Efficient arithmetic functional modules for FPGA-based VLSI designs

Xia, Fei

2005

Xia, F. (2005). Efficient arithmetic functional modules for FPGA-based VLSI designs.
Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/3786>

<https://doi.org/10.32657/10356/3786>

Nanyang Technological University

Downloaded on 26 Apr 2023 08:21:35 SGT

a365164 /

**Efficient Arithmetic Functional Modules
For
FPGA-based VLSI Designs**

Xia Fei

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2005

TK
7874.75
X6
2005

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

20-06-2005

Date

Xia Fei

Xia Fei

Acknowledgements

The work presented in this thesis has been carried out under the supervision of Associate Professor Jong Ching Chuen, Division of Circuits and Systems, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. I wish to express my sincere appreciation to him for the strict encouragement, guidance and critical comments, which have been enriching throughout my entire research and study.

I would like to thank Mr. Chua Eng Kee for his generous time and patience. I also like to thank Miss Xu Fei, Mr. Li Juhui and Mr. Wang Chao. Their helpful opinions and friendly assistance are very much appreciated.

Last but not least, I would like to thank the staff of the Integrated System Research Lab. for providing me with an excellent working environment and technical support.

Summary

The demand of DSP applications and their VLSI implementations is ever increasing. Arithmetic computations in these systems are often required and become an important aspect of the designs. In this work, the efficiency of arithmetic modules in speed and hardware cost is studied and improved to optimize their performance. Particularly, the algorithms for computing the trigonometric functions such as sine, cosine, hyperbolic Sine/Cosine, reciprocal and logarithm are studied. Their hardware implementations are carried out.

Based on the study and investigation, a hybrid CORDIC (COordinate Rotational DIgital Computer) is developed and presented in this thesis. The hybrid CORDIC combines a LUT (look-up table) with an iterative process of the conventional CORDIC algorithm. The simulation results show that the hybrid CORDIC can achieve good speed and area efficiency when the LUT and the iterative steps are properly balanced. The hybrid CORDIC is extended to handle the hyperbolic sine and cosine functions and also adopted for the development of the reciprocal function.

The study on the logarithm function leads to the development of an accurate base-2 binary-to-binary logarithm function. The logarithm developed achieves a better accuracy with minimum extra hardware cost.

Table of Content

Acknowledgements i

Summary.....ii

List of Tablesvi

List of Figures.....vii

Chapter 1 1

Introduction..... 1

 1.1 Motivation 1

 1.2 Objectives5

 1.3 Major Contributions of the Thesis..... 7

 1.4 Organization of the Thesis..... 9

Chapter 2 11

Background of CORDIC Algorithm 11

 2.1 What is CORDIC? 11

 2.2 Existing CORDIC Architectures 18

Chapter 3 22

Hybrid CORDIC for Trigonometric Functions 22

 3.1 Conventional Sin/Cos Function Generation 22

 3.2 Hybrid Technique 26

 3.2.1 Theoretical Orientation 26

 3.2.2 Methodology 27

 3.2.3 Scale Factor 30

 3.2.4 RTL Model of Hybrid CORDIC 31

- 3.2.5 Simulation Results 35
- 3.2.6 Selection of Resolution for Look-up Table..... 38
- 3.3 Extensions to the Hyperbolic Coordinate..... 40
 - 3.3.1 Simulation Results 42
- 3.4 Improved Hybrid CORDIC 44
 - 3.4.1 Methodology 44
 - 3.4.2 Performance Evaluation 46
- Chapter 4 51**
- Hybrid CORDIC for Reciprocation..... 51**
 - 4.1 Reciprocation in Conventional CORDIC 51
 - 4.2 Reciprocation Based on Hybrid Technique..... 55
 - 4.2.1 Selection of the Entry and Bit-width of Look-up Table 56
 - 4.2.2 RTL Model of Reciprocation 58
 - 4.2.3 Simulation Results 59
- Chapter 5 62**
- Improved Mitchell’s Logarithm Approximation..... 62**
 - 5.1 Introduction of Mitchell’s Method 64
 - 5.2 The Proposed Improved Method 69
 - 5.2.1 Accuracy Comparison..... 76
 - 5.2.2 Hardware Implementation..... 79
- Chapter 6 81**
- Conclusion and Recommendations 81**
 - 6.1 Conclusion 81
 - 6.2 Recommendations for Further Research 83
- Bibliography 85**
- Appendices..... 93**
 - Appendix A..... 93
 - VHDL Model of Hybrid CORDIC for Trigonometric Functions 93

Appendix B.....99

VHDL Model of Hybrid CORDIC for Hyperbolic Sin/Cos Functions.....99

Appendix C.....104

VHDL Model of Hybrid CORDIC for Reciprocal Function.....104

Appendix D.....128

VHDL Model of Proposed Binary to Binary-logarithm Approximation Method..128

List of Tables

Table 2.1: Functions calculated by the CORDIC algorithm..... 17

Table 3.1: Elementary rotation angle θ_i 24

Table 3.2: Comparison of errors 36

Table 3.3: Synthesized results of CORDIC 38

Table 3.4: The entry number of LUT and the required rotations..... 39

Table 3.5: Comparison of results error for hyperbolic CORDIC 42

Table 3.6: Synthesized results of CORDIC for hyperbolic coordinate 43

Table 3.7: Area comparison of both hybrid CORDIC in terms of N..... 49

Table 3.8: Comparison of the delay of both hybrid CORDIC 50

Table 3.9: Comparison of the area of both hybrid CORDIC 50

Table 4.1: The entry and bit-width of look-up table 57

Table 4.2: Synthesized results of reciprocal model with different multipliers 59

Table 4.3: Errors compared to c function 60

Table 4.4: Synthesized results of CORDIC in reciprocal function..... 61

Table 5.1: The correction function, corresponding errors and the constant to be added
..... 76

Table 5.2: Comparison of percentage error 78

Table 5.3: The synthesized results of the 3 methods 80

List of Figures

Figure 2.1: Rotation trajectory for the circular CORDIC system ($m = 1$).....	15
Figure 2.2: Rotation trajectory for the linear CORDIC system ($m = 0$).....	15
Figure 2.3: Rotation trajectory for the hyperbolic CORDIC system ($m = -1$).....	16
Figure 3.1: Vector rotation by 30° using a conventional CORDIC algorithm.....	25
Figure 3.2: Rotation trajectory in a conventional CORDIC	28
Figure 3.3: Rotation trajectory in the hybrid CORDIC	29
Figure 3.4: The VHDL entity interface of hybrid CORDIC.....	31
Figure 3.5: Fixed-point format for input.....	32
Figure 3.6: Fixed-point format for output.....	32
Figure 3.7: The architecture diagram of the hybrid CORDIC	35
Figure 3.8: The architecture diagram of the conventional CORDIC.....	35
Figure 3.9: The sine function values of CORDIC	37
Figure 4.1: Rotation trajectory in vectoring mode of CORDIC	54
Figure 4.2: Fixed-point format for the reciprocal model	58
Figure 4.3: the reciprocal function values of CORDIC	61
Figure 5.1: Comparison of $\log_2 N$ and Mitchell's approximation.....	66
Figure 5.2: The percentage error of Mitchell's approximation of $\log_2 N$	66
Figure 5.3: Comparison of SanGregory's, the proposed improved and the Mitchell's approximation.....	77
Figure 5.4: The percentage error of the proposed improved method	77
Figure 5.5: Error distribution of SanGregory's, the proposed improved and the Mitchell's approximation	78
Figure 5.6: Fixed-point format for the output result	79

Chapter 1

Introduction

1.1 Motivation

Arithmetic units are universal and often performance-critical building blocks in VLSI (Very Large Scale Integration) systems nowadays. Many applications such as DSP, graphic, audio and video processors and communication products require arithmetic computations like sine, cosine, division, logarithm, etc.

A given computing application may be implemented using different technologies, with a large range of tradeoffs among performance, unit and non-recurring costs (including development effort). The choices may include the followings:

- A software implementation, targeting off-the-shelf microprocessors, is easy to develop and reproduce, but will not always provide the best performance and meet the requirements.
- For cost or performance reasons, some applications can be implemented as application specific integrated circuits (or ASIC). An ASIC provides the best possible performance and may have a very low unit cost, at the expense of a very high development cost.

Chapter 1**Introduction**

- An intermediate approach is the use of reconfigurable circuits, or field-programmable gate arrays (FPGA).

In each case, the computation is broken down into elementary operations, executed by elementary hardware elements, or *arithmetic modules*. In the software approach, the modules used are those provided by the microprocessor instruction sets or high level programming functions. In ASIC or FPGA approaches, these modules have to be built by designers, or taken from module libraries.

A hardware implementation may lead to better performances than a software implementation for two main reasons: parallelism and specialization. The second factor, from the arithmetic point of view, means that specific data types and specific modules may be used which would require costly emulation on a processor.

A circuit may be optimized for speed or area (circuit cost). Computation speeds have increased dramatically during the past three decades resulting from the development of various technologies. The execution speed of an arithmetic operation is a function of two factors. One is the circuit technology and the other is the algorithm used. It can be rather confusing to discuss both factors simultaneously; for instance, a ripple-carry adder implemented in GaAs technology may be faster than a CMOS carry-look-ahead adder for input data with short word-length.

On the other hand, FPGA devices with millions of gates, huge on-chip memory and even hardwired multipliers can now provide alternatives to implement very complex systems. This makes the design approaches based on high-level synthesis even more

Chapter 1

Introduction

attractive. However, complex arithmetic modules for functions such as the trigonometric functions, the reciprocal function and the logarithm function are not being used as frequently as they should be in DSP and VLSI systems by designers nor in high-level synthesis systems.

All the above considerations motivate the study of arithmetic modules for VLSI design and their implementations in FPGA devices. The aim of this project is to study some arithmetic functional modules that are frequently required in VLSI systems and to improve the efficiency of such arithmetic modules at the algorithmic level.

Many algorithms are available for the implementation of elementary modules. Today research is active on algorithms for the following operations:

- Basic operations (addition, subtraction, multiplication), and their variations (multiplication and accumulation, multiplication or division by constants, etc.);
- Algebraic functions (division, inverse, and square root, and in general powering to an integer, and polynomials);
- Elementary functions (sine, cosine, exponential, etc.);
- Combinations of the above operations.

This work focuses on the CORDIC (Coordinate Rotational Digital Computer) algorithm [1] and the Mitchell's binary logarithm algorithm [2]. The CORDIC algorithm only uses shifts and additions to perform trigonometric and other transcendental functions such as square root. Despite its simplicity, it has some innate

Chapter 1

Introduction

restrictions that impede its performance: the convergence rate of the rotation angle is still not fast enough. And if some unnecessary rotations are skipped, the scale factor will be a variable, which introduces considerable overhead in computing variable scale factors. While in the case of the Mitchell's algorithm, it is an approximation method for binary-to-binary logarithm. This method has only modest accuracy and limited application for general-purpose computation. In this project, both the algorithms are improved to overcome their shortcomings.

1.2 Objectives

The main aim of this work is to investigate the CORDIC and Mitchell's algorithms, which are commonly used in arithmetic computation because of their simple hardware implementation, and to develop more efficient, in terms of speed and hardware cost, arithmetic modules based on these algorithms.

Motivated by the performance constraints existing in the CORDIC and the Mitchell's algorithms and increasing need for high performance arithmetic function modules, the objectives of this project are therefore as listed below.

1. To study the convergence of the CORDIC algorithm and to search for solutions to disrupt its iterative character so that the computation need not be executed step by step and therefore the speed of the algorithm is increased.
2. To apply the improved CORDIC algorithm to compute the sine and cosine functions and to extend it to the liner and hyperbolic modes of this algorithm so as to realize the sinh, cosh and reciprocal functions, hence, to obtain various arithmetic modules with different tradeoff between area and speed.
3. To investigate the accuracy of Mitchell's algorithm for logarithm and improve the accuracy by making some corrections on the original algorithm without introducing high hardware cost.

Chapter 1

Introduction

4. To develop synthesizable Register-Transfer-Level (RTL) models of these algorithms for efficient implementations in FPGA devices.
5. To synthesize these RTL models into FPGA devices as well as to verify their functionalities.

1.3 Major Contributions of the Thesis

The main contributions of this work are described below.

1. A hybrid method is proposed to speed up the convergence of the CORDIC algorithm. This approach combines a look-up table with an algorithmic iterative procedure of the conventional CORDIC algorithm. With a small-resolution LUT replacing some CORDIC rotations, fast access to the arbitrary precision is obtainable from a systematic iteration algorithm. When the proportion between the resolution of the LUT and the CORDIC rotations is properly balanced, the hybrid CORDIC can achieve faster speed than the conventional CORDIC with reasonable hardware area. It can also offer many choices of different combinations of area and speed to suit different design requirements. The functional resources in terms of area and speed can be varied easily by changing the resolution of LUT and the number of rotation steps correspondingly.
2. This hybrid CORDIC is subsequently applied to the sine and cosine functions. By parallelizing the direction determination of the remaining rotation stages, the corresponding rotations can be executed concurrently and therefore improve the computation speed further. The hybrid scheme is also extended to compute the sinh, cosh and reciprocal functions, aiming to achieve faster speed.
3. By investigating the error distribution of the Mitchell's logarithm approximation, it is found that the most significant errors occur when the input

Chapter 1

Introduction

values are less than 2^8 . Dividing the input range of 0 to $2^8 - 1$ into 8 regions according to the Most Significant Bit (MSB), the correction function for each region is tried out, aiming at making the approximations closer to the exact logarithm. The elegant accuracy, improved by more than 20% in the percentage error range when compared with a published technique, is achieved with less hardware cost in its implementation.

1.4 Organization of the Thesis

The thesis consists of 6 chapters and is organized as follows:

Chapter 1 gives an introduction on the importance of the arithmetic function module designs. The motivations and objectives as well as the contributions of the work are presented.

Chapter 2 introduces the concept and the theory of the CORDIC algorithm in detail. The arithmetic functions that can be realized by CORDIC are illustrated. Some existing CORDIC implementation techniques and their advantages and drawbacks are discussed.

Chapter 3 focuses on the proposed hybrid CORDIC algorithm. The sine and cosine functions are implemented based on the hybrid CORDIC algorithm. A VHDL model is developed for the hybrid CORDIC. The hardware implementation is compared between the conventional CORDIC and the hybrid CORDIC algorithms. An improved hybrid CORDIC is also introduced to perform the sin/cos function, and the area and delay of these modules are presented for comparison.

Chapter 4 addresses the reciprocal function based on the hybrid CORDIC. The concept of the design method is the same as the sin/cos function but the operation is in a different mode of the CORDIC algorithm. The implementation and the simulation results are also presented.

Chapter 1

Introduction

Chapter 5 presents a new method for computing the approximation of logarithm (base 2) for binary integers based on the Mitchell's algorithm. The accuracy of proposed method is much improved compared to the Mitchell's method. And this approximation method is performed using only combinational logic, thus requires only simple hardware. The RTL model and its simulation results are presented. The results are compared with those obtained from two existing techniques.

Chapter 6 concludes the thesis and gives some recommendations for further work.

Chapter 2

Background of CORDIC Algorithm

2.1 What is CORDIC?

The COordinate Rotational Digital Computer (CORDIC) algorithm is an efficient algorithm for the computation of elementary functions, especially the trigonometric functions. The algorithm was formally introduced by Volder in [1] for solving real-time navigational problems and unified to compute elementary functions such as trigonometric functions, multiplication and division etc. by Walther in [3]. It is attractive for hardware implementation because it uses only elementary shift-and-add steps to perform vector rotation. During the early time, the CORDIC algorithm was widely used in HP calculators to compute sine and cosine values. Today with its performance and hardware advantage in VLSI systems, CORDIC has become an attractive alternative to traditional arithmetic units for special-purpose hardware design. Also, the CORDIC algorithm can be applied to many applications where rotation-based arithmetic functions are heavily utilized, such as digital signal processing [4-6] and the conventional CORDIC processors [7-10] for general purpose. The basic CORDIC algorithm is described as follows.

Chapter 2**Background of CORDIC Algorithm**

Given a vector $v(x, y)$, its magnitude L and angle α , the rotation angle θ , and the rotated output vector $v'(x', y')$, the relationship between v' and v is described by Equations 2.1 and 2.2.

$$x' = L \cos(\alpha + \theta) \quad (2.1)$$

$$= L \cos \alpha \cdot \cos \theta - L \sin \alpha \cdot \sin \theta$$

$$y' = L \sin(\alpha + \theta) \quad (2.2)$$

$$= L \sin \alpha \cdot \cos \theta + L \cos \alpha \cdot \sin \theta$$

Because of $x = L \cos \alpha$ and $y = L \sin \alpha$, Equations 2.3 and 2.4 can be obtained from Equations 2.1 and 2.2.

$$x' = x \cos \theta - y \sin \theta \quad (2.3)$$

$$= \cos \theta (x - y \tan \theta)$$

$$y' = y \cos \theta + x \sin \theta \quad (2.4)$$

$$= \cos \theta (y + x \tan \theta)$$

If the rotation angle θ is divided into a set of small angles for rotations in a set of steps, θ can be approximated by $\theta = \sum \delta_i \theta_i$, where $\delta_i = \{-1, 1\}$. δ_i is the sign of rotation (positive for counter-clockwise and negative for clockwise rotation).

Therefore, the new coordinates after each rotation (x_{i+1}', y_{i+1}') can be expressed as Equations 2.5 and 2.6.

$$x_{i+1}' = \cos \theta_i (x_i' - \delta_i \cdot y_i' \tan \theta_i) \quad (2.5)$$

$$y_{i+1}' = \cos \theta_i (y_i' + \delta_i \cdot x_i' \tan \theta_i) \quad (2.6)$$

Chapter 2

Background of CORDIC Algorithm

There are several permissible values that may be chosen for the rotation steps. If the iteration is chosen as $tg\theta_i = 2^{-i}$, $\theta_i = tg^{-1}2^{-i}$. The reason why this value is selected is that it is easier to implement in hardware. So equation (2.5) and (2.6) can be changed to:

$$x_{i+1}' = \cos\theta_i(x_i' - y_i' \cdot \delta_i \cdot 2^{-i}) \quad (2.7)$$

$$y_{i+1}' = \cos\theta_i(y_i' + x_i' \cdot \delta_i \cdot 2^{-i}) \quad (2.8)$$

Let $k_i = \frac{1}{\cos\theta_i}$, and since

$$1 + tg^2\theta = \frac{1}{\cos^2\theta}$$

$$\frac{1}{\cos\theta} = \sqrt{1 + tg^2\theta}$$

therefore,

$$k_i = \sqrt{1 + tg^2\theta} = \sqrt{1 + 2^{-2i}} \quad (2.9)$$

$K = \prod_{i=0}^n k_i$ is defined as the scale factor.

From the Equations 2.7, 2.8 and 2.9, the following equation can be obtained.

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = k_i \begin{pmatrix} x_{i+1}' \\ y_{i+1}' \end{pmatrix} = \begin{pmatrix} 1 & -\delta_i \cdot 2^{-i} \\ \delta_i \cdot 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \quad (2.10)$$

Hence, the basic CORDIC iteration can be defined as follow by drawing K out.

$$x_{i+1} = x_i - \delta_i \cdot 2^{-i} \cdot y_i \quad (2.11)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (2.12)$$

$$z_{i+1} = z_i - \delta_i \cdot \theta_i \quad (2.13)$$

where z_{i+1} is the residue rotation angle in the i-th iteration.

Chapter 2**Background of CORDIC Algorithm**

From the above equations, it can be seen that the computations of x_{i+1} or y_{i+1} require only an i -bit shift and an add/subtract operation.

The basic CORDIC algorithm has two modes: rotation and vectoring. The former rotates a vector by a given angle, while the latter determines the angle between a vector and the positive x -axis. Both modes are based on the same primitive operations and can be used for computing different trigonometric operations.

The basic CORDIC method can be generalized to provide a more powerful tool for function evaluation. The generalized CORDIC is defined as follow:

$$x_{i+1} = x_i - m\delta_i \cdot 2^{-i} \cdot y_i \quad (2.14)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (2.15)$$

$$z_{i+1} = z_i - \delta_i \cdot \theta_i \quad (2.16)$$

where the variable $m \in \{1, 0, -1\}$ specifies a circular, linear or hyperbolic coordinate system, respectively as listed below.

$$m = 1 \quad \text{Circular rotations} \quad \theta_i = \tan^{-1} 2^{-i}$$

$$m = 0 \quad \text{Linear rotations} \quad \theta_i = 2^{-i}$$

$$m = -1 \quad \text{Hyperbolic rotations} \quad \theta_i = \tanh^{-1} 2^{-i}$$

Figures 2.1, 2.2 and 2.3 show the trajectories for a vector v for successive CORDIC iterations in circular, linear and hyperbolic modes respectively.

Figure 2.1: Rotation trajectory for the circular CORDIC system ($m = 1$)

Figure 2.2: Rotation trajectory for the linear CORDIC system ($m = 0$)

Figure 2.3: Rotation trajectory for the hyperbolic CORDIC system ($m = -1$)

Generalizations of the basic CORDIC approach allow the same technique to be used to compute a broader set of functions as listed in Table 2.1. Additionally, CORDIC has been proposed for computations in a variety of discrete transforms [11-13], filters [14-16], and communication systems [17-19] for its efficient implementation in vector rotation. In [11], a CORDIC unit is used as the basic processing element in a unified architecture for the DFT (discrete Fourier transform)/DHT (discrete Hartley transform)/DCT (discrete cosine transform)/DST (discrete sine transform). A specialized CORDIC approach without scale factor compensation is proposed to implement DCT in [12]. In [14-16], the CORDIC algorithm is applied to design orthogonal IIR digital filters for its hardware efficiency. And direct digital frequency

Chapter 2

Background of CORDIC Algorithm

synthesis (DDFS) [17-19] which is widely used in communication systems, also adopts the CORDIC architecture to generate a sinusoid instead of a rather large ROM.

m	mode	initialization	output	initialization	output
1	Rotation	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x' = K \cdot (x \cos \theta - y \sin \theta)$ $y' = K \cdot (y \cos \theta + x \sin \theta)$ $z' = 0$	$x_0 = \frac{1}{K}$ $y_0 = 0$ $z_0 = \theta$	$x' = \cos \theta$ $y' = \sin \theta$ $z' = 0$
1	Vectoring	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x' = K \cdot \text{sign}(x) \cdot \sqrt{x^2 + y^2}$ $y' = 0$ $z' = \theta + \tan^{-1}(\frac{y}{x})$	$x_0 = x$ $y_0 = y$ $z_0 = 0$	$z' = \tan^{-1}(\frac{y}{x})$
0	Rotation	$x_0 = x$ $y_0 = y$ $z_0 = z$	$x' = x$ $y' = y + x \cdot z$ $z' = 0$	$x_0 = x$ $y_0 = 0$ $z_0 = z$	$y' = x \cdot z$
0	Vectoring	$x_0 = x$ $y_0 = y$ $z_0 = z$	$x' = x$ $y' = 0$ $z' = z + \frac{y}{x}$	$x_0 = x$ $y_0 = 1$ $z_0 = 0$	$z' = \frac{1}{x}$
-1	Rotation	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x' = K \cdot (x \cosh \theta - y \sinh \theta)$ $y' = K \cdot (y \cosh \theta + x \sinh \theta)$ $z' = 0$	$x_0 = \frac{1}{K}$ $y_0 = 0$ $z_0 = \theta$	$x' = \cosh \theta$ $y' = \sinh \theta$ $z' = 0$
-1	Vectoring	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x' = K \cdot \text{sign}(x) \cdot \sqrt{x^2 - y^2}$ $y' = 0$ $z' = \theta + \tanh^{-1}(\frac{y}{x})$	$x_0 = x$ $y_0 = y$ $z_0 = 0$	$z' = \tanh^{-1}(\frac{y}{x})$

Table 2.1: Functions calculated by the CORDIC algorithm

2.2 Existing CORDIC Architectures

CORDIC has received much attention since it can be used to compute a wide variety of arithmetic functions for various applications using only simple hardware components. The conventional CORDIC is a highly sequential algorithm, in which the output values of the present iteration act as the inputs to the next iteration. For an iterative CORDIC structure, the speed performance of CORDIC operation is limited by the large iteration number, N , which is generally equal to the internal wordlength, W . This iterative nature of the algorithm is a major deterrence in optimizing the speed of execution of the iterations. Several methods [20-30] were proposed to overcome this problem at the algorithmic level. To improve the speed of each rotation, redundant arithmetic [20-23] was introduced and to reduce the number of iterations, higher Radix arithmetic [24-27] was also introduced.

