-Rapsons method must be in the range, $0 < z_0 < \frac{2}{B}$, to ensure that the method will converge. The higher the precision on this initial approximation the faster this method will converge towards the target precision. Depending on the chosen precision of the initial approximation different implementations which have different size and speed may be used.

2.5.1 Constant initial value

The simplest implementation for the initial value is to use a constant initial value. For the Newton-Raphson method the only demand for this value is that it always must be in the range $0 < z_0 < \frac{2}{B}$. Since B will be normalized to the range $B \in [1.0, 2.0\rangle$ the range of z_0 will be limited to $0 < z_0 < \frac{2}{2} \Rightarrow z_0 \in \langle 0.0, 1.0\rangle$. The value z_0 is an approximation for the reciprocal and will therefore be bounded by its range which is $[0.5, 1.0\rangle$. Since the range of z is more strict than the range of z_0 any value in z is a valid initial value.

Glodschmidt method has been showed to accumulate error for each iteration. This makes it necessary to do more complex calculations to find an appropriate constant value that can be used as start value.

On worst case input values this method will lead to one correct bit on the initial approximation. Since iterative division methods converge quadratic towards the finale value this will then require $\log_2(n)$ iterations.

2.5.2 Linear approximation

This method has been described by Hansen in [1]. Linear approximation is a method that calculates an approximate reciprocal based on a simplified mathematical expression of the reciprocal. This simplified expression is based on a sum of polynomials on the following form:

$$\frac{1}{B} = z(B) = c_0 + c_1 B + c_2 B^2 + \ldots + c_n B^n. \tag{2.22}$$

To achieve an implementation that has low complexity only the zero and first order part is included. This gives an expression on the form $z(B) = c_0 + c_1 B$. The constants c_0 and c_1 are then calculated to values that will give an expression as precise as possible while still keeping the implementation simple. The values that has been found in [1] that gives a simple implementation is the values at the edges of the range of B. This gives the following two equations that may be used to calculate c_0 and c_1.

$$\begin{aligned} z(1.0) &= c_0 + c_1 \cdot 1.0 = 1.0 \\ z(2.0) &= c_0 + c_1 \cdot 2.0 = 0.5. \end{aligned} \tag{2.23}$$

Solving equation (2.23) gives $c_0 = 1.5$ and $c_1 = -0.5$. Resulting in the following approximate equation for the reciprocal:

$$\frac{1}{B} \approx (3 - B)/2. \tag{2.24}$$

Equation (2.24) is easily implemented in hardware by one subtraction and one left shift. This implementation will result in a function that has at least 3 bits of precision over the entire range [1].

It is possible to increase the precision of this method by dividing the range of B in to smaller sections and calculate optimal constants for each of these smaller sections, see figure 2.3. These constants are then stored in lookup tables and retrieved for each calculation. This method will then require to table accesses one multiplication and one addition to calculate an approximate value.

The polynomial in equation (2.22) may also be used to implement an expression up to second order. This will give a more precise function approximation at the cost more complex implementation. By solving the following three equations for the edges and the midpoint of the range of B, a second order approximation is achieved.

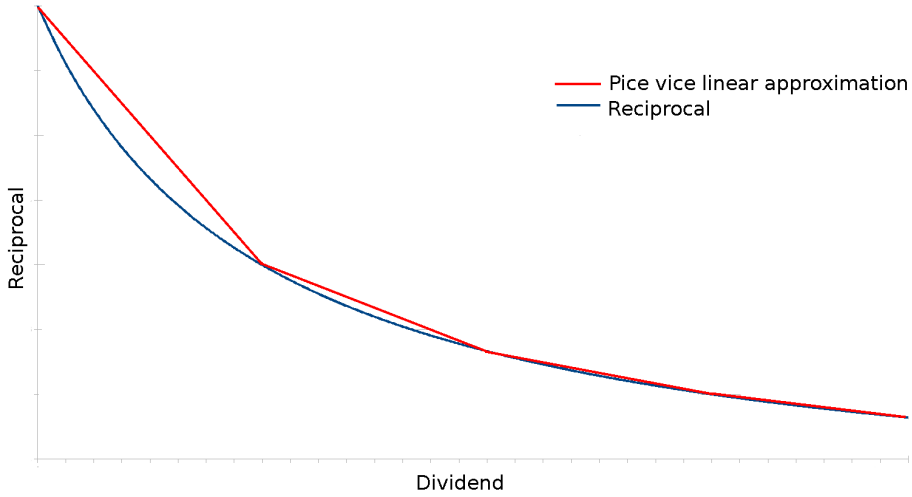

Figure 2.3: Piece vice linear approximation of the reciprocal $\frac{1}{B}$ given by the expression $\frac{1}{B} = c_0 + c_1 B$ with c_0 and c_1 stored in lookup tables

$$\begin{aligned} z(1.0) &= c_0 + c_1 \cdot 1.0 + c_2 \cdot 1.0^2 = 1.0 \\ z(1.5) &= c_0 + c_1 \cdot 1.5 + c_2 \cdot 1.5^2 = 1/1.5 \\ z(2.0) &= c_0 + c_1 \cdot 2.0 + c_2 \cdot 2.0^2 = 0.5 \end{aligned} \tag{2.25}$$

This will give the expression $z(B) = 2.166 - 1.5B + 0.333B^2$. The difference between this approximated function and the accurate value is given by $\varepsilon = z(B) - \frac{1}{B}$. This gives that the function will have an absolute maximal error of 0.0126 within the range of B, which corresponds to maximum of 6 bits of precision.

$$\text{precision} = \text{abs}(\log_2(0.0126)) = 6.310 \tag{2.26}$$

The constants calculated in this example will generate zero error at the limits of its range. By calculating other constants that distributes the error more evenly over the range of B it will be possible to slightly increase the precision of this second order approximation and achieve a few more bits of precision. An implementation of this second order approximation will require three multiplications and two subtractions or additions.

2.5.3 Table methods

As mentioned in section 2.5.2 table lookup can be used to increase the precision of linear approximation methods. Table methods may also be used in more general implementations to achieve good reciprocal approximation. Unlike linear approximation method or other methods that uses mathematical functions and methods to calculate an approximation towards the correct value, table methods store the function value in a lookup table. It should be noted that table methods may also be used get an approximation of other functions such as square root and trigonometric functions, but this has not been studied in this paper.

Table methods may be implemented in different ways. On solution is as described in previous section, to use table values as constants in a mathematical function that calculates an approximation towards the correct value. A more intuitive implementation is to store the function values it self in a lookup table. This method with k bits in and n bits out would require a lookup table with $2^k \cdot n$ bits. For most cases such an implementation will result in a lookup table that is much bigger than needed [5]. To improve on this several studies has been preformed to find methods to compress the tables.

Bipartite lookup tabels

One way to compress the size of the lookup tables is to use a method known as bipartite lookup tables. This method was presented by Das Sarma and Matula in [4]. The lookup table is divided in to two parts whose total size is smaller than the size of one complete table. The first table uses the most significant bits of the input word as address to a table holding a rough approximation to the reciprocal. The second table holds adjustment values that are added to the rough approximation from the first table. These adjustment values are divided in to sections of the input values range, and only stored once for each section. This way the total number of bits is reduced. The address in the second table is concatenated from the most, and the least significant bits in the input word. This is illustrated in Figure 2.4.

Figure 2.5 illustrates the values achieved after summation of the two tables. Since the adjustment values only are stored once in each segment there will be a saving in the total size.

The speed of this implementation will be limited by the speed of the memory interface and the summation logic used to add the two table values together. Das Sarma and Matula used a borrow-save representation of the values in the tables. These values are then added using a borrow-save adder circuit. This adder eliminates the carry propagation delay which limits the speeds of regular carry propagate adders.

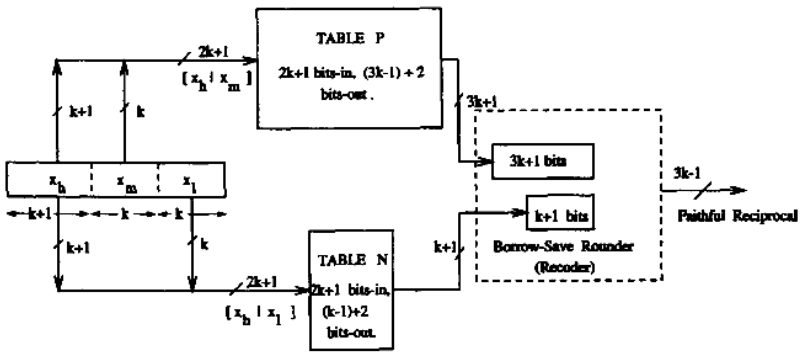

Figure 2.4: Illustration of bipartite lookup table [4], with a rough approximation stored in the P table while adjustment values are stored in the N table.

Multipartite lookup tables

Bipartite table methods compressed the size of lookup table by dividing the address space of the lookup tables in to two smaller tables. Multipartite table methods are an extension of this method by dividing the address space in to several parts. This has been studied in [5], [21] and [22]. As with bipartite lookup table methods the most significant bits of the input word is used to address a table that holds a rough approximation towards the reciprocal. The reminding bits of the input word are used in conjunction with some of the most significant bits to address other tables holding adjustment values to the rough approximate value. These other table values are added together with the initial approximate value to achieve a precise value for the reciprocal. This method is illustrated in figure 2.6.

Figure 2.7 illustrates how the table values will be added up to form an approximation towards the reciprocal using three tables. In the figure the red squares represents the values stored in the first table. While the blue dots represents the values stored in the first adjustment table. These values are only stored once for each segment which is highlighted in the figure. The green dots represent the values stored in the second adjustment table an gives adjustment values for even finer steps. In the illustrated example the first table holds 4 initial values, and the two other tables hold 8 adjustment values. This gives a total of 20 table entries as opposed to 64 if all values were to be stored in one table. The illustrated values use a linear approximation towards the reciprocal. By using higher order approximation it might be possible to achieve even better approximation.

By increasing the number of tables it will up to a point be possible to reduce the total memory needed for the tables. The optimal number of tables will be limited by the complexity of the addition logic, which will be lager and slower when the number of tables is increased. In addition, due to rounding error introduced by dividing each table in to smaller parts, extra guard bits has to be added, this will eventually increase the total tables size.

