Efficient Arithmetic for High Speed DSP Implementation on FPGAs

Steven Wilson Alexander (BEng/MSc)

A thesis submitted to

the Universities of

Edinburgh

Glasgow

Heriot-Watt

Strathclyde

For the Degree of

Doctor of Engineering in System Level Integration

Author's Declaration

I declare that no portion of the work in this thesis has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

I also declare that the work presented in this thesis is entirely my own contribution unless otherwise stated.

Steven W. Alexander

Abstract

This thesis presents the work that was carried out by the author to obtain the degree of Doctorate of Engineering (EngD). The author was sponsored for the duration of the degree by EnTegra Ltd, a company who develop hardware and software products and services for real time implementation of DSP and RF systems.

The field programmable gate array (FPGA) is being used increasingly in the field of DSP. This is due to the fact that the parallel computing power of such devices is ideal for today's truly demanding DSP algorithms. Algorithms such as the QR-RLS update are computationaly intensive and must be carried out at extremely high speeds (MHz). This means that the DSP processor is simply not an option. ASICs can be used but the expense of developing custom logic is prohibitive.

The increased use of the FPGA in DSP means that there is a significant requirement for efficient arithmetic cores that utilise the resources on such devices. This thesis presents the research and development effort that was carried out to produce fixed point division and square root cores for use in a new Electronic Design Automation (EDA) tool from EnTegra, which is targeted at FPGA implementation of DSP systems. Further to this, a new technique for predicting the accuracy of CORDIC systems computing vector magnitudes and cosines/sines is presented. This work allows the most efficient CORDIC design for a specified level of accuracy to be found quickly and easily without the need to run lengthy simulations, as was the case before. The CORDIC algorithm is a technique using mainly shifts and additions to compute many arithmetic functions and is thus ideal for FPGA implementation.

Acknowledgements

There are many people that I would like to thank for their help and support during the EngD degree. Firstly I owe a huge thank you to Professor R.W. Stewart who has been my academic supervisor for the past four years. The opportunities that he has presented to me and the experience that I have gained are invaluable in terms of personal and career development. I'd also like to thank my industrial supervisor Tim Bigg and EnTegra Ltd. for their support during the degree. A big thanks must also go to Sandie Buchanan and Sian Williams for their help and guidance in all EngD matters.

I have made some fantastic friends and colleagues during the EngD, each of which deserves thanks for either helping with some problem or simply providing some needed distraction. Everyone in Glasgow - Daniel, Iain, Garrey, Graham F, Jamie, Graham S, Louise, Karen, Amreet, Faisal, Ken and Neil - thanks to each of you, a great bunch of people! I'd also like to say a special thank you to Eugen Pfann for all his help with reviewing papers and reports as and when it was needed. His advice and direction has had a huge benefit to this work.

My family have been a great source of help and inspiration over the years. Dad, Jane, Alistair, Jen, Sean and Sandy - I thank you all for your love and support. I owe a special mention to my Mum who has worked so hard to provide for her family. She is the most selfless person I know and I owe so much of my success to her.

Finally I want to thank Kerry for supporting me throughout the EngD in what has been a challenging experience at times. Her sense of humour and ability to make me laugh have been a major help during this time. She is my best friend and partner and I couldn't have completed this work without her.

Acronyms & Symbols

ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
AMS	Analogue Mixed Signal
ASIC	Application Specific Integrated Circuit
CDMA	Code Division Multiple Access
CLB	Configurable Logic Block
CORDIC	COrdinate Rotational DIgital Computer
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ESL	Electronic System Level
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HDS	HDL Design Studio
IC	Integrated Circuit
IIR	Infinite Impulse Response
IOB	Input Output Block
IP	Intellectual Property
LAB	Logic Array Block

LMS	Least Mean Squares
LSB	Least Significant Bit
LUT	Look Up Table
MAC	Multiply Accumulate
MSB	Most Significant Bit
OQE	Overall Quantisation Error
PFU	Programmable Functional Unit
RE	Research Engineer
RF	Radio Frequency
RLS	Recursive Least Squares
ROM	Read Only Memory
RTL	Register Transfer Level
SOC	System On Chip
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integration

Contents

Chapter 1

Introduction

The industry sponsor for the duration of this EngD (October 2001 – October 2005) has been EnTegra Ltd. EnTegra develop hardware and software products and services for real time implementation of DSP and RF systems. The majority of the EngD project was spent at Strathclyde University under the guidance of Professor Bob Stewart, academic supervisor. Regular contact was maintained with the industrial supervisor Tim Bigg throughout the project.

1.1 Project Timeline

Figure 1.1 shows the activities that were carried out during the four years of the EngD project. During the first 10 months, 108 credits worth of technical modules were completed at ISLI. The final 12 credits required to obtain the necessary 120 were completed in early 2002. A further requirement of the EngD programme is that each student must obtain 60 credits of MBA level modules. Heriot-Watt Business School offers intensive MBA modules where the taught element can be completed in 6 full days rather than an entire semester. Hence, the decision was taken to study the business modules here to minimise the time spent away from the project. During 2004, three modules totalling 60 credits were completed.

In June 2002 work began on the project under the supervision of Professor

Stewart at Strathclyde University. Working from here, the research and development effort began into efficient algorithms for high speed DSP implementation on FPGAs. The first task was to research the CORDIC algorithm as it was felt that this technique had great potential in DSP. This has since been proven correct as downconverters using CORDIC have been seen in the market [45] as well as commercial FPGA EDA tools containing CORDIC cores [52]. The result of this research was a comprehensive report detailing the CORDIC algorithm. Through this examination a major problem with the CORDIC technique was found with respect to its use in fixed point DSP systems. The only way to find the worst case error in a fixed point CORDIC system was to run lengthy simulations and compare the output to a reference solution, which is not an ideal situation. In some DSP algorithms it is vitally important that the worst case error is known. This problem was to be solved later in the project.

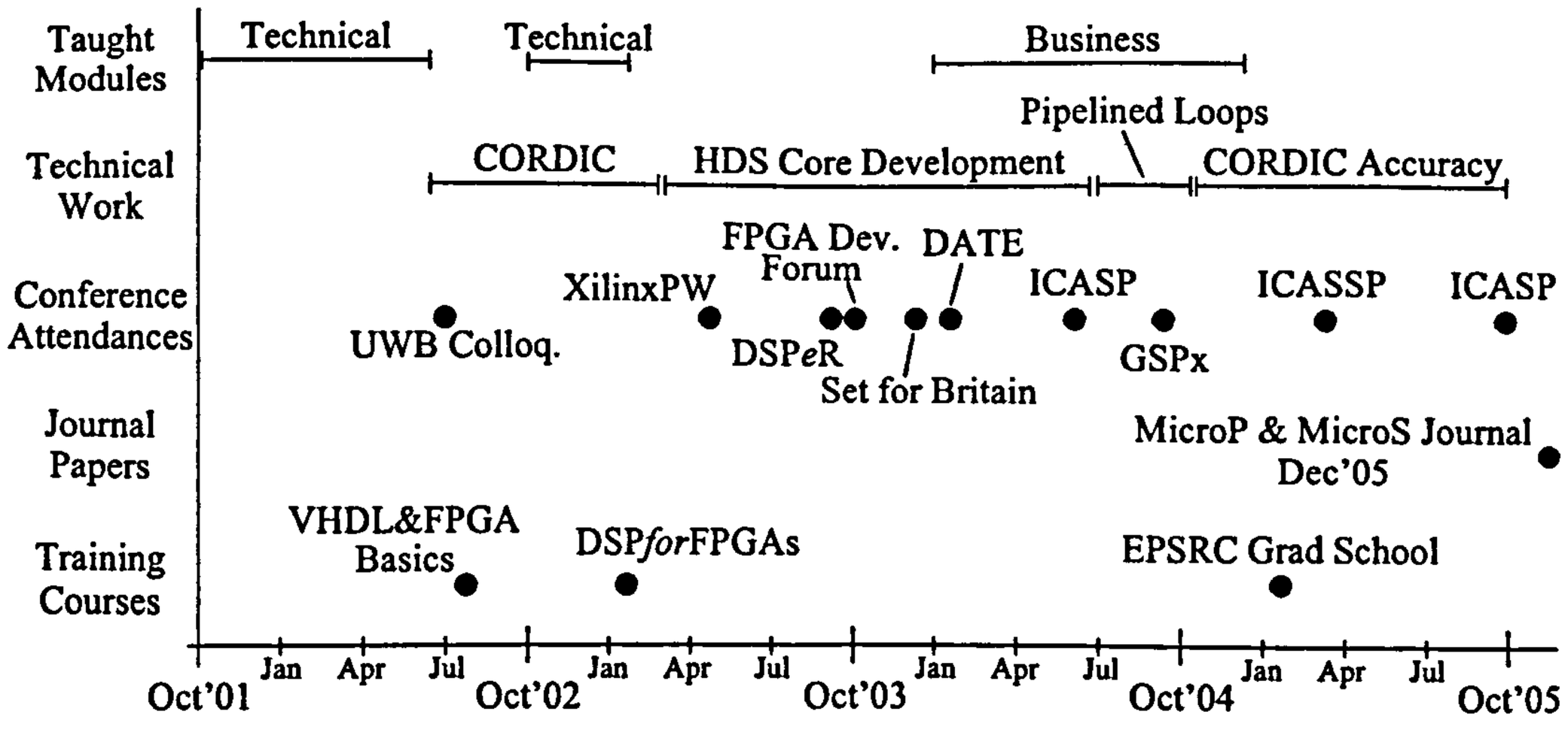


Figure: 1.1: Project Timeline

Following the CORDIC analysis, a major new product development began at EnTegra. A new software package called HDL Design Studio (HDS) [44] was to be developed which would work as a plugin to the existing DSP simulation tool SystemVue [50]. The idea behind HDS was that it would allow SystemVue simulations to be automatically converted into bit and cycle equivalent hardware

designs. Thus, the DSP development lifecycle could be reduced considerably as no hand coded hardware description language (HDL) would need to be written. The EngD role in this project was to develop Division and Square Root cores for inclusion in HDS. Hence, software and hardware functions for efficient implementation of a fixed-point Divider and Square Rooter were developed and verified.

The use of pipelined IP in feedback loops was a topic that had been debated many times amongst several colleagues at EnTegra and Strathclyde University during the EngD. Hence, the project supervisors agreed that a short paper should be written where an investigation into the validity of this technique was investigated and presented. This work had real value as many DSP algorithms use feedback. Also, pipelined IP cores are frequently provided in today's DSP EDA tools. The decision whether to use these cores in a pipelined or non-pipelined format within a feedback loop is not a trivial decision as was found during the investigation. The findings of this work were that non-pipelined feedback loops have a greater throughput, use less logic and consume less power than a pipelined equivalent. The results were presented at ICASSP 2005 in Philadelphia, PA.

The final piece of work that was undertaken during the EngD was to return to the CORDIC algorithm to try and solve the accuracy problem. By building on the work presented in a paper published in 1992 [18] a closed form solution was developed which allows the accuracy of fixed-point CORDIC systems to be found. Hence, rather than using the traditional trial and error approach, this breakthrough allows engineers to quickly find the defining parameters for a fixed-point CORDIC system computing an output to a desired level of accuracy. The initial work that was developed here was accepted in late 2005 for publication in the Journal of Microprocessors and Microsystems.

In addition to the research and development effort that was carried out during the EngD, several training courses and conferences were attended. Each of these events helped to improve the EngD students technical skills as well as his personal effectiveness.

1.2 Thesis Organisation

This document discusses the main outcomes of the EngD as well as the commercial and technical background to which the work was carried out. Also, details are given with respect to the compulsory aspects of the EngD, such as the completion of the technical and business modules.

The rest of this document is organised as follows. In Chapter 2, the Taught modules, both technical and business, are discussed along with the reasons for their selection. Following this, in Chapter 3 all publications that were accepted either for conference or journal publication are listed with a brief summary of each. In Chapters 4 and 5 the Commercial and Technical background to which the EngD work was carried out is set. Chapters 6 - 8 present the research and development effort during the EngD in some detail. Finally, the Conclusions are given in Chapter 9. Note that in Chapter 6, the screen dumps of any HDS tokens show the logo of a company called Steepest Ascent Ltd. [49] rather than EnTegra. This is because Steepest Ascent have taken ownership of the product since the EngD students involvement.

1.3 Contribution to Knowledge

The work presented in this thesis comprises a thorough analysis and discussion of the EngD project titled "Efficient Arithmetic for High Speed DSP Implementation on FPGAS". The following elements are regarded as novel contributions to knowledge and contributions to practical knowledge.

1.3.1 Novel contribution to knowledge

- An extremely accurate equation for predicting the Overall Quantisation Error (OQE) experienced by fixed point CORDIC systems computing vector magnitudes has been developed and verified. The equation is derived in terms of the number of iterations n and the number of fractional bits used in the data path

b. By using the equation it is now possible to find the combination of *n* and *b* resulting in the least hardware required to generate a desired level of accuracy from the output of a fixed point CORDIC system computing vector magnitudes.

- An OQE equation has been developed to predict the accuracy of fixed point CORDIC systems computing sines and cosines. Again the OQE is derived in terms of the number of iterations *n* and the number of fractional bits used in the data path *b*. This equation can be used to find the minimal set of parameters required by such systems to guarantee a desired level of accuracy from the cosine/sine output.

1.3.2 Contribution to practical knowledge

- Using direct methods of computation, fixed point divider and square root cores were developed. These cores are capable of generating bit accurate solutions when compared with floating point equivalent functions that are truncated to the same level of precision. They also have the added benefit of producing one bit of the solution per iteration and hence it is easy to know how many iterations it will take to guarantee a desired level of accuracy.

- An analysis of feedback loops has been carried out where the merits of pipelining such a structure has been considered. This work has shown that pipelining only serves to reduce the data throughput, increase the resource usage and use more power than when compared to a non-pipelined feedback loop.

Chapter 2

Taught Modules

In this section, the Technical and Business modules that were undertaken during the EngD degree are discussed. The Technical modules were all taken at the Institute for System Level Integration in Livingston over the first two years of the degree. Normally the technical modules are completed in the first year of the degree, however the RE started the EngD approximately one month late. Consequently a first term module was deferred until the second year to allow the remaining first term classes to be successfully caught up with.

The Business modules were all taken at Heriot-Watt University. The reason for choosing this university was the option to attend the classes during the weekends, which meant that it was possible to complete the Business modules while minimising the time spent away from the EngD project.

2.1 Technical Modules

The following Technical modules were taken at ISLI from October 2001 to May 2002. As mentioned already, an optional module (Microprocessors and Microcontrollers) was deferred until October 2002 to reduce the workload in the first term while progressing with the remaining classes.

Foundation Modules	Credits
Introduction to Hardware Design Automation	6
Introduction to Embedded Software Engineering	6
Compulsory Modules	
Embedded Software 1 (System on Chip)	6
VLSI Design	12
IP Block Authoring	12
IP Block Integration	12
System Partitioning	12
Optional Modules	
Embedded Software 2 (Operating Systems)	12
Communications Algorithms	12
Mobile Communications	6
Broadband & Digital Networks	6
Multimedia & Video	6
Microprocessors & Microcontrollers	12
	120

Table 1: Technical Module Breakdown

2.2 Business Modules

A requirement of each RE is that they complete 60 credits of Master of Business Administration (MBA) modules. These can be taken at any of the four participating EngD Universities. Heriot-Watt was chosen as they offer taught MBA classes during the weekends. Table 2 lists the modules that were taken and completed successfully during 2004.

These modules were chosen as they offered the best fit to the EngD project. Marketing was important as at the time EnTegra were developing a new product in

HDS. In addition to this, the ability to manage projects now and in the future whilst understanding the financial implications was of course important. Hence, Accounting and Project Management were chosen.

Business Modules	Credits
Accounting	20
Project Management	20
Marketing	20
	60

Table 2: Business Module Breakdown

Chapter 3

Publications

In this section, the publications submitted and accepted during the EngD degree are given.

3.1 Poster Presentations

- **Rapid Prototyping of DSP Systems for FPGA Implementation Using HDL Design Studio** - this paper was presented at the *Institute for Communications and Signal Processing Research Colloquium*, Glasgow in June 2004, which was run by the EEE department at Strathclyde University. The paper focused on the development of a new Electronic Design Automation (EDA) tool which EnTegra were developing. HDL Design Studio (HDS) was designed as a plugin to the SystemVue DSP software simulation package. Using HDS, bit/cycle accurate VHDL designs ready for hardware implementation can be rapidly generated from SystemVue software simulations. The paper illustrated the benefits of the tool, including reduced design times via a design example.

- **The Effects of Pipelining Feedback Loops in High Speed DSP Systems** - this paper was presented at the *International Conference on, Acoustics, Speech and Signal Processing (ICASSP)*, Philadelphia, PA, USA in March 2005. The purpose of this paper was to present findings that showed feedback loops

containing IP that is pipelined are slower in terms of data throughput and consume more power than equivalent non-pipelined loops. This is significant for two reasons. Firstly, many of today's DSP EDA tools supply ready pipelined IP cores and secondly these cores are increasingly being used in Adaptive Equalisation systems which often require feedback loops. The work involved in this paper involved using HDS which provided an opportunity to market the tool at the conference.