In conventional number systems, every addition or subtraction involves carry propagation. Independent of the adder architecture, the delay of the resulting carry ripple path is always a function of the wordlength. In the redundant arithmetic based CORDIC, the addition operation in each rotation is performed using carry-save addition or signed digit addition, resulting in constant delay irrespective of the wordlength [20-23]. Redundant number systems offer the opportunity to implement carry-free or limited carry propagation addition and subtraction with a small delay independent of the wordlength used. Since the CORDIC algorithm consists of a sequence of additions/subtractions, the use of redundant number systems is highly attractive. However, the calculation of the sign of a redundant number is quite

complicated compared to the conventional number systems where only the most significant bit has to be inspected.

The radix-4 CORDIC algorithm is an extension of the radix-2 algorithm (with powers of 4 are used instead of powers of 2). CORDIC processors that use radix-4 in the second half of the microrotations are presented in [24] and [25]. In [26], a general purpose pipelined CORDIC processor is proposed, where all the microcotations are radix-4 and the scale factor is not a constant. In this algorithm, the radix-4 microrotation is obtained as the repetition of a radix-2 microrotaion, and then the first half of the radix-4 microrotations have the complexity double that of a radix-2 microrotation, hence losing the advantage of using radix-4 microrotaions. Elisardo [27] proposed another radix-4 CORDIC algorithm for circular coordinates in the rotation mode. The number of iterations required is half of that required in radix-2 (without scaling factor compensation stages) and the complexity of each iteration is kept at the same level as the radix-2, thereby demonstrating the expected benefits of the radix-4 algorithm. However, radix-4 based CORDIC suffers from a variable scaling factor. The scaling factor needs to be computed online and its compensation is more complex than in full radix-2 or half radix-2 (radix-2 in the first half and radix-4 in the second half) architectures.

The Flat CORDIC [28-29] is a new variation of the popular CORDIC algorithm which aims to eliminate the iterative dependency of the CORDIC algorithm. It reduces the rotation number of basic CORDIC by half and decreases the computation in basic CORDIC by discarding the items that are not significant. In an N-bit Flat

Chapter 2**Background of CORDIC Algorithm**

CORDIC, the final values of x_n and y_n are expressed in terms of x_0 and y_0 for parallel implementation of the CORDIC algorithm. This is achieved by successive substitutions of x_i and y_i in the basic CORDIC equations. All the terms of which the cumulative index values are greater than a certain value will not affect the accuracy of computation. So not all terms are required for the computation of the functions, and these items can be eliminated to increase the computation speed. However, the Flat CORDIC generates only sine, cosine, sinh, cosh and exponential functions and cannot be applied to the vectoring mode of the CORDIC algorithm.

Another CORDIC architecture MVR-CORDIC was proposed by Cheng-Shing Wu et al. [30]. The major feature of the MVR-CORDIC is that the iteration number can be reduced significantly by reformatting the directional sequence, $u(i)$, which controls the rotation direction of each elementary iteration angle. In the development of the MVR-CORDIC algorithm, the following modifications are made on the microrotation procedure of the conventional CORDIC algorithm.

- (1) Skip some microrotation angles.
- (2) Repeat some rotation angles.
- (3) Confine the number of microrotations to R_m .

In the basic CORDIC, the required microrotations N is often set to be the internal wordlength, W . While in the modified algorithm the iteration number in the microrotation phase can be confined to R_m . By some searching algorithms, R_m is defined to be much less than N , usually equal to W [30]. This will lead to speed improvement in the CORDIC operation. But the scaling factor computation is still a major bottleneck in the entire CORDIC procedure.

Chapter 2

Background of CORDIC Algorithm

Some of the algorithms mentioned above are still iterative in nature and greatly restrict the speed of the CORDIC algorithm, while others obtain certain speed improvement with the tradeoff of complex hardware for variable scaling factor. So a low latency CORDIC architecture with tolerable hardware increase will speed up the computation of system much. A hybrid technique is developed in this work, in order to speed up the computation, and at the same time, not to increase the hardware complexity too much. The hybrid technique developed is presented in the next chapter.

Chapter 3

Hybrid CORDIC for Trigonometric Functions

3.1 Conventional Sin/Cos Function Generation

The computation of the sine and cosine functions and the angle rotations are often required in digital signal processing and communication applications. There are various designs that accomplish these tasks.

In most cases, Look-Up Tables (LUTs) establish directly the Sine and Cosine functions. With increasing resolution, LUTs become prohibitively large. For a W -bit wide amplitude, $W > 12$, a pure LUT solution is impractical because of excessive memory requirements. For a given amplitude resolution W , the LUT capacity is $W \times 2^p$ bits, where p is the wordlength of address. Therefore, a standard high-resolution, such as $W=16$, would require an 8-Mbit ROM. In the aspect of chip area, power budget and performance, LUTs thus create bottlenecks in the generation of the trigonometric functions. ROM compression can be achieved by exploiting the quarter wave symmetry of the sine/cosine functions and such trigonometric identities. The angle in the full range can be mapped into an angle ϕ , $0 \leq \phi \leq \pi/4$. Hence the LUT entries may be restricted to the first octant of the unit circle. If the LUT size is further

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

compressed, additional hardware is needed for decompression and spurious contributions to the output spectrum may result.

High-resolution Sin/Cos generation is thus frequently implemented in hardware without using LUTs, e.g. the CORDIC algorithm. Due to the minimalist hardware, the CORDIC algorithm is the computation engine behind most handheld calculator chips, and also many mathematics coprocessors. The CORDIC method is the first and foremost way to compute the standard trigonometric functions (sine, cosine, etc.) and their inverses (arctangent, etc.).

In the CORDIC method, the amplitudes are obtained by iteratively rotating the vector (x, y) in the polar plane. Increments or decrements of the rotation angle, z , in steps of $\theta_i = \arctg(2^{-i})$ amount to summing and successive binary shifts (i.e. divisions by 2) of x_i and y_i . The algorithm computes the set of equations shown in Section 2.1 and repeated here as Equations 3.1, 3.2 and 3.3.

$$x_{i+1} = x_i - \delta_i \cdot 2^{-i} \cdot y_i \quad (3.1)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (3.2)$$

$$z_{i+1} = z_i - \delta_i \cdot \theta_i \quad (3.3)$$

For each integer value of i , $0 \leq i \leq 15$, the corresponding values of 2^{-i} and θ_i are shown in Table 3.1.

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

i	2^{-i}	θ_i	i	2^{-i}	θ_i
0	1	45°	8	0.0039062	0.223811°
1	0.5	26.565051°	9	0.0019531	0.111906°
2	0.25	14.036243°	10	0.0009766	0.055953°
3	0.125	7.125016°	11	0.0004883	0.027976°
4	0.0625	3.576334°	12	0.0002441	0.013988°
5	0.03125	1.789911°	13	0.0001221	0.006994°
6	0.015625	0.895174°	14	0.0000610	0.003497°
7	0.0078125	0.447614°	15	0.0000305	0.001749°

Table 3.1: Elementary rotation angle θ_i

The positive/negative sense of each rotation, δ_i , must be chosen such that the procedure converges, i.e. $\delta_i = \text{sign}(z_i)$. Starting with $x_0 = 1$ (or another suitable amplitude) and $y_0 = 0$, the sequence of the x (or y) values then converges against the amplitude value of the $\text{const} \times \text{Cos}$ (or $\text{const} \times \text{Sin}$) function for a given target phase angle $\theta = z_0$. The constant depends only on the number of iterations, as the algorithm is not length conserving. The solution is to start out with an amplitude value already divided by this constant. Each iteration renders one bit of the binary amplitudes, e.g. for $W=16$ at least 16 iterations have to be performed. After W iterations in the rotation mode, when z is sufficiently close to 0, the CORDIC equations become:

$$x' = K(x_0 \cos \theta - y_0 \sin \theta) \tag{3.4}$$

$$y' = K(y_0 \cos \theta + x_0 \sin \theta) \tag{3.5}$$

$$z' = 0 \tag{3.6}$$

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

Starting with $x = 1/K$ and $y=0$, as z tends to 0, x and y converge to $\cos z$ and $\sin z$ respectively. Once $\cos z$ and $\sin z$ are known, $\tan z$ can be obtained through division. Figure 3.1 illustrates the process of rotating a vector by 30° from the x-axis. Note how the rotation converges towards the target angle in an iterative fashion and the effects of gain on the size of the vector.

Figure 3.1: Vector rotation by 30° using a conventional CORDIC algorithm

The main shortcoming of the CORDIC-based Sin/Cos generation is its long latency, with the input to each subrotation stage depending on the output of the previous one. In some systems, such as in frequency-hopping applications, a minimal latency time is desirable. In view of this requirement, a different approach is needed. A hybrid technique, which overcomes this drawback, is developed and presented in the next section.

3.2 Hybrid Technique

3.2.1 Theoretical Orientation

In the implementation of the sine/cosine generator based on the CORDIC algorithm, the main problem is the iterative nature of CORDIC, which results in slow computation. The obvious solution is to reduce the number of iterations or to reduce the time for each iteration. As discussed in Chapter 2, some attempts to overcome such a problem were proposed to reduce the number of iterations. However, a signal may be seriously distorted by the approximation and quantization noise in practical implementations. In this project, a hybrid scheme, which combines a traditional look-up table (LUT) with the algorithmic iterative procedures of the CORDIC algorithm, is proposed. The iterative CORDIC algorithm is rigorous for infinite bit resolution, permitting in principle arbitrary precision, while the look-up table size increases rapidly with resolution and is thus limited to a small-resolution input data regime. If the LUT and the CORDIC portions are properly balanced, a hybrid scheme, which combines the advantage of a LUT and a CORDIC, can be obtained. The hybrid approach, therefore, offers the advantages of both the fast access and the power efficiency of a reasonably sized LUT and the arbitrary precision obtainable from the systematic iteration algorithm CORDIC.

3.2.2 Methodology

With increasing resolution, a LUT for sine/cosine amplitudes may become prohibitively large. In this project, a LUT with the resolution of 1 degree, which contains all the amplitude values for sine and cosine within $0 \leq \theta \leq \pi/4$, is used. The reason for requiring only $\pi/4$ is that the angles outside $\pi/4$ can be handled by performing initial rotations (pre-rotations) of 180° and/or 90° , thereby reducing the angle to one that exists in the first quadrant. And the sine and cosine values of θ within $\pi/4 < \theta \leq \pi/2$ can also be represented by cosine and sine values of $(\pi/2 - \theta)$ respectively. So the LUT just contains the sine and cosine values within $0 \leq \theta \leq \pi/4$. Note that a LUT, which contains the value of sine or cosine within $0 \leq \theta \leq \pi/2$, will also work. Combining such a small sized LUT and the CORDIC algorithm, the hybrid CORDIC is achieved. The operation of the hybrid CORDIC is described below and its implementation is presented in Section 3.2.4.

When a rotation angle θ is input, the amplitude of the integer part of θ is sent to θ_0 . Then the corresponding sine and cosine values of θ_0 are obtained from the LUT and assigned to x_1 and y_1 . This can be viewed as the pseudo first rotation of the hybrid CORDIC. This first rotation angle equals to the integer part of the input angle θ . Therefore the accumulated rotation angle followed is less than 1 degree. Because $\arctan(2^{-6}) = 0.895$ degree is less than 1 degree, the second rotation, which is actually the first rotation, starts with $\theta_1 = \arctan(2^{-6})$. In the conventional CORDIC, mircorotations are executed step by step starting from $\theta_0 = \arctan(2^0)$, followed

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

$\theta_1 = \arctan(2^{-1})$, $\theta_2 = \arctan(2^{-2})$ and so on. Altogether there are N steps, which are generally equal to the internal wordlength W . For $W=16$, 16 iterations are executed.

Figure 3.2 shows the rotation steps of a conventional CORDIC. For clarity, not all the iterations are presented in the figure. In the hybrid scheme, the initial vector comes close to the destination vector more quickly. The first microrotation makes the vector rotate an angle that is almost equal the sum of the angles performed by the first 6 microrotations in a conventional CORDIC. Therefore only one step is taken to replace the original 6 steps in a conventional CORDIC. For a computing process with $W=16$, only 10 iterations are needed and almost 40% of the rotations are eliminated. Figure 3.3 shows the rotation steps in the hybrid CORDIC. It can be seen that the computation speed can be significantly improved by reducing the number of rotations.

Figure 3.2: Rotation trajectory in a conventional CORDIC

Figure 3.3: Rotation trajectory in the hybrid CORDIC

Another issue to be addressed is the rotations of angles outside the first quadrant. This problem can be handled by performing an initial rotation (pre-rotations) of 90° , 180° or both, thereby reducing the angle to one that is in the first quadrant. As shown below, these pre-rotations can be carried out simply by changing the sign of the inputs or by swapping them and do not cause any additional scaling.

Pre-rotation by 180° : $(x, y) \leftarrow (-x, -y)$, $\theta = \theta - 180^\circ$

Pre-rotation by 90° : $(x, y) \leftarrow (-y, x)$, $\theta = \theta - 90^\circ$

3.2.3 Scale Factor

At first glance, the vector scaling introduced by the CORDIC algorithm does not seem to pose a significant problem. Actually, the scaling factor compensation process heavily influences the execution speed of the CORDIC algorithm. The correction of a possibly variable scale factor generally requires two divisions or at least two multiplications with the corresponding reciprocal values. As a result, the time required to incorporate the scaling factor compensation is comparable to that needed to compute all the terms of the CORDIC expression. Thus, it is crucial that the scaling operation is speeded up to ensure that the overall computation time is kept to a minimum to realize a high speed CORDIC algorithm. A constant scale factor is enough for the hybrid CORDIC, which avoids the influence caused by the variable one. As described in Section 3.2.2, the first rotation just makes the vector rotate a certain angle with the same magnitude, which does not cause the scaling. So only scaling caused by the following rotations need to be considered. It means that the scaling factors of the initial 6 rotations can be omitted, so for a 16-iteration CORDIC,

the scaling factor is $K = \prod_{i=6}^{15} k_i = \prod_{i=6}^{15} \sqrt{1 + 2^{-2i}}$, instead of the scale factor of a

conventional CORDIC $K = \prod_{i=0}^{15} k_i = \prod_{i=0}^{15} \sqrt{1 + 2^{-2i}}$. The scale factor can be computed

in advance and the result is a constant. Using a fixed point number representation, a scaling multiplication operation can be realized by a corresponding number of shift and add operations.

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

3.2.4 RTL Model of Hybrid CORDIC

A fully synthesizable RTL model was developed in VHDL to implement build the architecture of hybrid CORDIC. The model implements a pipelined CORDIC and operates on fixed-point input values.

Figure 3.4 shows the VHDL entity interface named `hybrid_fast`. It consists of seven ports `clk`, `writestate`, `x`, `y`, `a`, `xout` and `yout`. The `x` and `y` ports are inputs for the coordinate data. The `xout` and `yout` ports are the output ports for the cosine and sine function values, respectively. The port `a` is the input for the rotation angle. The port `clk` is the input for the clock signal, and the port `writestate` is to enable and disable the operation of the computation.

Figure 3.4: The VHDL entity interface of hybrid CORDIC

The fixed-point format used in this model is based on 16-bit word length. Two formats are used in the model. The first format is used to represent the input

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

coordinates and the rotation angle. One bit is used for the sign, eight bits for encoding the integer portion of the number, and the remaining seven bits for the fractional representation, as shown in Figure 3.5. The outputs for the sine and cosine values are in a different 16-bit signed fixed point format. One bit is used for the sign, one bit is for the integer portion, and the remaining 14 bits for the fractional portion, as shown in Figure 3.6. The reason for employing different formats for input and output is that the output sine and cosine values are smaller than or equal to one. Therefore, if it uses the same representation as the input, most of the integer bits will be 0. In order to make full use of the bits, the second format is adopted.

Figure 3.5: Fixed-point format for input

Figure 3.6: Fixed-point format for output

Since the fraction part of output values has a 14-bit length, 14 rotation stages are required for the conventional CORDIC. For this 14-stage conventional CORDIC, the scale factor is

$$K = \prod_{i=6}^{14} k_i = \prod_{i=6}^{14} \sqrt{1 + 2^{-2i}} = 1.000163$$

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

As described in Section 3.2.3, the scale factor of the hybrid CORDIC can be

represented by $K = \prod_{i=6}^{n-1} k_i = \prod_{i=6}^{n-1} \sqrt{1 + 2^{-2i}}$. In order to remove the gain inherent to the

algorithm, a factor of $1/K = \prod_{i=6}^{15} \frac{1}{\sqrt{1 + 2^{-2i}}} = 0.999837$ needs to be multiplied with the

rotated values. Since $1/K$ is constant, in fixed-point representation, the scale factor compensation i.e. a constant multiplication can be implemented by a series of shift and add/subtract operations. Thus $x \leftarrow x * 0.999837$ can be replaced by $x \leftarrow x - (x \gg 13) - (x \gg 15)$, which equals to 0.999847. Obviously the advantage of this approach is that it eliminates the need of a multiplier.

The hybrid CORDIC model is pipelined. In contrast to a universal CORDIC processing element, the dominating motivation for a pipelined architecture is to have a high throughput. Additionally, it is advantageous if relatively long streams of data have to be processed in the same manner since it takes a number of clock cycles to fill the pipeline and also to flush the pipeline if the control flow changes (a different function is to be calculated). Although pipeline registers are usually inserted in between the single CORDIC iterations, they can, in principle, be placed anywhere since the unfolded algorithm is purely feed forward. The main advantage of pipelined CORDIC architectures compared to the recursive implementations is the possibility to implement hard-wired shifts rather than the area and time consuming barrel shifters. However, the shifts can be hard-wired only for a single fixed shift sequence. Nevertheless, a small number of different shifts can be implemented using multiplexers which are still much faster and less area consuming than the barrel shifters needed for the folded recursive architecture. A similar consideration holds for

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

the rotation angles. If only a single shift sequence is implemented, the angles can be hard-wired into the adders/subtractors. A small number of alternative rotation angles per stage can be implemented using a small combinatorial logic steering the selection of a particular rotation angle. In the hybrid CORDIC all the shifts and many additions have one constant argument. They can be implemented in simpler hardware.

For a conventional CORDIC, 14 bits are used to represent the fractional parts of the output values, therefore 14 pipelined stages are implemented for a desired output accuracy in the conventional CORDIC. In addition, 2 stages are used for pre-rotation to reduce the input angle to within 90 degree. Therefore, 16 pipelined stages are required in the conventional CORDIC. Compared to the conventional CORDIC, the hybrid CORDIC has 6 stages less. Thus there are only 10 pipelined stages. Figure 3.7 shows the architecture of the hybrid CORDIC developed in this work, where z_0 is the input rotation angle and z' is the integer part of z_0 . For comparison, the architecture of the conventional CORDIC is presented in Figure 3.8.

Figure 3.7: The architecture diagram of the hybrid CORDIC

Figure 3.8: The architecture diagram of the conventional CORDIC

3.2.5 Simulation Results

This section describes the simulation carried out to evaluate the performance of the hybrid CORDIC on the sin/cos generation. The VHDL model described above is

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

simulated in Modelsim and synthesized with FPGA Advantage. For comparison, a RTL model of the conventional CORDIC was also developed in VHDL and synthesized. The input to the models is a list of fixed-point numbers defining the vectors to be rotated and the rotation angles. The expected output is a list of numbers in the same fixed-point representation, encoding the coordinates of the rotated vectors.

In order to evaluate the accuracy of the results generated by the hybrid CORDIC, a group of 100 values of θ (rotation angle) random generated are input while x and y are forced to 1, 0 respectively. The corresponding values of $\sin\theta$ and $\cos\theta$ generated by C functions are used for comparison. Table 3.2 shows the comparison between the conventional CORDIC and the hybrid CORDIC in terms of the absolute value of error and the mean square error when compared with the values obtained from the C functions.

	Conventional CORDIC		Hybrid CORDIC	
	$\sin\theta$	$\cos\theta$	$\sin\theta$	$\cos\theta$
mean error	0.867%	0.766%	0.561%	0.450%
mean square error	0.015%	0.014%	0.013%	0.012%

Table 3.2: Comparison of errors

It can be seen from Table 3.2 that the results of the hybrid CORDIC are as accurate as those of the conventional CORDIC. The sine values obtained from the C function, the conventional CORDIC and the hybrid CORDIC are plotted in Figure 3.9. From the

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

Figure 3.9, it is obvious that the curve obtained from the hybrid CORDIC is almost overlap with the curve obtained from the C function, and the curve from the conventional CORDIC is also close to the curve from the C function.

Figure 3.9: The sine function values of CORDIC

The conventional CORDIC and the hybrid CORDIC models are both synthesized in the LeonardoSpectrum tool [31]. Use the Altera’s FPGA EP20K200EFC484 [32] as the configured device, which has 8320 logic cells and a maximum clock frequency of 180Mhz, the synthesized results are shown in Table 3.3. The hybrid CORDIC uses less logic cells but requires 128 bits of ROM. In timing comparison, the prerotation to refine θ within $[0, \pi/4]$ costs 2 clock cycles, plus the 14 stage rotations, 16 clock cycles are needed for the basic CORDIC. The time of prerotation is also necessary for the hybrid CORDIC, but since 6 stages have been eliminated only 8 rotations are

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

needed. Hence the hybrid CORDIC takes only 10 clock cycles to reach the results.

With a comparable size, the latency is reduced by 35.2%.