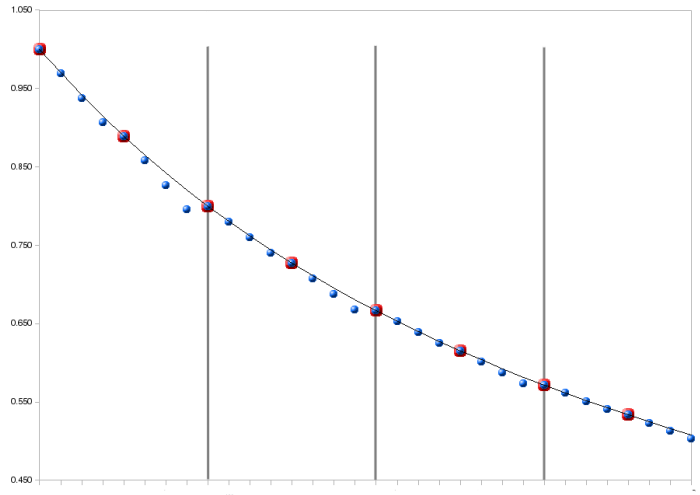

Figure 2.5: Illustration of values achieved after summation of the two tables. Red dots represent the rough approximation, while blue dots represent values after adding with the adjustment values.

Symetry

To improve the compression achieved by bipartite or multipartite lookup tables a method described by Stine and Schulte in [23] can be implemented. By adjusting the values stored in the first table, the adjustment values will be almost symmetrical around its midpoint. This is illustrated in figure 2.8. Since the values are symmetrical it is possible to reduce the size of the adjustment table by only storing half of the values. This implementation comes at the cost of some extra XOR gates on each of the adjustment tables.

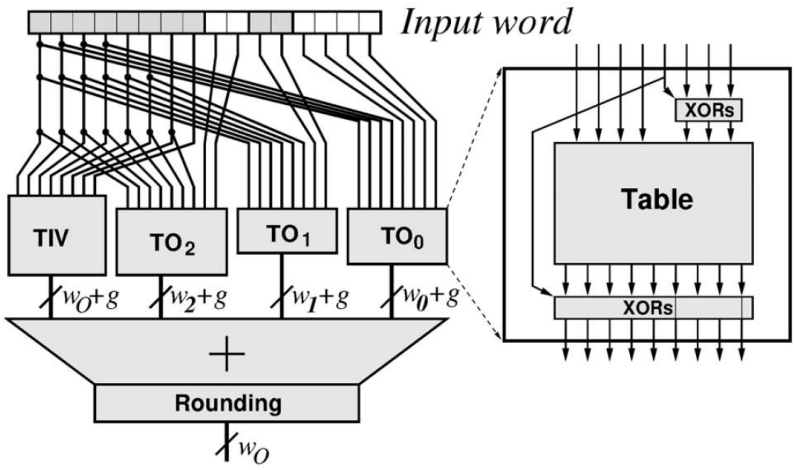

Figure 2.6: Example of implementation of multipartite table lookup method with utilisation of symmetry [5]

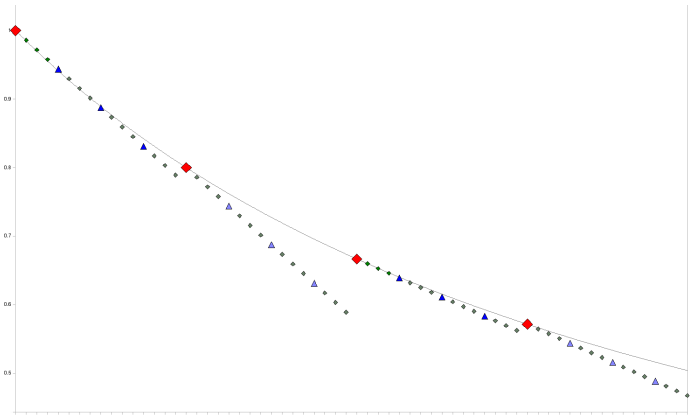

Figure 2.7: Illustration of valuse achived after adding together the multipartite tables.

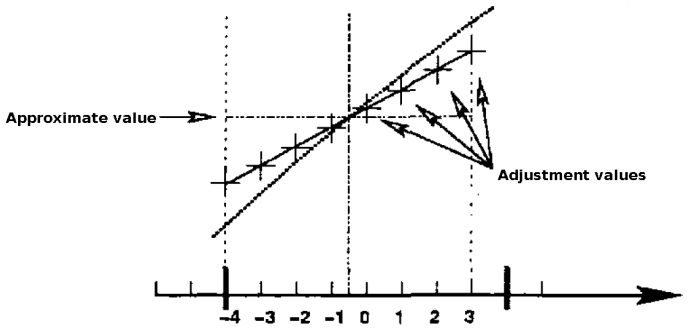

Figure 2.8: Illustration of symmetrical adjustment values [5]

CHAPTER

3

IMPLEMENTATION OF TABLE GENERATOR

The theory study on iterative division methods showed that they depend on an initial approximation of the reciprocal. Depending on the precision of this initial approximation the complexity, latency and area of the iterative circuit will wary. Based on the authors observations and on previous results from Rognerud [2] and Stafto [3] on the subject it was found that the best initial value implementation would be to use multipartite lookup tables.

To find a optimal implementation of multipartite lookup tables is was decided do a C++ implementation of a method described by Dinechin and Tisserand in [5] and [21]. This method is valid for all functions that are monotonically increasing or decreasing in the evaluated range for both the function it self and its derivative. In [5] the method has been described in a general case for several function implementations such as trigonometric and square root in addition to reciprocal function. In this assignment only implementation for reciprocal function has been evaluated.

Dinechin and Tisserand have described four steps involved in generating the lookup tables. These steps are as following:

1. Choose the number of table partitions that is wanted for the implementation. The larger the number of partition the more complex the addition and rounding circuit will be, but there is a possibility of achieving smaller tables.

2. Find all possible ways to decompose the input word in to table addresses.
3. For all decomposition calculate the approximation error and keep only those that have an error smaller than what may be permissible.
4. Calculate the actual table values and synthesize the tables and addition logic for some of the smallest table sizes. Choose which implementation to use based on size and speed of the implementation.

3.1 Lookup table decomposition

The C++ program developed starts by finding all decompositions of the input word according to figure 3.1. The part A is used to address a table called table of initial values (TIV) that holds a rough approximation, while a concatenation of part C_i and B_i is used to address tables, called table of offset (TO), that hold the adjustment values. In [4] Sarma and Matula implemented a method for bipartite lookup tables. They divided the input word in to predefined parts, where the part A was 2/3 of the input bits while C_i and B_i was 1/3 of the input bit width each. Compared to Sarma and Matulas method Dinechin and Tisserands method will be able to test all decompositions and find that which will give the smallest lookup table size.

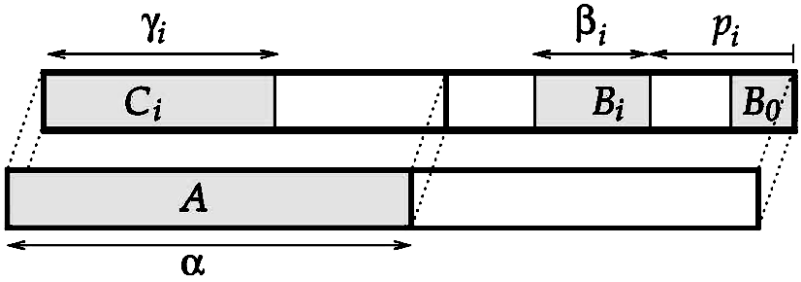

Figure 3.1: Input word decomposition [5]. The names in the figure are used in the following equations. A, B_i and C_i represents the value of the input word segments while α, β_i, γ_i and p_i represents the bit widths of these segments.

3.2 Calculating error

When choosing which decomposition to implement several criteria have to be evaluated. The most important of them is that the output result must be correct. This means that the output error must be less than on unit of least precision (ULP) of the output value. This gives the following limit on the output error when using w_O bits out:

$$\varepsilon_{\text{totoal}} < 0.5 \cdot 2^{-w_O}. \tag{3.1}$$

Dinechin and Tisserand [5] has described three different calculation errors that is involved in the lookup tables. These are:

- The input discretization or quantization error measures the fact that an input number usually represents a small interval centers around this number.
- The approximation or method error measures the difference between the pure mathematical function f and the approximated mathematical function.
- Finally, the actual computation involves rounding error due to the discrete nature of the final and intermediate values.

In the implemented method described by Dinechin and Tisserand the first error is ignored. This is because the input values to the division circuit are regarded as infinite precise. The two other errors, the approximation error and the rounding error will need to be evaluated for each of the decompositions. Since these errors will be different for each of the decompositions they must be calculated for every one of them, and those that do not satisfies the error limit must be discarded.

The approximation error is a result of the method being a linear approximation towards the reciprocal. The total error is calculated by summing up the error contributed by each of the TOs. Figure 3.2 illustrates this error. Since the derivate of the reciprocal function is monotonic decreasing over the entire valid range of the approximation, the error will be greatest at the borders of the interval. By adjusting the slope and the midpoint of the graphs such that the error is evenly distributed a minimal error will be achieved. This will then give an error for each TO expressed by:

$$\varepsilon_1 = -\varepsilon_2 = -\varepsilon_3 = \varepsilon_4 = \varepsilon_i^D(C_i). \tag{3.2}$$

By solving equation (3.2) it is possible to find the maximal error that each TO will contribute. This error is given by:

$$\begin{aligned} \varepsilon_i^D(C_i) &= \frac{f(x_2) - f(x_1) - f(x_4) + f(x_3)}{4} \\ &= \frac{\frac{1}{x_2} - \frac{1}{x_1} - \frac{1}{x_4} + \frac{1}{x_3}}{4}. \end{aligned} \tag{3.3}$$

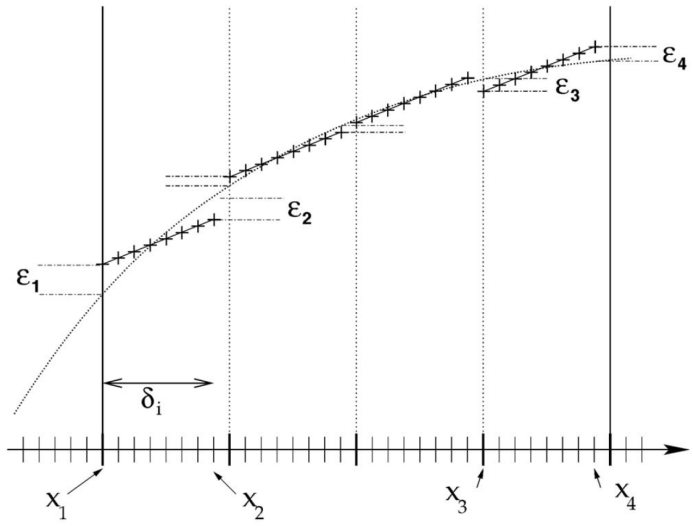

Figure 3.2: Illustration of the errors occurring in each TO [5].

$$\delta_i = 2^{-w_i+p_i}(2^{\beta_i} - 1) \tag{3.4}$$
$$x_1 = 1 + 2^{-\gamma_i}(C_i) \tag{3.5}$$
$$x_2 = x_1 + \delta_i \tag{3.6}$$
$$x_3 = x_1 + 2^{-\gamma} - 2^{-w_i+p_i-\beta_i} \tag{3.7}$$
$$x_4 = x_3 + \delta_i. \tag{3.8}$$