3.2 Oral Presentations

- **HDL Design Studio - An Integrated Design Flow for the Implementation of DSP Systems on FPGAs** - this paper was presented at the *Global Signal Processing Expo (GSPx)*, Santa Clara, CA, USA in September 2004. In this paper HDL Design Studio was presented in more detail with a more complex design example (adaptive LMS equaliser). Some additional features were also presented, including the fixed point analysis tools. These tools allow the number of overflows and underflows resulting from a simulation to be observed. Also, the maximum and minimum values from a specific output can also be observed along with the required range and precision that is necessary to represent these numbers fully. These features are unique to the DSP EDA tool market and are extremely useful for designing DSP systems that demand numerical stability and integrity to be maintained.

- **An Improved Algorithm for Assessing the Overall Quantisation Error in CORDIC Systems Computing a Vector Magnitude** - this paper was presented at the *Institute for Communications and Signal Processing Research Colloquium (ICASP)*, Jordanhill, Glasgow in October 2005. The focus of this paper was on analysing the accuracy of the output from a CORDIC system computing a vector magnitude. In DSP it is vital that the accuracy of signals is known if numerical

stability and integrity is to be maintained. There was very little work covering this aspect of the CORDIC algorithm and traditionally engineers simply used a trial and error approach to find a system that produced an output with enough accuracy. Clearly this was not an ideal situation. However, by building on work presented in [18] an accurate formula was developed for finding efficient CORDIC systems to compute the magnitude of a vector to a specified accuracy.

- **Tools for Implementation of DSP Functionality in FPGAs: CORDIC Vector Magnitude Calculation Using HDS3** - this paper was presented at the *Electronica Conference*, Munich, Germany in November 2006. In this paper the operation of the CORDIC algorithm and specifically how CORDIC can be used to calculate vector magnitudes is explained. The HDS3 design software is introduced (updated version HDS) and selected CORDIC implementations are developed in HDS3 using a variety of structures. These structures are synthesised for an FPGA target and the performance and resource usage presented. A single CORDIC cell was also synthesised as an example of targeting a CPLD target.

3.3 Journal Publications - Confirmed

- **An Improved Algorithm for Assessing the Overall Quantisation Error in FPGA Based CORDIC Systems Computing a Vector Magnitude** - this paper was accepted in December 2005 for publication in the *Special Issue on FPGA-based Reconfigurable Computing, Journal of Microprocessors and Microsystems*. Note that it is not due to be published until early 2007. This paper covered the work that was presented at *ICASP '05* in more detail. In addition to discussing the derivation of a formula for assessing the error in CORDIC systems computing a vector magnitude, the improvements in terms of FPGA hardware utilisation and clock speed were illustrated compared with other

techniques for computing the same function.

3.4 Journal Publications - Awaiting Notification

- **Assessing The Overall Quantisation Error In CORDIC Systems Computing Cosines And Sines -** this paper was submitted in January 2007 to the *IEEE Transactions on Circuits and Systems II*. This paper presented the work that was carried out to find the most efficient CORDIC systems for computing cosines and sines to a required level of accuracy.

Chapter 4

Commercial Relevance

In this section, the commercial background to which the EngD project was carried out will be discussed. Further to this, the contribution that the project has made towards the commercial interests of EnTegra Ltd. and the research community will also be highlighted.

4.1 IC Design - The Current Situation

In 1965 Gordon Moore [25] predicted that, for the foreseeable future, the number of transistors on an integrated circuit would double every 12 - 24 months. Over 40 years later the trend that he predicted still holds true, although for how much longer no one really knows. However, the rapid progress of IC process technology during these 40 years has resulted in a phenomenon known as the "Design Gap" [5][7][16]. This gap is the difference between the total number of transistors on current IC's and the number of transistors that are actually utilised in today's designs as show in Figure 4.1. There are several contributing factors to this gap which include mask costs, time to market demands and increased system complexity. Together, these factors mean that today's engineers cannot design and verify systems that utilise the full capability of current IC's in a time that gets the product to market fast enough.

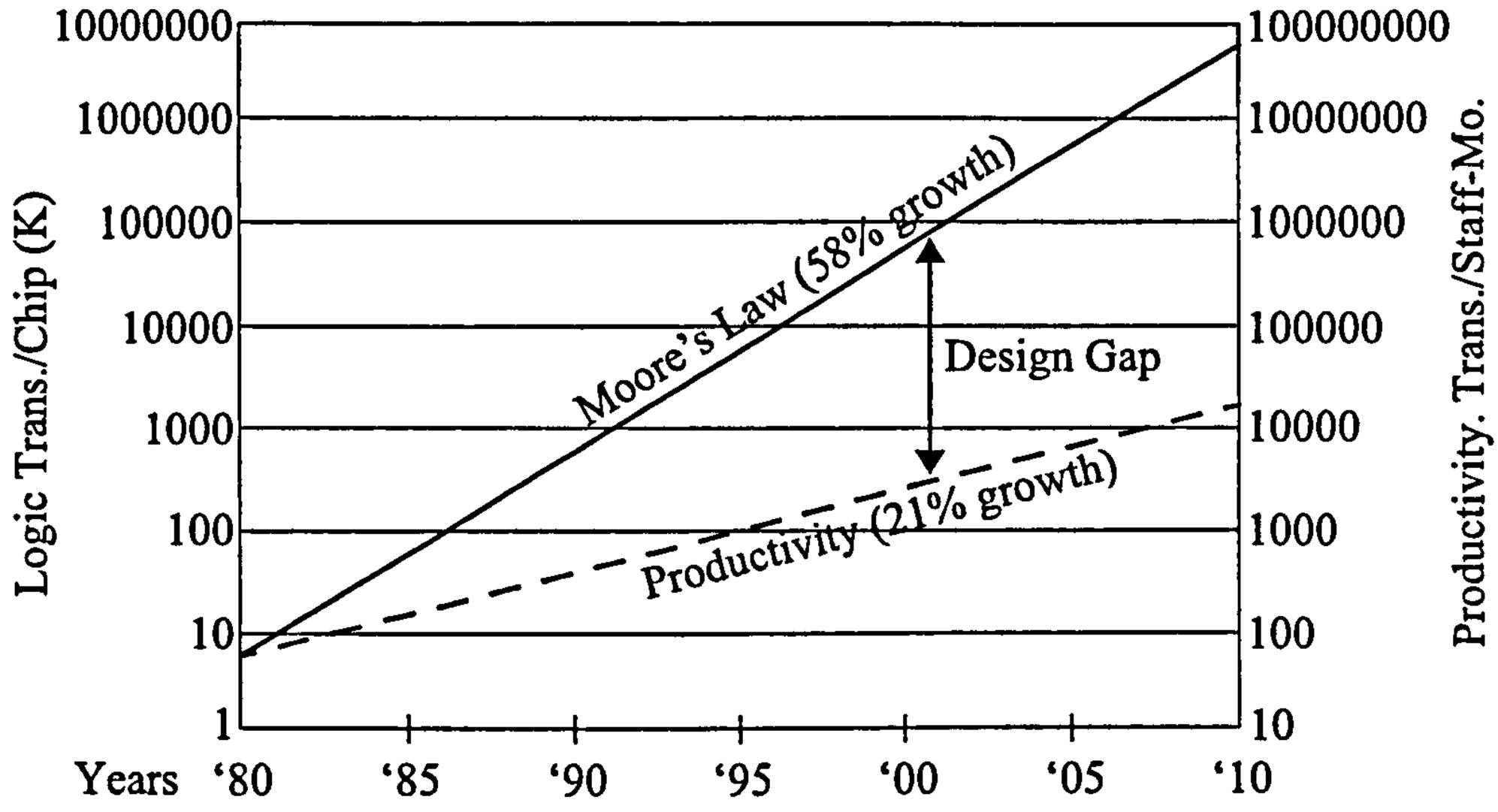


Figure: 4.1: Design Complexity versus Designer Productivity

At present, the solution to the design gap is believed to involve changes in all three interacting dimensions of the design environment. These are:

- **Design IP** - these are the building blocks of a design. Traditionally these are written in an HDL (hardware) or C/Assembly (software).

- **Design Tools** - these are the application programs and techniques that designers use to capture, verify, refine, and translate design descriptions for particular tasks and subsystems. Historically, tools such as RTL compilation and verification, code assemblers and compilers, and standard-cell placement and routing have comprised the essential tool box for complex chip design.

- **Design Methodology** - is the design team's strategy for combining the available IP and tools into a systematic process for implementing the target silicon and software. A methodology specifies which elements and tools are available, describes how the tools are used at each step of the design refinement, and outlines the sequence of design steps. Typically the sequence involves four steps in the following order: hardware-software partitioning, detailed RTL block

design and verification, chip integration of RTL blocks, processors and memories, and post-silicon software bring-up.

In the next section the commercial interests of EnTegra Ltd. with respect to today's design tools are discussed.

4.1.1 Electronic Design Automation (EDA) Tools

There are many different types of electronic design tool that come under the EDA label. For example, System On Chip (SOC) designers may use Electronic System Level (ESL) tools. These tools allow software and hardware to be partitioned and designed within the one tool. Often these tools use extensions to the C/C++ languages or variations of these. These include Handel-C [43], SystemC [51] and SpecC [48]. Other areas of design include Analogue Mixed Signal (AMS), Radio Frequency (RF) and Electro Magnetic (EM). However, Digital Signal Processing (DSP) is the area that EnTegra Ltd. operate in and hence this shall be the focus here.

The traditional approach to DSP system design is to use a simulation package such as SystemVue or Simulink [47] to build and verify a system before creating an equivalent software design by writing C code to target a DSP processor. Alternatively, an equivalent hardware design is created by writing VHDL/Verilog. However, this method is becoming less common. The reason for this is that Field Programmable Gate Arrays (FPGAs) are being used in DSP more than ever before, meaning that support for this design flow is increasing. Consequently several new tools have now entered the market [55][40] which allow bit/cycle accurate FPGA designs to be automatically generated from software simulations. This means that design times have been dramatically cut due to the fact that HDL code does not need to be hand crafted anymore [9]. So, where previously it may have taken several weeks/months to hand code an HDL design and test, verify and implement it, this process is now automated and takes hours/days. There are arguments that suggest that well constructed hand crafted HDL code will generally outperform automatically generated code. However,

the response to this tends to be that the amount of time saved writing code can be spent exploring more of the design space to find an optimal solution that might otherwise not have been found using a traditional approach.

In Figure 4.2 it is clear that making changes to a design using the traditional flow takes a significant amount of time. First, the software simulation has to be altered and checked against the original specification. The main time consumer is the manual coding of the altered design, which of course must be re-implemented, tested and debugged. With today's design flow, once the software simulation has been altered, the new HDL design is generated in minutes. The added bonus of this approach is that the generated hardware designs are bit and cycle accurate compared to the original software model. Also, testbenches can be automatically generated, thus verification takes much less time.

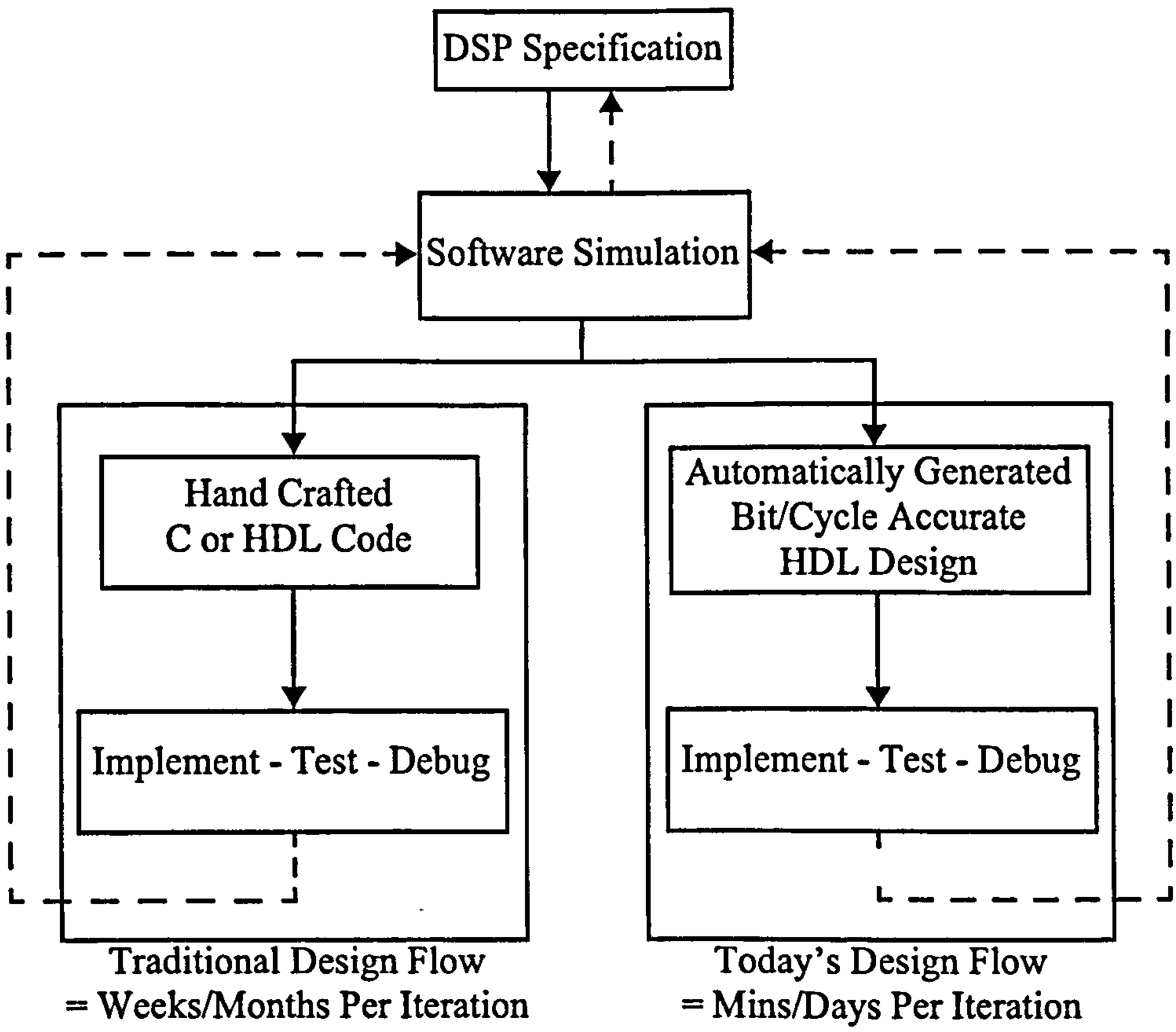


Figure: 4.2: Traditional DSP Design Flow versus New Design Flow

4.1.2 Commercial Interest

SystemVue is a DSP/RF software simulation package that was originally developed by Elanix, Los Angeles, CA. However, in recent years due to company buyouts, its ownership has moved to Eagleware and more recently Agilent Technologies [50]. For several years now EnTegra Ltd. have been involved with SystemVue as a UK distributor. In addition to this, they have also developed several DSP training courses that teach and use SystemVue during laboratory sessions. However, a major part of EnTegra's involvement with SystemVue has been in the development of several software libraries for use with the tool. These include 3G, CDMA2000, Equaliser and Adaptive libraries [42].

One of the major competitors to SystemVue is Simulink from The Mathworks [47]. Simulink is a block based software simulation package, similar to SystemVue, which is purchased as a plugin to the popular Matlab tool. However, Simulink differs from SystemVue in that it supports many different engineering disciplines. It is tailored to each industry through the purchase of blocksets which provide industry specific functionality. So for example, to build DSP systems would require at least the Signal Processing blockset to be purchased.

In 2001 The Mathworks, a leader in modelling and simulation software, started a partnership with the leading FPGA vendor, Xilinx. This partnership involved creating a new design flow that allowed Simulink DSP systems to be automatically generated into equivalent HDL designs targeted at Xilinx FPGAs. The result of this collaboration was System Generator [55]. This new tool was basically a Xilinx blockset for use with Simulink. By building DSP systems using Xilinx blocks it was possible to generate equivalent VHDL or Verilog designs for any Xilinx chip. Shortly after System Generator was released, a similar tool called DSP Builder [40] was developed. This was the offering from the other major FPGA vendor, Altera, which again was available as a blockset for use with Simulink.

Both of these tools have a common feature, which is that neither of them generate HDL code that can be used independently of the tool. Instead, a top level HDL

file is generated which instantiates electronic design files (edif) for each of the components in the design. This means for example, that if a multiplier block is used in the design, then it is represented using an edif file instead of actual HDL code. The reason for this is that it protects the IP of each core. It is extremely difficult to obtain useful information regarding IP from an edif file.

Another common feature of both tools is that neither of them have support for integrating the rest of the system with the DSP component. System Generator and DSP Builder are aimed at DSP design, however in a real system this is only part of the overall system. Once the DSP section has been designed it must be integrated with the rest of the system.