	Used logic cells	Used memory bits	Clock (cycles)	Clock Frequency (MHz)	Process Time (ns)	Latency reduced
Conventional CORDIC	1076	0	16	101.2	1581	0
Hybrid CORDIC	823	128	10	97.5	1025	35.2%

Table 3.3: Synthesized results of CORDIC

3.2.6 Selection of Resolution for Look-up Table

In this work, only one resolution of the LUT is implemented for the sine and cosine functions. With comparable area cost, the latency of computation is reduced. When the resolution of look-up table varies, the size of the LUTs changes consequently. Correspondently the number of rotations will increase or decrease. For example, given a LUT with the resolution of 0.1 degree, namely the first rotation angle equals the integer portion plus the first bit of fraction portion of the desired rotation angle, only the fraction part from the second bit will be rotated subsequently. For an angle of 45.586° , only 0.086° is required to be rotated. Since $\arctan(2^{-10}) = 0.055953^\circ$ degree is less than 0.1° , the first rotation starts with $\theta_1 = \arctan(2^{-10})$. Therefore compared to the conventional CORDIC, 10 rotations are saved. Table 3.4 shows the number of entries of look-up table and the required rotations for a 16-bit width input.

Chapter 3 Hybrid CORDIC for Trigonometric Functions

Configuring with a LUT of smaller resolution, the power efficiency of reasonably sized LUT can also be achieved. If a LUT with smaller resolution is adopted the number of followed rotations can be small and the computation will be faster.

Entry of LUT	Bit width	ROM size (bit)	Followed rotations
8	16	144	13
16	16	256	12
32	16	512	11
90	16	1440	10
128	16	2048	9
256	16	4096	8
512	16	8192	7
1024	16	16384	6

Table 3.4: The entry number of LUT and the required rotations

3.3 Extensions to the Hyperbolic Coordinate

Hyperbolic CORDIC rotations can be used to compute the hyperbolic sine and cosine functions. For hyperbolic CORDIC iterations, it is more difficult to ensure convergence. This is because unlike $\tan^{-1}(2^{-(i+1)}) \geq 0.5 \tan^{-1}(2^{-i})$, the corresponding relation for $\tanh$, namely, $\tanh^{-1}(2^{-(i+1)}) \geq 0.5 \tanh^{-1}(2^{-i})$, does not hold in general.

A relatively simple solution is to repeat steps for $i=4, 13, 40, 121, \dots, j, 3j+1, \dots$ to ensure convergence (each term is 1 more than 3 times of the preceding term). In other words, the iterations corresponding to the foregoing values of i are executed twice. So the sequence of θ_i will be $\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_4, \theta_5, \dots$ etc. The effect of these repetitions on performance is minimal because in practice the iteration always stops for $m < 121$. These repeated steps have already been taken into account in computing the shrinkage constant K given earlier.

In the hyperbolic CORDIC rotations, each input angle θ , can be expressed as:

$$\theta = \sum_{i=1}^k \delta_i \tanh^{-1}(2^{-i}) + \delta_4' \tanh^{-1}(2^{-4}) + \delta_{13}' \tanh^{-1}(2^{-13}) + \delta_{40}' \tanh^{-1}(2^{-40}) + \dots \quad (3.7)$$

After W iterations in the rotation mode, when z is sufficiently close to 0, the CORDIC equations become:

$$x' = K \cdot (x \cosh \theta - y \sinh \theta) \quad (3.8)$$

$$y' = K \cdot (y \cosh \theta + x \sinh \theta) \quad (3.9)$$

$$z' = 0 \quad (3.10)$$

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

Since $\sinh$ and $\cosh$ can be computed by CORDIC and $\sinh(x) = \frac{e^x - e^{-x}}{2}$ and

$\cosh(x) = \frac{e^x + e^{-x}}{2}$, the value of e^x can be obtained by simply adding them together.

Similar to the proposal in Section 3.2, the hybrid technique can also be applied in the hyperbolic CORDIC. A LUT that stores the values of $\sinh$ and $\cosh$ functions is employed, where the most significant several bits of θ are used as the address. The most significant m bits of $\sinh$ and $\cosh$ can be read from the LUT, which are actually the values of x_m and y_m after m rotations in the conventional CORDIC. So the first m rotation stages can be replaced by the LUT and the delay of m stage of shift and addition can be saved.

Because of the disappearance of the m stages, the scaling caused by the first m rotations does not exist any more. Only scaling caused by the following $(n-m)$ rotations needs to be considered. Thus for a hybrid CORDIC, the scaling factor is

$$K = \prod_{i=m}^{n-1} k_i = \prod_{i=m}^{n-1} \sqrt{1 - \tanh^2 \theta_i}, \text{ instead of the scale factor of the conventional}$$

$$\text{CORDIC } K = \prod_{i=0}^{n-1} k_i = \prod_{i=0}^{n-1} \sqrt{1 - \tanh^2 \theta_i}. \text{ The scale factor is a constant and can also be}$$

computed in advance.

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

3.3.1 Simulation Results

A simulation was carried out to evaluate the performance of the hybrid CORDIC on the sinh/cosh generation. A 16-stage conventional CORDIC and a 9-stage CORDIC combined with a ROM (hybrid CORDIC) were modelled in VHDL.

A group of 100 values of θ (rotation angle) randomly generated is used to test the accuracy of the results generated by the hybrid CORDIC. The corresponding values of $\sinh \theta$ and $\cosh \theta$ generated by the C function are used for comparison. Table 3.5 shows the comparison between the conventional CORDIC and the hybrid CORDIC in terms of the absolute value of error and mean square error when compared with the results obtained from the C functions.

	Conventional CORDIC		Hybrid CORDIC	
	$\sinh \theta$	$\cosh \theta$	$\sinh \theta$	$\cosh \theta$
mean error	0.365%	0.327%	0.351%	0.324%
mean square error	0.003%	0.002%	0.002%	0.002%

Table 3.5: Comparison of results error for hyperbolic CORDIC

The area and latency between the conventional CORDIC and the hybrid CORDIC for hyperbolic coordinate are compared, using the Altera's FPGA EP20K200EFC484 as the configured device. The area cost of the hybrid CORDIC is much smaller than that of the conventional CORDIC, but the hybrid CORDIC requires a ROM of 128 bits in addition. Since the access to the ROM replaces the delay for 7 stages, the latency for

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

executing the hybrid CORDIC is reduced by 37.5% when compared with the conventional CORDIC. Table 3.6 shows the synthesized results.

	Used logic cells	Used memory bits	Clock (cycles)	Clock Frequency (MHz)	Process Time (ns)	Latency reduced
Conventional CORDIC	1067	0	16	102.2	1565	0
Hybrid CORDIC	618	128	10	102.2	978	37.5%

Table 3.6: Synthesized results of CORDIC for hyperbolic coordinate

3.4 Improved Hybrid CORDIC

3.4.1 Methodology

From the description in Section 3.3, it can be seen that the iterative nature of the algorithm is a major deterrence in optimizing the speed of execution of the CORDIC. By reducing the first several stages of the CORDIC, the length of the sequential rotations becomes shorter. However the speed bottleneck also lies in the sequential determination of the value of $\delta_i = \text{sign}(z_i)$, $i=1, 2, 3, \dots, N$. The operations on x_i and y_i can be executed only after the corresponding z_i has been decided with the correct rotation direction. If the direction determination for all z_i can be parallelized, the corresponding computations on x_i and y_i can also be executed concurrently.

In [33], a parallel architecture was proposed for the CORDIC. Wang et al. partition the binary representation of $\theta = \sum_{i=0}^{N-1} b_i 2^{-i}$ (b_i being the bits in the N -bit binary representation) into two parts: the m most-significant bits and the remaining $N - m$ bits. Each of these groups of bits is the binary representation of an angle, namely:

$$\theta_H = \sum_{i=0}^{m-1} b_i 2^{-i} \quad (3.11)$$

$$\theta_L = \sum_{i=m}^{N-1} b_i 2^{-i} \quad (3.12)$$

where m is the smallest index value such that $2^{-m} - \tan^{-1} 2^{-m} < 2^{-N}$. It has been shown in [33] that $m = \lceil (N - \log_2 3) / 3 \rceil$.

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

In this way, the rotation θ is partitioned into two rotations: θ_H and θ_L . θ_H is the most significant while θ_L adjusts the position of the rotated vector to achieve the final expected (or nearby) position. θ_L always has zeros in the m most significant bits. The angle θ_H is completely independent of θ_L only if it does not change the $N - m$ least significant bits of the residue rotation angle with respect to θ_L . To achieve this goal, the exact rotation θ_H must be performed separately from θ_L . The CORDIC iterations related to the most significant part are performed as in the conventional CORDIC algorithm. In order to avoid errors, each rotation direction can be evaluated only after the previous iteration has been completed. The iterations related to the least significant part can be simplified as in the linear coordinate system. All rotation directions associated with such iterations are available in parallel, since each of them is given by the corresponding bit of the residue rotation angle (1 and 0 identify positive and negative rotation, respectively). So the rotation directions for θ_L can be predicted directly from the binary value of the initial input angle. For θ_L the delay waiting for the z stage to decide the rotation direction can be removed.

In [33], the rotation angle is divided into 2 parts and the delay for the lower part of the angle decreases. Furthermore, the proposed hybrid CORDIC can also reduce the delay for the first several rotation stages by replacing them with a LUT. The combination design of both the methods can have both their advantages. As described in Section 3.2.6, the number of the replaced stages can be chosen while the resolution of the LUT varies correspondently. For this case, the number of the replaced stages is set to not less than $(m-1)$ in order to take the advantages of the method in [33]. Then, the

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

rotation angle θ is also portioned into two parts θ_H' and θ_L' : the M most-significant bits and the remaining $N-M$ bits. Here, M is equal or larger than $(m-1)$.

$$\theta_H' = \sum_{i=0}^{M-1} b_i 2^{-i} \quad (3.13)$$

$$\theta_L' = \sum_{i=M}^{N-1} b_i 2^{-i} \quad (3.14)$$

For θ_H' , a LUT is adopted instead of using the conventional rotation stages as in [33]. And for θ_L' the iterations are realized in the same way as in [33]. Therefore, the computation speed can be further improved and is faster than both the hybrid method proposed in Section 3.2 and the method in [33]. Since M is not a constant anymore and can vary in a range, the corresponding changes in delay and area are investigated in the next section.

3.4.2 Performance Evaluation

The delay and area of the improved hybrid CORDIC is analyzed here. T_{FA} and A_{FA} denote the delay and the area of an one-bit full adder cell. The newly proposed improved hybrid CORDIC algorithm is also compared with the hybrid CORDIC and the method in [33] in this section. To make a fair comparison, it is assumed that all these methods which can predict the partial rotation directions concurrently use carry save adders (CSA) in these direction-predicted rotations plus an additional final stage where fast carry propagate adders (CPA) are employed to convert the operands of carry-save forms back to non-redundant binary representations. The common delay of

Chapter 3

Hybrid CORDIC for Trigonometric Functions

the T_{CPA} in the final stage is omitted in the following comparisons. The delay and area required in each of these direction-predicted rotations with word size of N bits are $2T_{FA}$ and $4N \times A_{FA}$ respectively.

For the conventional CORDIC, assuming that a fast N -bit CPA has delay of $\lceil \log_2 N \rceil \times T_{FA}$ with hardware cost of $N \lceil \log_2 N \rceil \times A_{FA}$, the total latency of the CORDIC is $N \lceil \log_2 N \rceil \times T_{FA}$ and the corresponding hardware cost is $N^2 \lceil \log_2 N \rceil \times A_{FA}$.

In the hybrid CORDIC, a ROM of size $2^j \times 2N$ is used to replace the first j stages of the conventional CORDIC. According to M. J. Flynn et al.'s theory [34], the area of one byte ROM takes 3.5 gates, which is equivalent to $A_{FA}/2$ using 0.35 μm CMOS technology. The delay and area for the $2^j \times 2N$ ROM is $\log_2 2^j \times T_{FA}/2$ and $2^j \times \lceil 2N/8 \rceil \times A_{FA}/2$, respectively. Thus the delay and area costs of the hybrid CORDIC are $[j/2 + (N-j) \lceil \log_2 N \rceil] \times T_{FA}$ and $[2^{j-1} \times \lceil 2N/8 \rceil + (N-j)N \lceil \log_2 N \rceil] \times A_{FA}$.

In [33], the rotation directions after the first m iterations are derived from the z remainders at the end of the m -th iteration. However, the first m iterations still adopt the conventional sequential approach in the Z -datapath. Thus, the hardware complexity for the X/Y micro-rotations and the Z operation are almost the same as that in the conventional CORDIC except that the execution time for Z operation is reduced to only one-third of conventional CORDIC. So the total latency of the

Chapter 3**Hybrid CORDIC for Trigonometric Functions**

CORDIC is $(\lceil \log_2 N \rceil \times N/3 + 4N/3) \times T_{FA}$ with the hardware cost of $(8N^2/3 + \lceil \log_2 N \rceil \times N^2/3) \times A_{FA}$.

For the improved hybrid CORDIC, a ROM of size $2^M \times 2N$ is adopted and the rotation directions of the remainder $(N-M)$ rotations can be obtained by the binary representation of the least significant part of θ . From the above calculation, it is clear that the total delay of this method is $[M/2 + 2(N-M)] \times T_{FA}$ and the total hardware cost is $[2^{M-1} \times \lceil 2N/8 \rceil + 4N(N-M)] \times A_{FA}$.

For $N=16$, the area cost of the hybrid CORDIC is $[2^{j-1} \times \lceil 2 \times 16/8 \rceil + (16-j) \cdot 16 \cdot \lceil \log_2 16 \rceil] \times A_{FA} = [2^{j-1} \times 4 + (16-j) \cdot 64] \times A_{FA}$. And the hardware cost of the improved hybrid CORDIC is $[2^{M-1} \times \lceil 2 \times 16/8 \rceil + 4 \times 16 \cdot (16-M)] \times A_{FA} = [2^{M-1} \times 4 + (16-M) \cdot 64] \times A_{FA}$. When $j = M$ i.e. using LUTs of the same size, the areas of both the methods are equal for $N=16$. Compare the area of the hybrid CORDIC $[2^{j-1} \times \lceil 2N/8 \rceil + (N-j)N \lceil \log_2 N \rceil] \times A_{FA}$ with the area of the improved hybrid CORDIC $[2^{M-1} \times \lceil 2N/8 \rceil + 4N(N-M)] \times A_{FA}$. When $j = M$, $2^{j-1} \times \lceil 2N/8 \rceil$ is equal to $2^{M-1} \times \lceil 2N/8 \rceil$, which shows that the first term of each equation is the same. Observing the remaining terms of both the equations, $(N-j)N \lceil \log_2 N \rceil$ is less than $4N(N-M)$ when $N \leq 9$ and greater than $4N(N-M)$ when $N > 16$, because $\lceil \log_2 N \rceil < 4$ when $N \leq 9$ and $\lceil \log_2 N \rceil > 4$ when $N > 16$, see Table 3.7. Therefore, when N is larger than 16, the area of the improved hybrid CORDIC is less than that of

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

the corresponding hybrid CORDIC. When N is not larger than 9, the area of the improved hybrid CORDIC is greater than that of the corresponding hybrid CORDIC.

$j = M$	$N > 16$	$9 < N \leq 16$	$N \leq 9$
Area of the Hybrid CORDIC (A_H) $[2^{j-1} \times \lceil 2N/8 \rceil + (N-j)N \lceil \log_2 N \rceil] \times A_{FA}$	$(N-j)N \lceil \log_2 N \rceil$	$(N-j)N \lceil \log_2 N \rceil$	$(N-j)N \lceil \log_2 N \rceil$
Area of the Improved CORDIC (A_I) $[2^{M-1} \times \lceil 2N/8 \rceil + 4N(N-M)] \times A_{FA}$	$>$ $4N(N-M)$	$=$ $4N(N-M)$	$<$ $4N(N-M)$
	$A_H > A_I$	$A_H = A_I$	$A_H < A_I$

Table 3.7: Area comparison of both hybrid CORDIC in terms of N

The delay and area of the hybrid CORDIC and the improved hybrid CORDIC are computed and compared with the method in [33] when $N=16$. Table 3.8 shows the decrease of the delay of both the hybrid CORDICs compared with that of [33]. It is clear that the improved hybrid CORDIC reduces the delay sharply. The area comparison is shown in Table 3.9. From Table 3.8 and Table 3.9, it can be seen that the delay of the hybrid CORDIC is longer than that of [33] when $M \leq 6$ and shorter than that when $M > 6$. While the delay of the improved hybrid CORDIC is always less than that of [33]. In hardware cost, both the hybrid and the improved hybrid CORDIC methods are more efficient in area than the method of [33] when $M < 8$. It is obvious that when $M=7$, both delay and area decrease are achieved in improved hybrid CORDIC.

Chapter 3 **Hybrid CORDIC for Trigonometric Functions**

M (m=5)	ROM entries	Number of remaining rotations	Delay of [33]	Hybrid CORDIC		Improved Hybrid CORDIC	
				Delay	$\Delta\%$ (compared with [33])	Delay	$\Delta\%$ (compared with [33])
4	2^4	12	42.67	50	17.19%	26	-39.06%
5	2^5	11	42.67	46.5	8.98%	24.5	-42.58%
6	2^6	10	42.67	43	0.78%	23	-46.09%
7	2^7	9	42.67	39.5	-7.42%	21.5	-49.61%
8	2^8	8	42.67	36	-15.62%	20	-53.12%
9	2^9	7	42.67	32.5	-23.83%	18.5	-56.64%
10	2^{10}	6	42.67	29	-32.03%	17	-60.16%
11	2^{11}	5	42.67	25.5	-40.23%	15.5	-63.67%
12	2^{12}	4	42.67	22	-48.44%	14	-67.19%

Table 3.8: Comparison of the delay of both hybrid CORDIC

M (m=5)	ROM entries	Number of remaining rotations	Area of [33]	Hybrid CORDIC		Improved Hybrid CORDIC	
				Area	$\Delta\%$ (compared with [33])	Area	$\Delta\%$ (compared with [33])
4	2^4	12	1024	800	-21.87%	800	-21.87%
5	2^5	11	1024	768	-25.00%	768	-25.00%
6	2^6	10	1024	768	-25.00%	768	-25.00%
7	2^7	9	1024	832	-18.75%	832	-18.75%
8	2^8	8	1024	1024	0%	1024	0%
9	2^9	7	1024	1472	43.75%	1472	43.75%
10	2^{10}	6	1024	2432	137.5%	2432	137.5%
11	2^{11}	5	1024	4416	331.25%	4416	331.25%
12	2^{12}	4	1024	8448	725%	8448	725%

Table 3.9: Comparison of the area of both hybrid CORDIC

Chapter 4

Hybrid CORDIC for Reciprocation

4.1 Reciprocation in Conventional CORDIC

The computation of reciprocal has been considered of importance for many years since it appears in many applications. For instance, the reciprocal is usually employed in scientific computing and digital signal processing, besides being employed as a seed in the multiplier-based iterative division algorithms, or even substituting these algorithms (a division operation can be performed by means of a reciprocal calculation and a multiplication). In today's FPGA technology, hardware multipliers are commonly available. Some FPGA devices provide tens and hundreds of hardware multipliers. If the reciprocal of a value can be obtained, the division operation can be achieved by using a hardware multiplier. This chapter presents a hybrid technique for the reciprocal function.

Different from the sine and cosine function generation, the reciprocal function is realized in the vectoring mode of linear rotations of the CORDIC algorithm. For division, $m=0$ and $\theta = 2^{-i}$ are used in the CORDIC equations, repeated here as Equations 4.1, 4.2 and 4.3.

Chapter 4**Hybrid CORDIC for Reciprocation**

$$x_{i+1} = x_i - m\delta_i \cdot 2^{-i} \cdot y_i \quad (4.1)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (4.2)$$

$$z_{i+1} = z_i - \delta_i \cdot \theta_i \quad (4.3)$$

Thus the equations become:

$$x_{i+1} = x_i \quad (4.4)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (4.5)$$

$$z_{i+1} = z_i - \delta_i \cdot 2^{-i} \quad (4.6)$$

In the vectoring mode, the objective is to rotate the given input vector (x, y) with a magnitude $\sqrt{x^2 + y^2}$ and an angle $\phi = -\tan^{-1}(\frac{y}{x})$ towards the x-axis. With $z_0 = 0$, the control scheme is such that during the n iterations y_n is driven to zero: $\delta_i = -\text{sign}(x_i) \cdot \text{sign}(y_i)$. Depending on the sign of x_0 , the vector is then rotated towards the positive ($x_0 \geq 0$) or negative ($x_0 < 0$) x-axis, as shown in Figure 4.1. If $y_n = 0$ holds, z_n contains the negative total accumulated rotation angle after n iterations which is equal to ϕ , as given in Equation 4.7.

$$z_n = \phi = -\sum_{i=0}^{n-1} \delta_i \cdot 2^{-i} \quad (4.7)$$

In the linear rotations corresponding to $m=0$, the end point of the vector is kept on the line $x = x_0$ and the vector length is always the length between the origin of

Chapter 4**Hybrid CORDIC for Reciprocation**

coordinates and the point on the line $x = x_0$. Hence, the scaling factor is 1 and the rotations are true linear rotations in this case.

From Equation 4.5, it can be obtained:

$$\begin{aligned}
 0 &= y + \sum_{i=0}^{n-1} \delta_i \cdot 2^{-i} \cdot x_i \\
 y &= -\sum_{i=0}^{n-1} \delta_i \cdot 2^{-i} \cdot x \\
 y/x &= -\sum_{i=0}^{n-1} \delta_i \cdot 2^{-i}
 \end{aligned} \tag{4.8}$$

Thus Equations 4.9, 4.10 and 4.11 characterize the results of the linear CORDIC rotations.

$$x' = x \tag{4.9}$$

$$y' = 0 \tag{4.10}$$

$$z' = \frac{y}{x} \tag{4.11}$$

The reciprocal of x can be obtained by initializing with: $z=0$ and $y=1$.

Due to the iterative nature of CORDIC, the computation of reciprocal function has a slow linear convergence, which makes it too slow for many real-time applications. To solve this problem, a hybrid technique is developed in this work and applied to CORDIC in the vectoring mode of linear rotations.

Figure 4.1: Rotation trajectory in vectoring mode of CORDIC

4.2 Reciprocation Based on Hybrid Technique

A high-speed method based on a hybrid CORDIC for computing the reciprocal function is proposed here. This method uses a look-up table to obtain an initial approximation to the reciprocal. After the initial approximation, several CORDIC rotations are executed to produce an accurate reciprocal.

The reciprocal look-up table is constructed by assuming that the input value x is normalized such that $1 \leq x < 2$ and represented by j bits in fixed point. These j bits are used to index the LUT, which provides j output bits. They are taken as the fraction part of the reciprocal approximation. Such a LUT is termed as a j -bit-in j -bit-out reciprocal table of size $2^{j-1} \cdot j$ bits.