As seen in equations (3.4) to (3.8) the error from each TO is dependent on the segment given by C_i. Since this metode works on monotonic increasing or decreasing function this error will always be greatest at the limits of C_i when $C_i = 0$ or $C_i = 2^{\gamma_i} - 1$. For this implementation when $f = \frac{1}{x}$ the point with the greatest error will be when $C_i = 0$. The total error for the decomposition is then given by adding up the approximation error for all m TOs:

$$\varepsilon_{approx} = \sum_{i=0}^{m-1} \varepsilon_i^D(0). \tag{3.9}$$

The total error in each decomposition is the sum of the approximation errors and the rounding error introduced when adding together the table values. This will then give that the maximum approximation error is expressed by:

$$\varepsilon_{approx} < 0.5 \cdot 2^{-(w_O+1)}. \tag{3.10}$$

To ensure that the rounding error introduced will be less than the error limit, some extra guard bits have to be added to the table values. The number of extra guard bits needed is calculated by the following formula:

$$g = \left\lceil -w_O - 1 + \log_2 \left(\frac{0.5m}{0.5 \cdot^{-w_O-1} - \varepsilon_{approx}} \right) \right\rceil. \tag{3.11}$$

3.3 Calculating table size

After the number of guard bits has been calculated it is possible to calculate the total number of bits needed to store for each decomposition. Based on this size it is possible to choose which implementation that will give the optimal size. It is worth noting that when implementing these tables in FPGA, the decomposition giving the least number of bits may not be the optimal implementation. This is because the embedded block RAM in the FPGAs is optimized for certain data and address bit widths. If on table is not able to fit in to one RAM block due to data or address width mismatch it will need to use more blocks and thereby might waste some bits in each RAM block. For this reason several of the smallest decompositions should be synthesized to see which will actually give the smallest implementation.

The total size of the TIV table is given by $2^{\alpha}(w_O + g)$. Since the range of th TOs is smaller than the range of the TIV they do not need the same bit width. The maximum range of the output values in each segment, given by C_i, in each TO is calculated by multiplying the average slope of each segment with the range of the input value in each segment. The average slope of each segment is given by following equation:

$$\begin{aligned} s_i(C_i) &= \frac{f(x_2) - f(x_1) + f(x_4) - f(x_3)}{2\delta_i} \\ &= \frac{\frac{1}{x_2} - \frac{1}{x_1} + \frac{1}{x_4} - \frac{1}{x_3}}{2\delta_i}. \end{aligned} \tag{3.12}$$

The values for x_i and δ is given by equations (3.4) to (3.8). As with the maximum approximation error, the maximum slope will be when $C_i = 0$. The maximum range of the TO_i is then given by $r_i = |s_i(0) \cdot \delta_i|$. The output width of TO_i is then given by:

$$w_{TO_i} = \left\lceil w_O + g - \log_2\left(\frac{0.5}{r_i}\right)\right\rceil. \tag{3.13}$$

By utilizing symmetry in the TOs, the size of TO_i is then given by $2^{\gamma_i+\beta_i-1}(w_{TO_i}-1)$. The total bit size for the implementation is then given by adding up the size of the TIV and the TOs.

3.4 Filling the tables

The optimal value for TIV(A) is found to be the mid point of the range interval represented by A. This interval is limited by x_l and x_r given by the following equation:

$$\begin{aligned} x_l &= 1 + 2^{-\alpha}A \\ x_r &= x_l + \sum_{i=0}^{m-1}\delta_i. \end{aligned} \tag{3.14}$$

The accurate value, before rounding, for the TIV is then given by:

$$\widetilde{TIV}(A) = \frac{f(x_l) + f(x_r)}{2}. \tag{3.15}$$

and the accurate values for TO_i is given by:

$$\widetilde{TO}_i(C_iB_i) = s(C_i)\cdot 2^{-w_{TO_i}+p_i}\left(B_i + \frac{1}{2}\right). \tag{3.16}$$

The infinite precise table values must be rounded to integer values that are possible to store in the tables. This will then give the following table values for the TO_i:

$$TO_i(C_iB_i) = \left\lfloor \frac{2^{w_O+g}}{0.5}\cdot \widetilde{TO}_i(C_iB_i)\right\rfloor. \tag{3.17}$$

And for the TIV when the number m of TOs is odd:

$$TIV(A) = \left\lfloor 2^{w_O+g}\cdot\frac{\widetilde{TIV}(A) - 0.5}{0.5} + \frac{m-1}{2} + 2^{g-1}\right\rfloor, \tag{3.18}$$

or if m TOs is even:

$$TIV(A) = \left\lfloor 2^{w_O+g} \cdot \frac{\widetilde{TIV}(A) - 0.5}{0.5} + \frac{m}{2} + 2^{g-1} \right\rfloor . \tag{3.19}$$

After calculating the table value the C++ program generated synthesizable VHDL code for the lookup tables.

3.5 Results from the implementation

In the implemented C++ code it was also implemented a test sequence that calculated the output value for all input values as it would have been when implemented in hardware. Unfortunately this test revealed that for some input values the output value would be one ULP above the correct value. This error in the output value seemed to be randomly distributed over the input range, and it was not possible to detect any correlation between the table ranges and the wrong output values.

Some effort was done to try to fix this error but unsuccessful. Instead it was decided to continue with focus on the division algorithms and implementation.

CHAPTER

4

EVALUATION OF DIVISION ALGORITHMS

In chapter 2 the most known division algorithms have been thoroughly investigated. Based on this theory an evaluation of the different algorithms will be performed in this chapter.

4.1 Digit recurrence algorithms

Several digit recurrence algorithms were presented in section 2.1. Common for all of these algorithms is that they calculate a number of digits for each iteration. More complex implementation with higher radix allows for more digits to be calculated at each iteration. Common for all implementation is that there are one or more comparisons involved. These will often be implemented using subtractors which will have a carry chain which is equal in length to the bit width. This carry chain will be a major limiting factor for the speed of such an implementation.

In this project the focus has been on an implementation with high throughput. A radix-2 implementation of a digit recurrence algorithm will calculate one digit for each iteration. For such an implementation to achieve a throughput of one data set on each clock, the circuit must implement as many stages as there are bits in the operands. In [24] an iterative radix-2 digit recurrence circuit has been presented, here each stage needs three n bit wide registers. One register holds the divisor one register holds the current remainder, while the

last register holds the currently calculated bits in the quotient and the reminding bits in the dividend. This results in an implementation that needs at least $3n^2$ bits of memory to hold the intermediate values, where n is the bit width. A 64-bit implementation in FPGA will then need to use $3 \cdot 64^2 = 12288$ slice flip flops for this data. In addition there will be some logic involved with each stage. Due to the design of the logic slices in FPGAs most of this logic will be placed in side of the slices already used to store the data, and therefore not necessarily take up much more space.

By implementing each stage with higher radix such as radix-4 which calculates 2 bits of the answer in each iteration, the depth of the total circuit will be halved and therefore also the need for temporary storage. This saving in memory usage comes at the cost of increased size of the logic. It is believed that the size of the selection logic will be greater than the reduction in memory usage, and therefore create a circuit with greater size.

4.2 Goldschmidt method

The Goldschmidt method is one of the multiplicative division methods, which all have quadratic rate of convergence. This means that for each iteration the number of correct bits is doubled.

In [8] a pipelined FPGA implementation of a multiplicative division algorithm using Goldschmidt method has been evaluated. In contrast to this assignment their focus has been on low latency rather than high throughput. The fact that the two multiplications in Goldschmidt method are independent makes this method ideal for low latency implementations. By executing each multiplication in parallel in each own multiplier, or executed after each other in a pipelined multiplier, the total number of cycles will be reduced.

The input to the Goldschmidt method is both the dividend and the divisor. For the method to calculate correct these operands must be normalized as described in section 2.4. This is no problem in most implementations that have been investigated since they are developed for floating point numbers which always is normalized. Since this implementation is for integer numbers they has to be normalized first, which means that there has to be two normalizing circuits, each consisting of a high bit decoder and a barrel shifter.

One advantage of Goldschmidt method is that it calculates on both the dividend and divisor and therefore the output will be the quotient. This eliminates the need for any additional stages to calculate this.

The weakness of Goldschmidt method is that it accumulates truncation error for each iteration it performs. To control this additional guard bits must be added to the operand bit width in all stages. This results that all iteration stages implemented must have full

bit width.

Figure 4.1 illustrates an assumed implementation of the Goldschmidt method.

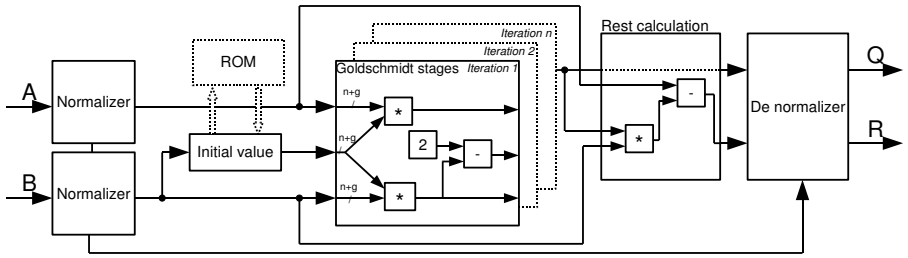

Figure 4.1: Drafted implementation of the Goldschmidt method

The output of the final stage in the Goldschmidt method is the normalized quotient. This then has to be de-normalized. In this assignment the remainder is of equal importance. This has to be calculated after the iteration stages. The remainder is calculated using equation (4.1). To calculate the remainder both the dividend and divisor is needed. This means that these have to be stored through the entire operation.

$$\text{Rest } = A - \left\lfloor \frac{A}{B} \right\rfloor \cdot B \tag{4.1}$$

4.3 Newton-Raphson method

As with the Goldschmidt method the Newton-Raphson method is a multiplicative division method with quadratic rate of convergence. The Newton-Raphson method is often the preferred choice when implementing division circuits [20]. The reason for this is because this method is self correcting, which means that truncation error in each iteration is corrected in the next iteration. This makes this method easier to verify and the size of the implementation will also often be smaller due to reduced size of the multipliers. This comes at the cost of longer latency since the two multiplications is dependent.

In figure 4.2 a draft of an assumed implementation using Newton-Raphson method is presented. Since the Newtons-Raphson method only calculates on the divisor, only the divisor need to be normalized. As with Goldschmidts method the divisor is used to find an initial approximation to the reciprocal. The approximated reciprocal and the divisor are then applied in to the Newton-Raphson stages. Each of these stages consists of two multiplications in series with a subtraction between. The number of Newton-Raphson stages needed in the implementation is dependent on the precision of the initial approximation.

While the output of the Goldschmidt method is the quotient, the output of the Newton-Raphson method is the reciprocal. This means that the quotient has to be calculated in an extra quotient calculation stage. Finally after calculation of the quotient, the remainder is calculated the same way as in the Goldschmidt method.