These issues were brought to the attention of EnTegra through speaking to engineers who had used the software. When EnTegra decided to develop their own tool, this information was crucial to the design of HDS.

4.2 DSP - From Sequential Processors To Parallel Arrays

Traditionally, specialised processors have been used for implementing DSP algorithms in software. These specialised processors differ from CPUs in that they often have single cycle instructions that are particularly useful in DSP, such as multiply-accumulate. The problem with this type of DSP implementation is processing throughput. A processor can only carry out one instruction at a time, each of which consumes clock cycles. As there are a fixed number of clock cycles between each data sample arriving, there is a limit to the number of instructions that can be executed before the next sample arrives. This limitation has meant that many DSP algorithms simply could not be implemented with this type of technology due to the high data processing requirements that they have. The solution to this problem is either to increase the clock speed so that more instructions can be executed in between samples or move to a parallel implementation where more than one function is computed during a clock period. Unfortunately technology limits the speed at which modern DSP

processors can be clocked. This leaves a parallel implementation as the solution.

Application Specific Integrated Circuits (ASICs) offer a high degree of parallelism, however the significant investment required to develop and manufacture such devices means that they cannot always be used. Fortunately, FPGAs have the parallelism required as well as several other attributes that are ideal for use in DSP systems. Over the last twenty years they have evolved from simple devices that were used as glue logic to the sophisticated multi-functional devices that exist today. This has had a significant effect on the DSP industry and its markets.

4.2.1 FPGAs For DSP

Today's FPGAs have many resources specifically targeted at DSP design. These include dedicated embedded multipliers, fast carry logic, flexible memory and more recently, embedded processors within the FPGA fabric. As has already been discussed, DSP has been dominated by the microprocessor for many years, which has led to engineers developing algorithms with a software implementation in mind. However, a rethink is now required as there are alternative approaches to implementing many algorithms which are more suited to a hardware implementation. One such approach is the CORDIC algorithm [33][34] which has been around since the 1950's. This technique can be used to compute many different functions using only shifts, additions/ subtractions and table look-ups. Hence, it is ideal for implementing on FPGAs.

Another area where FPGAs have created an opportunity is in the development of truly demanding DSP techniques. The serial nature of DSP processors has already been highlighted. FPGAs permit the design of computationaly intensive systems such as Adaptive Equalisers. There are two techniques which are often associated with Adaptive Equalisation, which are the Least Mean Squares (LMS) [37][38] and the Recursive Least Squares (RLS) [13][15][23]. The order of the computational requirements for these techniques where the filter length is N is:

LMS: $O(N)$ MACs/s *RLS:* $O(N^2)$ MACs/s

Clearly, the RLS technique requires significantly more computations than the LMS. However, this is not the only issue for the RLS as it also requires division and square root functions. These functions alone have traditionally been avoided in DSP due to their high computational requirements. However, there is significant commercial interest in RLS systems as they outperform LMS systems by equalising much faster and producing a cleaner output.

4.3 Project Contribution

The EngD project has made a substantial contribution to the commercial interest of EnTegra Ltd. and the DSP research community. This has been achieved through the following mini-projects which combine to make up the thesis. Each of these mini-projects are covered in detail in later chapters.

4.3.1 HDL Design Studio IP Development

As has been discussed already, EnTegra have been involved in developing a new software package called HDL Design Studio (HDS). The benefit of this tool is that it is used as an additional library within the SystemVue software simulation tool. By designing systems using this library it is possible to automatically generate equivalent hardware designs therefore removing the need to hand-code VHDL/Verilog.

The design of HDL Design Studio has involved creating two separate IP repositories. The first contains the IP for simulating any HDS functions within SystemVue. This IP is written using the C++ language. The second repository contains the equivalent hardware functionality. This IP is written using VHDL. Hence, for each C++ function there is an equivalent bit/cycle accurate VHDL representation.

Many of the functions that were written for the HDS repositories were

minimal, such as addition and subtraction. However, division and square root are two of the more complicated functions and these were assigned to the EngD project. From a business perspective these functions were significant for two reasons. The first was that neither Xilinx or Altera had direct division or square root functions within their tools. The second, and probably the most important reason for developing these cores, was that Adaptive Equalisation techniques are becoming more and more popular due to the fact that the technology now exists to implement them. These techniques often require division and square root functionality and hence, the inclusion of such cores within HDS would hopefully appeal to engineers working in this area.

A significant amount of the EngD project was spent developing, testing and upgrading the division and square root cores. However, they are now embedded within HDL Design Studio and are part of the commercial product that is available on the market today.

4.3.2 Analysis Of Pipelined Feedback Loops

Feedback loops are a common feature of Adaptive Equalisation techniques. A common feature of FPGA systems is that they are often pipelined to increase the clock rate. This effectively costs nothing on an FPGA as registers are in plentiful supply. However, should pipelining be used in a feedback loop? This was a question that had been debated several times amongst colleagues at Strathclyde University and EnTegra. With FPGAs being increasingly used for Adaptive Equalisation it was felt that this was an ideal time to try and answer this question by using HDL Design Studio. This of course had the added benefit of generating publicity for HDS should any of the work lead to a publication.

Through experiment and analysis it was found that feedback loops containing pipeline registers operate with a slower throughput, consume more power and use more resources than feedback loops without pipelining. This was an important finding and was significant in the development of the IP for HDS. Until this point, much of the

IP developed for HDS, as with it's competitors, was pipelined. However, based on these findings, some of the cores, including the division and square root cores, were updated to allow all pipeline registers to be removed. Hence, should they be used within a feedback loop, an optimal design could be realised. This work also led to a paper being accepted for a conference publication at the *International Conference on, Acoustics, Speech and Signal Processing* (ICASSP), Philadelphia, 2005, which also offered an ideal opportunity to promote HDS to the research community.

4.3.3 CORDIC Quantisation Error Analysis

The CORDIC algorithm is a technique that can be used to compute many different functions. It is ideal for FPGA implementation because it requires mainly shifts and additions/subtractions which are easily achieved on this type of device. For this reason, both Xilinx and Altera have included CORDIC cores within their tools.

The problem with the CORDIC algorithm is that there has been very little work done to assess the accuracy of the output it generates. It is easy to setup a CORDIC core that computes a desired function but there is no method of assessing its accuracy other than running time consuming simulations and comparing the output to a reference design. To overcome this problem, a project was undertaken to develop a formula for quantifying the error in such systems. If this could be achieved, the need to run tedious simulations could be avoided, thus saving time. Also, it would allow the most efficient CORDIC design to found for a specific level of accuracy. A further benefit of this work would be the ability to create a CORDIC core where all that was required from the user was to specify the desired function and a level of required accuracy. Using the error analysis work, the core could then automatically configure itself to use as few resources as possible to achieve this. None of the tools that compete with HDS have anything as sophisticated as this, and hence it would be a significant addition to the software.

The result of this project was that an algorithm was successfully developed to

predict the accuracy of CORDIC systems computing vector magnitude and sine/cosine calculations. Unfortunately, due to time restrictions the work has not been carried out to its full potential. There are many other CORDIC functions that have yet to be analysed. However, the methods that have been applied so far could be extended to the remaining CORDIC functions. Also, it is still possible that this work will lead to a new core within HDS, which will generate efficient CORDIC systems based on the derived algorithm. Further to this, the work that has been completed so far has resulted in the first journal publication of this EngD project. In early 2007, the paper titled "An Improved Algorithm for Assessing the Overall Quantisation Error in FPGA Based CORDIC Systems Computing a Vector Magnitude" will be published in the *Special Issue on FPGA-based Reconfigurable Computing, Journal of Microprocessors and Microsystems*.

Chapter 5

Technical Background

In this section the technical background is discussed to illustrate why the work that has been carried out is relevant, valuable and solves a real industrial problem. Much of the technical background focuses on the development of the FPGA, the effect this has had on the DSP industry and how it has enabled the realisation of computationaly demanding tasks such as adaptive equalisation using the least squares technique.

5.1 The Limit Of A DSP Processor

A processor has to use sequential clock cycles to perform an algorithm. Usually it takes one clock cycle for every operation that must be carried out in an algorithm. This means that there is a limit to the number of operations that can be carried out on one data sample before the next one arrives. This limit defines the operating envelope for a processor.

Table 3 illustrates the comparison between an Arithmetic Logic Unit (ALU) with a single multiplier clocked at 100MHz against an ALU with 2 multipliers clocked at 200MHz. By comparing the number of operations that both ALU's can complete per sample period it is clear that the bigger ALU is 4 times more capable.

For applications where the sampling rate is relatively low, like Audio for example (8KHz - 20KHz) a DSP processor is an excellent option. However, for a

control loop algorithm such as an FIR filter with 512 weights, it becomes unusable. Some modern communications systems (CDMA chip rates) use sample rates of 1.2288MHz and 3.84MHz [30], which means that only 26 operations can be carried out between samples. This is not enough to do anything really useful. For technologies such as Video (27MHz) and HDTV (74MHz) there is very little that can be done with a processor. Hence, for high sample rate applications a more parallel approach is required and this is where FPGAs excel in DSP.

Sample Rate	Operations per sample period	
	Single Multiplier ALU @ 100 MHz	Two Multipliers ALU @ 200 MHz
8KHz	12,500	50,000
44.1KHz	2,267	9,070
300KHz	333	1,333
1.2288MHz	81	325
3.84MHz	26	104
27MHz	3	14
74MHz	1	5
102.4MHz	Not Possible	3

Table 3: DSP Processor Operating Envelope

5.2 FPGAs

Twenty years ago, Field Programmable Gate Arrays (FPGAs) were used for tasks such as glue logic and routing [8]. Today, however, they are complex devices capable of housing an entire system [4]. Here, the basics of current FPGAs will be discussed as well as the features that are targeted directly at DSP applications.

5.2.1 FPGA Architecture Basics

The basic FPGA architecture follows the structure shown in Figure 5.1. The device is made up of block memory, Input/Output Blocks (IOBs), logic blocks and routing, although routing has been omitted from the figure for clarity. The FPGA is usually a fully SRAM based device, which is configured following the application of power or reconfigured during operation. The building blocks can be configured to provide many functions, with programmable interconnect used to join these small functions to form larger functions and systems. This flexibility means that the FPGA provides a huge degree of freedom for the creation of DSP processing functions.

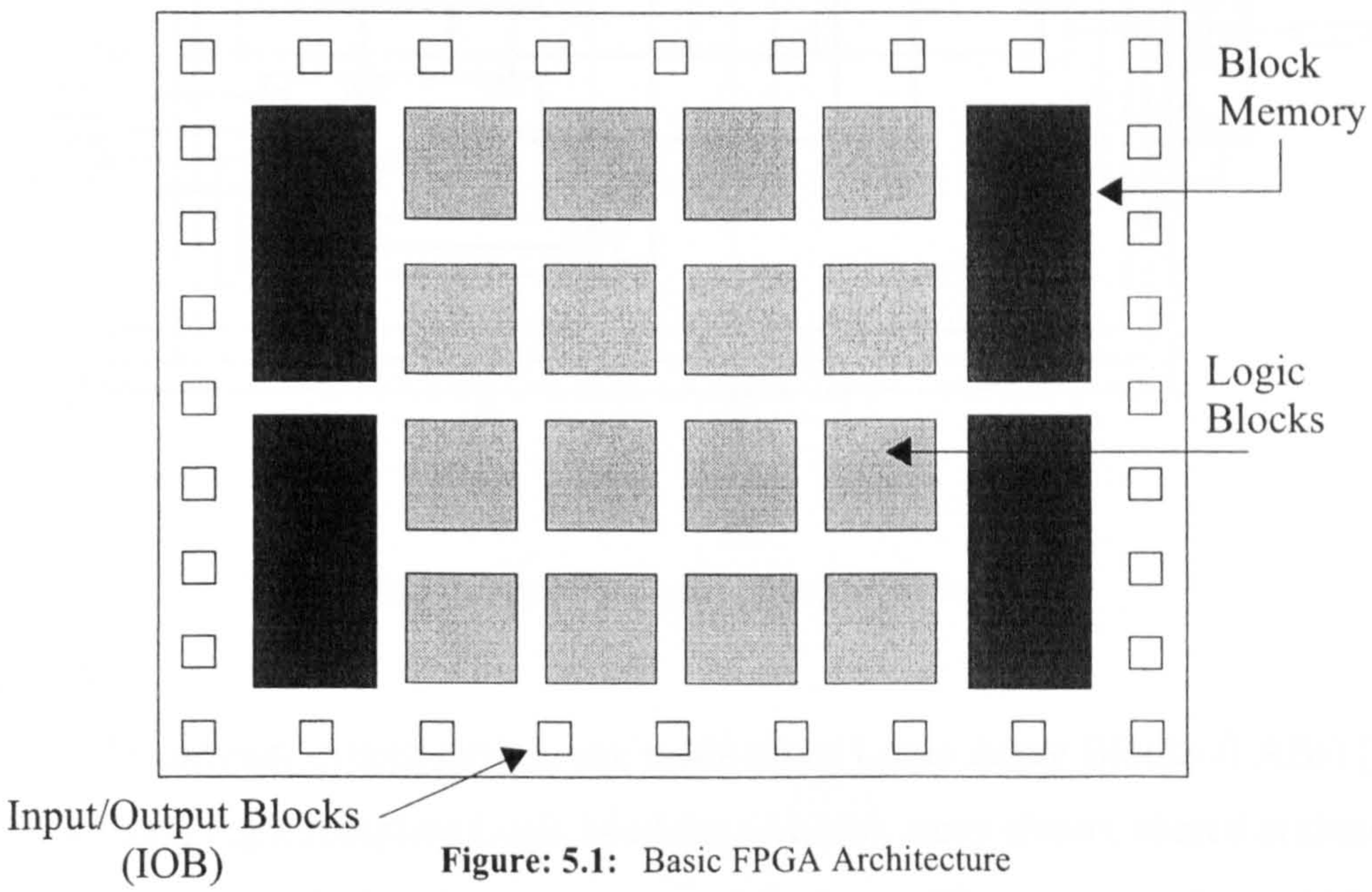


Figure: 5.1: Basic FPGA Architecture

The logic blocks within an FPGA and the names that are given to them differ from vendor to vendor. However, as will be shown, the logic shares many similarities across vendors, although may be in different sizes or widths. For example, the basic logic block in Xilinx devices is called the Configurable Logic Block (CLB) which is made up of Slices [56]. Depending on the device, each CLB can contain either 2 or 4 Slices. Each Slice contains two 4-input LUTs, flip-flops, multiplexers, arithmetic logic, carry chain and dedicated routing. The contents of a single Slice can be seen in

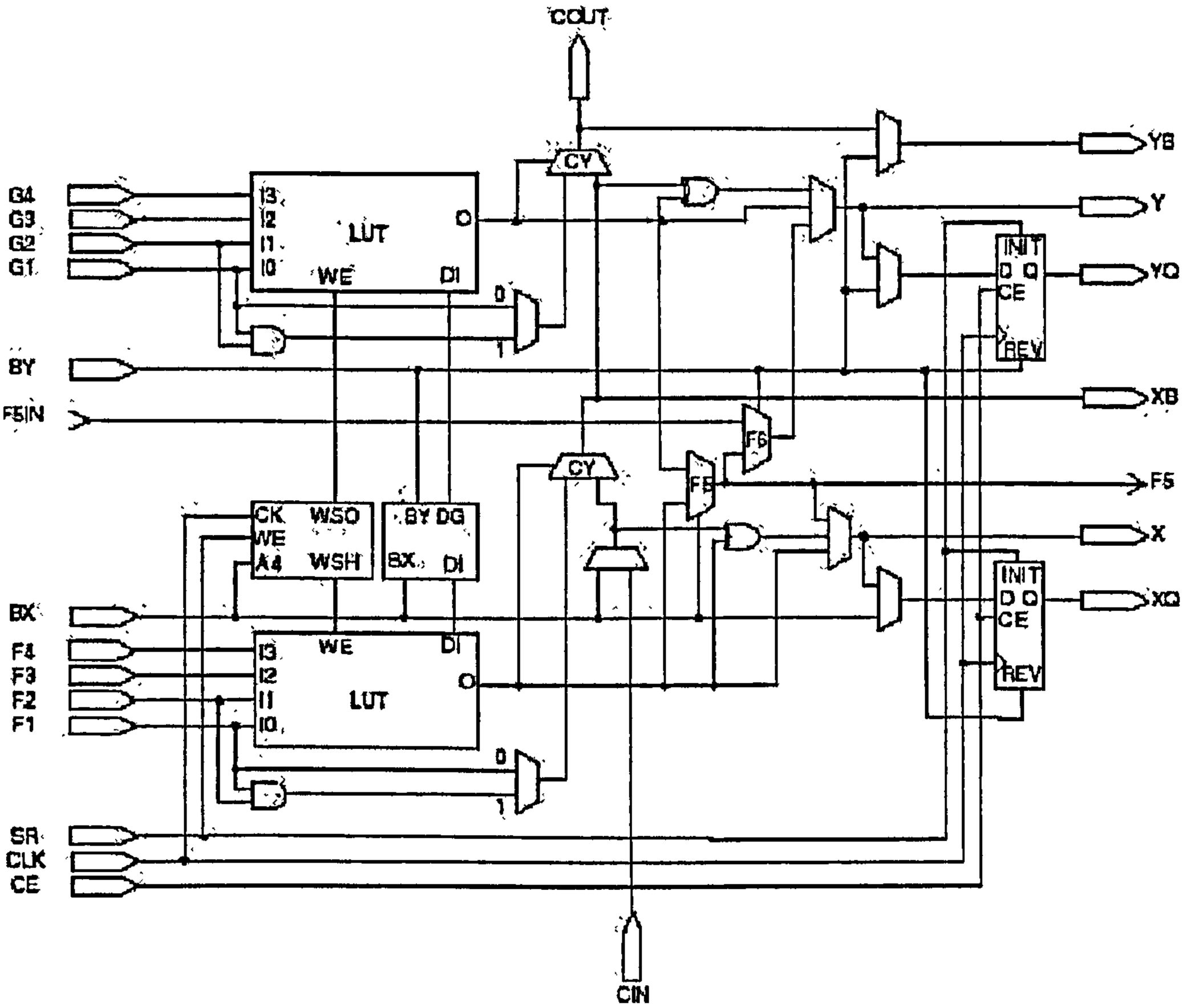


Figure: 5.2: Xilinx Slice Logic - Reproduced from [56]

Figure 5.2.