From Equations 4.4-4.6, with $z_0 = 0$ and $y_0 = 1$, Equations 4.12-4.14 can be obtained:

$$x_{i+1} = x_i \quad (4.12)$$

$$y_{i+1} = y_i + \delta_i \cdot 2^{-i} \cdot x_i \quad (4.13)$$

$$z_{i+1} = -\delta_i \cdot 2^{-i} \quad (4.14)$$

As described in Section 4.1, z will approach to the reciprocal of x closely through iterations. It can be seen that in the i -th rotation, a value 2^{-i} will be added to or subtracted from z . Since the initial value of z is 0, z can achieve the result of the

Chapter 4**Hybrid CORDIC for Reciprocation**

reciprocal with i -bit width fraction after i iterations. So the approximation l from the look-up table equals to the z value after $(j-1)$ rotations, as given in Equation 4.15.

$$l = z_j = -\delta_i \cdot 2^{-j+1} \quad (4.15)$$

Thus Equation 4.13 becomes:

$$y_j = y_0 - l \cdot x \quad (4.16)$$

As a result, the corresponding value of y after $j-1$ iterations can be obtained from the approximation of LUT. Starting with the y_j , z_j and x (constant), the accurate reciprocal will be approached by followed iterative rotations. It can also be seen as the first $j-1$ rotations are replaced by one step. The most complicated operation involved here is the calculation of y_j , which involves a multiplication between l and x and a subtraction of this quantity from the initial value y_0 . Any input data out of the range $1 \leq x < 2$ can be shifted into the range. After the reciprocal of the normalized input is obtained, the reciprocal of the original input can be achieved by shifting in the same way. For example, the input value is 48; in order to make it into the range the data is shifted right by 5 bits, then $x=1.5$. Note that $1/48$ is $1/2^5$ of $1/1.5$, so after $1/x$ is obtained, the result should be shifted to right by 5 bits as well.

4.2.1 Selection of the Entry and Bit-width of Look-up Table

With the entry and bit-width of look-up table varies, the size of LUT changes correspondently. Table 4.1 shows the entry and bit-width of the look-up table for a

Chapter 4 **Hybrid CORDIC for Reciprocation**

16-bit width input. p is the bit width of the input number to index the LUT and q is bit-width of the value stored in the table. The entry number of the look-up table is 2^{p-1} . From the LUT the first p bits of the reciprocal can be obtained. Then through $w - p$ rotations the w -bit width reciprocal is achieved.

p	q	Entry of LUT	ROM size (bit)	Rotations followed
6	6	64	384	10
7	7	128	896	9
8	8	256	2048	8
9	9	512	4608	7
10	10	1024	10240	6
11	11	2048	22528	5
12	12	4096	49152	4

Table 4.1: The entry and bit-width of look-up table

The information in Table 4.1 offers the potential for optimizing the reciprocal function generator by suitably balancing LUT size against the number of CORDIC stages. The different entry and bit-width of LUT can be selected according to the need of the applications, hence the tradeoff between area and speed can be decided for the most appropriate situation. Each group of p and q represents a different reciprocal functional resource in custom design.

4.2.2 RTL Model of Reciprocation

A RTL model is presented in this section. It is based on the Hybrid CORDIC model proposed in Chapter 3. The model also implements a pipelined architecture and operates on fixed-point values, except the modifications described below.

The fixed-point format based on a 16-bit word is adopted in this model again. Different from the first model, only one type of data format is used and both the input and output use the same format. One bit is for the sign, one bit is for the integer portion, and the other 14 bits are for the fractional portion. The format is shown in Figure 4.2. Because the input data x is normalized to $1 \leq x < 2$ before input to the model and the reciprocal of x is always between 0 and 1, only one bit is required to represent the integer part. Therefore, this format can provide full use of the bits.

Figure 4.2: Fixed-point format for the reciprocal model

In this model a LUT with 12-bit input and 12-bit output is selected. Since one bit is occupied by the sign and the integer bit is always 1, the actual entry number is 2^{12-1-1} , i.e. 1024. Thus the LUT has a size of $2^{10} \times 12 = 12288$ bits.

In the model described in Chapter 3, only adders and shifters are needed. While this reciprocal model employs a multiplier to compute the y_j value after pseudo $j-1$

Chapter 4

Hybrid CORDIC for Reciprocation

rotations. Either a 8×8 multiplier or a 16×16 multiplier can be used. When a 8×8 multiplier is employed, the approximation l from the LUT and x both need to be truncated to 8 bits, which causes much data loss and leads to inaccurate result. For a 16×16 multiplier, the input data has no truncation and only the output is required to be truncated to 16 bits. So the truncation error is much less than that in a 8×8 multiplier. Moreover, the replacement of the 8×8 multiplier with the 16×16 multiplier does not increase the area much. Table 4.2 shows the comparison of area between the two models employing these 2 types of multiplier.

Multiplier used	Resource used	
	Logic cells	Memory bits
8×8	2291	2048
16×16	2777	2048

Table 4.2: Synthesized results of reciprocal model with different multipliers

4.2.3 Simulation Results

To evaluate the performance of the reciprocal model based on the hybrid CORDIC, the simulations of the VHDL model were carried out with Modelsim. With $y=1$ and $z=0$, a group of 100 values of x which are randomly generated are input to the model. From Table 4.3, it can be seen that the reciprocals generated by the hybrid CORDIC are not as accurate as the conventional CORDIC when compared to the C functions. However, the results are still very close to the values produced by the C function, as shown in Figure 4.3.

	Conventional CORDIC	Hybrid CORDIC
	$1/x$	$1/x$
mean error	0.00825%	0.06898%
mean square error	0.00000068%	0.0000669%

Table 4.3: Errors compared to c function

To compare the area and speed between the conventional CORDIC and the hybrid CORDIC, both of them are synthesized with the LeonardoSpectrum tool. The area costed by the hybrid CORDIC is comparable with that of the conventional CORDIC, choosing the Altera’s FPGA EP1SGX10CF672C [35] as the configured device. An embedded multiplier of this FPGA device is employed in the hybrid CORDIC. In timing comparison, for a 14-bit width fraction part, the conventional CORDIC executes 14 rotations, so 14 clock cycles are needed. The time of fetching the value from the LUT costs 2 clock cycles and the operation of multiplication needs another 2 clock cycles. So for the hybrid CORDIC, with the followed 4 rotations, altogether 8 clock cycles are needed. With comparable logic cells, the speed is improved by 20%. Table 4.4 shows the area and timing results of the conventional CORDIC and the hybrid CORDIC. Therefore the hybrid CORDIC achieves shorter latency without much area increase by making use of an embedded multiplier in FPGA device.

Chapter 4

Hybrid CORDIC for Reciprocation

Figure 4.3: the reciprocal function values of CORDIC

	Used logic cells	Embedded Multiplier Employed	Clock (cycles)	Clock Frequency (MHz)	Process Time (ns)	Latency reduced
Conventional CORDIC	918	0	14	156.6	894	0
Hybrid CORDIC	997	1	8	156.6	510.9	42.9%

Table 4.4: Synthesized results of CORDIC in reciprocal function

Chapter 5

Improved Mitchell's Logarithm Approximation

The logarithm is a useful arithmetic function in various areas of science and engineering, for example the decibel scale for the loudness of sound. In many applications, particularly Digital Signal Processing (DSP), it is often necessary to apply the logarithm function to a data stream. With DSP becoming more and more widely used in electronic systems, there have been increasing demand for VLSI hardware implementations of multipliers and dividers that are faster and more area efficient. The logarithm has been long used in mathematics to simplify arithmetic computations. By using logarithms, multiplication and division can be simplified to addition and subtraction. Exponential functions can also be reduced to multiplication and division operations. Therefore a fast and simple hardware module for the logarithm computation is desired.

Muller [36] presents hardware-oriented as well as software-oriented algorithms to compute elementary functions and discusses issues related to the accurate floating-point implementation of these functions. These algorithms include the CORDIC algorithm, Newton-Raphson iteration, rational approximations, digit-by-digit algorithms, and polynomial approximations. Two of these hardware-oriented algorithms are presented by W. F. Wong [37] and M. J. Schulte [38] and these are

Chapter 5**Improved Mitchell Logarithm Approximation**

based on look-up tables and polynomial approximation of functions, respectively. Hardware implementations that provide results of accuracy within 0.5 unit of last place (ulp) for functions, such as division, logarithm, reciprocal square root, arc tangent, sine, and cosine are described in [37]. Fast speed and accurate results are achieved by using rectangular multiplier and lookup table. Hardware designs that generate an exact rounded result for the functions of reciprocal, square root, 2^x , and $\log_2 x$ are presented in [38]. By adjusting the coefficients based on the error in the original approximation, exactly rounded results are obtained. Also Yi Wan et al. [39] adopt a look-up table to compute the logarithm, and propose a factorization approach to reduce the look-up table size and a nonlinear approximation method to reduce the computational complexity.

Although these methods for computing the logarithm can derive very accurate results, the techniques are usually expensive in terms of time and/or hardware (ROM and functional units). For some applications, a trade-off can be made by sacrificing the accuracy of the conversion in order to achieve speed and to reduce power and area. This is less of a problem for DSP systems as they can often deal with the extra noise introduced. The work here focuses on the hardware implementation and optimisation of the logarithm developed by Mitchell [2].

5.1 Introduction of Mitchell's Method

The Mitchell's method is described as follows.

A binary number N , in the interval $2^j \leq N < 2^{k+1}$ (where $j=0, \pm 1, \pm 2, \dots$; $k=0, \pm 1, \pm 2, \dots$; and $k \geq j$) can be written as

$$N = \sum_{i=j}^k 2^i z_i, \text{ where } z_i = '0' \text{ or } '1'. \quad (5.1)$$

Assuming that z_k is a '1', since it is the MSB, N can be represented by

$$N = 2^k + \sum_{i=j}^{k-1} 2^i z_i \quad (5.2)$$

By taking out the factor 2^k , N becomes

$$N = 2^k (1 + \sum_{i=j}^{k-1} 2^{i-k} z_i) \quad (5.3)$$

By defining $m = \sum_{i=j}^{k-1} 2^{i-k} z_i$, Equation 5.3 can be rewritten as:

$$N = 2^k (1 + m) \quad (5.4)$$

Applying $\log_2$ to Equation 5.4, Equation 5.5 is obtained.

$$\log_2 N = k + \log_2(1 + m) \quad (5.5)$$

It is obvious that m consists of all the bits to the right of the leading one bit in the binary number N . And m can be seen as the binary fraction or the mantissa of this binary representation. In Mitchell's method, the actual $\log_2 N$ is approximated as $(\log_2 N)'$ by replacing $\log_2(1 + m)$ with m . So Mitchell's approximation for $\log_2 N$ is:

Chapter 5**Improved Mitchell Logarithm Approximation**

$$(\log_2 N)' = k + m \quad (5.6)$$

The error is the difference between the approximation and the exact value, i.e.

$$Error = \log_2 N - (\log_2 N)' = \log_2(1 + m) - m \quad (5.7)$$

As an example, consider the integer 13, which has a binary representation of “1101”. It is observed that the most significant ‘one’ occurs in the 2^3 bit position. Thus k , which is called the characteristic of the logarithm, is equal to 3, or “0011” in binary. The mantissa “0.101” consists of all the bits to the right of the leading-one. So the fraction of the logarithm approximation is 0.625. Therefore the approximated value of $\log_2 13$ is equal to 3.625 which is “0011.1010” in binary, where the exact log value is equal to 3.7004. Figure 5.1 illustrates the actual value and the Mitchell’s approximation of $\log_2 N$ for $8 \leq N < 16$.

Mitchell’s method finds the approximate value of the $\log_2$ of a binary number based on a straight-line approximation to the logarithm function for intervals between the powers of two. It is easy to implement the approximation function in hardware but this straight-line-approximation method suffers from inaccuracy. The worst-case percentage error $(= \frac{Error}{\log_2 N} \times 100\%)$ occurs at $N = 3$ (between 2^1 and 2^2) with the percentage error equal to 5.36%. Figure 5.2 shows the percentage error of the Mitchell’s approximation of $\log_2 N$, for $0 \leq N < 2^{16}$.

Chapter 5 Improved Mitchell Logarithm Approximation

Figure 5.1: Comparison of $\log_2 N$ and Mitchell's approximation

Figure 5.2: The percentage error of Mitchell's approximation of $\log_2 N$

Chapter 5**Improved Mitchell Logarithm Approximation**

The limitation of the accuracy is a major deterrence for using the Mitchell's approximation. Several improvements [40-42] were proposed to make the approximation more accurate.

Combet [40] divided the power-of-two intervals into four regions and applied straight-line approximations in each region. The coefficients of each line were selected by a trial and error method to reduce the error in the resulting logarithm approximation. Hall [41] minimized the mean square error also by using four subintervals and selecting the optimal straight-line approximations. These techniques produce more accurate results, but they are difficult to implement in hardware. SanGregory [42] divided the interval into two in order to obtain a better result compared to Mitchell. This method is based on dividing the interval into two regions of different slopes that increased and decreased in the lower and upper regions, respectively. The slope adjustment leads to the intersection of the midpoint of each line segment, with the actual logarithm function forcing the error to be zero at these two midpoints, besides the zero error at the beginning and end of the interval. This approximation reduces the worst error to 1.54% and employs less hardware cost compared with Combet's [40] and Hall's [41] method. The percentage error of this technique ranges over $-1.5403\% \leq error < 0.4314\%$. However, its results are not sufficiently accurate to meet the requirements of some applications.

The objective of the work is to develop binary logarithm functions that can be implemented with small and fast circuits with a very small error. An improved algorithm based on Mitchell's method was developed to increase the accuracy of Mitchell's approximation algorithm without adding significant hardware in

Chapter 5 **Improved Mitchell Logarithm Approximation**

implementation. In order to minimize the hardware used for the error correction, the approach taken is based on linear approximations to $\log_2 N$, as described in the next section.

5.2 The Proposed Improved Method

From the error distribution of the Mitchell's approximation (Figure 5.2), it can be observed that the maximum error occurs at $N=3$ and all the significant errors occur when the input values are less than 1000. When N is larger than 256, the maximum error of the Mitchell's approximation is less than 1.01%. So if the error of the approximation for $0 < N < 256$ can be limited to around 1.01%, the maximum error of the approximation for all the values can be decreased. From Figure 5.1, it is noticed that the Mitchell's approximation of $\log_2 N$ is always smaller than the exact value. In fact, the characteristic of the approximation logarithm, i.e. k , is completely accurate, while the fraction part is always less than the one of the exact logarithm. Once the more accurate fraction approximation can be found, the overall accuracy of the logarithm function will be improved. Since the value of approximation for fraction part in Mitchell's method becomes greater when the input value grows larger, the fraction approximation of N' (assuming that N' is an input value greater than N) is larger than the fraction approximation of N and thus closer to the fraction of exact logarithm value. Therefore, if the original approximation $(\log_2 N)'$ is replaced with the sum of the characteristic of $(\log_2 N)'$ and the fraction of $(\log_2 N')$ to approximate the exact value of $\log_2 N$, then the new approximation will be more accurate. Take $N=3$ as an example, i.e. $N="00000011"$ in binary representation, its Mitchell's approximation is "0001.1000", which is 1.5, less than the exact value of $\log_2 3 = 1.585$. The characteristic, $k="0001"$, matches the one of the exact logarithm value perfectly. If the fraction "0.1000" is replaced with "0.1001", or 0.5625 in decimal, the value is more accurate than 0.5. In order to obtain the fraction "0.1001",

Chapter 5**Improved Mitchell Logarithm Approximation**

the input value “00000011” is changed to “00011001”, namely $N' = “00011001”$, by being multiplied with 2^3 and adding 1, which can be easily implemented by shifting and addition operations in hardware. Therefore the more accurate fraction approximation can be achieved by applying a simple shift-and-add operation to the input values.

Further analysis shows that for $N=3$, the above approximation of “0001.1001” is more accurate than the Mitchell’s approximation, but its error is 1.42% and still much greater than 1.01%. If the input “00000011” is multiplied by 2^5 and added with 5, the result is “01100101”, of which the fraction part of the Mitchell’s logarithm approximation is “0.100101”. Here N' is “01100101”. So the new approximation is “0001.100101”, or 0.578125 in decimal, and much closer to 0.585, the fraction part of the exact value of $\log_2 3$. The error is 0.4314%, which is much less than 1.01%.

In this work, the input value range ($0 < N < 65536$) is divided into 16 sections. For each section, the characteristic of the logarithm of input values is the same. Noticing that the maximum error of first 8 sections (where N is less than 256) in Figure 5.2 is much greater than 1.01%, a correction function of the fraction for each section has to be found in order to improve the accuracy. The following guidelines are used when searching for the correction equations, where $error_{max}$ and $error_{min}$ are the maximum and the minimum error of each section respectively.

1. To make $error_{max}$ of each section as less than 1.01% as possible.
2. To make $|error_{min}|$ as small as possible.
3. To minimize the absolute value of $(error_{max} - error_{min})$ in each section.

Chapter 5**Improved Mitchell Logarithm Approximation**

The manual design technique, which is used to search for the correction functions for the first 8 sections, is described as followed.

When $k = 0$, there is no error for the approximation. So no correction is needed.

When $k = 1$, only $N=2$ and $N=3$ are in this section. There is no error in Mitchell's approximation for $N=2$, so only $N=3$ needs to be considered for correction. Noticing that the exact value of $\log_2 3$ is "0001.100101011100" while the Mitchell's approximation is "0001.100000000000", the significant error exists in the fourth and the sixth bit position of the fraction. In order to make the approximation as "0001.100101000000" instead of "0001.100000000000", adjustments have to be done to the fourth and the sixth bit position of the fraction. Then by multiplying "00000000000000011" (i.e. 3 in decimal) with 32 and adding 5 to it, "0000.100101000000", the fraction part of the result "0000000001100101", is more accurate than the fraction part of Mitchell's approximation "0000.100000000000". So the correction function for fraction is $N' = N \cdot 2^5 + 5$. The maximum and the minimum error of this section are shown below:

$$error_{\max} = 0.4314\%$$

$$error_{\min} = 0$$

$$|error_{\max} - error_{\min}| = 0.4314\%$$

When $k = 2$, since the maximum error in Mitchell's algorithm exists at the midpoint between powers of two for input values, which can be seen from Figure 5.2 obviously, the Mitchell's approximation for $N=6$ has the maximum error in this section. To

Chapter 5**Improved Mitchell Logarithm Approximation**

minimize the maximum error in this section, the fourth and the sixth bit of the fraction for the Mitchell's approximation "0010.100000000000" need to be adjusted. Being multiplied with 16 and added 5, the more accurate fraction part "0000.100101000000" of the result "0000000001100101" is obtained. The error of approximation for $N=6$ decreases from 3.2882% to 0.2645%. So the correction function should be $N' = N \cdot 2^4 + 5$. But when this correction function is applied to all the input values of this section other than $N=4$, of which the approximation has not error, the minimum error of this section is -0.7398% occurring at $N=7$. The absolute value of the minimum error is quite large, which does not fulfil the guideline of the searching. Because the maximum error of the input values for $N > 256$ is 1.01% at least, the $error_{\max}$ of this section can be sacrificed in certain range to reduce the $|error_{\min}|$. That is to say the correction function should be adjusted so that the $|error_{\min}|$ can be minimized as possible while the $error_{\max}$ only need to be keep around 1.01%. The reason that $|error_{\min}|$ became so large is that the correction function gives too much increment to the approximation. Therefore the augment, which the correction function added to the approximation, need to reduced. Under such rules, the correction function is adjusted and corresponding $error_{\max}$ and $error_{\min}$ are worked out. After iterative refinement, the correction function for this section, $N' = N \cdot 2^2 + 1$, is selected. The maximum and the minimum error of this section are presented:

$$error_{\max} = 0.8690\%, \quad error_{\min} = -0.1833\%, \quad |error_{\max} - error_{\min}| = 1.0523\%.$$

Chapter 5**Improved Mitchell Logarithm Approximation**

With the same searching technique, the correction functions for the other 6 sections are also derived. The following is the correction functions and corresponding $error_{\max}$ and $error_{\min}$.

When $k = 3$, the correction function for fraction is $N' = N \cdot 2^3 + 3$.

$$error_{\max} = 1.0856\%, \quad error_{\min} = -0.3835\%, \quad |error_{\max} - error_{\min}| = 1.4691\%.$$

When $k = 4$, the correction function for fraction is $N' = N \cdot 2^3 + 5$.

$$error_{\max} = 1.0390\%, \quad error_{\min} = -0.4515\%, \quad |error_{\max} - error_{\min}| = 1.4905\%.$$

When $k = 5$, the correction function for fraction is $N' = N + 1$.

$$error_{\max} = 0.9923\%, \quad error_{\min} = -0.3801\%, \quad |error_{\max} - error_{\min}| = 1.3724\%.$$

When $k = 6$, the correction function for fraction is $N' = N + 1$.

$$error_{\max} = 1.0802\%, \quad error_{\min} = -0.1619\%, \quad |error_{\max} - error_{\min}| = 1.2421\%.$$

When $k = 7$, the correction function for fraction is $N' = N + 1$.

$$error_{\max} = 1.0405\%, \quad error_{\min} = -0.1116\%, \quad |error_{\max} - error_{\min}| = 1.1521\%.$$

In several sections such as $k = 3$ and $k = 4$, if the $error_{\max}$ is reduced to less than 1.01%, the $|error_{\min}|$ would become very large and $|error_{\max} - error_{\min}|$ would be large too, which cannot meet the requirement. All the 3 errors should be kept as small as possible. Thus these correction functions are chosen in such a way that the $error_{\max}$ is allowed to be slightly greater than 1.01% when the $|error_{\min}|$ and $|error_{\max} - error_{\min}|$ can be kept small.

Chapter 5**Improved Mitchell Logarithm Approximation**

Therefore, with the integer part of the original binary input N and the fraction part of the N' after correction, the obtained logarithm approximation is much more accurate than the Mitchell's method. For $N < 256$, $error_{\max} = 1.0856\%$, $error_{\min} = -0.4515\%$ and $|error_{\max} - error_{\min}| = 1.5371\%$, compared to $error_{\max} = 5.36\%$, $error_{\min} = 0$ and $|error_{\max} - error_{\min}| = 5.36\%$ from the Mitchell's method.

While improving the accuracy, the correction equations are also modified in order to minimize the hardware cost for implementing them.

When implementing these correction functions to hardware, it is found that the correction function for fraction of each section can be implemented by adding a constant to the fraction part of this section. Taking $N = "0000000000001011"$ as example, $k = 3$ and the Mitchell's approximation is $"0011.011000000000"$. After the correction for the fraction part of the approximation with $N' = N \cdot 2^3 + 3$, the improved fraction, namely the fraction of $N' = "0000000001011011"$, is $"0000.011011000000"$. Observing the fractions before and after correction, it is obvious that only a constant $"0000.000011000000"$ is added to the original fraction. It also can be regarded as the fifth and the sixth bit of the fraction of Mitchell's approximation change from '0' to '1'. Therefore the constants $"0000.000101000000"$, $"0000.000100000000"$, $"0000.000011000000"$, $"0000.000010100000"$, $"0000.000010000000"$, $"0000.000001000000"$ and $"0000.000000100000"$ are added to the fraction of the Mitchell's approximation for $k = 1, 2, 3, 4, 5, 6, 7$ respectively.