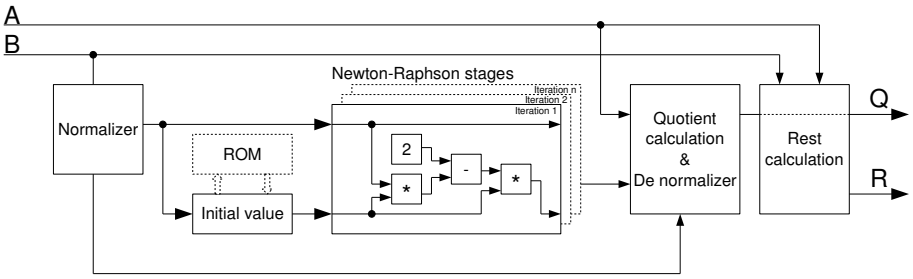

Figure 4.2: Drafted implementation of the Newton-Raphsons method

4.4 Selection of division algorithm

The digit recurrence class of algorithms is the easiest class of division circuit to implement when the radix of such implementations is kept low. The draw back of this method is its slow convergence rate, which is linear versus quadratic for multiplicative methods. To compensate for this slow rate of convergence the number of implemented stages must be high to still keep a throughput of one data set on each clock. In section 2.1 and estimated 12288 slice flip flops is needed to implement a fully pipelined implementation with 64 bit operands and throughput of one on each clock period. It is believed that an increase in radix of such an implementation will increase the complexity of the system and thus increase the total size with more what is gained by the reduction in number stages. Based on the size for this implementation it is found that this method is not suited for this implementation.

The input values in this implementation are integers. This means that for both the multiplicative methods the input parameters has to be normalized. One advantage the Newton-Raphson method has compared to Goldschmidt method is that it only computes on the divisor and hence only one normalizing circuit is needed. This comes at some cost, while the Goldschmidt method gives the quotient out directly, the Newton-Raphson method gives out the reciprocal. This means that the Newton-Raphson method needs an extra multiplication of the dividend and the reciprocal to find the quotient.

Both Newton-Raphson and Goldschmidt method converges quadratic from an initial value towards the target precision. This means that the precision of the initial approximation will equally influence the number of stages needed in both methods. In addition the method used to calculate the initial approximation may be implemented equally for both methods.

None of the multiplicative methods calculates the remainder, so this has to be calculated later. To calculate the remainder both the dividend and divisor is needed, which means that they must be stored thought the division operation. The number of bits needed to store this data is dependent on the pipeline depth of the method implemented. This gives the Goldschmidt method an advantage since the multiplications is executed in parallel, which give a shorter depth.

In the Newton-Raphson method it is possible to reduce the size of the first stages since truncation error in each stage is corrected in next stage. Due to the reduced size of the multipliers is may also be beneficial to reduce the precision of the initial approximation. This will reduce the size of the lookup tables used, while only adding some more iteration stages which has significantly smaller size than the reminding stages. Equal saving in space will not be possible with Goldschmidts method since each multiplier needs full bit width in addition to guard bits to control the accumulated truncation error.

Based on these observations the Newton-Raphson method was found to be the best solution for this implementation.

CHAPTER

5

IMPLEMENTATION OF NEWTON-RAPHSON SIMULATION MODEL

In previous chapter it was found that the Newton-Raphson method would be the best to implement. This was based on the assumption that it would be possible to truncate the first stages in the implementation and thereby save area. To verify this assumption, and to find how many bits that is needed in each stage an analysis of the truncation error in the Newton-Raphsons method is performed, and then verified in a simulation model.

5.1 Truncation error in Newton-Raphson method

To find the needed bit width through the Newton-Raphson stages it is necessary to find the maximum allowable error at the output, and then calculate the needed bit width backwards through the circuit.

The output quotient Q form the division circuit will be a n bit integer number. This number will be in the range $Q \in [0, 2^n - 1]$ and the least significant bit will obviously have value 0 or 1. This means that for the output error to not be significant, it must be less than one before the output is truncated. As shown for the Newton-Raphson method, the

quotient is calculated by a final multiplication between the reciprocal and the dividend. By adding in the truncation error ε in each term it is possible to derive an expression for the allowable error in the final reciprocal:

$$Q + \varepsilon_Q = \left(\frac{1}{B} + \varepsilon_{\frac{1}{B}out}\right) \cdot (A + \varepsilon_A)\,. \tag{5.1}$$

In equation (5.1) $\varepsilon_{\frac{1}{B}out}$ and ε_A represents the truncation error in the reciprocal out from the Newton-Raphson stages and the dividend respectively. The input dividend may be regarded as infinite precise and the error associated by it may therefore be regarded as zero. This makes it possible to derive the following equation for the output error ε_Q:

$$\begin{aligned} Q + \varepsilon_Q &= \left(\frac{1}{B} + \varepsilon_{\frac{1}{B}out}\right) \cdot (A + \varepsilon_A) \\ \varepsilon_Q &= \left(Q - \frac{1}{B} \cdot A\right) + \varepsilon_{\frac{1}{B}out} \cdot A \\ \varepsilon_Q &= \varepsilon_{\frac{1}{B}out} \cdot A. \end{aligned} \tag{5.2}$$

In equation (5.2) it is shown that the final error is dependent on the truncation error in the reciprocal and the dividend. The value of the, n bit, dividend A will lay in the range $A \in [0, 2^n - 1]$, this means that the maximum error out will be $\varepsilon_{\frac{1}{B}out} \cdot (2^n - 1) \approx \varepsilon_{\frac{1}{B}out} \cdot 2^n$. This gives the following bound on the error output from the Newton-Rapson stages:

$$\begin{aligned} &0 \leq \varepsilon_Q < 1 \\ &0 \leq \varepsilon_{\frac{1}{B}out} \cdot 2^n < 1 \\ &0 \leq \varepsilon_{\frac{1}{B}out} < 2^{-n}. \end{aligned} \tag{5.3}$$

From equation (2.5) in section 2.2.1 we know that the Newton-Raphson method is expressed as $z_i(2 - Bz_i)$. By inserting error terms for all truncations that occurs in an implementation it is possible to evaluate how the truncations influence the total error out.

$$\begin{aligned} z_{i+1} &= z_i(2 - Bz_i) \\ &= \max\left[((z_i + \varepsilon_{z_i}) \cdot (2 - ((B + \varepsilon_B) \cdot (z_i + \varepsilon_{z_i})) + \varepsilon_1)), z_i(2 - Bz_i) + \varepsilon_2\right] \end{aligned} \tag{5.4}$$

In equation (5.4) ε_{z_i} represents the error introduced by the initial approximation to the reciprocal. ε_B represents the error introduced by truncating the divisor, and ε_1 represents

the truncation error introduced by truncating the first multiplication. In addition there is a possibility to introduce truncation error by truncating the second multiplier. Since this error is added after the other calculations it will not influence the result directly. By ensuring that the truncation error out of the last multiplier is less than the truncation error in the rest of the circuit it will have no influence.

By expanding equation (5.4) and substituting z_i with $\frac{1}{B}$ and the following expression for the truncation error in one Newton-Raphson stage is achieved:

$$\begin{aligned}\frac{1}{B} + \varepsilon_{\frac{1}{B}out} &= \left(\frac{1}{B} + \varepsilon_{\frac{1}{B}in}\right) \cdot \left(2 - B\frac{1}{B} - B\varepsilon_{\frac{1}{B}in} - \frac{1}{B}\varepsilon_B - \varepsilon_B\varepsilon_{\frac{1}{B}in} + \varepsilon_1\right) \\ &= \left(\frac{1}{B} + \varepsilon_{\frac{1}{B}in}\right) \cdot \left(1 - B\varepsilon_{\frac{1}{B}in} - \frac{1}{B}\varepsilon_B - \varepsilon_B\varepsilon_{\frac{1}{B}in} + \varepsilon_1\right) \\ &= \frac{1}{B} - \frac{\varepsilon_B}{B^2} - 2\frac{\varepsilon_B\varepsilon_{\frac{1}{B}in}}{B} - B\varepsilon^2_{\frac{1}{B}in} - \varepsilon_B\varepsilon^2_{\frac{1}{B}in} + \frac{1}{B}\varepsilon_1 + \varepsilon_1\varepsilon_{\frac{1}{B}in}.\end{aligned} \tag{5.5}$$

The divisor B in the Newton-Raphson module is a normalised representation of the divisor to the range $B \in [1, 2\rangle$. This is used to simplify equation (5.5) by substituting all fractions of $\frac{1}{B}$ and $\frac{1}{B^2}$ with its maximum value of 1. Also by subtracting with the reciprocal $\frac{1}{B}$ in the equation all that is left is the truncation errors. This gives:

$$\varepsilon_{\frac{1}{B}out} = -\varepsilon_B - 2\varepsilon_B\varepsilon_{\frac{1}{B}in} - B\varepsilon^2_{\frac{1}{B}in} - \varepsilon_B\varepsilon^2_{\frac{1}{B}in} + \varepsilon_1 + \varepsilon_1\varepsilon_{\frac{1}{B}in}. \tag{5.6}$$

From equation (5.6) it is possible to evaluate the error contributed by each truncation. This is gives the following equations:

$$-\varepsilon_B - 2\varepsilon_B\varepsilon_{\frac{1}{B}in} - \varepsilon_B\varepsilon^2_{\frac{1}{B}in} = -(1 + \varepsilon_{\frac{1}{B}in})^2 \cdot \varepsilon_B \approx -\varepsilon_B \tag{5.7}$$

$$\varepsilon_1 + \varepsilon_1\varepsilon_{\frac{1}{B}in} = (1 + \varepsilon_{\frac{1}{B}in})\varepsilon_1 \approx \varepsilon_1 \tag{5.8}$$

$$B\varepsilon^2_{\frac{1}{B}in} < 2\varepsilon^2_{\frac{1}{B}in}. \tag{5.9}$$

Equations (5.7) and (5.8) is the total error contributed by truncation of the divisor, and truncation after the first multiplier respectively. These errors are influenced by the error $\varepsilon_{\frac{1}{B}}$ on the approximation of the reciprocal. This error is always much smaller than one, which leads to the simplification that $(1 + \varepsilon_{\frac{1}{B}in}) \approx 1$.

The error introduced by the initial approximation of the reciprocal is given in equation (5.9). Again since B is in the range $B \in [1, 2\rangle$, it is possible to substitute B with its

maximum value. It is worth noting here that the error term $\varepsilon_{\frac{1}{B}in}$ is squared, this confirms that the Newton-Raphson method has quadratic rate of convergence.