In contrast, Altera devices are made up of Logic Array Blocks (LABs) [41] which contain eight Adaptive Logic Modules (ALMs), carry chains, shared arithmetic chains, control signals, local interconnect and registers. The contents of an ALM can be seen in Figure 5.3. Note that the Combinational Logic block contains two LUTs which can be configured in several different ways offering a wide range of options.

Lattice Semiconductors is another major FPGA vendor. The building blocks used in their devices are called Programmable Functional Units (PFUs) [46] which are made up of four Slices. Each Slice contains LUTs, multiplexers, carry logic and flip-flops. A single Lattice Slice is shown in Figure 5.4.

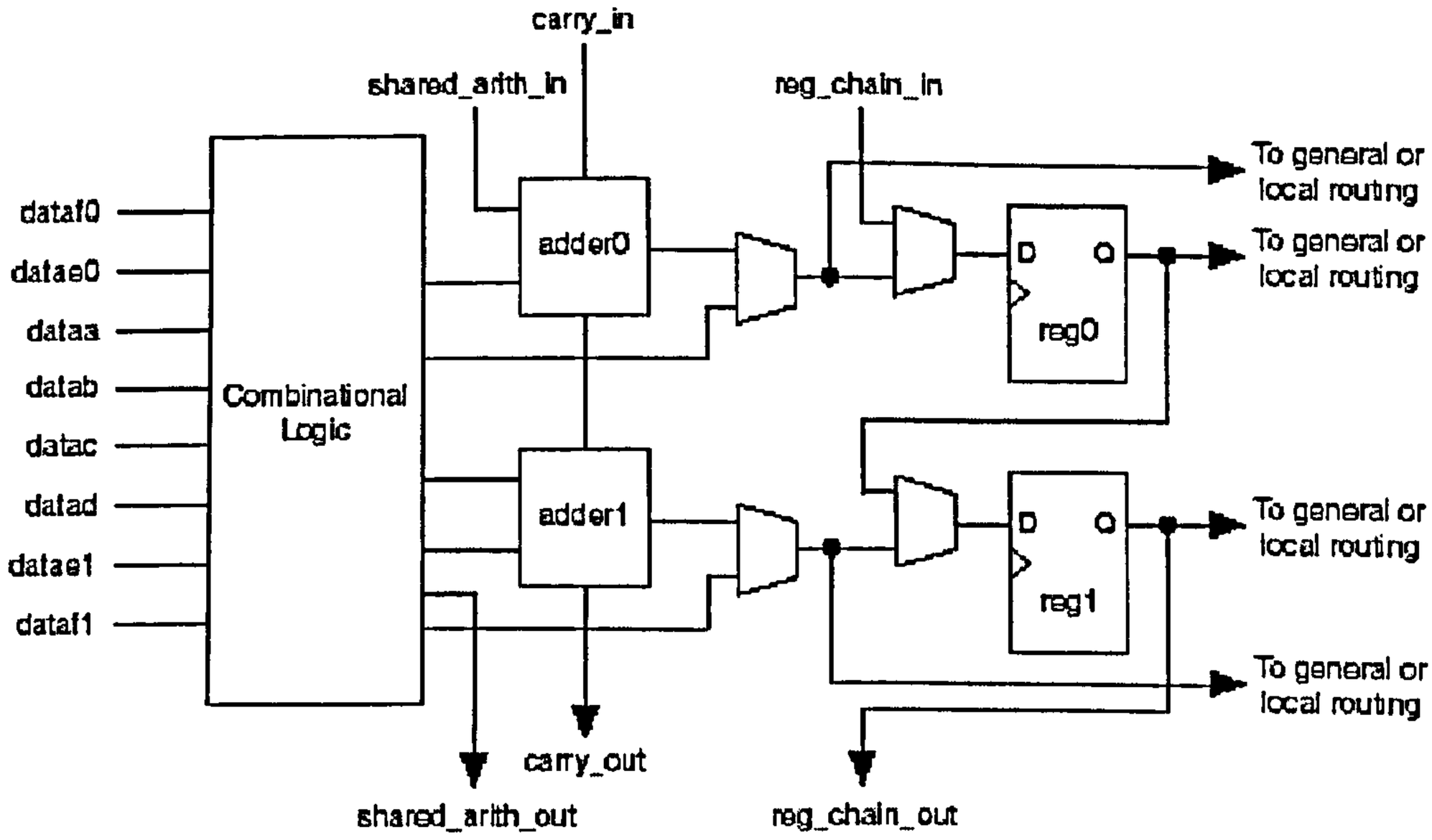


Figure: 5.3: Altera Adaptive Logic Module (ALM) - Reproduced from [41]

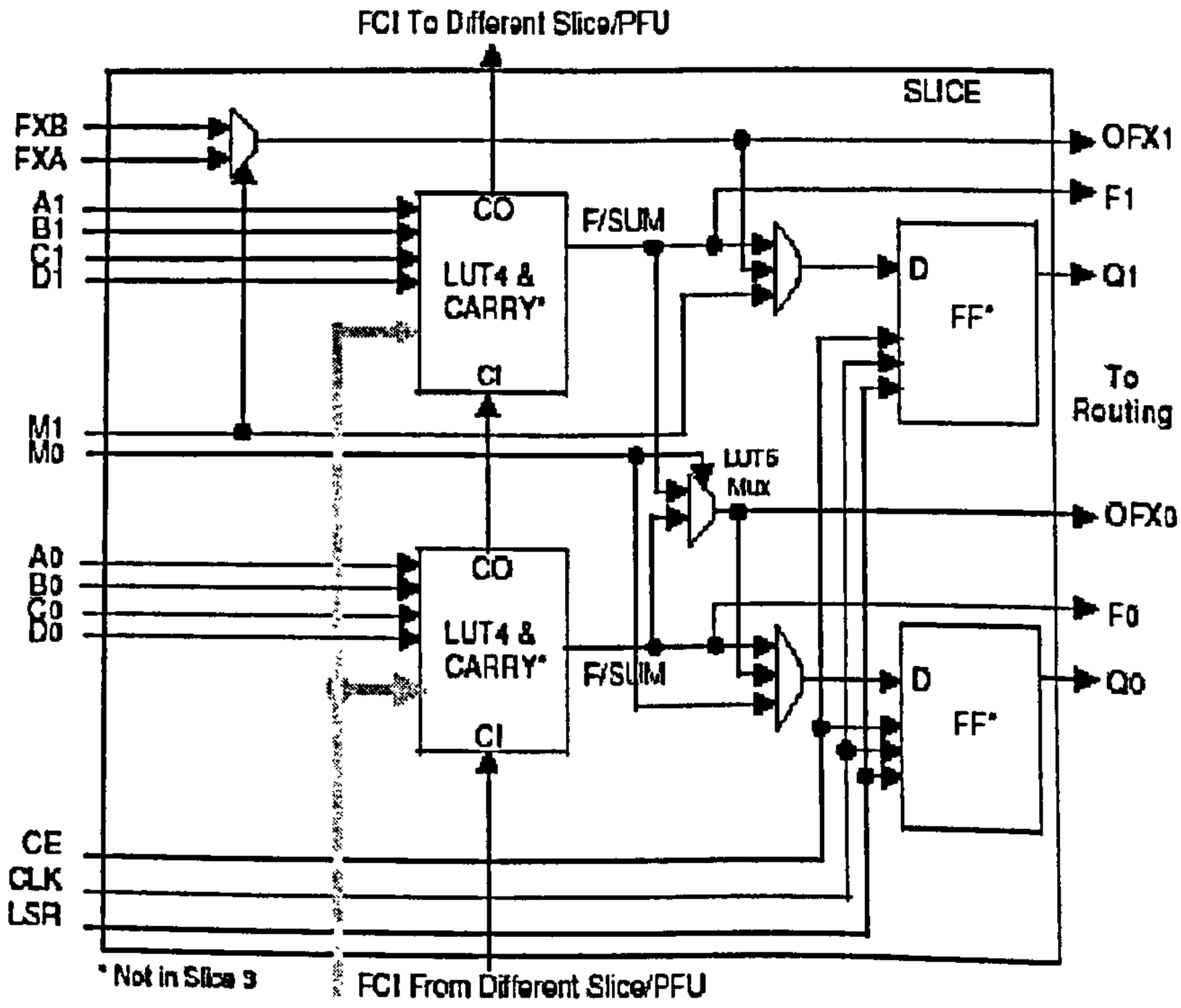


Figure: 5.4: Lattice Slice Logic - Reproduced from [46]

5.2.2 DSP Functionality

The FPGA architecture shown in Figure 5.1 shows the fundamental components of today's devices. However, there are many more features that are not included in this figure which are extremely useful in DSP design. These include the following:

- *Embedded Multipliers*: Multiplication is one of the most common operations in DSP. The largest FPGAs that exist today have several hundred dedicated multipliers embedded into the FPGA fabric. These are typically set up for 18 bit inputs and produce a 36 bit output although they can be cascaded for larger input widths. In the latest devices the multipliers can be clocked at up to 400MHz [53] when fully pipelined.

- *Fast Carry Chains*: Addition is another very common operation in DSP, hence fast carry chains have been included to allow the carry bit to propagate between each full adder as fast as possible. For large 32 bit carry chains, 150 - 200MHz is achievable. The full adders can be constructed from the logic within the logic blocks.

- *MULTAND*: Xilinx has included a single AND gate with each LUT in its Virtex family specifically for creating distributed multipliers. A parallel multiplier can be made from a series of cells, where each cell is made from one Full Adder and an AND gate. Hence, in the FPGA world, a LUT is used to form the Full Adder and the single AND gate is used to complete the multiplication cell. A single AND gate takes up very little of the FPGA fabric and yet if it didn't exist, other FPGA resources would be used to form the AND gate which would be an inefficient use of this logic.

- *Flexible Memory*: The LUTs that are included within an FPGA are flexible and can be used in several different ways. LUTs can be used as dual port RAM, ROM and as Shift Registers. In some devices LUTs can be split to form two LUTs of differing sizes [41]. These options have many uses within DSP and extend the

flexibility of FPGAs even further.

- *Dedicated DSP Modular Blocks:* This is the latest DSP feature to be included in today's FPGAs. As well as the array of logic blocks that have always existed on FPGAs, modular blocks containing hardware multipliers and adders/subtractors are now being included in the FPGA fabric [26].

5.3 Rethinking Algorithm Implementation

Due to the extensive use of the DSP processor, many algorithms have been developed to make use of the operations available on these devices. This has meant that algorithms requiring square root and division operations have been avoided or worked around as they are too demanding in terms of computational requirements. However, now that the FPGA has arrived, a whole new set of resources are available to use. This has meant than the way in which many algorithms can been implemented has had to be re-thought. An ideal example of this is the re-emergence of the CORDIC algorithm which was developed in the 1950's for use in Radar systems [33]. This technique can be used to compute many different functions using only shifts, additions/subtractions and table lookups, all of which are ideal for FPGA implementation. Figure 5.5 gives a summary of the functions that can be directly computed using this technique. There are many more functions that can then be computed by combining more than one CORDIC block. So for example, a single CORDIC core operating in Rotation mode with a Circular coordinate system, can compute $\cos z$ and $\sin z$. The results could then be passed to another CORDIC core computing *x/y* (Vectoring mode, Linear coordinate system) which would yield $\tan z = \frac{\sin z}{\cos z}$.

There has always been a problem with CORDIC regarding the accuracy of the results it produces. It is an iterative technique and the more iterations that are performed the more accurate the solution will be. Traditionally engineers have used a lengthy trial and error approach to CORDIC design. Obviously this could take a long

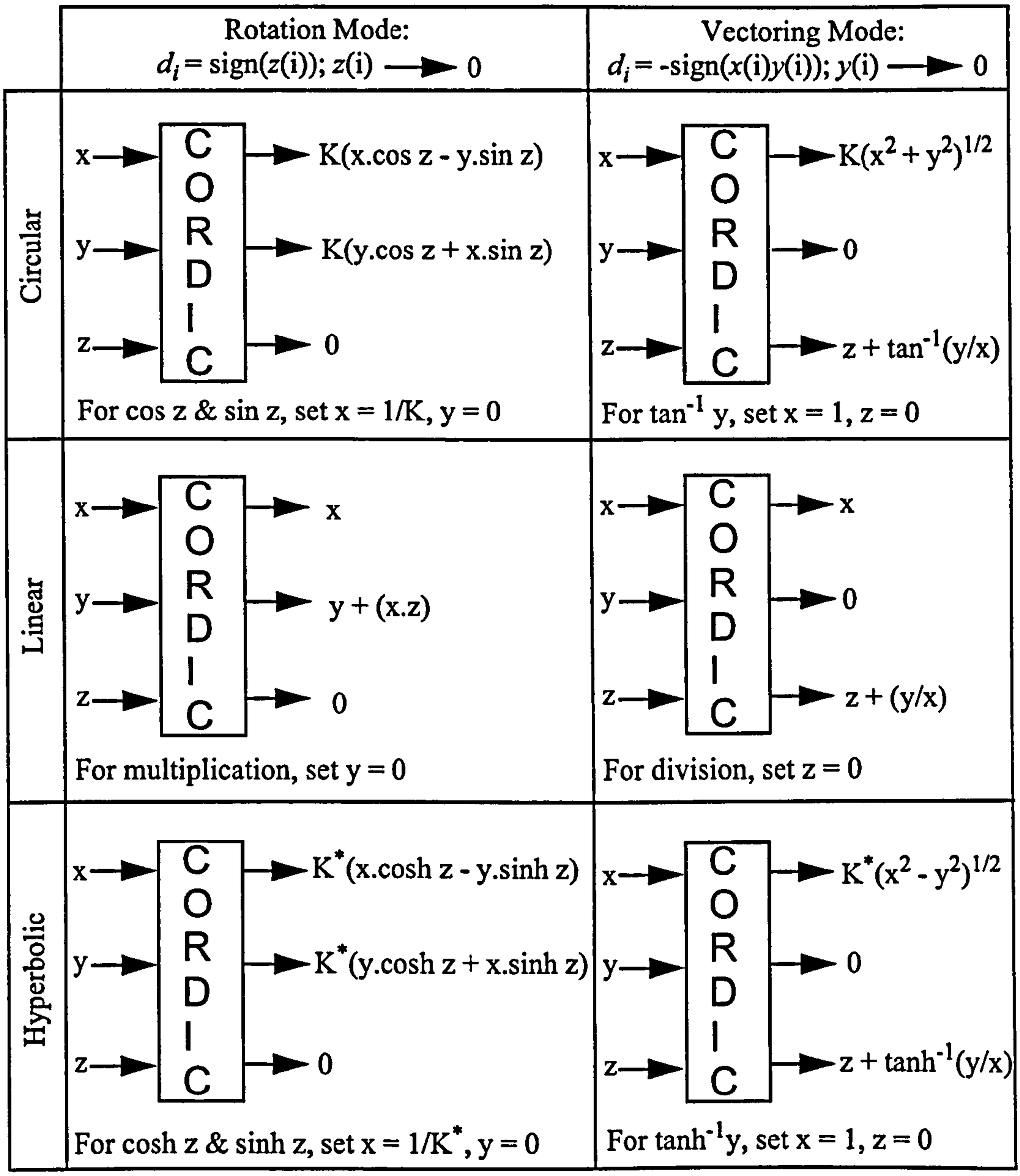


Figure: 5.5: CORDIC Output

time and was not an ideal situation. Hence, one of the mini-projects carried out during the EngD was to find a formula for predicting the error of CORDIC systems, thus removing the need for lengthy simulations. The CORDIC algorithm and the work that has been carried out in this area during the EngD is covered in detail in Chapter 7.