Chapter 5**Improved Mitchell Logarithm Approximation**

In order that the correction circuits can share the implemented hardware as much as possible, the constants for $k = 6, 7$ are changed to the same one as $k = 5$, “0000.000010000000”. That is to say the corrections for $k = 5, 6, 7$ can all be implemented only by adjusting the fifth bit of the fraction of Mitchell’s approximation from ‘0’ to ‘1’. The correction functions for $k = 6, 7$ are modified consequently and shown below.

When $k = 6$, the correction function for fraction changes to $N' = N + 2$.

$$error_{\max} = 0.8402\%, \quad error_{\min} = -0.4069\%, \quad |error_{\max} - error_{\min}| = 1.2471\%.$$

When $k = 7$, the correction function for fraction is $N' = N + 4$.

$$error_{\max} = 0.7286\%, \quad error_{\min} = -0.4464\%, \quad |error_{\max} - error_{\min}| = 1.1750\%.$$

For $k = 5, 6, 7$, after modification they share the same correction circuits, thus the area of implemented hardware can be reduced. Although the $|error_{\min}|$ and $|error_{\max} - error_{\min}|$ for modified correction functions increased for $k = 6, 7$ compared to original ones, the $|error_{\min}|$ and $|error_{\max} - error_{\min}|$ after optimisation are also less than the overall $|error_{\min}|$, 0.4515%, and $|error_{\max} - error_{\min}|$, 1.5371%, for $N < 256$, which does not affect the maximum and the minimum errors of all 16 sections. Table 5.1 summarises the final correction functions, the corresponding errors and the constants to be added to the fraction for each section when $N < 256$.

Chapter 5 **Improved Mitchell Logarithm Approximation**

k	Correction function	$error_{\max}$	$error_{\min}$	$ error_{\max} - error_{\min} $	Constant to be added (in binary)
0	0	0	0	0	0
1	$N' = N \cdot 2^5 + 5$	0.4314%	0	0.4314%	0000.000101000000
2	$N' = N \cdot 2^2 + 1$	0.8690%	-0.1833%	1.0523%	0000.000100000000
3	$N' = N \cdot 2^3 + 3$	1.0856%	-0.3835%	1.4691%	0000.000011000000
4	$N' = N \cdot 2^3 + 5$	1.0390%	-0.4515%	1.4905%	0000.000010100000
5	$N' = N + 1$	0.9923%	-0.3801%	1.3724%	0000.000010000000
6	$N' = N + 2$	0.8402%	-0.4069%	1.2471%	0000.000010000000
7	$N' = N + 4$	0.7286%	-0.4464%	1.1750%	0000.000010000000

Table 5.1: The correction function, corresponding errors and the constant to be added

5.2.1 Accuracy Comparison

The proposed improved approximation is shown in Figure 5.3. The overall $error_{\max}$ of the proposed method is 1.0856%, the $error_{\min}$ is -0.4515% and the maximum absolute value of $(error_{\max} - error_{\min})$ is 1.5371%, as shown in Figure 5.4. Figure 5.5 illustrates the error distribution of the proposed improved approximation and the original Mitchell's approximation.

Chapter 5 **Improved Mitchell Logarithm Approximation**

Figure 5.3: Comparison of SanGregory's, the proposed improved and the Mitchell's approximation

Figure 5.4: The percentage error of the proposed improved method

Chapter 5

Improved Mitchell Logarithm Approximation

Figure 5.5: Error distribution of SanGregory’s, the proposed improved and the Mitchell’s approximation

The maximum percentage error of the proposed improved method 1.0856% is less than Mitchell’s 5.3605%, but greater than SanGregory’s 0.4314%. However, the absolute value of the minimum error 0.4515% is much less than SanGregory’s 1.5403% [42]. So the percentage range of the proposed improvement is 1.5371%, which is less than SanGregory’s 1.9717%, as shown in Table 5.2.

	Mitchell [2]	SanGregory [42]	Proposed Improved
Max. -ve percentage error	0	-1.5403	-0.4515
Max. +ve percentage error	5.3605	0.4314	1.0856
Percentage error range	5.3605	1.9717	1.5371

Table 5.2: Comparison of percentage error

5.2.2 Hardware Implementation

The 16-bit binary-to-binary logarithm converters based on Mitchell’s method, SanGregory’s method and the proposed improved method were all implemented. These logarithm converters were coded in VHDL and synthesized into implementation on FPGA.

The fixed-point format based on a 16-bit word is adopted in the implementation. The input values are represented by 16-bit binary integer. While for the output result, since the binary logarithm of a 16-bit binary integer is less than 16, 4 bits are used to represent the characteristic of the result logarithm, and the other 12 bits are to represent the fraction part of the binary logarithm, as shown in Figure 5.6.

Figure 5. 6: Fixed-point format for the output result

The VHDL modules of the 3 logarithm converters are all synthesized with Leonardo Spectrum tool. Using the Altera’s FPGA EP20K200EFC484 as the configured device, from the summary of the synthesis, it can be seen that the IOs and memory bits used by the 3 logarithm converters are the same. The SanGregory’s method and the proposed improved method sacrifice more logic cells to achieve the elegant accuracy. While achieving more accurate results, the Proposed Improved method costs less (127-123= 4) logic cells of the FPGA than the SanGregory’s method, as shown in

Chapter 5 **Improved Mitchell Logarithm Approximation**

Table 5.3. Therefore besides the elegant accuracy, the proposed improved method is more efficient in hardware implementation compared with SanGregory’s method. Although SanGregory’s method has the fastest clock frequency, it also takes 5 clock cycles, the most of the 3 methods, to complete a computation. The proposed improved method costs 3 clock cycles to computer input values for $N < 256$ and need only 2 clock cycles, same as the Mitchell’s method, for other input values. One more clock cycle is needed for $N < 256$ in order to judge whether the fraction part of the Mitchell’s approximation is 0. If the condition is true, the logarithm of N is an integer, i.e. k . Then no correction is needed for the input value. If the fraction part of the Mitchell’s approximation is not equal to 0, the corresponding constant is added to the fraction part. For $N \geq 256$, no correction is required and hence, only 2 clock cycles are needed.

	Used IOs	Used Logic cells	Used Memory bits	Hardware cost comparison	Clock (cycles)	Clock Frequency (MHz)
Mitchell’s method	33	88	0	100%	2	65.4
SanGregory’s method	33	127	0	144.3%	5	69.3
Proposed improved method	33	123	0	139.8%	3 ($N < 256$) 2 ($N \geq 256$)	61.7

Table 5.3: The synthesized results of the 3 methods

This proposed improved logarithm module costs 123 logic cells, only 1.48% of all 8320 logic cells of the FPGA device EP20K200EFC484, which contains approximately 200,000 logic gates. Nowadays FPGA devices with million logic gates are common. The hardware cost of the proposed module is not significant for the resource of a FPGA device and is suitable for being used in FPGA-based designs.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

A hybrid scheme of combining ROM-based look-up tables and hardware implementations of the iterative CORDIC algorithm is proposed and implemented. This scheme combines the respective advantages of fast access and power efficiency of reasonably sized LUTs and the arbitrary precision obtainable from a rigorous iteration algorithm. When the proportion between the resolution of LUTs and the CORDIC rotations is varied, many types of function modules with various area and speed can be established to satisfy different application requirements in terms of computational load and memory requirements. The hybrid CORDIC is further extended to implement the sine, cosine, sinh, cosh and reciprocal functions. The computation speed of the hybrid CORDIC is improved further by computing the directions of all the followed rotations in advance and executing in parallel. Both area and speed improvement are achieved in the sin/cos function modules of this improved hybrid CORDIC when the entry of LUT is properly selected. When replacing the first 7 rotation stages with a LUT, the delay is reduced by 49.61% and the hardware cost is 18.75% lower, compared with the Wang et al.'s method [33]. A tradeoff of area for speed is obtained for the sine/cosine, sinh/cosh and reciprocal functions. When a

small LUT is employed to replace a few stages, the speed will be improved slightly without much area increases. While a larger LUT replaces more stages, the speed will be accelerated greatly with the increasing cost of memory. The proposed hybrid CORDIC is also very suitable for implementation in FPGA devices, which provides millions of gates, large memory and hardwired multipliers.

Also presented in the thesis is a method to improve the accuracy of the Mitchell's algorithm, which is an approximation method for binary-to-binary logarithm. The error distribution of the Mitchell's algorithm was investigated. It was found that the most significant errors occurred when the input values were less than 2^8 . Hence, the input values of this range are divided into 8 regions according to their MSB. Several correction functions for each region are tried out to make the approximations closer to the exact logarithm values. The better accuracy is achieved without costing much more hardware in implementation. The proposed improved method uses only combinational logic to reduce the maximum percentage error to 1.0856% and the minimum percentage error to -0.4515%. Thus the percentage error range is limited to 1.5371%, far less than the Mitchell's 5.3605%. The minimum percentage error is also better than the one from SanGregory [42], which is -1.5403%. Although the maximum percentage error is not as good as the one from SanGregory, which is 0.4314%, the percentage error range is better than SanGregory's 1.9717%, improved by 22.04%.

6.2 Recommendations for Further Research

In addition to the work done and presented here, there are still further improvements and development that can be carried out for the designs of arithmetic modules. Some recommendations for further research are suggested below.

In this work, the hybrid CORDIC modules for sin/cosine, sinh/cosh and reciprocal functions have been designed. Since CORDIC is widely used in communication, DSP and other VLSI systems, these efficient modules can be employed to realise these applications to achieve better overall performance. Different applications have different hardware resource and speed requirements. It would be useful and interesting to study how the hybrid CORDIC with different LUT sizes and speed can be employed in different applications. Based on this study, a generalized RTL model of the hybrid CORDIC can be developed. The generalized model can then be used as a parameterized library module, which would be reusable in different designs depending on the requirements. Furthermore, the parameterized model can be used for synthesis of VLSI systems, where complex trigonometric functions are required. This will extend the capability of VLSI Synthesis Systems, which currently use only simple arithmetic functions.

Logarithms have long been used as a tool in mathematics to simplify the process of multiplication, division, roots, powers, etc. They reduce multiplication and division problems to addition and subtraction. Power and root calculations are reduced to multiplication and division. Since a simple hardware implementation binary-to-binary

Chapter 6

Conclusion and Recommendations

logarithm converter with good accuracy can be designed, the development of a simple approach to convert the binary logarithm to binary is worth exploring. Once this anti-logarithm converter is built, a module combining the logarithm and anti-logarithm converters can be established. Multiplication and division can be performed by addition and subtraction with this module, which will not have high hardware cost. Therefore, it is suggested to further the work to extend the proposed logarithm converter to a logarithm number system [43-45] where the multiplication, division, exponential and root functions can be computed efficiently. And the logarithm modules can also be employed in VLSI synthesis systems.

Bibliography

Bibliography

- [1] J. Volder, "The CORDIC Trigonometric Computing Technique," IRE Transactions on Electronic Computing, Volume: EC-8, pp.330-334, Sept. 1959.

- [2] J.N. Mitchell , "Computer Multiplication and Division Using Binary Logarithms," IRE Transactions on Electronic Computers, Volume: 11, pp.512-517, Aug. 1962.

- [3] J.S. Walther, "A unified algorithm for elementary functions," Spring Joint Computer Conference, proc., pp.379-385, 1971.

- [4] Sang Yoon Park, Nam Ik Cho, Sang Uk Lee, Kichul Kim and Jisung Oh, "Design of 2K/4K/8K-point FFT processor based on CORDIC algorithm in OFDM receiver," Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on, Volume: 2, 26-28 Aug. 2001.

- [5] S. W. Mondwurf, "Versatile COFDM demodulator design based on the CORDIC-algorithm," Consumer Electronics, IEEE Transactions on, Volume: 48, Issue: 3, Aug. 2002.

Bibliography

- [6] Jen-Chih Kuo, Ching-Hua Wen and An-Yeu Wu, "Implementation of a programmable 64/spl sim/2048-point FFT/IFFT processor for OFDM-based communication systems," Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on, Volume: 2, 25-28 May 2003.
- [7] Shen-Fu Hsiao, "A high-speed constant-factor redundant CORDIC processor without extra correcting or scaling iterations," Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on, Volume: 1, 30 May-2 June 1999.
- [8] Shaoyun Wang, V. Piuri, J. Swartzlander, "A unified view of CORDIC processor design," Circuits and Systems, 1996. IEEE 39th Midwest symposium on, Volume: 2, 18-21 Aug. 1996.
- [9] J. Kwak, V. Piuri and J. Swartzlander, "Fault-tolerant high-performance CORDIC processors," Defect and Fault Tolerance in VLSI Systems, 2000. Proceedings. IEEE International Symposium on, 25-27 Oct. 2000.
- [10] Tso Bing Juang and Shen Fu Hsiao, "Low power and fast CORDIC processor for vector rotation," Circuits and Systems, 1999. 42nd Midwest Symposium on, Volume: 1, 8-11 Aug. 1999.
- [11] B. Das, S. Banerjee, "Unified CORDIC-based chip to realize DFT/DHT/DCT/DST," Computers and Digital Techniques, IEE Proceedings, Volume: 149 Issue: 4, July 2002.

Bibliography

- [12] Sungwook Yu; J. Swartzlander, "A scaled DCT architecture with the CORDIC algorithm," *Signal Processing, IEEE Transactions on*, Volume: 50 Issue: 1, Jan. 2002.
- [13] Jue-Hsuan Hsiao, Liang-Gee Ghen, Tzi-Dar Chiueh and Chun-Te Chen, "High throughput CORDIC-based systolic array design for the discrete cosine transform," *Circuits and Systems for Video Technology, IEEE Transactions on*, Volume: 5, Issue: 3, June 1995.
- [14] Yu Hen Hu and H. E. Lian, "CALF: a CORDIC Adaptive Lattice Filter," *Signal Processing, IEEE Transactions on*, Volume: 40, 1992.
- [15] J. Ma, K. Parhi, E. F. Deprettere, "Pipelined CORDIC-based cascade orthogonal IIR digital filters," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, Volume: 47 Issue: 11, Nov. 2000.
- [16] S. Shiraishi, M. Haseyama, H. Kitajima, "A cost-effective and high-precision architecture for CORDIC-based adaptive lattice filters," *Circuits and Systems*, 2002. ISCAS 2002. *IEEE International Symposium on*, Volume: 5, 26-29 May 2002.
- [17] E. Grayver, B. Daneshrad, "Direct digital frequency synthesis using a modified CORDIC," *Circuits and Systems*, 1998. ISCAS '98. *Proceedings of the 1998 IEEE International Symposium on*, Volume: 5, 31 May-3 June 1998.

Bibliography

- [18] Chang Yong Kang, J. Swartzlander, "An analysis of the CORDIC algorithm for direct digital frequency synthesis," Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on, 17-19 July 2002.
- [19] F. Cardells-Tormo and J. Valls-Coquillat, "Optimisation of direct digital frequency synthesisers based on CORDIC," Electronics Letters, Volume: 37, Issue: 21, 11 Oct 2001.
- [20] N. Takagi, T. Asada, S. Yajima, "Redundant CORDIC methods with a constant scaling factor for Sine and Cosine computation," Computers, IEEE Transactions on, Volume: 40, Issue: 9, Sept. 1991.
- [21] D. Timmermann, H. Hahn, B. J. Hosticka, "Low latency time CORDIC algorithms," Computers, IEEE Transactions on, Volume: 41, Issue: 8, Aug. 1992.
- [22] H. Dawid, H. Meyr, "The differential CORDIC algorithm: constant scale factor redundant implementation without correcting iterations," Computers, IEEE Transactions on, Volume: 45, Issue: 3, Mar. 1996.
- [23] S. F. Hsiao and C. Y. Lau, "Design of a unified arithmetic processor based on redundant constant-factor CORDIC with merged scaling operation," Computers and Digital Techniques, IEE Proceedings-, Volume: 147, Issue: 4, July 2000.

Bibliography

- [24] E. Antelo, J. D. Bruguera, and E.L.Zapata, "Unified Mixed Radix-2-4 Redundant CORDIC Processor," *Computers, IEEE Transactions on*, Volume: 45, no.9, pp. 1,086-1,073, Sept. 1996.
- [25] J. A. Lee and T. Lang, "Constant-Factor Redundant CORDIC for Angle Calculation and Rotation," *Computers, IEEE Transactions on*, Volume: 41, no.8, pp. 1,016-1,025, Aug. 1992.
- [26] J. D. Bruguera, E. Antelo, and E. L. Zapata, "Design of a Pipelined Radix-4 CORDIC Processor," *J. Parallel Computing*, Volume: 19, no.7, pp. 729-744, 1993.
- [27] Elisardo Antelo, Julio Villalba, Javier D. Bruguera, and Emilio L. Zapata, "High Performance Rotation Architectures Based on the Radix-r CORDIC Algorithm," *Computers, IEEE Transactions on*, Volume: 46, no.8, Aug. 1997.
- [28] B. Gisuthan, T. Srikanthan, "Flat CORDIC: a unified architecture for high-speed generation of trigonometric and hyperbolic functions", *Circuits and Systems*, 2000. Proceedings of the 43rd IEEE Midwest Symposium, Volume: 3, pp. 1,414 –1,417, 2000.
- [29] B. Gisuthan, T. Srikanthan and K.V. Asari, "A high speed flat CORDIC based neuron with multi-level activation function for robust pattern recognition," *Computer Architectures for Machine Perception*, 2000. Proceedings. Fifth IEEE International Workshop on, 11-13 Sept. 2000

Bibliography

[30] Cheng-Shing Wu, An-Yeu Wu, "Modified vector rotational CORDIC (MVR-CORDIC) algorithm and architecture," Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, Volume: 48, Issue: 6, Jun 2001.

[31] http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/index.cfm

[32] [APEX 20K Programmable Logic Device Family Data Sheet](#)

[33] S. Wang, V. Piuri, and E. E. Swartzlander, "Hybrid CORDIC Algorithms," Computers, IEEE Transactions on, Volume: 46, pp. 1202-1207, Nov. 1997.

[34] M. J. Flynn and S. F. Oberman, "Advanced Computer Arithmetic Design," New York: Wiley, pp. 206, 2001.

[35] [Stratix GX FPGA Family Data Sheet](#)

[36] J. -M. Muller, "Elementary Functions: Algorithms and Implementation," Birkhauser, 1997.

[37] W.F. Wang and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," Computers, IEEE Transactions on, Volume: 43, no. 3, pp. 278-294, Mar. 1994.

Bibliography

- [38] M.J. Schulte and E.E. Swartzlander Jr., "Hardware Designs for Exactly Rounded Elementary Functions," *Computers, IEEE Transactions on*, Volume: 43, no. 8, pp. 964-973, Aug. 1994.
- [39] Y. Wan and C. L. Wey, "Efficient algorithms for binary logarithmic conversion and addition," *Computers and Digital Techniques, IEE Proceedings-*, Volume: 146, Issue: 3, May 1999.
- [40] M. Combet, H. Zonneveld, and L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers," *IEEE Transactions on Electronic Computers*, Volume: 14, pp. 863-867, Dec. 1965.
- [41] E. L. Hall, D.D. Lynch, and S.J. Dwyer III, "Generation of Products and Quotients Using Approximate Binary Logarithms for Digital Filtering Applications," *Computers, IEEE Transactions on*, Volume: 19, pp. 97-105, Feb. 1970.
- [42] S. L. SanGregory, R.E. Siferd, C. Brother, and D. Gallagher, "A Fast, Low-Power Logarithm Approximation with CMOS VLSI Implementation," *Proc. IEEE Midwest Symposium Circuits and Systems*, Aug. 1999.
- [43] E. E. Swartzlander and A. G. Alexopoulos, "The signed logarithm number system," *Computers, IEEE Transactions on*, Volume: C-24, pp.1238-12142, Dec. 1975.

Bibliography

[44] F. J. Taylor, R. Gill, J. Joseph and J. Radke, "A 20 bit logarithmic number system processor," Computers, IEEE Transactions on, Volume: 37, Issue: 2, Feb. 1988.

[45] J. N. Coleman, E. I. Chester, C. I. Softley and J. Kadlec, "Arithmetic on the European logarithmic microprocessor," Computers, IEEE Transactions on, Volume: 49, Issue: 7, July 2000.