This gives the following expression for the error in the Newton-Raphson stages:

$$\varepsilon_{\frac{1}{B}out} \approx \varepsilon_1 - \varepsilon_B + 2\varepsilon^2_{\frac{1}{B}out}. \tag{5.10}$$

It was showed in equation (5.3) that the total error out of the Newton-Raphson stages must be in the range $0 \leq \varepsilon_{\frac{1}{B}out} < 2^{-n}$. To keep this error bound while still keeping the bit width as small as possible this error must be evenly distributed amongst the three error terms. This gives:

$$0 \leq \varepsilon_1 < \frac{2^{-n}}{4} \tag{5.11}$$

$$0 \leq -\varepsilon_B < \frac{2^{-n}}{4} \tag{5.12}$$

$$0 \leq 2\varepsilon^2_{\frac{1}{B}in} < 2 \cdot \frac{2^{-n}}{4}. \tag{5.13}$$

From equation (5.11) we see that the error introduced by truncating after the first multiplier must be less than $2^{-(n+2)}$. This will be achieved by keeping at least $n+2$ bits of precision after the decimal point. The total number of bits needed for each signal is determined by its range and its precision. For this signal the range is determined by the range of the two inputs to the multiplier. The lower range of these two inputs is 1.0 and 0.5, this gives $1.0 \cdot 0.5 = 0.5$ as the lower range of the output, while the upper range of the inputs is 2.0 and 1.0, which gives $2.0 \cdot 1.0 = 2.0$ as the upper range. The range of the output is then $[0.5, 2.0\rangle$.

To achieve the range up to 2.0, one bit is needed left of the decimal point. This gives that a total of $n+3$ bits is needed out of the first multiplier.

The maximum acceptable error introduced by truncating the divisor is given by equation (5.12). Since truncation will create positive error, this error bound will obviously not hold. To solve this, a constant has to be added so that the error will always be positive. This gives

$$\begin{aligned} 0 &\leq 2^{-(n+2)} - \varepsilon_B < 2^{-(n+2)} \\ -2^{-(n+2)} &\leq -\varepsilon_B < 0 \\ 2^{-(n+2)} &\geq \varepsilon_B > 0. \end{aligned} \tag{5.14}$$

As for error ε_1 the error bounds for ε_B is kept by using $n+2$ bits after the decimal point of the divisor. The exception for this is the last stage. Here the entire width is used and since this may be regarded as infinite precise no more extra guard bits is needed. The range of the B is $[1.0, 2.0\rangle$ This is the same as for the output of the first multiplier which gives that this has to have $n+3$ bits.

The precision of the initial approximation is given by equation (5.13). This gives that the error must be less than $\varepsilon_{\frac{1}{B}in} < 2^{\frac{n}{2}+1}$. This gives that the input reciprocal must have $n/2+1$ bits of precision after the decimal point. The range of the reciprocal has a maximum of 1, this means that no bits is needed left of the decimal point to represent the number. This gives that the total needed bitwidth in is $n/2+1$.

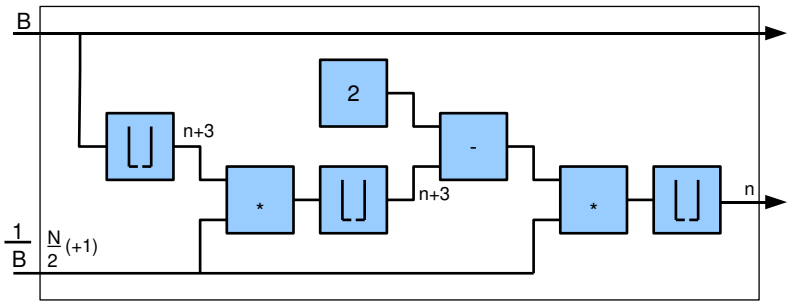

Figure 5.1: Illustration of the needed bit width in the Newton-Raphson stages. The $\lfloor\ \rfloor$ illustrates where the truncation is performed.

Figure 5.1 gives an illustration of the truncation and the bitwidths through one Newton-Raphson stage. It may be noted that the calculations has found that the precision on the input approximation need to be $n/2+1$ of the output precision. This contradicts the previous notion that Newton-Raphson method wills double the number of correct bits for each iteration. It is presumed that this extra bit needed is a result of worst case calculations used thought the calculations. In equation (5.6) all fractions of $\frac{1}{B}$ was substituted with its maximal value of 1, and all B's were substituted with its maximal value of 2. This two worst cases will obviously not occur simultaneously, and it might therefore still be possible to use only $n/2$ bits of precision on the input approximation. This will be tested in the next section.

In the case that the tests revile that the extra input precision is needed then the precision out of the second last stage must be increased. To achieve this the input of the second last stage must be $\frac{\frac{n}{2}+1}{2}+1 = \frac{n}{4}+\frac{1}{2}+1$ bits. And then for the third last stage the needed input precision will be $\frac{n}{4}+\frac{1}{4}+\frac{1}{2}+1$ bits. The needed extra bits on the initial approximation may be expressed as a geometric series, and the value of the n'th term will go towards two as n goes towards infinite. From this is can be concluded that by adding two extra guard bits on the initial approximation this error will be guarantied to be less than the maximum limit.

5.2 Test of truncation calculation

To verify the calculated bit width through the Newton-Raphson stages were correct, a simulation model was implemented in VHDL and simulated using ModelSim from Mentor Graphics. Since this simulation model was designed to verify the calculated bit widths through the Newton-Raphson stages, it was not implemented as a synthesizable code. This was done by implementing the entire code in one sequential block.

5.2.1 Normalization logic

The normalization of the operands were implemented using a while loop that left-shifted the divisor until the most significant bit were '1' or until the divisor were sifted n positions, which implied a division by zero.

5.2.2 Initial value calculation

For simplicity it was decided not to implement any of the initial approximation methods examined in section 2.5. Instead the actual reciprocal was calculated using real numbers in VHDL, and then truncate it to the implemented bit width.

During development of the reciprocal approximation module a possible problem was discovered. After truncating of the divisor some values will be rounded down to 1.0, as illustrated in:

$$\begin{aligned} &1.000011010\ldots \\ &1.0000 \end{aligned} \tag{5.15}$$

The input range of the Newton-Raphson method is limited to $\langle 1.0, 0, 5]$. Since truncation of the divisor might lead to an approximation of the reciprocal $\frac{1.0}{1.0}$ the answer will also be 1.0, which is just outside the valid range. This will result in overflow on the initial approximation, which means that the output will be 0.0 rather than 1.0. To solve this problem it was decided to decrease the reciprocal, when this accrue, by 1 ULP so that it would be represented as $0.111\ldots1$.

5.2.3 The Newton-Raphson stages

The Newton-Raphson stages used in the simulation model was implemented using the bit widths described in previous section, and $n/2$ bits on the initial approximation. In the

calculations there were some uncertainties about the input precision needed in each stage. To thoroughly test this it was decided to implement four stages, and iterate from 4 initial bits up to 64 bits out. It is believed that this will revile any errors that will propagate or accumulate through the stages.

In previous section it was found that an adjustment value had to be added to the truncated divisor to achieve correct result since the truncation error would be negative. This was implemented in the simulation model and tested thoroughly. The test showed this adjustment was not necessary and could be safely removed.

5.3 Test vectors used in the simulations

A fully test of the implementation with 64 bits dividend and divisor would require $2^{2 \cdot 64}$ input vectors. This is obviously not possible to simulate within reasonable time. To limit this, a number of presumed worst case test vectors were generated. These vectors included those that introduce most truncation error in each stage. To find these vectors three sets of vectors were generated.

The first set of vectors was walking one's patterns. In this set of vectors all possibilities with one bit high in both the dividend and the divisor is tested. A code for generating this test vector set is given in algorithm 5.1. These test vectors is presumed to detect all error due to implementation error in the normalization and de-normalization logic as well as some error in the Newton-Raphson stages.

Algorithm 5.1 Test vector set one. Nested loops for generating walking one's on both the dividend and the divisor.

```
1: for i = 0 to n − 1 do
2:    for j = 0 to n − 1 do
3:       A := (others=>'0')
4:       A(i) := '1';
5:       B := (others => '0')
6:       B(j) := '1'
7:       test A/B
8:    end for
9: end for
```

The second sets of test vectors were generated to detect truncation error in the Newton-Raphson stages. The vectors that will have most error due to truncation are those that have only one's at the end which are trimmed of. To find these vectors it was decided to implement a set of walking patterns. An illustration of these vectors is given in figure 5.2. Since these patterns were generated to test the Newton-Raphson module they were only applied to the divisor. Due to normalization of the divisor before entering the Newton-

1000...0001	walking pattern with one bit high
1000...0010	
⋮	
1100...0000	
1000...0011	walking pattern with two bits high
1000...0110	
⋮	
1110...0000	
1000...0111	walking pattern with three bits high
⋮	

Figure 5.2: Illustration of test vector set two and three. The patterns are generated with increasingly long sequences of one's that walk from right to left. Pattern three is an inverted representation of this pattern.

Raphson stages it was decided to set the most significant bit in all vectors to '1' to ensure that all test vectors would still be unique after normalization. The third set was generated by inverting the second set, but still keeping the most significant bit high. The entire test bench used to test the simulation model is listed in appendix A.

5.4 Results from simulations

To verify that the simulation model calculated correct answers, the correct answer was also calculated using the division and remainder operator in VHDL. These answers were then automatically compared in the test bench and a warning was printed about the occurrence of the error and which test vector that produced the error.

During the simulation it was found that some of the presumed worst case test vectors resulted in wrong output. These outputs were always one ULP below the correct value. Since these errors always were only one ULP wrong compared to the correct value it was decided to implement a correction stage at the end instead of adding two extra bits in every iteration stage. This stage is implemented as a digit recurrence stage. In this stage the divisor and the remainder is compared. If this comparison shows that the divisor is smaller than the remainder, the result is wrong. This is then fixed by subtracting the divisor from the remainder, and add one to the quotient.

New tests performed after the correction stage were implemented finished without detecting any errors.

CHAPTER

6

IMPLEMENTATION OF SYNTHESIZABLE CIRCUIT

After simulating on the simulation model a synthesizable circuit was developed. Through the development of this circuit it was a goal to achieve short critical paths so that the clock frequency will be as high as possible. In addition, since it is wanted to evaluate how different pipeline depth affects the circuit, the code is made so that it would be easy to change the pipeline depth.

Based on calculations on the needed bit widths through the Newton-Raphson stages and the size needed to implement a lookup tables that achieves faithful rounding, it was decided to implement a solution with only 4 bits precision on the initial approximation. This solution will only require $4 \cdot 2^4$ bits to be stored in a lookup table. Then to achieve 64 bits of final precision it is necessary to implement four Newton-Raphson stages. The increased size of the division circuit due to the added stages will be relative small compared to the last and largest stage. For each additional stage added the total increase in size will only be $\frac{1}{2^{k-1}}$ times that of the last stage when k is the total number of stages. In addition there will be an acceptable increase in pipeline stages in the circuit due to the added stages.

Figure 6.1 shows an overview of the implemented circuit. To enable easy modification of each block all signals is connected through the blocks and pipelined equally to the signal flow.