5.4 Adaptive Equalisation

In many communications applications, such as an Ethernet transceiver, the channel over which data is transmitted can change significantly. To cope with this scenario, Adaptive Equalisers are used in the receiver to adapt to the changes in the channel. An equaliser is basically a digital filter with an adaptive algorithm as shown in Figure 5.6. The digital filter can be either Finite Impulse Response (FIR) or Infinite Impulse Response (IIR), although FIR adaptive equalisers are more common due to the fact that they do not suffer from instability issues that can occur in systems with feedback such as the IIR. The Adaptive Algorithm tries to alter the weights in the filter so that the error signal $e(k)$ is reduced to zero. When this occurs, the output signal $y(k)$ has matched the desired signal $d(k)$ and the channel has been equalised. The success of the filter in minimising $e(k)$ will depend on the nature of the input signals, the length of the adaptive filter and the adaptive algorithm used.

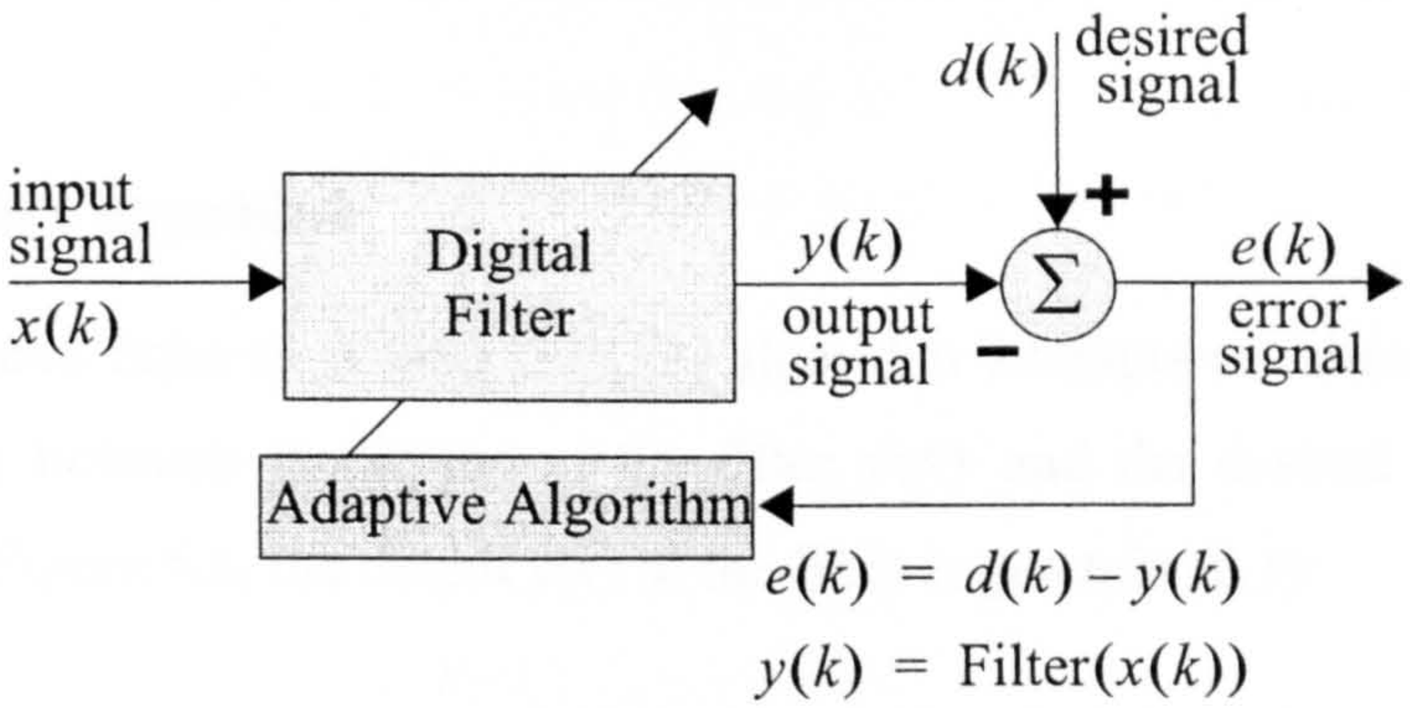


Figure: 5.6: Generalised Adaptive Equaliser

In this section, the background behind two of the available Adaptive Algorithms is presented. Until recently, the Least Mean Squares (LMS) [37][38] algorithm has been one of the most commonly used adaptive algorithms due to its minimal structure, stable performance and relatively low computational requirements. However, applications such as mobile communications now require extremely fast

adaptation which the LMS is not able to achieve. Fortunately another adaptive algorithm called the Recursive Least Squares (RLS) [13][15][23] technique is able to adapt fast enough and with a smaller equalisation error relative to the LMS as is shown in [1][29]. As is nearly always the case in DSP, improved performance comes at a cost and the RLS is no exception. It requires significantly more computational capability than the LMS and until recently has been difficult to realise due to economic and technological factors.

In section 5.4.3 it will be shown that one alternative method for calculating the LS solution is the QR-RLS approach [24], which requires Square Root and Division operations. These operations are expensive to implement and are often avoided in DSP. However, with the advancements in chip technology there is no reason why these functions cannot be used and so there is a need for them in today's EDA tools. It will also be shown that the QR-RLS has a significant amount of feedback within the algorithm. Although the feedback loops can be pipelined, an important question that needs answered is whether or not this is a good thing to do?

5.4.1 The LMS Algorithm

The Least Mean Squares (LMS) [37][38] algorithm attempts to minimise the mean squared error between the output of the filter $y(k)$ and the desired answer $d(k)$. Referring to Figure 5.6, the output $y(k)$ of the filter is given in (5.1):

$$y(k) = \sum_{n=0}^{N-1} w_n x(k-n) = w^T x(k) \tag{5.1}$$

where w is the weight vector,

$$w = [w_0 w_1 w_2 \ldots\ldots w_{N-1}]^T$$

and $x(k)$ is the data matrix:

$$x(k) = [x(k)\ x(k-1)\ x(k-2)\ldots\ldots x(k-N+1)]^T$$

Therefore, the FIR filter requires N multiply-accumulates (MACs) per iteration, where N is the number of weights in the filter.

The LMS algorithm coefficient vector update equation is given by:

$$w(k+1) = w(k) + 2\mu e(k)x(k) \tag{5.2}$$

where the error $e(k)$ is the difference between the desired and obtained output:

$$e(k) = y(k) - d(k)$$

and μ is a scalar called the *step-size*. Hence, the adaptive algorithm requires $N + 1$ MACs per LMS iteration. This means that the total number of MACs per LMS iteration is $2N + 1$ as shown in [37]. So, for a filter of length N, the LMS requires $O(N)$ MACs.

The step-size controls the speed of convergence of the LMS. For a very small step-size, the LMS will converge slowly but when it does converge the error signal $e(k)$ will be very close to zero. However, if a large step-size is used, the LMS will quickly converge to a solution but the power of $e(k)$ will be large. If too large a step-size is chosen, the LMS will go unstable and will not converge at all.

The key point to note on the LMS is that the only operations required to implement an adaptive equaliser using this algorithm are multiplication and addition. The number of these of operations required is $2N + 1$, where N is the number of weights.

5.4.2 The RLS Algorithm

The Recursive Least Squares (RLS) [13][15][23] algorithm is based on the Least Squares solution which minimises the total sum of squared errors for all inputs up to and including time k. The total squared error, $v(k)$, is:

$$v(k) = \sum_{s=0}^{k} [e(s)]^2 = e^2(0) + e^2(1) + e^2(2) + \ldots\ldots + e^2(k) = e_k^T e_k \tag{5.3}$$

where

$$e_k = [e(0)\ e(1)\ e(2).....e(k)]^T$$

The total sum of square errors can then be written as:

$$v(k) = e_k^T e_k = \|e_k\|_2^2$$
$$= [d_k - X_k w]^T [d_k - X_k w]$$
$$= d_k^T d_k + w^T X_k^T X_k w - 2 d_k^T X_k w$$

where

$$X_k = [x(0)\ x(1)\ x(2).....x(k)]^T$$

The equation for $v(k)$ is quadratic in w, therefore the minimum value of $v(k)$ is obtained by finding where the gradient vector is zero:

$$\frac{\partial}{\partial w} v(k) = 0$$

The function $v(k)$ is a hyperparabaloid when plotted in N dimensional space and there exists exactly one minimum point on its surface. The gradient vector is therefore:

$$\frac{\partial}{\partial w} v(k) = 2X_k^T X_k w - 2X_k^T d_k = -2X_k^T [d_k - X_k w]$$

and therefore:

$$-2X_k^T [d_k - X_k w] = 0$$
$$\Rightarrow X_k^T X_k w = X_k^T d_k$$

The Least Squares solution, denoted as w_{LS} and based on data received up to and including time k and can be formed from the above equation as:

$$w_{LS} = [X_k^T X_k]^{-1} X_k^T d_k \qquad (5.4)$$

Solving this equation is not easy as it requires inverting a matrix, which is extremely demanding in terms of the computation that it requires. The greater the number of weights N, the bigger X_k will be and the more demanding the inversion of the matrix will become, which requires multiplication and division operations. This would be hard enough to compute if it was only carried out once, but an inversion would have to be carried out for every new sample of $x(k)$ and $d(k)$. Hence, this is not a realistic

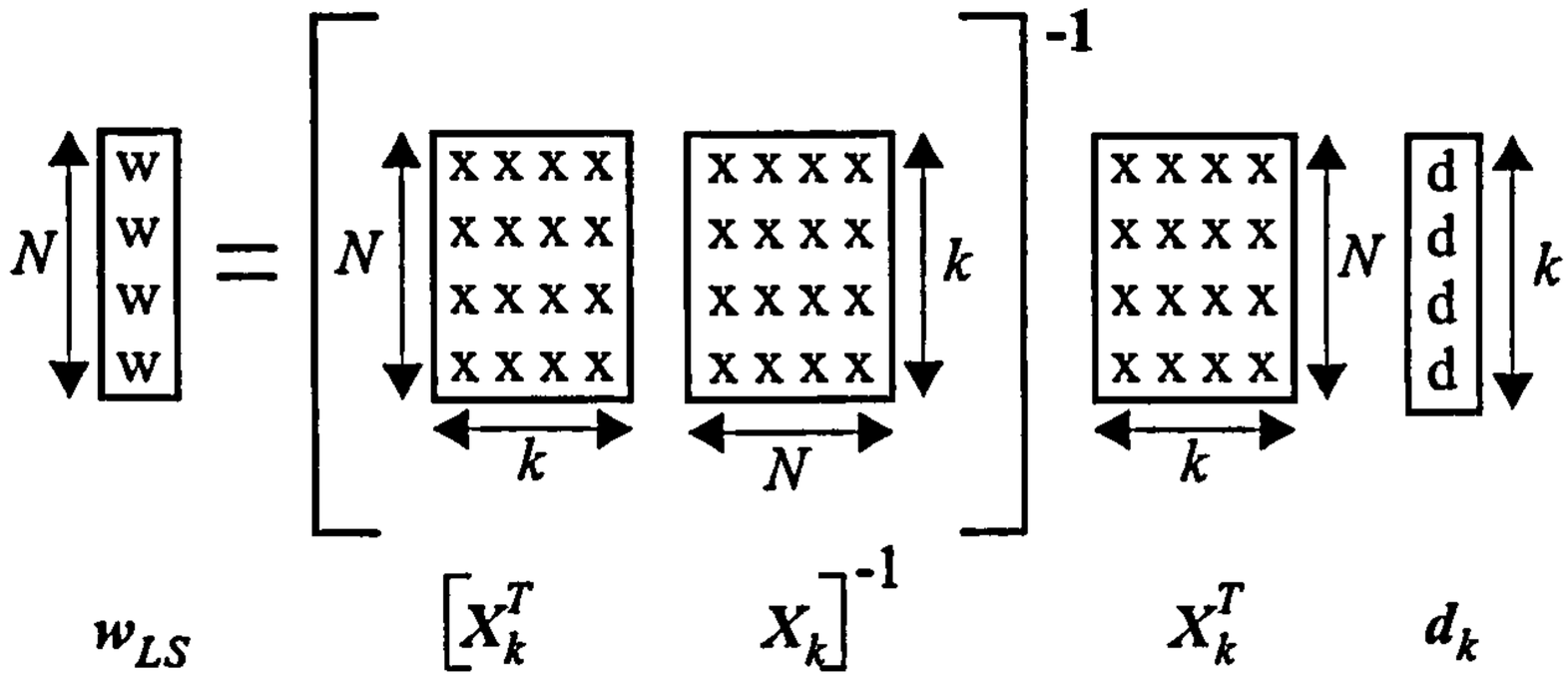


Figure: 5.7: Least Squares Matrix Solution

option for solving this equation. Figure 5.7 gives a visual indication of the size of the task involved in solving (5.4). In addition to the complexities of matrix inversion, forming the inner product $X^T.X$ is known to be extremely damaging from a numerical point of view. The square doubles the dynamic range, and hence the wordlength has to be doubled, otherwise precision is approximately halved. However, there is an approach which reduces the computational complexity of the Least Squares equation known as QR matrix decomposition and this is discussed in the following section.

5.4.3 QR Decomposition

The QR matrix decomposition [17] is an extremely useful technique in least squares signal processing systems where a full rank $m \times n$ data matrix X_k is decomposed into an upper triangular matrix, R and an orthogonal matrix Q ($Q^TQ = I$).

$$X_k = QR$$

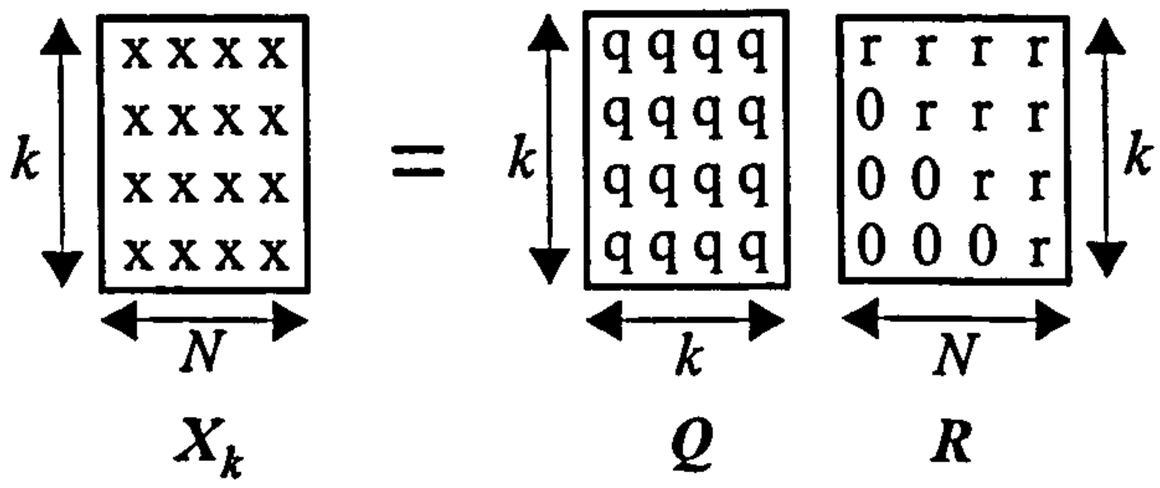


Figure: 5.8: QR Decomposition

Using the QR decomposition to solve the least squares equation gives:

$$w_{LS} = [X_k^T X_k]^{-1} X_k^T d_k = [(QR)^T (QR)]^{-1} (QR)^T d_k$$
$$= [R^T Q^T QR]^{-1} R^T Q^T d_k$$
$$= [R^T R]^{-1} R^T Q^T d_k$$
$$= R^{-1} R^{-T} R^T Q^T d_k$$
$$= R^{-1} Q^T d_k$$
$$w_{LS} = R^{-1} d_k'$$

where

$$d_k' = Q^T d_k$$

The result of the decomposition still requires the matrix R to be inverted before w_{LS} can be found. However, it will be shown shortly that this is not actually the case and that it is possible to solve w_{LS} without inverting any matrices.

To be able to solve w_{LS} using the QR decomposition requires R to be found starting from the data matrix X_k. This process can be carried out using Givens rotations to zero the lower half of the matrix X_k resulting in the matrix R. The series of Givens rotations is actually the matrix Q^T and is illustrated in the following example:

$$X = QR \rightarrow \underset{X}{\begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & -7 & 11 \end{bmatrix}} = \underset{Q}{\begin{bmatrix} -0.27 & -0.51 & -0.82 \\ -0.53 & -0.63 & 0.56 \\ -0.80 & 0.57 & -0.10 \end{bmatrix}} \underset{R}{\begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & 10.43 & 4.19 \\ 0 & 0 & -3.20 \end{bmatrix}}$$

The process of getting from X to R is computed using a series of Givens rotations which is achieved in a row-wise fashion. Each element in X that must be zeroed is done so by multiplying by the G matrix which is a modification of the identity matrix I:

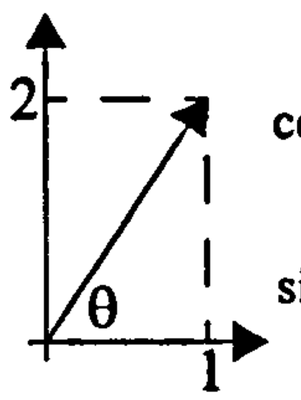
$$G_{i,j} = \begin{bmatrix} (\cos\theta)_{jj} & (\sin\theta)_{ji} & 0 \\ (-\sin\theta)_{ij} & (\cos\theta)_{ii} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $[i, j]$ is the location of the element to be zeroed.