Appendix A

Appendices

Appendix A

VHDL Model of Hybrid CORDIC for Trigonometric Functions

```

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY hybrid_highreso IS
PORT (
    Clk      : IN    std_logic;
    WriteState : IN    std_logic;
    a        : IN    std_logic_vector(15 downto 0);
    x        : IN    std_logic_vector(15 downto 0);
    y        : IN    std_logic_vector(15 downto 0);
    xout     : OUT   std_logic_vector(15 downto 0);
    yout     : OUT   std_logic_vector(15 downto 0)
);-- Declarations

END hybrid_highreso ;

-- renoir interface_end

ARCHITECTURE hybrid_highreso OF hybrid_highreso IS
constant      Int90      :      signed(15      downto      0)      :=
SIGNED(To_StdLogicVector(bit_vector'(x"2d00")));

signal a1, a2, a3, a4, a5, a6, a7 : signed(15 downto 0);
signal a8, a9: signed(15 downto 0);
signal x1, x2, x3, x4, x5, x6, x7 : signed(15 downto 0);
signal x8, x9 : signed(15 downto 0);
signal y1, y2, y3, y4, y5, y6, y7 : signed(15 downto 0);

```

Appendix A

```

signal y8, y9: signed(15 downto 0);
signal const_1:std_logic_vector(15 downto 0) ;
signal const_2:std_logic_vector(15 downto 0) ;

```

```

function SINE(n :signed(15 downto 0)) return signed is
    variable result :signed(15 downto 0);
begin
    case n is

```

```

        when "0000000000000000" => result := "0000000000000000";
        when "0000000010000000" => result := "0000000100011101";
        when "0000000100000000" => result := "0000001000111011";
        when "0000000110000000" => result := "0000001101011001";
        when "0000001000000000" => result := "0000010001110110";
        when "0000001010000000" => result := "0000010110010011";
        when "0000001100000000" => result := "0000011010110000";
        when "0000001110000000" => result := "0000011111001100";
        when "0000010000000000" => result := "0000100011101000";
        when "0000010010000000" => result := "0000101000000011";
        when "0000010100000000" => result := "0000101100011101";
        when "0000010110000000" => result := "0000110000110110";
        when "0000011000000000" => result := "0000110101001110";
        when "0000011010000000" => result := "0000111001100101";
        when "0000011100000000" => result := "0000111101111011";
        when "0000100000000000" => result := "0001000010010000";
        when "0000100010000000" => result := "0001000110100100";
        when "0000100100000000" => result := "0001001010110110";
        when "0000100110000000" => result := "0001001111000110";
        when "0000101000000000" => result := "0001010011010110";
        when "0000101010000000" => result := "0001010111100011";
        when "0000101010000000" => result := "0001011011101111";
        when "0000101100000000" => result := "0001011111111001";
        when "0000101110000000" => result := "0001100100000001";
        when "0000110000000000" => result := "0001101000000111";
        when "0000110010000000" => result := "0001101100001100";
        when "0000110100000000" => result := "0001110000001110";
        when "0000110110000000" => result := "0001110100001110";
        when "0000111000000000" => result := "0001111000001011";
        when "0000111010000000" => result := "0001111100000111";
        when "0000111100000000" => result := "0010000000000000";
        when "0000111110000000" => result := "0010000011110110";
        when "0001000000000000" => result := "0010000111101010";
        when "0001000010000000" => result := "0010001011011011";
        when "0001000100000000" => result := "0010001111001001";
        when "0001000110000000" => result := "0010010010110101";
        when "0001001000000000" => result := "0010010110011110";
        when "0001001010000000" => result := "0010011010000100";
        when "0001001100000000" => result := "0010011101100110";
        when "0001001110000000" => result := "0010100001000110";

```


Appendix A

```

when "0001010000000000" => result := "0010100100100011";
when "0001010010000000" => result := "0010100111111100";
when "0001010100000000" => result := "0010101011010011";
when "0001010110000000" => result := "0010101110100101";
when "0001011000000000" => result := "0010110001110101";
when "0001011010000000" => result := "0010110101000001";
when "0001011100000000" => result := "0010111000001001";
when "0001011110000000" => result := "0010111011001110";
when "0001100000000000" => result := "0010111110001111";
when "0001100010000000" => result := "0011000001001101";
when "0001100100000000" => result := "0011000100000110";
when "0001100110000000" => result := "0011000110111100";
when "0001101000000000" => result := "0011001001101110";
when "0001101010000000" => result := "0011001100011100";
when "0001101100000000" => result := "0011001111000110";
when "0001101110000000" => result := "0011010001101100";
when "0001110000000000" => result := "0011010100001110";
when "0001110010000000" => result := "0011010110101100";
when "0001110100000000" => result := "0011011001000110";
when "0001110110000000" => result := "0011011011011011";
when "0001111000000000" => result := "0011011101101100";
when "0001111010000000" => result := "00110111111111001";
when "0001111100000000" => result := "0011100010000010";
when "0001111110000000" => result := "0011100100000110";
when "0010000000000000" => result := "0011100110000101";
when "0010000010000000" => result := "0011101000000000";
when "0010000100000000" => result := "0011101001110111";
when "0010000110000000" => result := "0011101011101001";
when "0010001000000000" => result := "0011101101010110";
when "0010001010000000" => result := "0011101110111111";
when "0010001100000000" => result := "0011110000100011";
when "0010001110000000" => result := "0011110010000011";
when "0010010000000000" => result := "0011110011011110";
when "0010010010000000" => result := "0011110100110100";
when "0010010100000000" => result := "0011110110000101";
when "0010010110000000" => result := "0011110111010001";
when "0010011000000000" => result := "0011111000011001";
when "0010011010000000" => result := "0011111001011100";
when "0010011100000000" => result := "0011111010011001";
when "0010011110000000" => result := "0011111011010010";
when "0010100000000000" => result := "0011111100000111";
when "0010100010000000" => result := "0011111100110110";
when "0010100100000000" => result := "0011111110110000";
when "0010100110000000" => result := "0011111110000101";
when "0010101000000000" => result := "0011111110100110";
when "0010101010000000" => result := "0011111111000001";
when "0010101100000000" => result := "0011111111011000";
when "0010101110000000" => result := "0011111111101001";
when "0010110000000000" => result := "0011111111101110";

```

Appendix A

```

        when "0010110010000000" => result := "0011111111111101";
        when "0010110100000000" => result := "0100000000000000";
        when others => result := "0000000000000000";
    end case;
    return result;
end SINE;

begin

xout <= STD_LOGIC_VECTOR(x9);
yout <= STD_LOGIC_VECTOR(y9);

const_1<= "1111111110000000";
const_2<= "0000000001111111";

PROCESS(clk)
variable angle :std_logic_vector(15 downto 0) ;
variable ang :signed (15 downto 0);
variable r_ang :std_logic_vector(15 downto 0);

begin
angle := std_logic_vector(a2) and const_1;--"1111111110000000";
ang := signed(angle);
r_ang := std_logic_vector(a2) and const_2;--"0000000001111111";
if (clk'event and clk='1') then
    if (WriteState = '1') then
        if (a(15) = '1') then -- a<0
            a1 <= SIGNED(a) + SIGNED(To_StdLogicVector(bit_vector'(x"5a00")));
            if (a1 > Int90) then
                a2 <= a1 - Int90;
                x2<= SINE(ang); y2<=-SINE(Int90-ang);
            else
                a2<=a1;
                x2<=-SINE(Int90-ang); y2<=-SINE(ang);
            end if;
        else
            a1 <= SIGNED(a);
            if (a1 > Int90) then
                a2 <= a1 - Int90;
                x2<=-SINE(ang); y2<=SINE(Int90-ang);
            else
                a2 <= a1;
                x2<=sine(Int90-ang); y2<=sine(ang);
            end if;
        end if;

        if (r_ang="0000000000000000") then
            x9 <= x2; y9 <= y2;
        else

```


Appendix A

```

a3 <= signed(r_ang);
x3 <= x2; y3 <= y2;

if (a3(15) = '0') then -- a3>=0
  a4 <= a3 - SIGNED(To_StdLogicVector(bit_vector'(x"0073")));
  x4 <= x3 - SHR(y3,"110"); y4 <= y3 + SHR(x3,"110");
else
  a4 <= a3 + SIGNED(To_StdLogicVector(bit_vector'(x"0073")));
  x4 <= x3 + SHR(y3,"110"); y4 <= y3 - SHR(x3,"110");
end if;

if (a4(15) = '0') then -- a4>=0
  a5 <= a4 - SIGNED(To_StdLogicVector(bit_vector'(x"0039")));
  x5 <= x4 - SHR(y4,"111"); y5 <= y4 + SHR(x4,"111");
else
  a5 <= a4 + SIGNED(To_StdLogicVector(bit_vector'(x"0039")));
  x5 <= x4 + SHR(y4,"111"); y5 <= y4 - SHR(x4,"111");
end if;

if (a5(15) = '0') then -- a5>=0
  a6 <= a5 - SIGNED(To_StdLogicVector(bit_vector'(x"001d")));
  x6 <= x5 - SHR(y5,"1000"); y6 <= y5 + SHR(x5,"1000");
else
  a6 <= a5 + SIGNED(To_StdLogicVector(bit_vector'(x"001d")));
  x6 <= x5 + SHR(y5,"1000"); y6 <= y5 - SHR(x5,"1000");
end if;

if (a6(15) = '0') then -- a6>=0
  a7 <= a6 - SIGNED(To_StdLogicVector(bit_vector'(x"000e")));
  x7 <= x6 - SHR(y6,"1001"); y7 <= y6 + SHR(x6,"1001");
else
  a7 <= a6 + SIGNED(To_StdLogicVector(bit_vector'(x"000e")));
  x7 <= x6 + SHR(y6,"1001"); y7 <= y6 - SHR(x6,"1001");
end if;

if (a7(15) = '0') then -- a7>=0
  a8 <= a7 - SIGNED(To_StdLogicVector(bit_vector'(x"0007")));
  x8 <= x7 - SHR(y7,"1010"); y8 <= y7 + SHR(x7,"1010");
else
  a8 <= a7 + SIGNED(To_StdLogicVector(bit_vector'(x"0007")));
  x8 <= x7 + SHR(y7,"1010"); y8 <= y7 - SHR(x7,"1010");
end if;

if (a8(15) = '0') then -- a8>=0
  a9 <= a8 - SIGNED(To_StdLogicVector(bit_vector'(x"0004")));
  x9 <= x8 - SHR(y8,"1011"); y9 <= y8 + SHR(x8,"1011");
else
  a9 <= a8 + SIGNED(To_StdLogicVector(bit_vector'(x"0004")));
  x9 <= x8 + SHR(y8,"1011"); y9 <= y8 - SHR(x8,"1011");

```

Appendix A

```
        end if;  
        end if;  
        end if;  
    end if;  
  
end PROCESS;  
END hybrid_highreso;
```

Appendix B

Appendix B

VHDL Model of Hybrid CORDIC for Hyperbolic Sin/Cos Functions

```

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY hypercord_hyb IS
  PORT (
    Clk      : IN      std_logic;
    WriteState : IN      std_logic;
    a        : IN      std_logic_vector(15 downto 0);
    x        : IN      std_logic_vector(15 downto 0);
    y        : IN      std_logic_vector(15 downto 0);
    xout     : OUT     std_logic_vector(15 downto 0);
    yout     : OUT     std_logic_vector(15 downto 0)
  );-- Declarations

END hypercord_hyb ;
-- hds interface_end

ARCHITECTURE hypercord_hyb OF hypercord_hyb IS

  signal a1, a2, a3, a4, a5, a6, a7 : signed(15 downto 0);
  signal a8, a9, a10, a11, a12 : signed(15 downto 0);
  signal x1, x2, x3, x4, x5, x6, x7 : signed(15 downto 0);
  signal x8, x9, x10, x11, x12 : signed(15 downto 0);
  signal y1, y2, y3, y4, y5, y6, y7 : signed(15 downto 0);
  signal y8, y9, y10, y11, y12 : signed(15 downto 0);
  signal const : std_logic_vector(15 downto 0);
  signal ain : signed(6 downto 0);

  function SINH(n :signed(6 downto 0)) return signed is
    variable result :signed(6 downto 0);
  begin
    case n is
      when "0000000" => result := "0000000";

```

Appendix B

```

when "0000001" => result := "0000001";
when "0000010" => result := "0000010";
when "0000011" => result := "0000011";
when "0000100" => result := "0000100";
when "0000101" => result := "0000101";
when "0000110" => result := "0000110";
when "0000111" => result := "0000111";
when "0001000" => result := "0001000";
when "0001001" => result := "0001001";
when "0001010" => result := "0001010";
when "0001011" => result := "0001011";
when "0001100" => result := "0001100";
when "0001101" => result := "0001101";
when "0001110" => result := "0001110";
when "0001111" => result := "0001111";
when "0010000" => result := "0010000";
when "0010001" => result := "0010001";
when "0010010" => result := "0010010";
when "0010011" => result := "0010100";
when "0010100" => result := "0010101";
when "0010101" => result := "0010110";
when "0010110" => result := "0010111";
when "0010111" => result := "0011001";
when "0011000" => result := "0011010";
when "0011001" => result := "0011011";
when "0011010" => result := "0011100";
when "0011011" => result := "0011110";
when "0011100" => result := "0011111";
when "0011101" => result := "0100001";
when "0011110" => result := "0100010";
when "0011111" => result := "0100100";
when "0100000" => result := "0100101";
when "0100001" => result := "0100111";
when "0100010" => result := "0101000";
when "0100011" => result := "0101010";
when "0100100" => result := "0101100";
when others => result := "0000000";
end case;
return result;
end SINH;

function COSH(n :signed(6 downto 0)) return signed is
variable result :signed(6 downto 0);
begin
case n is
when "0000000" => result := "0100000";
when "0000001" => result := "0100000";
when "0000010" => result := "0100000";

```

Appendix B

```

        when "0000011" => result := "0100000";
        when "0000100" => result := "0100000";
        when "0000101" => result := "0100000";
        when "0000110" => result := "0100000";
        when "0000111" => result := "0100000";
        when "0001000" => result := "0100001";
        when "0001001" => result := "0100001";
        when "0001010" => result := "0100001";
        when "0001011" => result := "0100001";
        when "0001100" => result := "0100010";
        when "0001101" => result := "0100010";
        when "0001110" => result := "0100011";
        when "0001111" => result := "0100011";
        when "0010000" => result := "0100100";
        when "0010001" => result := "0100100";
        when "0010010" => result := "0100101";
        when "0010011" => result := "0100101";
        when "0010100" => result := "0100110";
        when "0010101" => result := "0100111";
        when "0010110" => result := "0100111";
        when "0010111" => result := "0101000";
        when "0011000" => result := "0101001";
        when "0011001" => result := "0101010";
        when "0011010" => result := "0101011";
        when "0011011" => result := "0101100";
        when "0011100" => result := "0101101";
        when "0011101" => result := "0101110";
        when "0011110" => result := "0101111";
        when "0011111" => result := "0110000";
        when "0100000" => result := "0110001";
        when "0100001" => result := "0110010";
        when "0100010" => result := "0110011";
        when "0100011" => result := "0110101";
        when "0100100" => result := "0110110";
        when others => result := "0000000";
    end case;
    return result;
end COSH;

begin

xout <= STD_LOGIC_VECTOR(x1);
yout <= STD_LOGIC_VECTOR(y1);

const <= "0000000111111111";

PROCESS(clk)

variable remainder :std_logic_vector(15 downto 0);

```


Appendix B

```

begin

remainder := a and const;--"0000000111111111";

if (clk'event and clk='1') then
  if (WriteState = '1') then
    a1 <= SIGNED(a);
    x1 <= SIGNED(x); y1 <= SIGNED(y);

    ain <= a1(15 downto 9);
    x2 <= COSH(ain)&"000000000";
    y2 <= SINH(ain)&"000000000";
    a2 <= SIGNED(remainder);

    if (a2(15) = '0') then -- a2>=0
      a3 <= a2 - SIGNED(To_StdLogicVector(bit_vector('x"0100"')));
      x3 <= x2 + SHR(y2,"110"); y3 <= y2 + SHR(x2,"110");
    else
      a3 <= a2 + SIGNED(To_StdLogicVector(bit_vector('x"0100"')));
      x3 <= x2 - SHR(y2,"110"); y3 <= y2 - SHR(x2,"110");
    end if;

    if (a3(15) = '0') then -- a3>=0
      a4 <= a3 - SIGNED(To_StdLogicVector(bit_vector('x"0080"')));
      x4 <= x3 + SHR(y3,"111"); y4 <= y3 + SHR(x3,"111");
    else
      a4 <= a3 + SIGNED(To_StdLogicVector(bit_vector('x"0080"')));
      x4 <= x3 - SHR(y3,"111"); y4 <= y3 - SHR(x3,"111");
    end if;

    if (a4(15) = '0') then -- a8>=0
      a5 <= a4 - SIGNED(To_StdLogicVector(bit_vector('x"0040"')));
      x5 <= x4 + SHR(y4,"1000"); y5 <= y4 + SHR(x4,"1000");
    else
      a5 <= a4 + SIGNED(To_StdLogicVector(bit_vector('x"0040"')));
      x5 <= x4 - SHR(y4,"1000"); y5 <= y4 - SHR(x4,"1000");
    end if;

    if (a5(15) = '0') then -- a9>=0
      a6 <= a5 - SIGNED(To_StdLogicVector(bit_vector('x"0020"')));
      x6 <= x5 + SHR(y5,"1001"); y6 <= y5 + SHR(x5,"1001");
    else
      a6 <= a5 + SIGNED(To_StdLogicVector(bit_vector('x"0020"')));
      x6 <= x5 - SHR(y5,"1001"); y6 <= y5 - SHR(x5,"1001");
    end if;

    if (a6(15) = '0') then -- a10>=0
      a7 <= a6 - SIGNED(To_StdLogicVector(bit_vector('x"0010"')));
      x7 <= x6 + SHR(y6,"1010"); y7 <= y6 + SHR(x6,"1010");

```

Appendix B

```

else
  a7 <= a6 + SIGNED(To_StdLogicVector(bit_vector'(x"0010")));
  x7 <= x6 - SHR(y6,"1010"); y7 <= y6 - SHR(x6,"1010");
end if;

if (a7(15) = '0') then -- a10>=0
  a8 <= a7 - SIGNED(To_StdLogicVector(bit_vector'(x"0008")));
  x8 <= x7 + SHR(y7,"1011"); y8 <= y7 + SHR(x7,"1011");
else
  a8 <= a7 + SIGNED(To_StdLogicVector(bit_vector'(x"0008")));
  x8 <= x7 - SHR(y7,"1011"); y8 <= y7 - SHR(x7,"1011");
end if;

if (a8(15) = '0') then -- a10>=0
  a9 <= a8 - SIGNED(To_StdLogicVector(bit_vector'(x"0004")));
  x9 <= x8 + SHR(y8,"1100"); y9 <= y8 + SHR(x8,"1100");
else
  a9 <= a8 + SIGNED(To_StdLogicVector(bit_vector'(x"0004")));
  x9 <= x8 - SHR(y8,"1100"); y9 <= y8 - SHR(x8,"1100");
end if;

if (a9(15) = '0') then -- a10>=0
  a10 <= a9 - SIGNED(To_StdLogicVector(bit_vector'(x"0002")));
  x10 <= x9 + SHR(y9,"1101"); y10 <= y9 + SHR(x9,"1101");
else
  a10 <= a9 + SIGNED(To_StdLogicVector(bit_vector'(x"0002")));
  x10 <= x9 - SHR(y9,"1101"); y10 <= y9 - SHR(x9,"1101");
end if;

if (a10(15) = '0') then -- a10>=0
  a11 <= a10 - SIGNED(To_StdLogicVector(bit_vector'(x"0002")));
  x11 <= x10 + SHR(y10,"1101"); y11 <= y10 + SHR(x10,"1101");
else
  a11 <= a10 + SIGNED(To_StdLogicVector(bit_vector'(x"0002")));
  x11 <= x10 - SHR(y10,"1101"); y11 <= y10 - SHR(x10,"1101");
end if;

end if;
end if;

end PROCESS;

END hypercord_hyb;

```

Appendix C

Appendix C

VHDL Model of Hybrid CORDIC for Reciprocal Function

```

LIBRARY IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY hybrid_div_MOD11 IS
PORT (
    Clk      : IN    std_logic;
    WriteState : IN    std_logic;
    a        : IN    std_logic_vector(31 downto 0);
    x        : IN    std_logic_vector(31 downto 0);
    y        : IN    std_logic_vector(31 downto 0);
    xout     : OUT    std_logic_vector(15 downto 0);
    yout     : OUT    std_logic_vector(15 downto 0);
    zout     : OUT    std_logic_vector(15 downto 0)
);-- Declarations

END hybrid_div_MOD11 ;

-- hds interface_end
ARCHITECTURE hybrid_div_MOD11 OF hybrid_div_MOD11 IS

    signal a1, a2, a3, a4, a5, a6: signed(31 downto 0);
    signal x1, x2, x3, x4, x5, x6: signed(31 downto 0);
    signal y1, y2, y3, y4, y5, y6: signed(31 downto 0);
    signal z : signed(31 downto 0);
    signal divin: signed(11 downto 0) ;
    signal divout:signed(11 downto 0) ;

    function DIV(n :signed(11 downto 0)) return signed is
        variable result :signed(11 downto 0);
        begin
            case n is
                when "010000000000" => result := "010000000000";
                when "010000000001" => result := "001111111111";
                when "010000000010" => result := "001111111110";
                when "010000000011" => result := "001111111101";
            end case;
        end function;


```


Appendix C

```

when "010000000100" => result := "001111111100";
when "010000000101" => result := "001111111011";
when "010000000110" => result := "001111111010";
when "010000000111" => result := "001111111001";
when "010000001000" => result := "001111111000";
when "010000001001" => result := "001111111011";
when "010000001010" => result := "001111111010";
when "010000001011" => result := "0011111110101";
when "010000001100" => result := "0011111110100";
when "010000001101" => result := "0011111110011";
when "010000001110" => result := "0011111110010";
when "010000001111" => result := "0011111110001";
when "010000010000" => result := "0011111110000";
when "010000010001" => result := "0011111101111";
when "010000010010" => result := "0011111101110";
when "010000010011" => result := "0011111101101";
when "010000010100" => result := "0011111101100";
when "010000010101" => result := "0011111101011";
when "010000010110" => result := "0011111101010";
when "010000010111" => result := "0011111101001";
when "010000011000" => result := "0011111101000";
when "010000011001" => result := "0011111100111";
when "010000011010" => result := "0011111100110";
when "010000011011" => result := "0011111100101";
when "010000011100" => result := "0011111100100";
when "010000011101" => result := "0011111100011";
when "010000011110" => result := "0011111100010";
when "010000011111" => result := "0011111100001";
when "010000100000" => result := "0011111100000";
when "010000100001" => result := "0011111100000";
when "010000100010" => result := "0011111011111";
when "010000100011" => result := "0011111011110";
when "010000100100" => result := "0011111011101";
when "010000100101" => result := "0011111011100";
when "010000100110" => result := "0011111011011";
when "010000100111" => result := "0011111011010";
when "010000101000" => result := "0011111011001";
when "010000101001" => result := "0011111011000";
when "010000101010" => result := "0011111010111";
when "010000101011" => result := "0011111010110";
when "010000101100" => result := "0011111010101";
when "010000101101" => result := "0011111010100";
when "010000101110" => result := "0011111010011";
when "010000101111" => result := "0011111010011";
when "010000110000" => result := "0011111010010";
when "010000110001" => result := "0011111010001";
when "010000110010" => result := "0011111010000";
when "010000110011" => result := "0011111001111";
when "010000110100" => result := "0011111001110";

```

Appendix C

```

when "010000110101" => result := "001111001101";
when "010000110110" => result := "001111001100";
when "010000110111" => result := "001111001011";
when "010000111000" => result := "001111001010";
when "010000111001" => result := "001111001010";
when "010000111010" => result := "001111001001";
when "010000111011" => result := "001111001000";
when "010000111100" => result := "001111000111";
when "010000111101" => result := "001111000110";
when "010000111110" => result := "001111000101";
when "010000111111" => result := "001111000100";
when "010001000000" => result := "001111000011";
when "010001000001" => result := "001111000010";
when "010001000010" => result := "001111000001";
when "010001000011" => result := "001111000001";
when "010001000100" => result := "001111000000";
when "010001000101" => result := "001110111111";
when "010001000110" => result := "001110111110";
when "010001000111" => result := "001110111101";
when "010001001000" => result := "001110111100";
when "010001001001" => result := "001110111101";
when "010001001010" => result := "001110111101";
when "010001001011" => result := "001110111101";
when "010001001100" => result := "001110111001";
when "010001001101" => result := "001110111000";
when "010001001110" => result := "001110110111";
when "010001001111" => result := "001110110110";
when "010001010000" => result := "001110110101";
when "010001010001" => result := "001110110100";
when "010001010010" => result := "001110110100";
when "010001010011" => result := "001110110011";
when "010001010100" => result := "001110110010";
when "010001010101" => result := "001110110001";
when "010001010110" => result := "001110110000";
when "010001010111" => result := "001110101111";
when "010001011000" => result := "001110101110";
when "010001011001" => result := "001110101110";
when "010001011010" => result := "001110101101";
when "010001011011" => result := "001110101100";
when "010001011100" => result := "001110101011";
when "010001011101" => result := "001110101010";
when "010001011110" => result := "001110101001";
when "010001011111" => result := "001110101001";
when "010001100000" => result := "001110101000";
when "010001100001" => result := "001110100111";
when "010001100010" => result := "001110100110";
when "010001100011" => result := "001110100101";
when "010001100100" => result := "001110100100";
when "010001100101" => result := "001110100100";