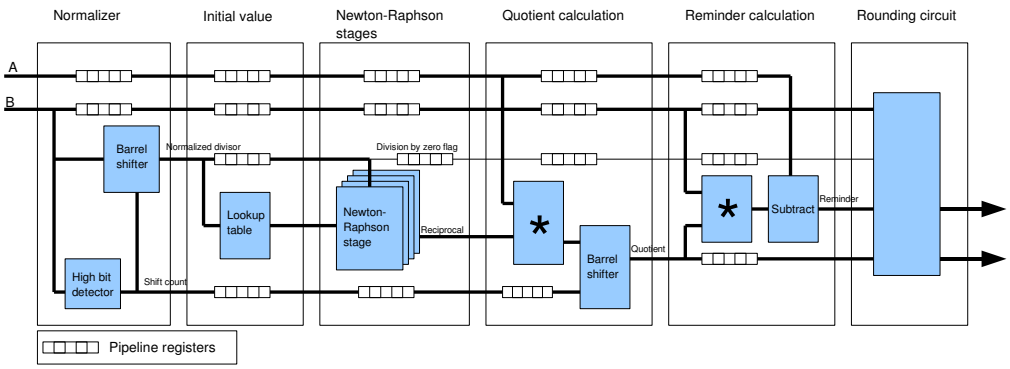

Figure 6.1: Illustration of the implemented circuit

6.1 Normalization circuit

The normalization circuit performs the task of normalizing the divisor as described in section 2.4. The process of normalizing the operands consists of two operations. First the position of the most significant '1' has to be detected. Then the divisor must be shifted left accordingly by a barrel shifter. Finally both the shifted divisor and the shift count is sent out of the module.

6.1.1 Most significant high bit detection

The circuit implemented to detect the position of the most significant '1' is illustrated in figures 6.2 and 6.3. The circuit in figure 6.2 is implemented to isolate the most significant high bit. As seen in the figure this is implemented by use of chains with "or" and "and" gates. This will then result in a "hot one" coding of the position.

One continuous chain of "or" and "and" gates would result in a critical path that is equal in length to the bit width of the signal. To resolve this it was decided to split the chains in to 8 parts and then use some extra logic to detect which of the eight detectors that has the most significant high bit. This is illustrated in figure 6.3. The three single output lines from the circuit will give the most significant position coding, while the 8 bit bus will give a "hot one" coding of the least significant position coding. These signals is then coded to binary form and then sent to the de-normalization circuit. The Critical path in the system will then be as illustrated by the red line in figure 6.3.

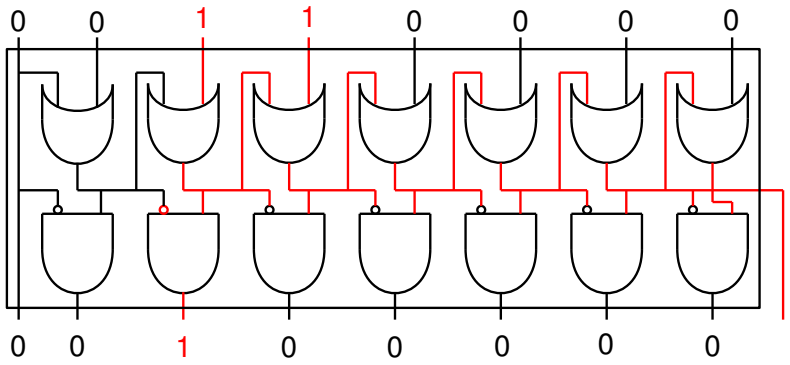

Figure 6.2: Chains with “or” and “and” to isolate the most significant bit with value ’1’.

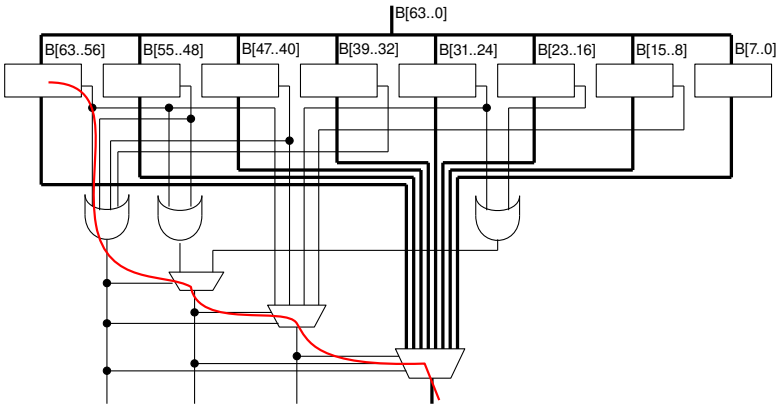

Figure 6.3: Circuit that is used to detect the most significant “or” and “and” chain that has a high bit.

6.1.2 Barrel shifter

The barrel shifter used to left shift the divisor is implemented in two stages. The first stages of the shifting is performed using three levels of multiplexers as illustrated in figure 6.4. These multiplexers are implements shifting of the divisor in multiples of 8. The control lines for these multiplexers are the three one bit control lines from figure 6.3. The last shifting of the divisor is implemented using a pipelined 64 times 8 bit multiplier. The eight control bits to this multiplier is the 8 “hot one” coded bits form the previous circuit applied in reversed range. This means that the most significant bit is connected to the least significant input on the multiplier. As an example, if the most significant bit is already one, then no shifting is required, which is equal to multiply by one which is achieved by reversing the range.

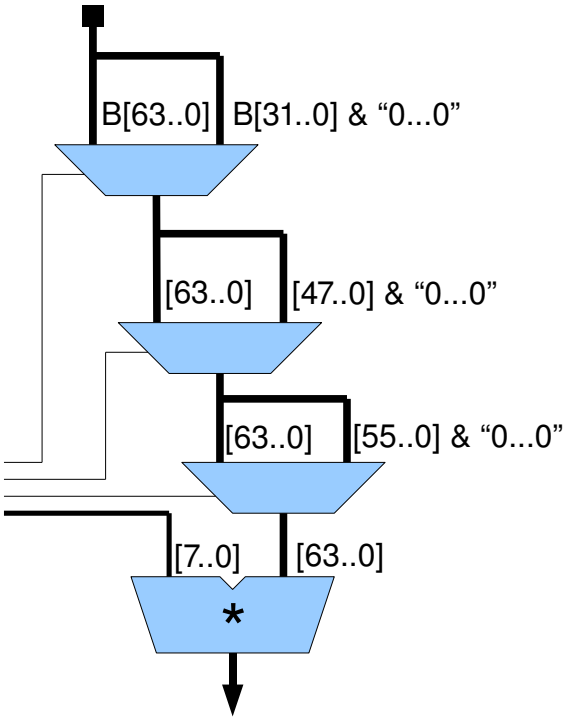

Figure 6.4: Illustration of barrel shifter. The shifter is implemented using three stage multiplexing and one multiplier to perform the least significant shifting.

6.2 Initial value

As mentioned it was in this implementation decided to have only 4 bits of initial approximation. This was based on the assumption that the increased size of the Newton-Raphson stages would be smaller than the size of the initial approximation circuitry and tables. In section 2.5 it was shown that the number of bits needed to store in a lookup table may be reduced by implementing methods with bi- and multipartite lookup tables. For this implementation it is believed that extra guard bits and addition logic needed to implementing partite lookup table methods will generate a table that is bigger than a simple 4 bit in 4 bit out lookup table. For this reason it was decided to implement the initial value as a simple lookup table with 4 address bits and 4 output bits.

6.3 Newton-Raphson stages

The main component in the division algorithm implemented is the Newton-Raphson stages. These stages contribute to most of the area and latency in the data pipe. Each stage is implemented as illustrated in figure 5.1, with the exception for the subtraction between the multipliers. This subtraction is implemented by inverting the output of the first multiplier and then adding one, which is an equal operation. This is illustrated in figure 6.5.

$$\begin{array}{rr} 2.00 & 10.00000 \\ -0.75 & -0.11000 \\ \hline = 1.25 & 1.01000 \end{array}$$

$$\begin{array}{rr} \overline{0.11000} \rightarrow & 1.00111 \\ & +0.00001 \\ \hline = 1.25 & 1.01000 \end{array}$$

Figure 6.5: Illustration of two's complement by using inversion.

In the circuit there are three parts that all influences the performance of the circuit. These are the two multiplications and the subtraction. The two multipliers are possible to implement using lookup tables (LUTS) in the FPGA or on the embedded multipliers. By implementing the multipliers using LUTS it is possible to implement specialized multiplication circuits such as carry save multiplier. This has been used in an ASIC implementation in [25]. In such a solution it will also be possible to implement the subtraction logic between the multipliers as a part of the multiplier logic and thereby remove some of the carry propagate delay introduced by the subtraction. An implementation of this type in FPGA will

requires high number of LUTS. A presumed better implementation which is used in this implementation is the embedded multipliers. These multipliers are optimized for pipelining and concatenation of arbitrary bit widths which makes them easy to implement in the system.

To enable the possibility of easy changing the pipe depth in the system all multipliers were coded as shown in listing 6.1. By changing the number n in the code the pipe depth of the circuit is changed. These pipeline registers is then pushed in to the multiplier by the FPGA synthesize tools. During development it was found that to achieve pipelining of the multiplier all registers in the pipe had to have synchronous reset rather than asynchronous. This is due to the fact that the embedded multipliers only supports synchronous reset[1].

Listing 6.1: VHDL implementation of pipelined multiplier with syncron reset

```
1  type pipe is array 0 to n-1 of std_logic_vector(63 downto 0);
2  signal mult_pipe : pipe;
3
4  process(clock)
5  begin
6    if clock'event and clock = '1' then
7      if reset = '1' then
8        mult_pipe <= (others => (others => '0'));
9      elsif enable = '1' then
10
11       mult_pipe(0) <= vec1 * vec2;
12
13       for i in mult_pipe'high downto 1 loop
14         mult_pipe(i) <= mult_pipe(i-1);
15       end loop;
16
17     end if;
18   end if;
19 end process;
```

6.4 Quotient and de-normalization circuit

The calculation of the quotient and the de-normalization is performed in two separate stages as illustrated in figure 6.6. The first stage multiplies the reciprocal with the dividend. While the next stage de-normalizes the result by right shifting it the same amount as the divisor was left shifted during the normalization. The barrel-shifter implemented to shift the result is implemented equally as the barrel shifter that normalizes the divisor.

[1]The DSP48^{TM}blocks in the Virtex family of FPGAs must have synchronous reset, while other devices may handle both synchronous and asynchronous reset.