Givens Rotation $G_{2,1}$ | Givens Rotation $G_{3,1}$ | Givens Rotation $G_{3,2}$

$$\begin{bmatrix} \boxed{1} & 5 & 9 \\ \boxed{2} & 6 & 10 \\ 3 & -7 & 11 \end{bmatrix} \rightarrow \begin{bmatrix} \boxed{2.24} & 7.60 & 12.97 \\ 0 & -1.79 & -3.58 \\ \boxed{3} & -7 & 11 \end{bmatrix} \rightarrow \begin{bmatrix} 3.74 & \boxed{-1.07} & 16.57 \\ 0 & -1.79 & -3.58 \\ 0 & \boxed{-10.26} & -3.82 \end{bmatrix} \rightarrow \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & 10.43 & 4.19 \\ 0 & 0 & -3.20 \end{bmatrix}$$

X R

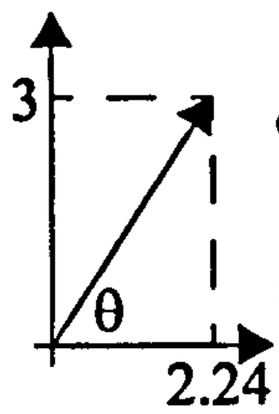


$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}} = 0.45 \qquad \sin\theta = \frac{y}{\sqrt{x^2+y^2}} = 0.89$$

$G_{2,1}$ X X'

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 5 & 9 \\ \textcircled{2} & 6 & 10 \\ 3 & -7 & 11 \end{bmatrix} = \begin{bmatrix} 2.24 & 7.60 & 12.97 \\ 0 & -1.79 & -3.58 \\ 3 & -7 & 11 \end{bmatrix}$$

$$\begin{bmatrix} 1\cos\theta + 2\sin\theta + 0 = 2.24 & 5\cos\theta + 6\sin\theta + 0 = 7.6 & 9\cos\theta + 10\sin\theta + 0 = 12.97 \\ -1\sin\theta + 2\cos\theta + 0 = 0 & -5\sin\theta + 6\cos\theta + 0 = -1.79 & -9\sin\theta + 10\cos\theta + 0 = -3.58 \\ 0 + 0 + 3 = 3 & 0 + 0 - 7 = -7 & 0 + 0 + 11 = 11 \end{bmatrix}$$

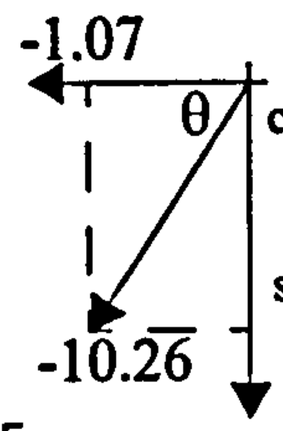


$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}} = 0.60 \qquad \sin\theta = \frac{y}{\sqrt{x^2+y^2}} = 0.80$$

$G_{3,1}$ X' X''

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 2.24 & 7.60 & 12.97 \\ 0 & -1.79 & -3.58 \\ \textcircled{3} & -7 & 11 \end{bmatrix} = \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & -1.79 & -3.58 \\ 0 & -10.26 & -3.82 \end{bmatrix}$$

$$\begin{bmatrix} 2.24\cos\theta + 0 + 3\sin\theta = 3.74 & 7.6\cos\theta + 0 - 7\sin\theta = -1.07 & 12.97\cos\theta + 0 + 11\sin\theta = 16.57 \\ 0 + 0 + 0 = 0 & 0 - 1.79 + 0 = -1.79 & 0 - 3.58 + 0 = -3.58 \\ -2.24\sin\theta + 0 + 3\cos\theta = 0 & 7.6\sin\theta + 0 - 7\cos\theta = -10.26 & -12.97\sin\theta + 0 + 11\cos\theta = -3.82 \end{bmatrix}$$



$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}} = -0.10 \qquad \sin\theta = \frac{y}{\sqrt{x^2+y^2}} = -1.00$$

$G_{3,2}$ X'' R

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & -1.79 & -3.58 \\ 0 & \textcircled{-10.26} & -3.82 \end{bmatrix} = \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & 10.43 & 4.19 \\ 0 & 0 & -3.20 \end{bmatrix}$$

$$\begin{bmatrix} 3.74 + 0 + 0 = 3.74 & -1.07 + 0 + 0 = -1.07 & 16.57 + 0 + 0 = 16.57 \\ 0 + 0 + 0 = 0 & 0 - 1.79\cos\theta - 10.26\sin\theta = 10.43 & 0 - 3.58\cos\theta - 3.82\sin\theta = 4.19 \\ 0 + 0 + 0 = 0 & 0 + 1.79\sin\theta - 10.26\cos\theta = 0 & 0 + 3.58\sin\theta - 3.82\cos\theta = -3.2 \end{bmatrix}$$

The series of Givens rotations $G_{2,1}$, $G_{3,1}$, $G_{3,2}$ combine to give Q^T. Once the matrix R has been formed it is possible to find w_{LS} without carrying out matrix inversion. This process is called *backsubstitution*. Going back to the QR

$$Q^T \begin{bmatrix} x_{11} & x_{12} & x_{13} & d_1 \\ x_{21} & x_{22} & x_{23} & d_2 \\ x_{31} & x_{32} & x_{33} & d_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & d_1' \\ 0 & r_{22} & r_{23} & d_2' \\ 0 & 0 & r_{33} & d_3' \end{bmatrix}$$
$$X \qquad d_k \qquad\qquad R \qquad d_k'$$

$$\Rightarrow \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} d_1' \\ d_2' \\ d_3' \end{bmatrix}$$
$$R \qquad w_{LS} \qquad d_k'$$

Figure: 5.9: Single QR Decomposition

decomposition it was shown that:

$$w_{LS} = R^{-1} Q^T d_k$$

$$\Rightarrow R w_{LS} = Q^T d_k$$

$$\Rightarrow Q^T X w_{LS} = Q^T d_k = d_k'$$

Using this version of the decomposition it can be seen that Q^T is applied to both the X and the d_k matrices. Thus, in practice these two matrices are combined and Q^T is applied to this single matrix as illustrated in Figure 5.9. From here, the process of *backsubstitution* can be used to find w_{LS}. In theory this process is easy. However, it is computationally expensive as it requires division, multiplication and addition. From Figure 5.9 it can be seen that for this example the weights can be found by starting with w_3 as can be seen below.

$$w_3 = \frac{d_3'}{r_{33}} \qquad w_2 = \frac{d_2' - r_{23} w_3}{r_{22}} \qquad w_1 = \frac{d_1' - r_{12} w_2 - r_{13} w_3}{r_{11}}$$

5.4.4 QRD-RLS

The QR decomposition shown so far only illustrates how the weights are found once.

$$\left[\begin{array}{c|c} R(k) & d_k' \\ \hline x_{k+1}^T & d_{k+1} \end{array}\right] = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & d_1' \\ 0 & r_{22} & r_{23} & d_2' \\ 0 & 0 & r_{33} & d_3' \\ \hline x_1 & x_2 & x_3 & d \end{array}\right] = \begin{bmatrix} 3.741 & 11.759 & 6.4143 & 0.534 \\ 0.000 & -3.273 & -1.091 & -3.273 \\ 0.000 & 0.000 & 17.962 & -2.449 \\ 4 & 9 & -13 & -2 \end{bmatrix}$$

$$\begin{bmatrix} 3.741 & 11.759 & 6.4143 & 0.534 \\ 0.000 & -3.273 & -1.091 & -3.273 \\ 0.000 & 0.000 & 17.962 & -2.449 \\ 4 & 9 & -13 & -2 \end{bmatrix} \rightarrow \begin{bmatrix} 5.477 & 14.605 & -5.112 & -1.095 \\ 0.000 & -3.273 & -1.091 & -3.273 \\ 0.000 & 0.000 & 17.962 & -2.449 \\ 0.000 & -2.439 & -13.565 & -1.756 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 5.477 & 14.605 & -5.112 & -1.095 \\ 0.000 & -4.082 & -8.981 & -3.674 \\ 0.000 & 0.000 & 17.962 & -2.449 \\ 0.000 & 0.000 & -10.224 & 0.547 \end{bmatrix} \rightarrow \left[\begin{array}{ccc|c} 5.477 & 14.605 & -5.112 & -1.095 \\ 0.000 & -4.082 & -8.981 & -3.674 \\ 0.000 & 0.000 & 20.668 & -2.399 \\ \hline 0.000 & 0.000 & 0.000 & -0.735 \end{array}\right]$$

$$\left[\begin{array}{c|c} R(k+1) & d_{k+1}' \\ \hline 0 & * \end{array}\right]$$

Figure: 5.10: Recursive QR Decomposition

In a real system the weights are computed for every new $x(k)$ and $d(k)$. However, rather than update the X and d_k matrices and repeat the process illustrated in Figure 5.9, a recursive technique can be used which cuts down the computational requirement [15][24]. This recursive technique can be seen in Figure 5.10. Here, an additional row is used to allow the new x (4, 9, -13) and d (-2) data to be added to the upper triangular matrix containing $R \;\; d_k'$. Hence, the original matrix grows from a 3×4 to a square matrix of dimension 4×4. The decomposition then proceeds to zero each of the new x data entries working from left to right. Once complete, the backsubstitution process can be carried out to find the new w_{LS}. Note that although there are now 4 rows in the matrix, there are still only 3 weights to be found as in Figure 5.9.

The QR decomposition can be computed using a triangular array, as developed in [23] and [36], which is shown in Figure 5.11. This diagram shows that the array is made up of two types of cell. The cells on the left edge of the array are used

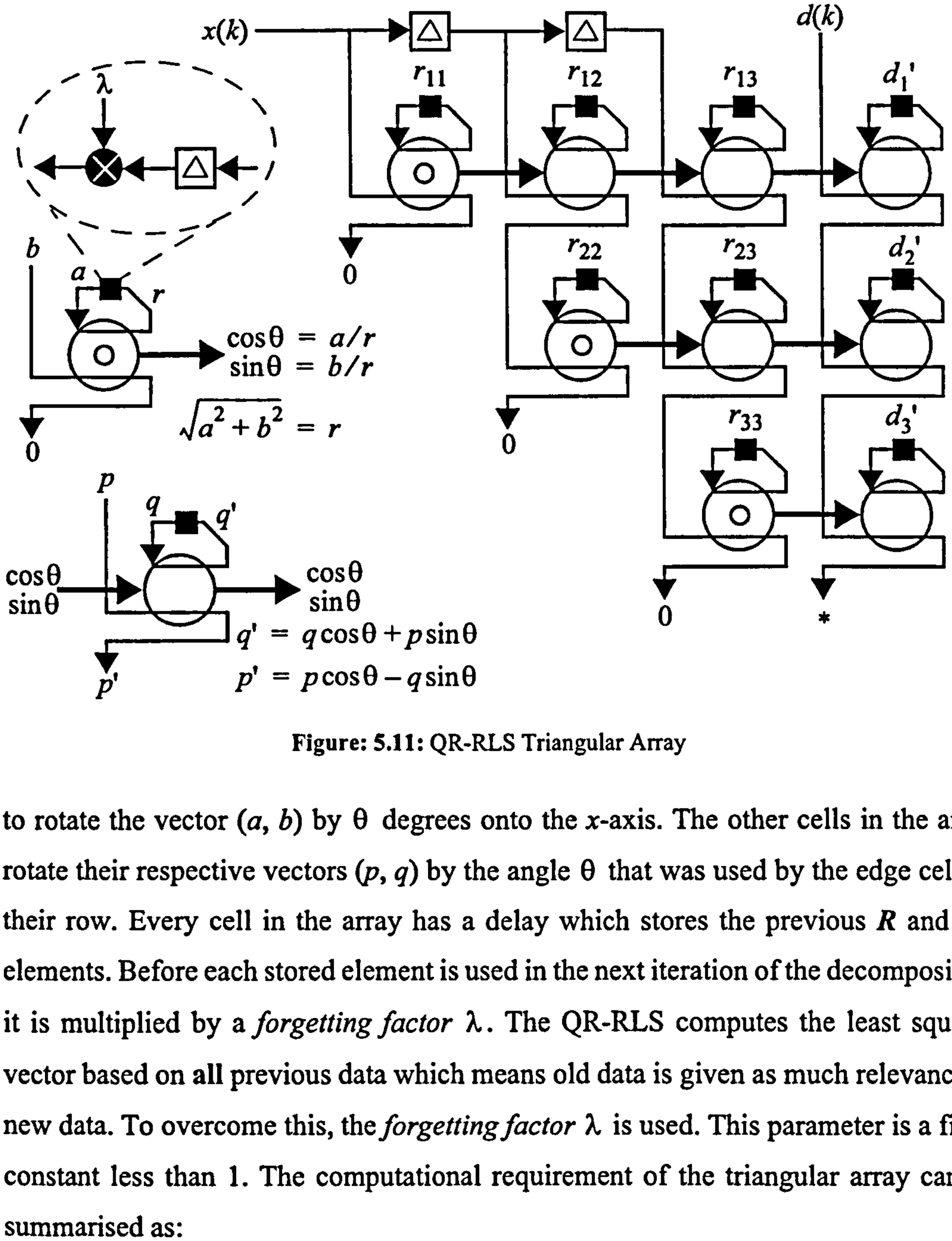
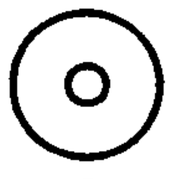


Figure: 5.11: QR-RLS Triangular Array

to rotate the vector (a, b) by θ degrees onto the x-axis. The other cells in the array rotate their respective vectors (p, q) by the angle θ that was used by the edge cell on their row. Every cell in the array has a delay which stores the previous R and d_k' elements. Before each stored element is used in the next iteration of the decomposition it is multiplied by a *forgetting factor* λ. The QR-RLS computes the least squares vector based on **all** previous data which means old data is given as much relevance as new data. To overcome this, the *forgetting factor* λ is used. This parameter is a fixed constant less than 1. The computational requirement of the triangular array can be summarised as:

Each  requires 5 multiplies, 1 addition, 1 division, 1 square root

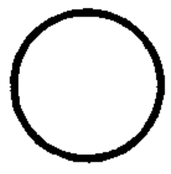
Each  requires 5 multiplies, 1addition, 1 subtraction

Figure: 5.12: QR-RLS Computational Requirement

There are other QR-RLS techniques that do not require the use of square root operations as presented in [3][12][14] where the idea is to store values in a squared form. However, this comes at a cost as the dynamic range is doubled and hence the wordlength has to be doubled, otherwise precision is approximately halved.

Earlier it was shown that the LMS technique required $O(N)$ MACs per sample [37][38]. This means that the LMS will scale well if the number of weights are increased. Unfortunately this is not the case for the RLS and the QR-RLS as can be seen in Figure 5.11. The computational requirement is a function of $O(N^2)$ which does not scale well [15][24]. In addition to this the QR-RLS requires square roots and divides which are also very costly in terms of hardware. This is the reason for the delay in moving to the RLS technique even though it offers better performance. However, today's technology is now at a stage where the QR-RLS and the RLS are realistic options. This means that to exploit the advantages of the QR-RLS method that the development of efficient square root and division cores is required.

Chapter 6

Division & Square Root Core Development

This chapter discusses the work that was carried out to develop a Square Root and Division core for inclusion in HDL Design Studio. Both cores were based on direct methods for computing these functions. The direct methods discussed are straight forward "paper and pencil" techniques that generate a solution with a known level of accuracy within a known number of cycles which is a major benefit in Digital Signal Processing (DSP) Systems. Other techniques for computing division and square roots exist but these usually involve an iterative approach such as Newton's method. For DSP design this is not ideal as the number of iterations required to achieve a desired level of accuracy are unknown.

As well as illustrating these techniques, software and hardware implementations for both operations are discussed. The software implementations must model the delay and output of the equivalent hardware implementations. This is vitally important as HDL Design Studio is marketed as producing bit and cycle accurate VHDL code.

6.1 Long Division

Long Division is a method for dividing two integers. Many people learn how to use long division while at school but tend to forget how to use it.

Decimal Long Division

To illustrate long division, consider the following calculation: 2467/13.

```
      0189.7692
  13 |2467.0000
     0
     --
     24
     13
     --
     116
     104
     ---
     0127
     0117
     ----
     0010 0
     0009 1
     ------
     0000 90
     0000 78
     -------
     0000 120
     0000 117
     --------
     0000 0030
     0000 0026
     ---------
     0000 0004
```

The working can be broken down into the following steps:

1. $\lfloor 2/13 \rfloor = 0$

2. 0*13 = 0

3. 2 - 0 = 2

4. bring down the 4 and append to 2 to get 24

1. $\lfloor 24/13 \rfloor = 1$

2. 1*13 = 13

3. 24 - 13 = 11

4. bring down the 6 and append to 11 to get 116

1. $\lfloor 116/13 \rfloor = 8$

2. 8*13 = 104

3. 116 - 104 = 12

4. bring down the 7 and append to 12 to get 127

1. $\lfloor 127/13 \rfloor = 9$

2. 9*13 = 117

3. 127 - 117 = 10

4. bring down the 0 and append to 10 to get 100

1. $\lfloor 100/13 \rfloor = 7$

2. 7*13 = 91

3. 100 - 91 = 9

4. bring down the 0 and append to 9 to get 90

etc.