```


Appendix C

```

when "010001100110" => result := "001110100011";
when "010001100111" => result := "001110100010";
when "010001101000" => result := "001110100001";
when "010001101001" => result := "001110100000";
when "010001101010" => result := "001110011111";
when "010001101011" => result := "001110011111";
when "010001101100" => result := "001110011110";
when "010001101101" => result := "001110011101";
when "010001101110" => result := "001110011100";
when "010001101111" => result := "001110011011";
when "010001110000" => result := "001110011011";
when "010001110001" => result := "001110011010";
when "010001110010" => result := "001110011001";
when "010001110011" => result := "001110011000";
when "010001110100" => result := "001110010111";
when "010001110101" => result := "001110010110";
when "010001110110" => result := "001110010110";
when "010001110111" => result := "001110010101";
when "010001111000" => result := "001110010100";
when "010001111001" => result := "001110010011";
when "010001111010" => result := "001110010010";
when "010001111011" => result := "001110010010";
when "010001111100" => result := "001110010001";
when "010001111101" => result := "001110010000";
when "010001111110" => result := "001110001111";
when "010001111111" => result := "001110001111";
when "010010000000" => result := "001110001110";
when "010010000001" => result := "001110001101";
when "010010000010" => result := "001110001100";
when "010010000011" => result := "001110001011";
when "010010000100" => result := "001110001011";
when "010010000101" => result := "001110001010";
when "010010000110" => result := "001110001001";
when "010010000111" => result := "001110001000";
when "010010001000" => result := "001110000111";
when "010010001001" => result := "001110000111";
when "010010001010" => result := "001110000110";
when "010010001011" => result := "001110000101";
when "010010001100" => result := "001110000100";
when "010010001101" => result := "001110000100";
when "010010001110" => result := "001110000011";
when "010010001111" => result := "001110000010";
when "010010010000" => result := "001110000001";
when "010010010001" => result := "001110000000";
when "010010010010" => result := "001110000000";
when "010010010011" => result := "001101111111";
when "010010010100" => result := "001101111110";
when "010010010101" => result := "001101111101";
when "010010010110" => result := "001101111101";

```

Appendix C

```

when "010010010111" => result := "001101111100";
when "010010011000" => result := "001101111011";
when "010010011001" => result := "001101111010";
when "010010011010" => result := "001101111010";
when "010010011011" => result := "001101111001";
when "010010011100" => result := "001101111000";
when "010010011101" => result := "001101111011";
when "010010011110" => result := "001101111011";
when "010010011111" => result := "001101111010";
when "010010100000" => result := "0011011110101";
when "010010100001" => result := "0011011110100";
when "010010100010" => result := "0011011110100";
when "010010100011" => result := "0011011110011";
when "010010100100" => result := "0011011110010";
when "010010100101" => result := "0011011110001";
when "010010100110" => result := "0011011110001";
when "010010100111" => result := "0011011110000";
when "010010101000" => result := "001101101111";
when "010010101001" => result := "001101101110";
when "010010101010" => result := "001101101110";
when "010010101011" => result := "001101101101";
when "010010101100" => result := "001101101100";
when "010010101101" => result := "001101101100";
when "010010101110" => result := "001101101011";
when "010010101111" => result := "001101101010";
when "010010110000" => result := "001101101001";
when "010010110001" => result := "001101101001";
when "010010110010" => result := "001101101000";
when "010010110011" => result := "001101100111";
when "010010110100" => result := "001101100110";
when "010010110101" => result := "001101100110";
when "010010110110" => result := "001101100101";
when "010010110111" => result := "001101100100";
when "010010111000" => result := "001101100100";
when "010010111001" => result := "001101100011";
when "010010111010" => result := "001101100010";
when "010010111011" => result := "001101100001";
when "010010111100" => result := "001101100001";
when "010010111101" => result := "001101100000";
when "010010111110" => result := "001101011111";
when "010010111111" => result := "001101011111";
when "010011000000" => result := "001101011110";
when "010011000001" => result := "001101011101";
when "010011000010" => result := "001101011100";
when "010011000011" => result := "001101011100";
when "010011000100" => result := "001101011011";
when "010011000101" => result := "001101011010";
when "010011000110" => result := "001101011010";
when "010011000111" => result := "001101011001";

```


Appendix C

```

when "010011001000" => result := "001101011000";
when "010011001001" => result := "001101010111";
when "010011001010" => result := "001101010111";
when "010011001011" => result := "001101010110";
when "010011001100" => result := "001101010101";
when "010011001101" => result := "001101010101";
when "010011001110" => result := "001101010100";
when "010011001111" => result := "001101010011";
when "010011010000" => result := "001101010011";
when "010011010001" => result := "001101010010";
when "010011010010" => result := "001101010001";
when "010011010011" => result := "001101010001";
when "010011010100" => result := "001101010000";
when "010011010101" => result := "001101001111";
when "010011010110" => result := "001101001110";
when "010011010111" => result := "001101001110";
when "010011011000" => result := "001101001101";
when "010011011001" => result := "001101001100";
when "010011011010" => result := "001101001100";
when "010011011011" => result := "001101001011";
when "010011011100" => result := "001101001010";
when "010011011101" => result := "001101001010";
when "010011011110" => result := "001101001001";
when "010011011111" => result := "001101001000";
when "010011100000" => result := "001101001000";
when "010011100001" => result := "001101000111";
when "010011100010" => result := "001101000110";
when "010011100011" => result := "001101000110";
when "010011100100" => result := "001101000101";
when "010011100101" => result := "001101000100";
when "010011100110" => result := "001101000100";
when "010011100111" => result := "001101000011";
when "010011101000" => result := "001101000010";
when "010011101001" => result := "001101000010";
when "010011101010" => result := "001101000001";
when "010011101011" => result := "001101000000";
when "010011101100" => result := "001101000000";
when "010011101101" => result := "001100111111";
when "010011101110" => result := "001100111110";
when "010011101111" => result := "001100111110";
when "010011110000" => result := "001100111101";
when "010011110001" => result := "001100111100";
when "010011110010" => result := "001100111100";
when "010011110011" => result := "001100111011";
when "010011110100" => result := "001100111010";
when "010011110101" => result := "001100111010";
when "010011110110" => result := "001100111001";
when "010011110111" => result := "001100111001";
when "010011111000" => result := "001100111000";

```

Appendix C

```

when "010011111001" => result := "001100110111";
when "010011111010" => result := "001100110111";
when "010011111011" => result := "001100110110";
when "010011111100" => result := "001100110101";
when "010011111101" => result := "001100110101";
when "010011111110" => result := "001100110100";
when "010011111111" => result := "001100110011";
when "010100000000" => result := "001100110011";
when "010100000001" => result := "001100110010";
when "010100000010" => result := "001100110001";
when "010100000011" => result := "001100110001";
when "010100000100" => result := "001100110000";
when "010100000101" => result := "001100110000";
when "010100000110" => result := "001100101111";
when "010100000111" => result := "001100101110";
when "010100001000" => result := "001100101110";
when "010100001001" => result := "001100101101";
when "010100001010" => result := "001100101100";
when "010100001011" => result := "001100101100";
when "010100001100" => result := "001100101011";
when "010100001101" => result := "001100101010";
when "010100001110" => result := "001100101010";
when "010100001111" => result := "001100101001";
when "010100010000" => result := "001100101001";
when "010100010001" => result := "001100101000";
when "010100010010" => result := "001100100111";
when "010100010011" => result := "001100100111";
when "010100010100" => result := "001100100110";
when "010100010101" => result := "001100100101";
when "010100010110" => result := "001100100101";
when "010100010111" => result := "001100100100";
when "010100011000" => result := "001100100100";
when "010100011001" => result := "001100100011";
when "010100011010" => result := "001100100010";
when "010100011011" => result := "001100100010";
when "010100011100" => result := "001100100001";
when "010100011101" => result := "001100100001";
when "010100011110" => result := "001100100000";
when "010100011111" => result := "001100011111";
when "010100100000" => result := "001100011111";
when "010100100001" => result := "001100011110";
when "010100100010" => result := "001100011110";
when "010100100011" => result := "001100011101";
when "010100100100" => result := "001100011100";
when "010100100101" => result := "001100011100";
when "010100100110" => result := "001100011011";
when "010100100111" => result := "001100011010";
when "010100101000" => result := "001100011010";
when "010100101001" => result := "001100011001";

```


Appendix C

```

when "010100101010" => result := "001100011001";
when "010100101011" => result := "001100011000";
when "010100101100" => result := "001100010111";
when "010100101101" => result := "001100010111";
when "010100101110" => result := "001100010110";
when "010100101111" => result := "001100010110";
when "010100110000" => result := "001100010101";
when "010100110001" => result := "001100010100";
when "010100110010" => result := "001100010100";
when "010100110011" => result := "001100010011";
when "010100110100" => result := "001100010011";
when "010100110101" => result := "001100010010";
when "010100110110" => result := "001100010010";
when "010100110111" => result := "001100010001";
when "010100111000" => result := "001100010000";
when "010100111001" => result := "001100010000";
when "010100111010" => result := "001100001111";
when "010100111011" => result := "001100001111";
when "010100111100" => result := "001100001110";
when "010100111101" => result := "001100001101";
when "010100111110" => result := "001100001101";
when "010100111111" => result := "001100001100";
when "010101000000" => result := "001100001100";
when "010101000001" => result := "001100001011";
when "010101000010" => result := "001100001011";
when "010101000011" => result := "001100001010";
when "010101000100" => result := "001100001001";
when "010101000101" => result := "001100001001";
when "010101000110" => result := "001100001000";
when "010101000111" => result := "001100001000";
when "010101001000" => result := "001100000111";
when "010101001001" => result := "001100000111";
when "010101001010" => result := "001100000110";
when "010101001011" => result := "001100000101";
when "010101001100" => result := "001100000101";
when "010101001101" => result := "001100000100";
when "010101001110" => result := "001100000100";
when "010101001111" => result := "001100000011";
when "010101010000" => result := "001100000011";
when "010101010001" => result := "001100000010";
when "010101010010" => result := "001100000001";
when "010101010011" => result := "001100000001";
when "010101010100" => result := "001100000000";
when "010101010101" => result := "001100000000";
when "010101010110" => result := "001011111111";
when "010101010111" => result := "001011111111";
when "010101011000" => result := "001011111110";
when "010101011001" => result := "001011111101";
when "010101011010" => result := "001011111101";

```


Appendix C

```

when "010101011011" => result := "001011111100";
when "010101011100" => result := "001011111100";
when "010101011101" => result := "001011111101";
when "010101011110" => result := "001011111101";
when "010101011111" => result := "001011111101";
when "010101100000" => result := "001011111101";
when "010101100001" => result := "001011111100";
when "010101100010" => result := "001011111100";
when "010101100011" => result := "001011111100";
when "010101100100" => result := "001011111011";
when "010101100101" => result := "001011111011";
when "010101100110" => result := "001011111010";
when "010101100111" => result := "001011111010";
when "010101101000" => result := "001011111010";
when "010101101001" => result := "001011111010";
when "010101101010" => result := "001011111010";
when "010101101011" => result := "001011111010";
when "010101101100" => result := "001011111001";
when "010101101101" => result := "001011111001";
when "010101101110" => result := "001011111001";
when "010101101111" => result := "001011111001";
when "010101110000" => result := "001011111001";
when "010101110001" => result := "001011111000";
when "010101110010" => result := "001011111000";
when "010101110011" => result := "001011111000";
when "010101110100" => result := "001011111011";
when "010101110101" => result := "001011111011";
when "010101110110" => result := "001011111010";
when "010101110111" => result := "001011111010";
when "010101111000" => result := "001011111001";
when "010101111001" => result := "001011111001";
when "010101111010" => result := "001011111010";
when "010101111011" => result := "001011111010";
when "010101111100" => result := "001011111010";
when "010101111101" => result := "001011111010";
when "010101111110" => result := "001011111001";
when "010101111111" => result := "001011111001";
when "010110000000" => result := "001011111000";
when "010110000001" => result := "001011111000";
when "010110000010" => result := "001011111001";
when "010110000011" => result := "001011111001";
when "010110000100" => result := "001011111010";
when "010110000101" => result := "001011111010";
when "010110000110" => result := "001011111010";
when "010110000111" => result := "001011111010";
when "010110001000" => result := "001011111000";
when "010110001001" => result := "001011111001";
when "010110001010" => result := "001011111001";
when "010110001011" => result := "001011111001";

```

Appendix C

```

when "010110001100" => result := "0010111000010";
when "010110001101" => result := "0010111000001";
when "010110001110" => result := "0010111000001";
when "010110001111" => result := "0010111000000";
when "010110010000" => result := "0010111000000";
when "010110010001" => result := "0010110111111";
when "010110010010" => result := "0010110111111";
when "010110010011" => result := "0010110111110";
when "010110010100" => result := "0010110111110";
when "010110010101" => result := "0010110111101";
when "010110010110" => result := "0010110111101";
when "010110010111" => result := "0010110111100";
when "010110011000" => result := "0010110111100";
when "010110011001" => result := "0010110110111";
when "010110011010" => result := "0010110110111";
when "010110011011" => result := "0010110110110";
when "010110011100" => result := "0010110110110";
when "010110011101" => result := "0010110110101";
when "010110011110" => result := "0010110110101";
when "010110011111" => result := "0010110110100";
when "010110100000" => result := "0010110110100";
when "010110100001" => result := "0010110101111";
when "010110100010" => result := "0010110101111";
when "010110100011" => result := "0010110101110";
when "010110100100" => result := "0010110101110";
when "010110100101" => result := "0010110101101";
when "010110100110" => result := "0010110101101";
when "010110100111" => result := "0010110101100";
when "010110101000" => result := "0010110101100";
when "010110101001" => result := "0010110100111";
when "010110101010" => result := "0010110100111";
when "010110101011" => result := "0010110100110";
when "010110101100" => result := "0010110100110";
when "010110101101" => result := "0010110100011";
when "010110101110" => result := "0010110100011";
when "010110101111" => result := "0010110100000";
when "010110110000" => result := "0010110100000";
when "010110110001" => result := "0010110011111";
when "010110110010" => result := "0010110011111";
when "010110110011" => result := "0010110011110";
when "010110110100" => result := "0010110011110";
when "010110110101" => result := "0010110011101";
when "010110110110" => result := "0010110011101";
when "010110110111" => result := "0010110011100";
when "010110111000" => result := "0010110011100";
when "010110111001" => result := "0010110010111";
when "010110111010" => result := "0010110010111";
when "010110111011" => result := "0010110010110";
when "010110111100" => result := "0010110010110";

```


Appendix C

```

when "010110111101" => result := "001011001001";
when "010110111110" => result := "001011001001";
when "010110111111" => result := "001011001000";
when "010111000000" => result := "001011001000";
when "010111000001" => result := "001011000111";
when "010111000010" => result := "001011000111";
when "010111000011" => result := "001011000110";
when "010111000100" => result := "001011000110";
when "010111000101" => result := "001011000101";
when "010111000110" => result := "001011000101";
when "010111000111" => result := "001011000100";
when "010111001000" => result := "001011000100";
when "010111001001" => result := "001011000100";
when "010111001010" => result := "001011000011";
when "010111001011" => result := "001011000011";
when "010111001100" => result := "001011000010";
when "010111001101" => result := "001011000010";
when "010111001110" => result := "001011000001";
when "010111001111" => result := "001011000001";
when "010111010000" => result := "001011000000";
when "010111010001" => result := "001011000000";
when "010111010010" => result := "001010111111";
when "010111010011" => result := "001010111111";
when "010111010100" => result := "001010111110";
when "010111010101" => result := "001010111110";
when "010111010110" => result := "001010111101";
when "010111010111" => result := "001010111101";
when "010111011000" => result := "001010111100";
when "010111011001" => result := "001010111100";
when "010111011010" => result := "001010111011";
when "010111011011" => result := "001010111011";
when "010111011100" => result := "001010111011";
when "010111011101" => result := "001010111010";
when "010111011110" => result := "001010111010";
when "010111011111" => result := "001010111001";
when "010111100000" => result := "001010111001";
when "010111100001" => result := "001010111000";
when "010111100010" => result := "001010111000";
when "010111100011" => result := "001010110111";
when "010111100100" => result := "001010110111";
when "010111100101" => result := "001010110110";
when "010111100110" => result := "001010110110";
when "010111100111" => result := "001010110101";
when "010111101000" => result := "001010110101";
when "010111101001" => result := "001010110101";
when "010111101010" => result := "001010110100";
when "010111101011" => result := "001010110100";
when "010111101100" => result := "001010110011";
when "010111101101" => result := "001010110011";

```

Appendix C

```

when "010111101110" => result := "001010110010";
when "010111101111" => result := "001010110010";
when "010111110000" => result := "001010110001";
when "010111110001" => result := "001010110001";
when "010111110010" => result := "001010110000";
when "010111110011" => result := "001010110000";
when "010111110100" => result := "001010110000";
when "010111110101" => result := "001010101111";
when "010111110110" => result := "001010101111";
when "010111110111" => result := "001010101110";
when "010111111000" => result := "001010101110";
when "010111111001" => result := "001010101101";
when "010111111010" => result := "001010101101";
when "010111111011" => result := "001010101100";
when "010111111100" => result := "001010101100";
when "010111111101" => result := "001010101100";
when "010111111110" => result := "001010101011";
when "010111111111" => result := "001010101011";
when "011000000000" => result := "001010101010";
when "011000000001" => result := "001010101010";
when "011000000010" => result := "001010101001";
when "011000000011" => result := "001010101001";
when "011000000100" => result := "001010101000";
when "011000000101" => result := "001010101000";
when "011000000110" => result := "001010101000";
when "011000000111" => result := "001010100111";
when "011000001000" => result := "001010100111";
when "011000001001" => result := "001010100110";
when "011000001010" => result := "001010100110";
when "011000001011" => result := "001010100101";
when "011000001100" => result := "001010100101";
when "011000001101" => result := "001010100100";
when "011000001110" => result := "001010100100";
when "011000001111" => result := "001010100100";
when "011000010000" => result := "001010100011";
when "011000010001" => result := "001010100011";
when "011000010010" => result := "001010100010";
when "011000010011" => result := "001010100010";
when "011000010100" => result := "001010100001";
when "011000010101" => result := "001010100001";
when "011000010110" => result := "001010100001";
when "011000010111" => result := "001010100000";
when "011000011000" => result := "001010100000";
when "011000011001" => result := "001010011111";
when "011000011010" => result := "001010011111";
when "011000011011" => result := "001010011110";
when "011000011100" => result := "001010011110";
when "011000011101" => result := "001010011110";
when "011000011110" => result := "001010011101";

```


Appendix C

```

when "011000011111" => result := "001010011101";
when "011000100000" => result := "001010011100";
when "011000100001" => result := "001010011100";
when "011000100010" => result := "001010011011";
when "011000100011" => result := "001010011011";
when "011000100100" => result := "001010011011";
when "011000100101" => result := "001010011010";
when "011000100110" => result := "001010011010";
when "011000100111" => result := "001010011001";
when "011000101000" => result := "001010011001";
when "011000101001" => result := "001010011000";
when "011000101010" => result := "001010011000";
when "011000101011" => result := "001010011000";
when "011000101100" => result := "001010010111";
when "011000101101" => result := "001010010111";
when "011000101110" => result := "001010010110";
when "011000101111" => result := "001010010110";
when "011000110000" => result := "001010010101";
when "011000110001" => result := "001010010101";
when "011000110010" => result := "001010010101";
when "011000110011" => result := "001010010100";
when "011000110100" => result := "001010010100";
when "011000110101" => result := "001010010011";
when "011000110110" => result := "001010010011";
when "011000110111" => result := "001010010011";
when "011000111000" => result := "001010010010";
when "011000111001" => result := "001010010010";
when "011000111010" => result := "001010010001";
when "011000111011" => result := "001010010001";
when "011000111100" => result := "001010010001";
when "011000111101" => result := "001010010000";
when "011000111110" => result := "001010010000";
when "011000111111" => result := "001010001111";
when "011001000000" => result := "001010001111";
when "011001000001" => result := "001010001110";
when "011001000010" => result := "001010001110";
when "011001000011" => result := "001010001110";
when "011001000100" => result := "001010001101";
when "011001000101" => result := "001010001101";
when "011001000110" => result := "001010001100";
when "011001000111" => result := "001010001100";
when "011001001000" => result := "001010001100";
when "011001001001" => result := "001010001011";
when "011001001010" => result := "001010001011";
when "011001001011" => result := "001010001010";
when "011001001100" => result := "001010001010";
when "011001001101" => result := "001010001010";
when "011001001110" => result := "001010001001";
when "011001001111" => result := "001010001001";

```


Appendix C

```

when "011001010000" => result := "001010001000";
when "011001010001" => result := "001010001000";
when "011001010010" => result := "001010001000";
when "011001010011" => result := "001010000111";
when "011001010100" => result := "001010000111";
when "011001010101" => result := "001010000110";
when "011001010110" => result := "001010000110";
when "011001010111" => result := "001010000110";
when "011001011000" => result := "001010000101";
when "011001011001" => result := "001010000101";
when "011001011010" => result := "001010000100";
when "011001011011" => result := "001010000100";
when "011001011100" => result := "001010000100";
when "011001011101" => result := "001010000011";
when "011001011110" => result := "001010000011";
when "011001011111" => result := "001010000010";
when "011001100000" => result := "001010000010";
when "011001100001" => result := "001010000010";
when "011001100010" => result := "001010000001";
when "011001100011" => result := "001010000001";
when "011001100100" => result := "001010000000";
when "011001100101" => result := "001010000000";
when "011001100110" => result := "001010000000";
when "011001100111" => result := "001001111111";
when "011001101000" => result := "001001111111";
when "011001101001" => result := "001001111110";
when "011001101010" => result := "001001111110";
when "011001101011" => result := "001001111110";
when "011001101100" => result := "001001111101";
when "011001101101" => result := "001001111101";
when "011001101110" => result := "001001111101";
when "011001101111" => result := "001001111100";
when "011001110000" => result := "001001111100";
when "011001110001" => result := "001001111101";
when "011001110010" => result := "001001111101";
when "011001110011" => result := "001001111101";
when "011001110100" => result := "001001111101";
when "011001110101" => result := "001001111101";
when "011001110110" => result := "001001111101";
when "011001110111" => result := "001001111100";
when "011001111000" => result := "001001111100";
when "011001111001" => result := "001001111100";
when "011001111010" => result := "001001111100";
when "011001111011" => result := "001001111100";
when "011001111100" => result := "001001111011";
when "011001111101" => result := "001001111011";
when "011001111110" => result := "001001111010";
when "011001111111" => result := "001001111010";
when "011010000000" => result := "001001111010";