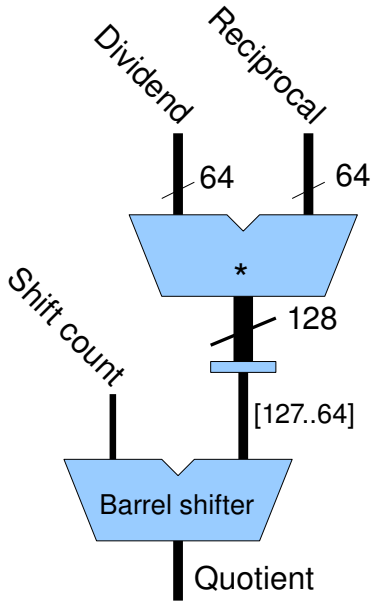

Figure 6.6: Illustration of quotient calculation and de-normalization circuit

6.5 Remainder calculation

After the quotient is calculated and de-normalized, it is used to calculate the remainder (REM). The remainder is calculated according to the following equation REM $= A - \lfloor Q \rfloor \cdot B$. The multiplier between the quotient and the reciprocal is coded in the same way as in the Newton-Raphson module to enable pipelining.

6.6 Adjustment circuit

During simulation of the model it was found that an adjustment stage was needed at the end of the circuit to achieve correct result. This adjustment stage performs a comparison between the calculated remainder and the divisor. If the divisor is found to be greater than the remainder, the divisor is subtracted from the remainder and one is added to the quotient. This is illustrated in algorithm 6.1.

The comparison between the remainder and the divisor is performed using a subtraction between the two. To prevent this subtraction from being performed twice the code was implemented such that the subtracted value were reused inside the if statement.

Algorithm 6.1 test for correctnes

1: **if** REM $\geq$ B **then**
2: $Q \leftarrow Q + 1$
3: REM $\leftarrow$ REM $- B$
4: **end if**
5: **return** Q, REM

In addition it was decided to handle the case of division by zero in this stage. In this case the output will be set to zero on both the quotient and the remainder.

6.7 Simulation results

The implemented circuit was tested using ModelSim. This test was performed with a modified version of the test bench used to test the simulation model. The modifications were done to handle that the implemented model is pipelined. The simulation completed correctly and no wrong results were found in the tested vectors.

6.8 Synthesize results

In this assignment the main focus has been to evaluate how different design choices affect the size, speed and throughput of the system. For this reason it was decided to synthesize several configurations of the circuit on two different FPGAs from Xilinx. These synthesizes has been performed using ISE Foundation design tool from Xilinx.

For comparison a code for a 16 bits division circuit obtained from KDA has been synthesized to FPGA using the same settings as done for the other code. This code implements a division circuit that uses a finite state machine (FSM) to control the data flow through one multiplier. In this circuit it is used a 8 bits initial approximation towards the correct reciprocal, and a total of five clock periods to complete on division. Since this circuit only uses one multiplier, only on data set can be calculated at one time. This then gives a throughput of one each fifth clock cycle.

Current systems at KDA use both Spartan3 and Virtex4 FPGAs from Xilinx running at 55MHz. This was used as a target for the developed circuit. By adjusting the pipeline depth of the circuit so that it would run at at least 55MHz it was possible to find the smallest possible size of the circuit. In addition to see the maximum performance of the circuit the pipeline depth was increased until that the maximum frequency was achieved. This was preformed for both 16 and 64 bits implementations on both Spartan3 and Virtex4

FPGAs. The results from the synthesize to Spartan3 is given in table 6.1, and the results from the synthesize to Virtex4 is given in table 6.2.

	Developed circuit				Ref. model
Bit widths	64		16		16
Optimization effort	Speed	Area	Speed	Area	-
Pipeline stages	29	14	16	6	-
Frequency MHz	87.292	60.644	206.471	67.179	49.460
Number of Slices	6894	5639	512	147	416
Number of Slice Flip Flops	8198	4265	848	191	82
Number of 4 input LUTs	9799	8624	476	233	772
used as logic	9092	8460	341	217	-
used as Shift registers	717	164	135	16	-
Number of IOs	259	259	67	67	68
Number of bonded IOBs	259	259	67	67	68
Number of MULT18X18s	26	26	9	9	1
Number of GCLKs	1	1	1	1	1

Table 6.1: Division circuit synthesize result on xc3s1500-5-fg676 (Spartan3) FPGA from Xilinx

It is observed that the size of the reference circuit is larger than the size of the slowest implementation of the developed circuit on all variables except on the number of multiplier blocks. It is believed that this is because of the multiplexers used to control the data flow through the multiplier will use much logic.

During development of the circuit it was found that an implementation using four Newton-Raphson stages and an initial approximation with 4 bits would give a total smaller circuit. To verify this, a circuit was synthesized with only two Newton-Raphson stages. In this circuit no table method has been implemented so this will come in addition to the synthesized results. This results is shown in table 6.3.

	Developed circuit				Ref. model
Bit widths	64		16		16
Optimization effort	Speed	Area	Speed	Area	-
Pipeline stages	54	10	19	6	-
Frequency MHz	219.730	69.238	324.465	92.523	96.279
Number of Slices	4009	1314	473	183	416
Number of Slice Flip Flops	6635	1747	763	168	78
Number of 4 input LUTs	3664	1592	466	308	755
used as logic	2496	1472	344	292	-
used as Shift registers	1168	120	122	16	-
Number of IOs	259	259	67	67	68
Number of bonded IOBs	259	259	67	67	68
Number of GCLKs	1	1	1	1	1
Number of DSP48s	65	65	8	6	1

Table 6.2: Division circuit synthesize result on 4vsx25ff668-12 (Virtex4) FPGA from Xilinx

Bit widths	64	
Optimization effort	Speed	Area
Pipeline stages	48	8
Frequency MHz	219.730	71.951
Number of Slices	3475	1216
Number of Slice Flip Flops	5630	1576
Number of 4 input LUTs	3342	1334
used as logic	2328	1328
used as Shift registers	1014	6
Number of IOs	259	259
Number of bonded IOBs	259	259
Number of GCLKs	1	1
Number of DSP48s	54	57

Table 6.3: Division circuit synthesize result with two Newton-Raphson stages on 4vsx25ff668-12 (Virtex4) FPGA from Xilinx

CHAPTER

7

DISCUSSION

7.1 Implemented solution

In the theory part of this report several methods for performing integer division were examined. It was found that the multiplicative classes of division algorithms would give the best implementation. This is due to their quadratic rate of convergence. The two multiplicative division algorithms evaluated were Newton-Raphsons method and Goldschmidts method. The complexity of these two methods is basically the same as both methods need two multiplications and one subtraction. The difference between the two is that the multiplications in Goldschmidts method are independent and may be executed in parallel, while they in the Newton-Raphsons method are dependent and must therefore be executed in sequence. This will decrease the latency of the Goldschmidts method compared to Newton-Rapsons method. The problem with Goldschmidts method was that it accumulated truncation error in each iteration. To compensate for this each iteration must be performed with full bit width in addition to some guard bits that is needed to control the accumulated error. In Newton-Rapsons method truncation error in each stage is corrected in the next stage. This results that, in an implementation where all iteration stages is implemented in sequence, the bit width of each stage will only be that of the calculated precision in that stage. The result of this is that a high throughput pipelined division circuit based on Newton-Raphsons method will be smaller than an implementation based on the Goldschmidts method.

One of the design choices made in this implementation were to use a simple low accuracy approximation towards the correct reciprocal. This choice was based in the assumption

that the reduced size of the first iteration stages in an implementation based on Newton-Raphsons method would be smaller than the size of the table methods needed to achieve a higher precision on the initial approximation. In the implemented circuit it was used a 4 bit initial approximation. This gave a table size of $4 \cdot 2^4 = 64$ bit. For comparison Das Sarma and Matula needed in [4] 960 bits to achieve 8 bit precision by using bipartite lookup tables, and Dinechin and Tisserand needed in [5] 14592 bits to achieve 16 precision by using multipartite lookup tables. This shows that the assumptions about the use of redused precision on the initial approximation and the addition of extra Newton-Raphson stages was correct. As the bit widths increase it will be even more reasonable to use a less precise initial approximation since the the total size of the circuit can be regarded as a geometric series that will add up to twice the size of the last stage, independent of the total number of stages in the circuit. The added stages will also add some additional pipeline stages. This will be acceptable since they are quite few due to the small size of the stages.

One of the issues in the assignment was to evaluate the possibilities to implement a module generator that can generate circuits with different parameters regarding size, speed and throughput. It was found, during the theory study and development of the circuit, that to develop such a generator most of the circuit code would need to be preprogrammed in to the generator for every configuration. Based on this it is assumed that it will be better to create library of modules that may easily be put together manually.

7.2 Possible improvements

As seen in figure 6.1 both of the input operands and the shift count needs to be pipelined through the circuit. These pipeline-registers takes up most of the slice flip flops in the circuit. By implementing this memory in the embedded block RAM on the FPGAs it would free up much logic to be used in other circuits. Since these RAM blocks are addressed through an address bus, such an implementation would need some address counter to keep track on the input and output point in the address space for the data stored.

An observation that was done when synthesizing for the Virtex4 FGPA with DSP48$^{\text{TM}}$ multiplier blocks is that they has an integrated accumulate function, adder/subtracter, to speed up filter implementation. It might be possible to modify the Newton-Raphson stage code such that the subtraction between the multipliers will be synthesized to this subtracter. This will then reduce the size of the logic since the signal do not need to be routed out to slices and then back in to the DSP48 blocks.

The implemented circuit has been developed to have a throughput of one data set each clock cycle. This makes the circuit use a lot of the embedded multiplier blocks in the FPGA, which may be a scarce resource for many implementations. To reduce the number of multipliers it is possible to looping the data flow several times through each multiplier.

This will then reduce the throughput of the circuit, but it will reduce the number of multipliers. This will come at the cost of added size of the other logic since the data flow must be switched through the circuit.

Another improvement that might reduce the latency through the circuit is to start earlier to calculate the quotient. The most significant bits of the reciprocal will be available already after the initial value stage. These bits may be used in parallel to the Newton-Raphson iteration to calculate the most significant bits of the quotient. At the end of the iterations it is then possible to use only a few pipeline stages to calculate the least significant bits of the quotient.

7.3 Future work

Before implementing the circuit, the needed bit widths were calculated. Based on these calculations the numbers of bits were implemented. Later during simulation and assumptions made about the calculations it was found that the calculated bit widths were over estimated and that the worst case error would never accrue. Based on this it was decided to reduce the bit widths. This assumption should be verified better by use of more thorough mathematical analysis and simulations.

In this work the primary focus has been on evaluating how different design choices will affect the division circuit. To test this, the implemented VHDL code was coded so that it would be easy to change the number of pipeline stages through the entire circuit. Due to this design of the circuit it is difficult to achieve an implementation that is optimal for a specific implementation. For the future the circuit should be implemented in a way that is optimized for a specific implementation.