Note that the number of cycles through the algorithm is dependent on the size of the dividend. In this case, the dividend is 8 digits wide and thus it takes 8 cycles to achieve the result. Note also that if more accuracy is desired then the number of zeroes after the decimal point on the dividend is simply extended. In this example, there are 4

zeroes which gives a result with 4 decimal points of accuracy.

Binary Long Division

To illustrate binary long division, again consider the case 2467/13 (100110100011/ 1101).

The example can be broken down into the following steps:

1. 1/1101 = 0
2. 0*1101 = 0
3. 1 - 0 = 1
4. append next digit in dividend (0) to result of step 3 (1) to get 10

1. 10/1101 = 0
2. 0*1101 = 0
3. 10 - 0 = 10
4. append next digit in dividend (0) to result of step 3 (10) to get 100

1. 100/1101 = 0
2. 0*1101 = 0
3. 100 - 0 = 100
4. append next digit in dividend (1) to result of step 3 (100) to get 1001

1. 1001/1101 = 0
2. 0*1101 = 0
3. 1001 - 0 = 1001
4. append next digit in dividend (1) to result of step 3 (1001) to get 10011

```
                000010111101.1100 = 189.75
1101 | 100110100011.0000
       0
       10
       00
       100
       000
       1001
       0000
       10011
       01101
       001100
       000000
       0011001
       0001101
       00011000
       00001101
       000010110
       000001101
       0000010010
       0000001101
       00000001011
       00000000000
       000000010111
       000000001101
       000000001010 0
       000000000110 1
       000000000011 10
       000000000011 01
       000000000000 010
       000000000000 000
       000000000000 0100
       000000000000 0000
       000000000000 0100
```

1. 10011/1101 = 1

2. 1*1101 = 1101

3. 10011 - 1101 = 110

etc.

Note again that the number of cycles through the algorithm depends on the width of the dividend. In this example the dividend is 16 bits wide, hence it takes 16 cycles to complete the calculation. Also, if *x* fractional bits are required from the result, then the dividend needs *x* zeroes padded onto the end.

The division method shown here is a non-restoring technique [20] which means that the partial remainder is never restored to its previous value. The divider designed for HDS is a full parallel, unrolled implementation that offers high throughput but with the expense of using more logic. In [22] a non-restoring divider is implemented using a bit serial/word parallel approach which offers a smaller implementation for slower data rates. There are restoring techniques [6] where the partial remainder is restored to a previous value if it is found to be negative. However, this type of algorithm uses more operations and is less efficient than a non-restoring technique. Another technique that produces 1-bit of the quotient per iteration is SRT division which refers to Sweeney, Robertson [28] and Tocher [32] who independently discovered the same algorithm around 1958. SRT division is similar to non-restoring division, but it uses a lookup table based on the dividend and the divisor to determine each quotient digit. It is a popular method for floating-point division on microprocessors. Finally, the Newton-Raphson [35] and Goldschmidt [11] algorithms differ from those already discussed in that they start with an estimate of the quotient and then proceed to generate a more accurate estimate with each iteration. These techniques can produce a result faster than the 1-bit per iteration schemes already discussed but they tend to be avoided in DSP due to the fact that the number of iterations required to generate a desired level of accuracy cannot be pre-determined easily.

6.2 Specification

The specification that was originally drawn up for the development of a Divider core for HDS included the following requirements:

Develop a parameterisable VHDL core able to:

- Perform division on any combination of inputs using signed fixed point and unsigned integers (HDS does not support unsigned fixed point data).
- Return either signed fixed point of unsigned integer output at the request of the user.
- Allow the user to specify output parameters such as the number of integer and fractional bits with the option to truncate, round or saturate.
- Be able to pipeline to increase throughput.

Develop a C++ model able to:

- Mimic the VHDL output exactly, hence must be bit and cycle accurate.

6.3 The Hardware Implementation

In this section the hardware implementation that was designed for calculating long division is discussed. The architecture is based on the long division technique discussed earlier.

6.3.1 The Top Level

The Divider core has a fixed number of I/O ports which include input ports for the Dividend, Divisor, Clock, Enable and Reset signals. The output ports are for the Quotient and Ready signals. To allow the core to be parameterisable, generic parameters are used within the VHDL. Hence, information on the width and data type

of the Dividend, Divisor and Quotient signals is passed to the generics as well as whether rounding, saturation or truncation is to be used. A summary of the information used to configure the top-level of the Division core can be seen in Figure 6.1 below.

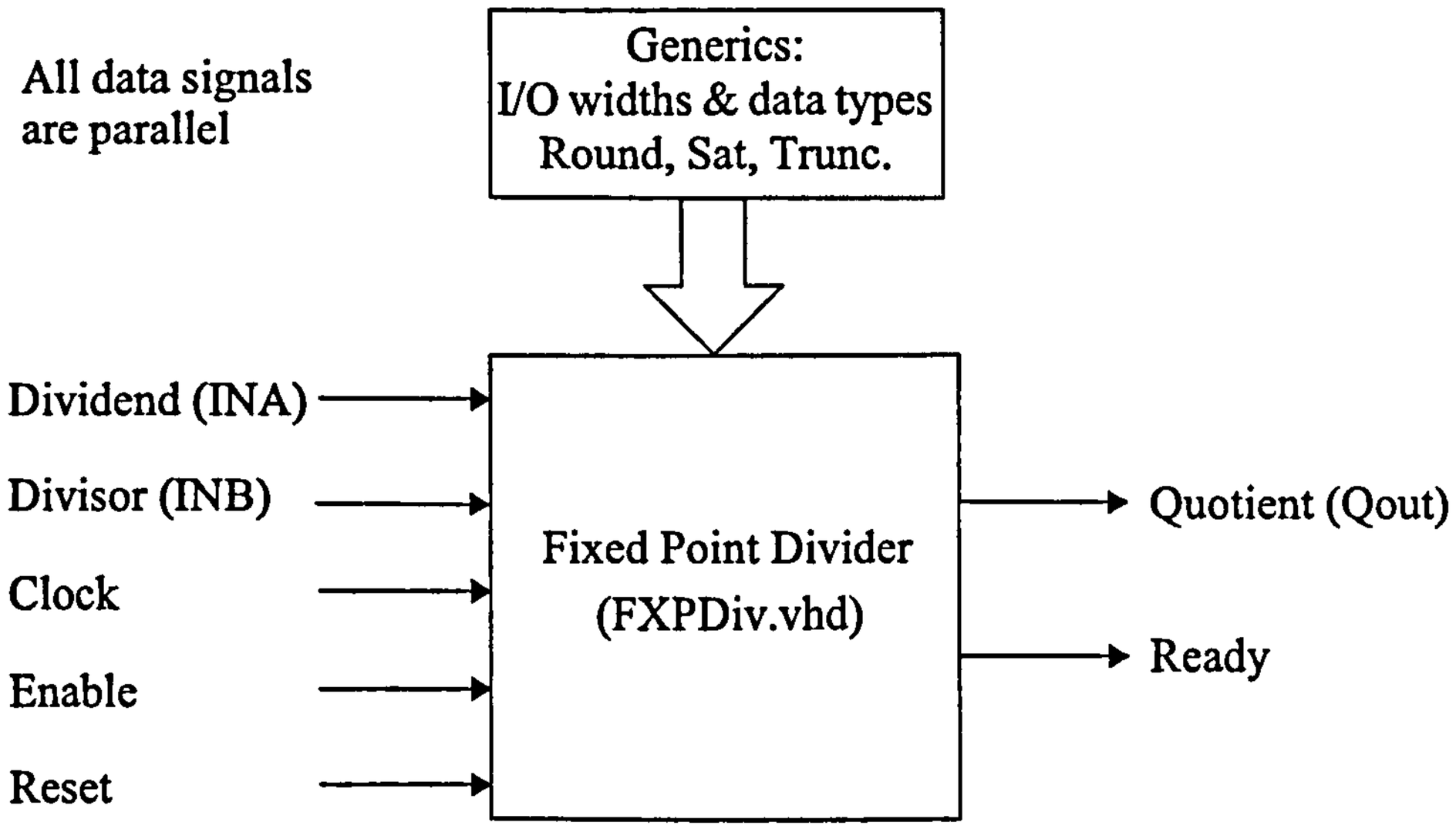


Figure: 6.1: Fixed Point Divider Top Level

6.3.2 Inside The Fixed Point Divider

Inside the divider there are several steps that must be carried out to allow the output to be generated correctly. Although the binary division example shows the division algorithm to be fairly straight forward, it does not illustrate how signed division, rounding, saturation or truncation is handled. Of course, all these options must be handled within the divider core. In Figure 6.2 the sequence of steps that are carried out within the divider are illustrated.

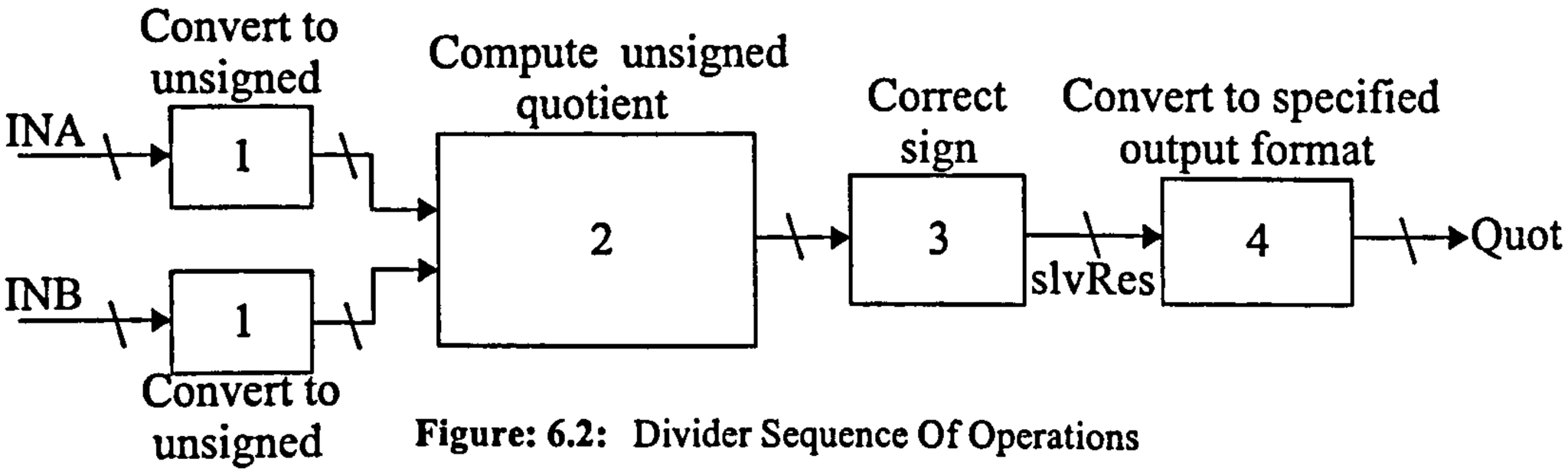


Figure: 6.2: Divider Sequence Of Operations

- Step 1: At this stage the numerator (INA) and the denominator (INB) are converted to equivalent unsigned numbers if they are in 2's complement format. Hence, negative signed data must be converted to equivalent unsigned values.

- Step 2: The unsigned quotient is computed using the algorithm shown in the example earlier. Note that this section of the divider is covered in greater detail in the following subsection.

- Step 3: Here, the unsigned quotient is converted back into a signed number if either of the inputs were signed. Of course, the sign of each quotient must be corrected too as some will be negative and some positive signed values. The VHDL signal at the output of this stage is *slvRes*.

- Step 4: Finally the *slvRes* output, which at this stage is signed, unless both inputs were unsigned, is converted into the desired output format specified by the user. Hence, if the user has chosen to Round and/or Saturate the output, it will occur here. There are some important points to note about the signals entering and leaving this stage of the divider. Firstly, the number of integer bits in the *slvRes* signal are exactly sufficient to avoid overflow. This is precomputed and will be discussed in the following subsection. The number of integer bits that finally exit the divider is specified by the user. Therefore, at this stage either truncation or sign extension is performed depending on the number of bits specified. This means the user must be aware that they are responsible for maintaining signal integrity. If they choose to have an output signal with only 2 integer bits but a minimum of 6 are required to avoid overflow then signal integrity will be compromised. It is not possible to only generate the number of integer bits that are specified by the user. This is because the divider generates the result MSB first. Hence, the full integer result is computed and then converted to the user specified format. The number of fractional bits that exist within *slvRes* is equal to the desired output fractional width if Rounding is **not** selected. If Rounding **is** selected, an extra fractional bit will exist to allow rounding to occur. Therefore,

if 8 fractional bits are required from the output but Rounding is enabled, then 9 fractional bits will exist in the *slvRes* signal. The result from the divider will of course have only 8 bits after rounding has been performed. Figure 6.3 shows the Fixed Point Divider parameters window where the width of the integer and fractional result is specified. The option to Round and Saturate is also shown.

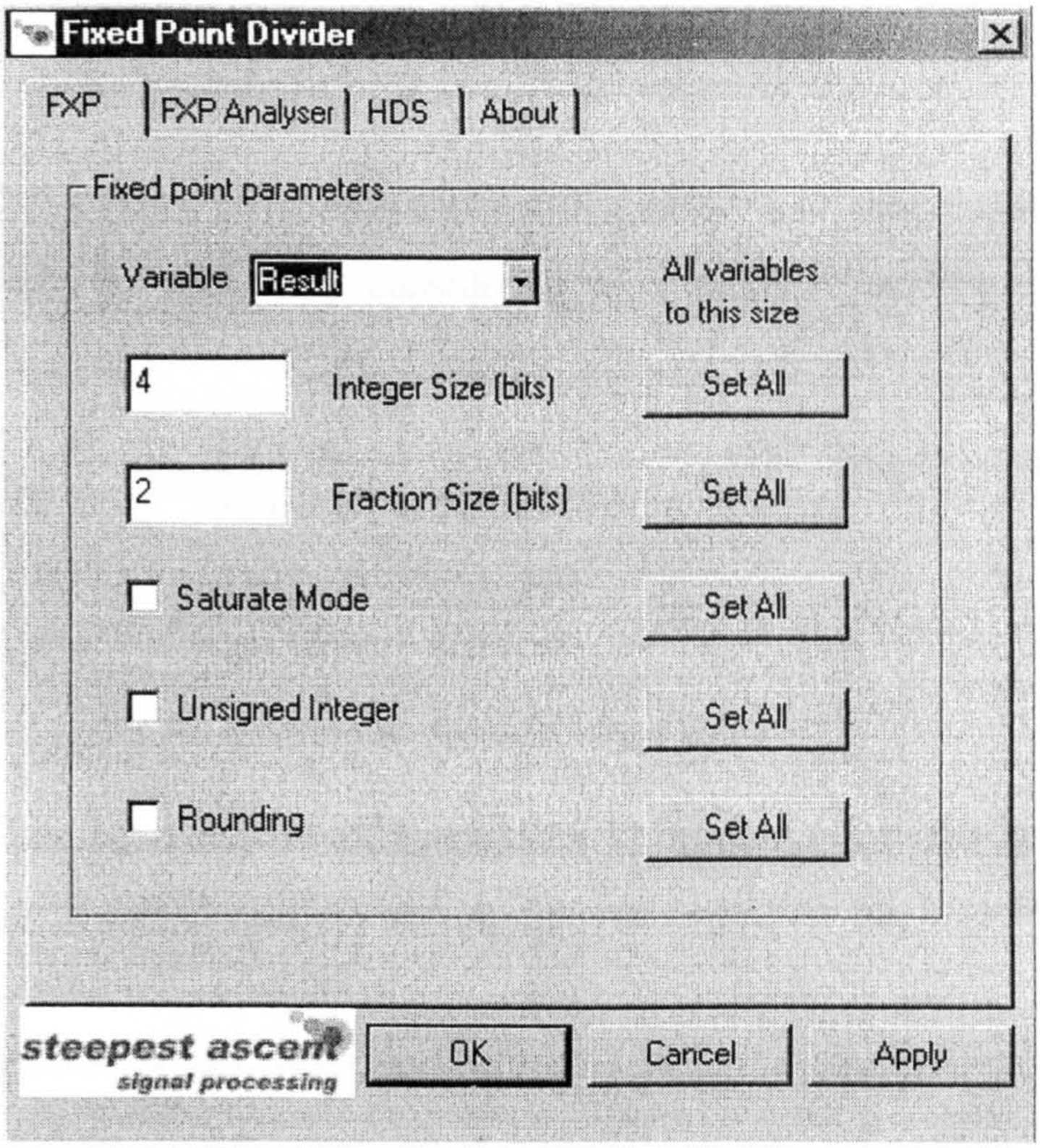


Figure: 6.3: Fixed Point Divider Parameters

6.3.3 Computing The Quotient

After the numerator and denominator signals have been converted into unsigned values, they are used to compute the unsigned quotient. In this section the hardware that is used to do this shall be focussed on.