```

Appendix C

```

when "011010000001" => result := "001001110101";
when "011010000010" => result := "001001110101";
when "011010000011" => result := "001001110101";
when "011010000100" => result := "001001110100";
when "011010000101" => result := "001001110100";
when "011010000110" => result := "001001110011";
when "011010000111" => result := "001001110011";
when "011010001000" => result := "001001110011";
when "011010001001" => result := "001001110010";
when "011010001010" => result := "001001110010";
when "011010001011" => result := "001001110010";
when "011010001100" => result := "001001110001";
when "011010001101" => result := "001001110001";
when "011010001110" => result := "001001110000";
when "011010001111" => result := "001001110000";
when "011010010000" => result := "001001110000";
when "011010010001" => result := "001001101111";
when "011010010010" => result := "001001101111";
when "011010010011" => result := "001001101111";
when "011010010100" => result := "001001101110";
when "011010010101" => result := "001001101110";
when "011010010110" => result := "001001101101";
when "011010010111" => result := "001001101101";
when "011010011000" => result := "001001101101";
when "011010011001" => result := "001001101100";
when "011010011010" => result := "001001101100";
when "011010011011" => result := "001001101100";
when "011010011100" => result := "001001101011";
when "011010011101" => result := "001001101011";
when "011010011110" => result := "001001101010";
when "011010011111" => result := "001001101010";
when "011010100000" => result := "001001101010";
when "011010100001" => result := "001001101001";
when "011010100010" => result := "001001101001";
when "011010100011" => result := "001001101001";
when "011010100100" => result := "001001101000";
when "011010100101" => result := "001001101000";
when "011010100110" => result := "001001101000";
when "011010100111" => result := "001001100111";
when "011010101000" => result := "001001100111";
when "011010101001" => result := "001001100111";
when "011010101010" => result := "001001100110";
when "011010101011" => result := "001001100110";
when "011010101100" => result := "001001100101";
when "011010101101" => result := "001001100101";
when "011010101110" => result := "001001100101";
when "011010101111" => result := "001001100100";
when "011010110000" => result := "001001100100";
when "011010110001" => result := "001001100100";

```


Appendix C

```

when "011010110010" => result := "001001100011";
when "011010110011" => result := "001001100011";
when "011010110100" => result := "001001100011";
when "011010110101" => result := "001001100010";
when "011010110110" => result := "001001100010";
when "011010110111" => result := "001001100001";
when "011010111000" => result := "001001100001";
when "011010111001" => result := "001001100001";
when "011010111010" => result := "001001100000";
when "011010111011" => result := "001001100000";
when "011010111100" => result := "001001100000";
when "011010111101" => result := "001001011111";
when "011010111110" => result := "001001011111";
when "011010111111" => result := "001001011111";
when "011011000000" => result := "001001011110";
when "011011000001" => result := "001001011110";
when "011011000010" => result := "001001011110";
when "011011000011" => result := "001001011101";
when "011011000100" => result := "001001011101";
when "011011000101" => result := "001001011101";
when "011011000110" => result := "001001011100";
when "011011000111" => result := "001001011100";
when "011011001000" => result := "001001011100";
when "011011001001" => result := "001001011011";
when "011011001010" => result := "001001011011";
when "011011001011" => result := "001001011010";
when "011011001100" => result := "001001011010";
when "011011001101" => result := "001001011010";
when "011011001110" => result := "001001011001";
when "011011001111" => result := "001001011001";
when "011011010000" => result := "001001011001";
when "011011010001" => result := "001001011000";
when "011011010010" => result := "001001011000";
when "011011010011" => result := "001001011000";
when "011011010100" => result := "001001010111";
when "011011010101" => result := "001001010111";
when "011011010110" => result := "001001010111";
when "011011010111" => result := "001001010110";
when "011011011000" => result := "001001010110";
when "011011011001" => result := "001001010110";
when "011011011010" => result := "001001010101";
when "011011011011" => result := "001001010101";
when "011011011100" => result := "001001010101";
when "011011011101" => result := "001001010100";
when "011011011110" => result := "001001010100";
when "011011011111" => result := "001001010100";
when "011011100000" => result := "001001010011";
when "011011100001" => result := "001001010011";
when "011011100010" => result := "001001010011";

```

Appendix C

```

when "011011100011" => result := "001001010010";
when "011011100100" => result := "001001010010";
when "011011100101" => result := "001001010010";
when "011011100110" => result := "001001010001";
when "011011100111" => result := "001001010001";
when "011011101000" => result := "001001010001";
when "011011101001" => result := "001001010000";
when "011011101010" => result := "001001010000";
when "011011101011" => result := "001001010000";
when "011011101100" => result := "001001001111";
when "011011101101" => result := "001001001111";
when "011011101110" => result := "001001001111";
when "011011101111" => result := "001001001110";
when "011011110000" => result := "001001001110";
when "011011110001" => result := "001001001110";
when "011011110010" => result := "001001001101";
when "011011110011" => result := "001001001101";
when "011011110100" => result := "001001001101";
when "011011110101" => result := "001001001100";
when "011011110110" => result := "001001001100";
when "011011110111" => result := "001001001100";
when "011011111000" => result := "001001001011";
when "011011111001" => result := "001001001011";
when "011011111010" => result := "001001001011";
when "011011111011" => result := "001001001010";
when "011011111100" => result := "001001001010";
when "011011111101" => result := "001001001010";
when "011011111110" => result := "001001001001";
when "011011111111" => result := "001001001001";
when "011100000000" => result := "001001001001";
when "011100000001" => result := "001001001000";
when "011100000010" => result := "001001001000";
when "011100000011" => result := "001001001000";
when "011100000100" => result := "001001000111";
when "011100000101" => result := "001001000111";
when "011100000110" => result := "001001000111";
when "011100000111" => result := "001001000110";
when "011100001000" => result := "001001000110";
when "011100001001" => result := "001001000110";
when "011100001010" => result := "001001000101";
when "011100001011" => result := "001001000101";
when "011100001100" => result := "001001000101";
when "011100001101" => result := "001001000100";
when "011100001110" => result := "001001000100";
when "011100001111" => result := "001001000100";
when "011100010000" => result := "001001000011";
when "011100010001" => result := "001001000011";
when "011100010010" => result := "001001000011";
when "011100010011" => result := "001001000011";

```


Appendix C

```

when "011100010100" => result := "001001000010";
when "011100010101" => result := "001001000010";
when "011100010110" => result := "001001000010";
when "011100010111" => result := "001001000001";
when "011100011000" => result := "001001000001";
when "011100011001" => result := "001001000001";
when "011100011010" => result := "001001000000";
when "011100011011" => result := "001001000000";
when "011100011100" => result := "001001000000";
when "011100011101" => result := "001000111111";
when "011100011110" => result := "001000111111";
when "011100011111" => result := "001000111111";
when "011100100000" => result := "001000111110";
when "011100100001" => result := "001000111110";
when "011100100010" => result := "001000111110";
when "011100100011" => result := "001000111101";
when "011100100100" => result := "001000111101";
when "011100100101" => result := "001000111101";
when "011100100110" => result := "001000111100";
when "011100100111" => result := "001000111100";
when "011100101000" => result := "001000111100";
when "011100101001" => result := "001000111100";
when "011100101010" => result := "001000111011";
when "011100101011" => result := "001000111011";
when "011100101100" => result := "001000111011";
when "011100101101" => result := "001000111010";
when "011100101110" => result := "001000111010";
when "011100101111" => result := "001000111010";
when "011100110000" => result := "001000111001";
when "011100110001" => result := "001000111001";
when "011100110010" => result := "001000111001";
when "011100110011" => result := "001000111000";
when "011100110100" => result := "001000111000";
when "011100110101" => result := "001000111000";
when "011100110110" => result := "001000111000";
when "011100110111" => result := "001000110111";
when "011100111000" => result := "001000110111";
when "011100111001" => result := "001000110111";
when "011100111010" => result := "001000110110";
when "011100111011" => result := "001000110110";
when "011100111100" => result := "001000110110";
when "011100111101" => result := "001000110101";
when "011100111110" => result := "001000110101";
when "011100111111" => result := "001000110101";
when "011101000000" => result := "001000110100";
when "011101000001" => result := "001000110100";
when "011101000010" => result := "001000110100";
when "011101000011" => result := "001000110100";
when "011101000100" => result := "001000110011";

```


Appendix C

```

when "011101000101" => result := "001000110011";
when "011101000110" => result := "001000110011";
when "011101000111" => result := "001000110010";
when "011101001000" => result := "001000110010";
when "011101001001" => result := "001000110010";
when "011101001010" => result := "001000110001";
when "011101001011" => result := "001000110001";
when "011101001100" => result := "001000110001";
when "011101001101" => result := "001000110001";
when "011101001110" => result := "001000110000";
when "011101001111" => result := "001000110000";
when "011101010000" => result := "001000110000";
when "011101010001" => result := "001000101111";
when "011101010010" => result := "001000101111";
when "011101010011" => result := "001000101111";
when "011101010100" => result := "001000101110";
when "011101010101" => result := "001000101110";
when "011101010110" => result := "001000101110";
when "011101010111" => result := "001000101110";
when "011101011000" => result := "001000101101";
when "011101011001" => result := "001000101101";
when "011101011010" => result := "001000101101";
when "011101011011" => result := "001000101100";
when "011101011100" => result := "001000101100";
when "011101011101" => result := "001000101100";
when "011101011110" => result := "001000101011";
when "011101011111" => result := "001000101011";
when "011101100000" => result := "001000101011";
when "011101100001" => result := "001000101011";
when "011101100010" => result := "001000101010";
when "011101100011" => result := "001000101010";
when "011101100100" => result := "001000101010";
when "011101100101" => result := "001000101001";
when "011101100110" => result := "001000101001";
when "011101100111" => result := "001000101001";
when "011101101000" => result := "001000101001";
when "011101101001" => result := "001000101000";
when "011101101010" => result := "001000101000";
when "011101101011" => result := "001000101000";
when "011101101100" => result := "001000100111";
when "011101101101" => result := "001000100111";
when "011101101110" => result := "001000100111";
when "011101101111" => result := "001000100111";
when "011101110000" => result := "001000100110";
when "011101110001" => result := "001000100110";
when "011101110010" => result := "001000100110";
when "011101110011" => result := "001000100101";
when "011101110100" => result := "001000100101";
when "011101110101" => result := "001000100101";

```

Appendix C

```

when "011101110110" => result := "001000100100";
when "011101110111" => result := "001000100100";
when "011101111000" => result := "001000100100";
when "011101111001" => result := "001000100100";
when "011101111010" => result := "001000100011";
when "011101111011" => result := "001000100011";
when "011101111100" => result := "001000100011";
when "011101111101" => result := "001000100010";
when "011101111110" => result := "001000100010";
when "011101111111" => result := "001000100010";
when "011110000000" => result := "001000100010";
when "011110000001" => result := "001000100001";
when "011110000010" => result := "001000100001";
when "011110000011" => result := "001000100001";
when "011110000100" => result := "001000100000";
when "011110000101" => result := "001000100000";
when "011110000110" => result := "001000100000";
when "011110000111" => result := "001000100000";
when "011110001000" => result := "001000011111";
when "011110001001" => result := "001000011111";
when "011110001010" => result := "001000011111";
when "011110001011" => result := "001000011111";
when "011110001100" => result := "001000011110";
when "011110001101" => result := "001000011110";
when "011110001110" => result := "001000011110";
when "011110001111" => result := "001000011101";
when "011110010000" => result := "001000011101";
when "011110010001" => result := "001000011101";
when "011110010010" => result := "001000011101";
when "011110010011" => result := "001000011100";
when "011110010100" => result := "001000011100";
when "011110010101" => result := "001000011100";
when "011110010110" => result := "001000011011";
when "011110010111" => result := "001000011011";
when "011110011000" => result := "001000011011";
when "011110011001" => result := "001000011011";
when "011110011010" => result := "001000011010";
when "011110011011" => result := "001000011010";
when "011110011100" => result := "001000011010";
when "011110011101" => result := "001000011010";
when "011110011110" => result := "001000011001";
when "011110011111" => result := "001000011001";
when "011110100000" => result := "001000011001";
when "011110100001" => result := "001000011000";
when "011110100010" => result := "001000011000";
when "011110100011" => result := "001000011000";
when "011110100100" => result := "001000011000";
when "011110100101" => result := "001000010111";
when "011110100110" => result := "001000010111";

```


Appendix C

```

when "011110100111" => result := "001000010111";
when "011110101000" => result := "001000010110";
when "011110101001" => result := "001000010110";
when "011110101010" => result := "001000010110";
when "011110101011" => result := "001000010110";
when "011110101100" => result := "001000010101";
when "011110101101" => result := "001000010101";
when "011110101110" => result := "001000010101";
when "011110101111" => result := "001000010101";
when "011110110000" => result := "001000010100";
when "011110110001" => result := "001000010100";
when "011110110010" => result := "001000010100";
when "011110110011" => result := "001000010100";
when "011110110100" => result := "001000010011";
when "011110110101" => result := "001000010011";
when "011110110110" => result := "001000010011";
when "011110110111" => result := "001000010010";
when "011110111000" => result := "001000010010";
when "011110111001" => result := "001000010010";
when "011110111010" => result := "001000010010";
when "011110111011" => result := "001000010001";
when "011110111100" => result := "001000010001";
when "011110111101" => result := "001000010001";
when "011110111110" => result := "001000010001";
when "011110111111" => result := "001000010000";
when "011111000000" => result := "001000010000";
when "011111000001" => result := "001000010000";
when "011111000010" => result := "001000001111";
when "011111000011" => result := "001000001111";
when "011111000100" => result := "001000001111";
when "011111000101" => result := "001000001111";
when "011111000110" => result := "001000001110";
when "011111000111" => result := "001000001110";
when "011111001000" => result := "001000001110";
when "011111001001" => result := "001000001110";
when "011111001010" => result := "001000001101";
when "011111001011" => result := "001000001101";
when "011111001100" => result := "001000001101";
when "011111001101" => result := "001000001101";
when "011111001110" => result := "001000001100";
when "011111001111" => result := "001000001100";
when "011111010000" => result := "001000001100";
when "011111010001" => result := "001000001100";
when "011111010010" => result := "001000001011";
when "011111010011" => result := "001000001011";
when "011111010100" => result := "001000001011";
when "011111010101" => result := "001000001010";
when "011111010110" => result := "001000001010";
when "011111010111" => result := "001000001010";

```

Appendix C

```

when "011111011000" => result := "001000001010";
when "011111011001" => result := "001000001001";
when "011111011010" => result := "001000001001";
when "011111011011" => result := "001000001001";
when "011111011100" => result := "001000001001";
when "011111011101" => result := "001000001000";
when "011111011110" => result := "001000001000";
when "011111011111" => result := "001000001000";
when "011111100000" => result := "001000001000";
when "011111100001" => result := "001000000111";
when "011111100010" => result := "001000000111";
when "011111100011" => result := "001000000111";
when "011111100100" => result := "001000000111";
when "011111100101" => result := "001000000110";
when "011111100110" => result := "001000000110";
when "011111100111" => result := "001000000110";
when "011111101000" => result := "001000000110";
when "011111101001" => result := "001000000101";
when "011111101010" => result := "001000000101";
when "011111101011" => result := "001000000101";
when "011111101100" => result := "001000000101";
when "011111101101" => result := "001000000100";
when "011111101110" => result := "001000000100";
when "011111101111" => result := "001000000100";
when "011111110000" => result := "001000000100";
when "011111110001" => result := "001000000011";
when "011111110010" => result := "001000000011";
when "011111110011" => result := "001000000011";
when "011111110100" => result := "001000000011";
when "011111110101" => result := "001000000010";
when "011111110110" => result := "001000000010";
when "011111110111" => result := "001000000010";
when "011111111000" => result := "001000000010";
when "011111111001" => result := "001000000001";
when "011111111010" => result := "001000000001";
when "011111111011" => result := "001000000001";
when "011111111100" => result := "001000000001";
when "011111111101" => result := "001000000000";
when "011111111110" => result := "001000000000";
when "011111111111" => result := "001000000000";
when others => result := "000000000000";
end case;
return result;
end DIV;

BEGIN

xout <= STD_LOGIC_VECTOR(x6(31 DOWNT0 16));
yout <= STD_LOGIC_VECTOR(y6(31 DOWNT0 16));

```

Appendix C

```
zout <= STD_LOGIC_VECTOR(a6(31 DOWNT0 16));
```

```
PROCESS(clk)
```

```
begin
```

```
if (clk'event and clk='1') then
```

```
if (WriteState = '1') then
```

```
  a1 <= SIGNED(a);
```

```
  x1 <= SIGNED(x); y1 <= SIGNED(y);
```

```
      divin <= x1(31 downto 20);
```

```
      divout <= DIV(divin);
```

```
      a2 <= divout & "00000000000000000000";
```

```
      x2 <= x1;
```

```
      z <= a2(31 downto 16)*x1(31 downto 16);
```

```
      y2 <= y1 - z;
```

```
if (y2(15) = '0') then -- y2>0
```

```
  a3 <= a2 + SIGNED(To_StdLogicVector(bit_vector('x"0008"')));
```

```
  x3 <= x2;
```

```
      y3 <= y2 - SHR(x2,"1100");
```

```
else
```

```
  a3 <= a2 - SIGNED(To_StdLogicVector(bit_vector('x"0008"')));
```

```
  x3 <= x2;
```

```
      y3 <= y2 + SHR(x2,"1100");
```

```
end if;
```

```
if (y3(15) = '0') then -- y3>0
```

```
  a4 <= a3 + SIGNED(To_StdLogicVector(bit_vector('x"0004"')));
```

```
  x4 <= x3;
```

```
      y4 <= y3 - SHR(x3,"1101");
```

```
else
```

```
  a4 <= a3 - SIGNED(To_StdLogicVector(bit_vector('x"0004"')));
```

```
  x4 <= x3;
```

```
      y4 <= y3 + SHR(x3,"1101");
```

```
end if;
```

```
      if (y4(15) = '0') then -- y4>0
```

```
        a5 <= a4 + SIGNED(To_StdLogicVector(bit_vector('x"0002"')));
```

```
        x5 <= x4;
```

```
            y5 <= y4 - SHR(x4,"1110");
```

```
      else
```

```
        a5 <= a4 - SIGNED(To_StdLogicVector(bit_vector('x"0002"')));
```

```
        x5 <= x4;
```

```
            y5 <= y4 + SHR(x4,"1110");
```

```
      end if;
```

```
      if (y5(15) = '0') then -- y5>0
```


Appendix C

```
    a6 <= a5 + SIGNED(To_StdLogicVector(bit_vector'(x"0001")));
    x6 <= x5;
        y6 <= y5 - SHR(x5,"1111");
else
    a6 <= a5 - SIGNED(To_StdLogicVector(bit_vector'(x"0001")));
    x6 <= x5;
        y6 <= y5 + SHR(x5,"1111");
end if;

end if;
end if;

end PROCESS;

END hybrid_div_MOD11;
```

Appendix D

Appendix D

VHDL Model of Proposed Binary to Binary-logarithm Approximation Method

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;

ENTITY mitchell_imp IS

    PORT (
        Clk      : IN    std_logic;
                x      : IN    std_logic_vector(15 downto 0);
        xout     : out   std_logic_vector(15 downto 0)
    );
    -- Declarations

END mitchell_imp ;

-- hds interface_end

ARCHITECTURE mitchell_imp OF mitchell_imp IS

    signal k      : integer range 0 to 15;
    signal mantissa : std_logic_vector(11 downto 0);
    signal mant_cor : unsigned(11 downto 0);
    signal log     : unsigned(15 downto 0);

BEGIN
    lod: process(x)
    begin
        for i in 15 downto 0 loop
            if x(i)='1' then
                k <= i;
                exit;
            end if;
        end loop;
    end process lod;
    process(clk)

```

Appendix D

```

begin
    if (clk'event and clk='1') then
        if k=1 then
            mantissa <= x(0) & "000000000000";
            If (mantissa = "000000000000") then
                mant_cor <= "000000000000";
            else
                mant_cor <= unsigned(mantissa) +
"000101000000";
            end if;
            log <= "0010" & mant_cor;

        elsif k=2 then
            mantissa <= x(1) & x(0) & "0000000000";
            If (mantissa = "000000000000") then
                mant_cor <= "000000000000";
            else
                mant_cor <= unsigned(mantissa) +
"000100000000";
            end if;
            log <= "0010" & mant_cor;

        elsif k=3 then
            mantissa <= x(2 downto 0) & "0000000000";
            If (mantissa = "000000000000") then
                mant_cor <= "000000000000";
            else
                mant_cor <= unsigned(mantissa)+
"000011000000";
            end if;
            log <= "0011" & mant_cor;

        elsif k=4 then
            mantissa <= x(3 downto 0) & "00000000";
            If (mantissa = "000000000000") then
                mant_cor <= "000000000000";
            else
                mant_cor <= unsigned(mantissa) +
"000010100000";
            end if;
            log <= "0100" & mant_cor;

        elsif k=5 then
            mantissa <= x(4 downto 0) & "00000000";
            If (mantissa = "000000000000") then
                mant_cor <= "000000000000";
            else
                mant_cor <= unsigned(mantissa) +
"000010000000";

```

Appendix D

```

        end if;
        log <= "0101" & mant_cor;

    elsif k=6 then
        mantissa <= x(5 downto 0) & "000000";
        If (mantissa = "000000000000") then
            mant_cor <= "000000000000";
        else
            mant_cor <= unsigned(mantissa) +
"000010000000";
        end if;
        log <= "0110" & mant_cor;

    elsif k=7 then
        mantissa <= x(6 downto 1) & "000000";
        If (mantissa = "000000000000") then
            mant_cor <= "000000000000";
        else
            mant_cor <= unsigned(mantissa) +
"000010000000";
        end if;
        log <= "0111" & mant_cor;

    elsif k=8 then
        mantissa <= x(7 downto 2) & "000000";
        log <= "1000" & unsigned(mantissa);

    elsif k=9 then
        mantissa <= x(8 downto 3) & "000000";
        log <= "1001" & unsigned(mantissa);

    elsif k=10 then
        mantissa <= x(9 downto 4) & "000000";
        log <= "1010" & unsigned(mantissa);

    elsif k=11 then
        mantissa <= x(10 downto 5) & "000000";
        log <= "1011" & unsigned(mantissa);

    elsif k=12 then
        mantissa <= x(11 downto 6) & "000000";
        log <= "1100" & unsigned(mantissa);

    elsif k=13 then
        mantissa <= x(12 downto 7) & "000000";
        log <= "1101" & unsigned(mantissa);

    elsif k=14 then
        mantissa <= x(13 downto 8) & "000000";

```


Appendix D

```
        log <= "1110" & unsigned(mantissa);

    elsif k=15 then
        mantissa <= x(14 downto 9) & "000000";
        log <= "1111" & unsigned(mantissa);
    else
        log <= "0000000000000000";
    end if;
end if;
end process;

xout <= std_logic_vector(log);

END mitchell_imp;
```