CHAPTER

8

CONCLUSION

In this paper several division methods and algorithms has been evaluated. Of those methods that were evaluated, the multiplicative methods Newton-Raphson method and Goldschmidt method were found to be best suited since they has quadratic rate of convergence. Goldschmidt method was shown to have an advantage over the Newton-Raphson method in that the two multiplications involved in each iteration could be executed in parallel and therefore reduce the latency. The problem with Goldschmidts method is that it accumulates truncation error for each iteration it performs. This will not happen with Newton-Raphson method since it is strictly convergent meaning that truncation error from last stage is corrected in next stage. This made it possible to decrease the size of the iteration stages used in the Newton-Raphson method.

The theory study showed that multiplicative methods need an initial approximation of the correct reciprocal. Several methods were investigated, and it was found that to achieve a precise initial approximation with the smallest possible size on the tables a multipartite table method should be implemented. A C++ program for generation these multipartite tables were implemented, but tests of this program showed that it generated on ULP error on some of the output values. It was later found that it would be better to use a less precise initial approximation, and therefore not use multipartite table methods. Due to this the work on the table generator was abandoned.

As mentioned it was decided not to implement multipartite table methods in the division circuit. In stead it was decided to use simple 4 bit in 4 bit out lookup table. This was due to the relative small size of the extra stages needed in an implementation with Newton-

Raphson method. Synthesis result on the complete circuit confirmed that the extra size added by the extra stages is compensated for by the simplicity of the initial value circuit.

Several configurations of the circuit were synthesized with optimization effort on both speed and size. The result showed that it is possible to achieve high speed at the cost of large size and pipeline depth, or it is possible to achieve good speed with small size.

Summarized it is found that Newton-Raphsons method is well suited for pipelined division implementation. It is also found that that it is optimal in a high throughput pipelined division circuit to use a rough initial approximation to the reciprocal, and implement additional iteration stages.

REFERENCES

[1] Simen Gimle Hansen. Multifunksjonell beregningsenhet for digital signalprosessering, April 2005. Universitetet i Oslo.

[2] Martin Rognerud. Konstruksjon av digital heltallsaritmetikk, kompakte initialverditabeller for multiplikative divisjons algoritmer. Master's thesis, NTNU, Juni 2007.

[3] Karl Marius Stafto. Konstruksjon av digital heltallsaritmetikk, multiplikativ divisjon. Master's thesis, NTNU, Juni 2008.

[4] D. Das Sarma and D.W. Matula. Faithful bipartite ROM reciprocal tables. *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*, pages 17–28, Jul 1995.

[5] F. de Dinechin and A. Tisserand. Multipartite table methods. *Computers, IEEE Transactions on*, 54(3):319–330, March 2005.

[6] S.F. Obermann and M.J. Flynn. Division algorithms and implementations. *Computers, IEEE Transactions on*, 46(8):833–854, Aug 1997.

[7] M.J. Flynn. On division by functional iteration. *Computers, IEEE Transactions on*, C-19(8):702–706, Aug. 1970.

[8] R. Goldberg, G. Even, and P.M. Seidel. An FPGA implementation of pipelined multiplicative division with ieee rounding. *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 185–196, April 2007.

[9] R. Stefanelli. A suggestion for a high-speed parallel binary divider. *Computers, IEEE Transactions on*, C-21(1):42–55, Jan. 1972.

[10] K. Tatas, D.J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis. A novel division algorithm for parallel and sequential processing. *Electronics, Circuits and Systems, 2002. 9th International Conference on*, 2:553–556 vol.2, 2002.

[11] M. Langhammer. Improved subtractive division algorithm. *ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International*, pages 343–347, Sep 1998.

[12] J. Ebergen, I. Sutherland, and A. Chakraborty. New division algorithms by digit recurrence. *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, 2:1849–1855 Vol.2, Nov. 2004.

[13] H.R. Srinivas and K.K. Parhi. A fast radix 4 division algorithm. *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, 4:311–314 vol.4, May-2 Jun 1994.

[14] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli. Digit-recurrence dividers with reduced logical depth. *Computers, IEEE Transactions on*, 54(7):837–851, July 2005.

[15] A.A. Ibrahem, H. Elsimary, and A.E. Salama. FPGA implementation of modified radix 2 SRT division algorithm. *Micro-NanoMechatronics and Human Science, 2005 IEEE International Symposium on*, 3:1419–1422 Vol. 3, Dec. 2003.

[16] D.L. Harris, S.F. Oberman, and M.A. Horowitz. SRT division architectures and implementations. *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pages 18–25, Jul 1997.

[17] Xiaojun Wang and B.E. Nelson. Tradeoffs of designing floating-point division and square root on virtex FPGAs. *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 195–203, April 2003.

[18] Ross L. Finney, Maurice D. Weir, and Frank R. Giordano. *Thomas' Calculus*. Addison Wesley, updated tenth edition, 2003.

[19] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 1964.

[20] G. Even, P.-M. Seidel, and W.E. Ferguson. A parametric error analysis of goldschmidt's division algorithm. *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pages 165–171, June 2003.

[21] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 128–135, 2001.

[22] P. Kornerup and D.W. Matul. Single precision reciprocals by multipartite table lookup. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 240–248, June 2005.

[23] James E. Stine and Michael J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, Desember 1998.

[24] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL design*. McGraw - Hill, second edition, 2005.

[25] Anders Torp, Rasmus Winther-Almsturp, and Michael Boesen. Design of reciprocal unit based on the newton-raphson approximation. Technical report, Department of Infomatics and Mathematical Modelling, Technical University of Denmark.

APPENDIX

A

SIMULATION TEST BENCH

To test that the calculated bit widths in the Newton-Raphson stages were correct, a simulation model was implemented. This simulation model was tested with the following test bench. The test bench generates sets of vectors that are presumed to be worst case input parameters, and applies them to the model. The resulting output from the model is then compared with the correct answer to verify the implementation.

Listing A.1: Simulation test bench

```
1  library ieee, work;
2
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5  use work.txt_util.all;
6
7  entity NewtonSim_tb is end NewtonSim_tb;
8
9  architecture behavior of NewtonSim_tb is
10
11   component sim_newton is
12     port( dividend :  in std_logic_vector(63 downto 0);
13           divisor :   in std_logic_vector(63 downto 0);
14           clock :     in std_logic;
15           quotient :  out std_logic_vector(63 downto 0);
16           rest :      out std_logic_vector(63 downto 0));
17   end component;
18
19   constant period : time := 10 ns;
20
21   signal tb_dividend :  std_logic_vector(63 downto 0);
22   signal tb_divisor :   std_logic_vector(63 downto 0);
23   signal tb_clock :     std_logic := '0';
24   signal tb_quotient:   std_logic_vector(63 downto 0);
25   signal tb_rest :      std_logic_vector(63 downto 0);
26
27 begin
28
29   --Instantiation of simulation model
30   st:sim_newton port map(
31       tb_dividend,
32       tb_divisor,
```

```
33          tb_clock,
34          tb_quotient,
35          tb_rest);
36
37      --Clocking process
38      clocking:process
39      begin
40        wait for period/2;
41        tb_clock <= not( tb_clock );
42      end process;
43
44      --Generate test vectors
45      process
46        variable divd, divs : std_logic_vector(tb_dividend'range);
47        variable pattern : std_logic_vector(tb_dividend'range);
48      begin
49
50        --Generate manual test vector1
51        divd := (others => '0');
52        divd(divd'high) := '1';
53
54        divs := (others => '0');
55        divs(divs'high) := '1';
56        divs(divs'high-3) := '1';
57
58        tb_dividend <= divd;
59        tb_divisor  <= divs;
60
61        wait until tb_clock'event and tb_clock = '1';
62
63        --Generate manual test vector2
64        divs(divs'high-4) := '1';
65
66        tb_dividend <= divd;
67        tb_divisor  <= divs;
68
69        wait until tb_clock'event and tb_clock = '1';
70
71        --Generate patterns of walking one's on dividend and divisor
72        for i in 0 to 63 loop
73          divd := (others => '0');
74          divd(i) := '1';
75          for j in 0 to 63 loop
76            divs := (others => '0');
77            divs(j) := '1';
78
79            tb_dividend <= divd;
80            tb_divisor  <= divs;
81
82            wait until tb_clock'event and tb_clock = '1';
83
84          end loop;
85        end loop;
86
87        --Generate patterns of walking one's on the dividend with high bit '1'
88        --Generate increasingly larger patterns of one's to apply to the divisor
89        for i in 0 to 63 loop
90          divd := (others=>'0');
91          divd(divd'high) := '1';
92          divd(i) := '1';
93
94          for p_len in 1 to 62 loop--for pattern lengths from 1 to 62
95
96            --generate pattern of one's
97            pattern := (others => '0');
98            for j in 0 to p_len - 1 loop pattern(j) := '1'; end loop;
99
100           --for all shifted positions of the pattern
101           for j in 0 to 64 - p_len loop
102             divs := (divs'high => '1', others => '0');
103             divs := divs or pattern;
104             pattern := pattern(pattern'high - 1 downto 0) & '0';
105
106             tb_dividend <= divd;
107             tb_divisor  <= divs;
108
109             wait until tb_clock'event and tb_clock = '1';
110
111           end loop;
112         end loop;
113       end loop;
114
115       --Generate patterns of walking one's on the dividend with high bit '1'
116       --Generate increasingly larger patterns of zero's to apply to the divisor
```

```
117       for i in 0 to 63 loop
118         divd := (others=>'0');
119         divd(divd'high) := '1';
120         divd(i) := '1';
121         for p_len in 1 to 62 loop
122
123           --generate pattern of zero's
124           pattern := (others => '1');
125           for j in 0 to p_len - 1 loop pattern(j) := '0'; end loop;
126
127           for j in 0 to 64 - p_len loop
128             divs := (others => '1');
129             divs := divs and pattern;
130             pattern := pattern(pattern'high - 1 downto 0) & '1';
131
132             tb_dividend <= divd;
133             tb_divisor  <= divs;
134
135             wait until tb_clock'event and tb_clock = '1';
136
137           end loop;
138         end loop;
139       end loop;
140       report "Simulation finished";
141       wait;
142     end process;
143
144     --Verify the answer
145     verify:process
146       variable q, r : std_logic_vector(63 downto 0);
147       variable s, d : string(1 to 64);--strings for text printing
148     begin
149
150       wait until tb_clock'event and tb_clock = '1';
151
152       q := std_logic_vector(unsigned(tb_dividend) / unsigned(tb_divisor));--Calculate the exact quotient
153       r := std_logic_vector(unsigned(tb_dividend) rem unsigned(tb_divisor));--Calculate the exact reminder
154
155         wait until tb_clock = '0';
156
157       if tb_quotient /= q or tb_rest /= r then --Compere result
158         s := str(tb_divisor);
159         d := str(tb_dividend);
160         report "Error: " & s & " : " & d;
161
162         s := str(tb_quotient);
163         d := str(q);
164         report "Sim is: " & s & ", ans: " & d;
165
166       end if;
167
168     end process;
169
170   end behavior;
```