Before the hardware is discussed, the number of bits that are required to represent the integer part of the result without overflow occurring is important to know.

As the division algorithm generates the most significant bits (MSB) of the quotient first, it is vital that enough iterations are performed to at least allow the full integer part of the solution to be generated. By using equation (6.1), the number of integer bits required can be easily computed.

$$\begin{aligned} \text{for signed division:} \quad & \frac{\pm\langle x, y\rangle}{\pm\langle a, b\rangle} = \pm\langle x + b + 1, ?\rangle \\ \text{for unsigned division:} \quad & \frac{\langle x, y\rangle}{\langle a, b\rangle} = \langle x + b, ?\rangle \end{aligned} \tag{6.1}$$

where $\pm\langle x, y\rangle$ represents a signed number with x integer and y fractional bits.

This can be proven by considering an example. Consider the following division:

$$\text{Consider: } \frac{\pm\langle 7,3\rangle}{\pm\langle 3,5\rangle} \longrightarrow \frac{largest}{smallest} = \frac{-64}{\pm 0.03125} = \mp 2048 \text{ —requires⟶ } \pm\langle 13, ?\rangle$$

$$\text{Consider: } \frac{\langle 7,3\rangle}{\langle 3,5\rangle} \longrightarrow \frac{largest}{smallest} = \frac{127.875}{0.03125} = 4092 \text{ —requires⟶ } \langle 12, ?\rangle$$

Figure: 6.4: Quotient Integer Width

Note that the number of fractional bits cannot be worked out in the same way. For example, 1/3 = 0.33333...,which recurs forever requiring an infinite number of fractional bits.

In the fixed point divider, an unsigned division is computed and the result is converted back into a signed number if required. Hence, the number of iterations through the algorithm is worked out according to the following:

$$\text{Iterations} = \underbrace{\text{INA Int. Width} + \text{INB Frac. Width}}_{\substack{\text{Iterations required to} \\ \text{generate the integer part} \\ \text{of the quotient}}} + \underbrace{\text{Desired Frac. Width}}_{\substack{\text{Iterations required to} \\ \text{generate the fractional part} \\ \text{of the quotient}}}$$

INA = numerator
INB = denominator

Note: if Rounding is enabled, an extra fractional bit is computed

Figure: 6.5: Required Iterations for Division Calculation

If the user does not require any fractional bits, then at least the full integer result will be computed. Also, if the user specifies that the output is to be Rounded, then an additional fractional bit will need to be computed and so one extra iteration will be performed.

The hardware used to compute the unsigned quotient is made up of a series of cells. Each cell computes one iteration of the division algorithm and thus generates one bit of the quotient. Hence, the number of cells is equal to the number of iterations. So for example, if the division was $\pm\langle 1, 4\rangle / \pm\langle 2, 2\rangle$ then 3 cells would be required to compute the integer part of the quotient, although once the result is converted back into signed format, there will be 4 integer bits. For this example, consider that the user wants 0 fractional bits of accuracy, but has enabled Rounding on the output. This means that 1 fractional bit will need to be computed so that Rounding can be performed, although the final result will have 0 fractional bits. Thus, a total of 4 cells are required to compute the unsigned quotient. This can be seen in Figure 6.6. Note that this figure has omitted the control signals (clock, reset, enable, ready) for clarity.

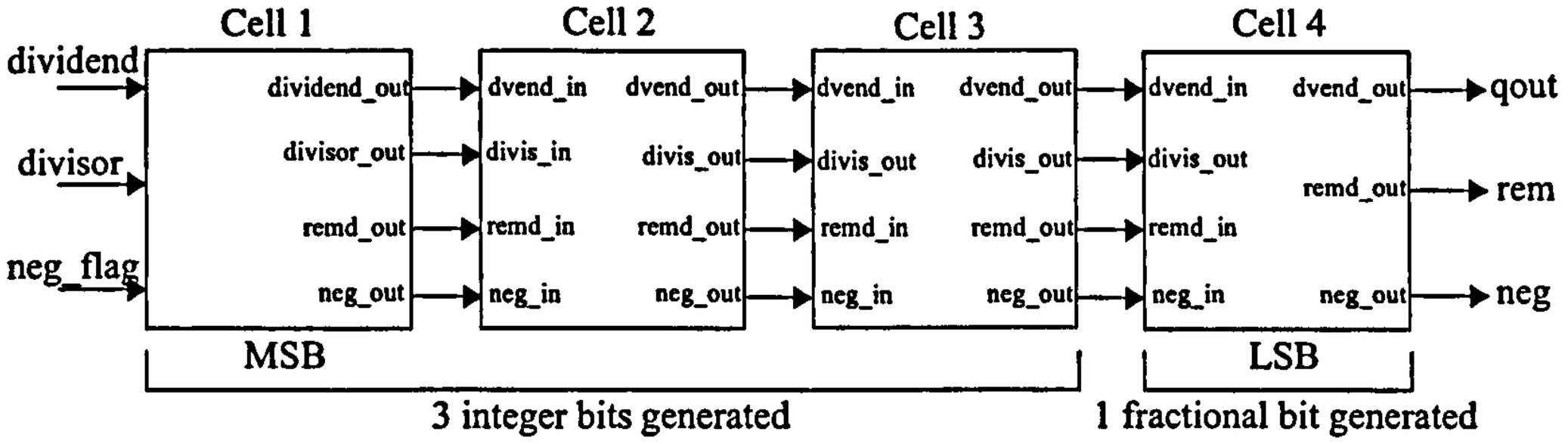


Figure: 6.6: 4 Cells Required to Compute 3 Integer Bits and 1 Fractional Bit

As mentioned previously, a *negative flag* must accompany each data sample pair to indicate whether or not the resulting unsigned quotient must be converted into a negative signed number. Figure 6.6 shows this flag passing through each cell accompanying the quotient that it relates to. The other signals passing through each cell are used as follows:

- *dividend_out:* is used to store the developing quotient as each bit is generated.

- *divisor_out:* is simply the divisor passing through each iteration. It does not change during the computation.
- *remd_out:* is used to store the partial remainder as it is generated at each iteration.

To look at the hardware that actually computes the division algorithm requires us to look within a cell. Figure 6.7 shows that a cell is made up of an adder, which is actually set up to subtract (invert bits of divisor_in and add 1), a comparator, two multiplexers and registers on each output signal. The registers in each cell are required to fully pipeline the design.

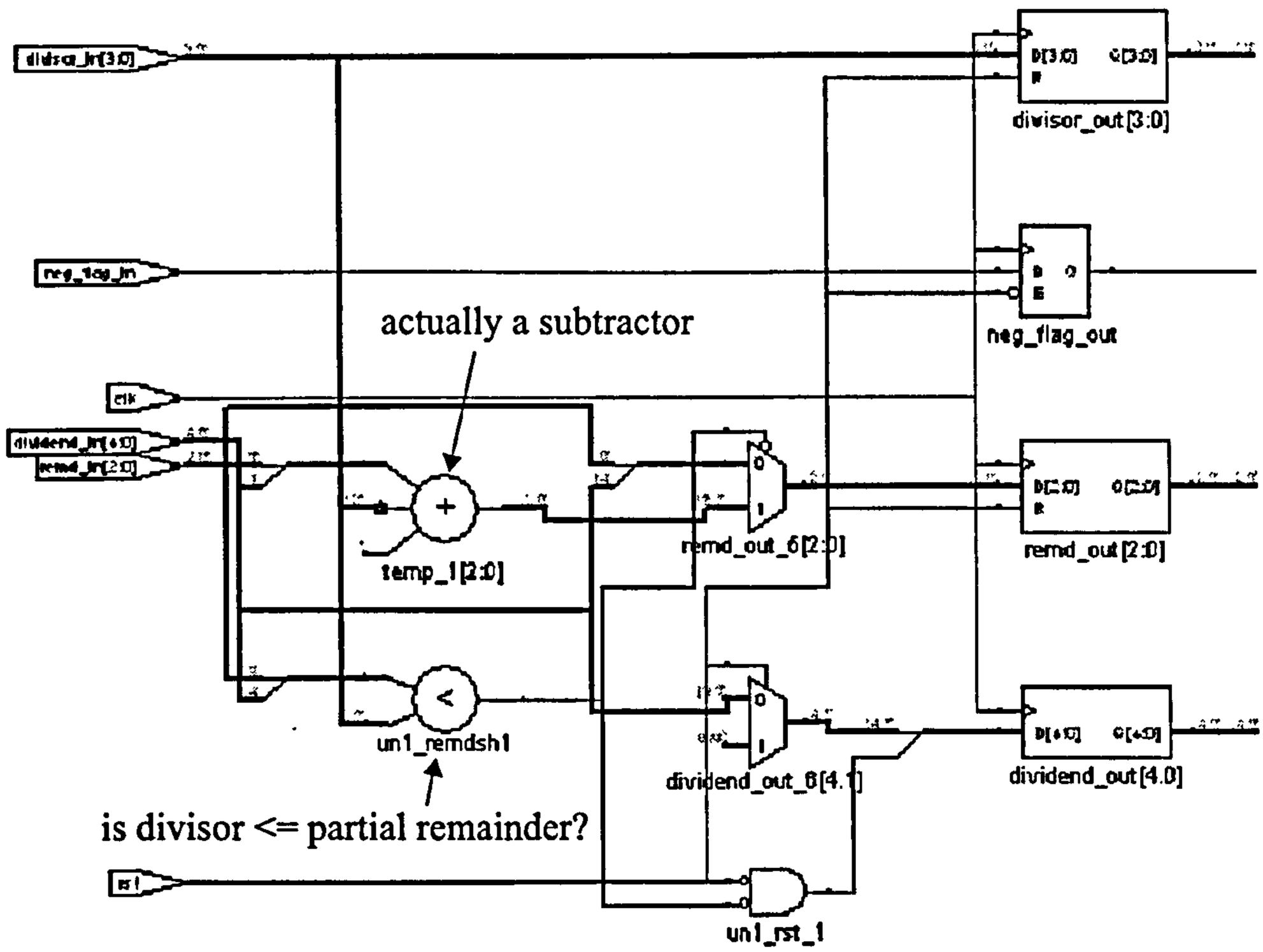


Figure: 6.7: Hardware Within A Division Cell

Note that two versions have been written. The first is for a fully pipelined divider, as shown in Figure 6.7, and the second implementation is for a fully combinatorial design where the pipelining registers on each data path in Figure 6.7 do not exist.

To understand how this hardware is used to generate the unsigned quotient, it is best to consider an example. Take the case $\pm\langle 5, 0\rangle / \pm\langle 5, 0\rangle$ and consider that the user has specified 0 fractional bits from the output with no Rounding. Hence, 5 cells are required to compute the unsigned quotient. Consider the division of 11/4. The individual steps that are carried out within each cell are now stepped through. Note that the ampersand (&) denotes the concatenation of two numbers.

Division Cell 1:

dividend_in = 01011, divisor_in = 00100, initial remdsh1 = 0000 & dividend_in(msb) = 00000;

divisor_in <= remdsh1 = False=> dividend_out = dividend_in & 0 = 10110

=> remd_out = remdsh1 (msb removed) = 0000

Division Cell 2:

dividend_in = 10110, divisor_in = 00100, remd_in = 0000;

remdsh1 = remd_in & dividend_in(msb) = 00001

divisor_in <= remdsh1 = False=> dividend_out = dividend_in & 0 = 01100

=> remd_out = remdsh1 (msb removed) = 0001

Division Cell 3:

dividend_in = 01100, divisor_in = 00100, remd_in = 0001;

remdsh1 = remd_in & dividend_in(msb) = 00010

divisor_in <= remdsh1 = False=> dividend_out = dividend_in & 0= 11000

=> remd_out = remdsh1(msb removed) = 0010

Division Cell 4:

dividend_in = 11000, divisor_in = 00100, remd_in = 0010;

remdsh1 = remd_in & dividend_in(msb) = 00101

divisor_in <= remdsh1 = True=> dividend_out = dividend_in & 1= 10001

=> remd_out = remdsh1(msb removed) - divisor_in(msb removed)

= 0101 - 0100 = 0001

Division Cell 5:

dividend_in = 10001, divisor_in = 00100, remd_in = 0001;

remdsh1 = remd_in & dividend_in(msb) = 00011

divisor_in <= remdsh1 = False=> dividend_out = dividend_in & 0 = 00010

=> remd_out = remdsh1(msb removed) = 0011

And so the result is: dividend_out = 00010 = 2

remd_out = 0011 = 3

6.3.4 Pipelined Design Latency

The latency through the pipelined design is dependent on the width of the numerator and denominator, and on the number of fractional bits required from the output. There are two registers that exist in the data path before the numerator and denominator reach the stage where the quotient is computed. These registers are used to break up the data path before the quotient is generated. Within the quotient generator there is a single delay in each cell. Hence, the total delay here is equal to the number of cells. Finally, there is a single delay before the quotient is passed out. The latency through the data path can be seen in Figure 6.8.

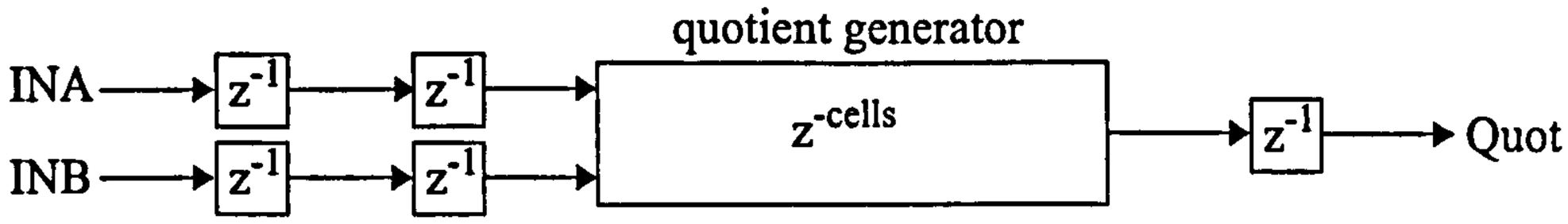


Figure: 6.8: Hardware Latency

Thus the latency is equal to:

$$\text{latency} = 2 + \text{number of cells} + 1 \qquad (6.2)$$

6.3.5 Folding The Pipeline

One feature of the divider that was researched but never fully implemented due to the

limitations of SystemVue, was the ability to optimise the core in terms of speed and area. The architecture illustrated so far is fully unrolled, thus for the pipelined divider the maximum data rate is equal to the maximum clock speed. This is the fastest implementation, however it is also the largest in terms of area. If area is the key constraint, rather than speed, it would be useful to optimise the design for this situation.

By considering an example it is possible to see just how this could be achieved. For example, if 4 cells are needed to compute a quotient, the architecture would look like that shown in Figure 6.9:

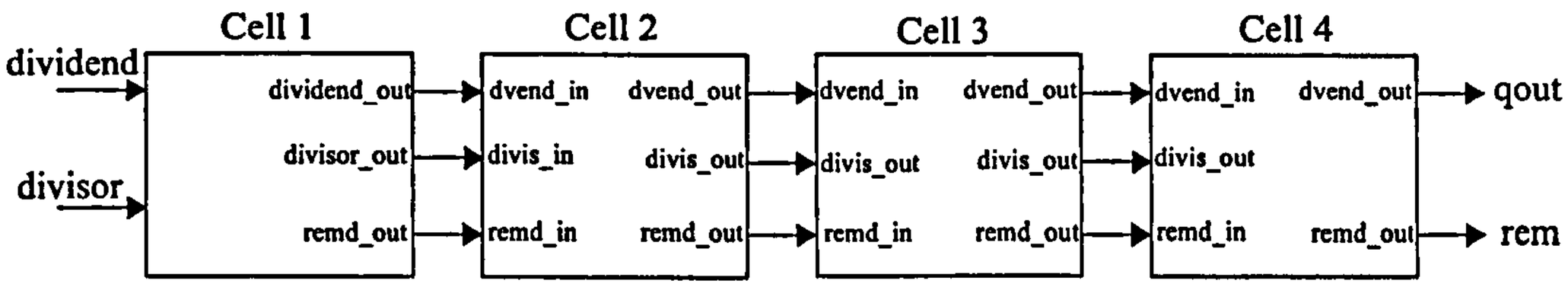


Figure: 6.9: 4 Cells Fully Unrolled

However, by "folding" the structure in two, the number of cells is halved. Then if the results from Cell 2 are fed back into Cell 1 the correct result can still be obtained. The trade-off is that data can only be fed into the structure at half the clock rate. Therefore, although the number of cells has been halved, the data rate has been halved too. If area is the key constraint rather than speed, then this is a better design than the fully unrolled approach. Figure 6.10 below illustrates the "folded" structure.

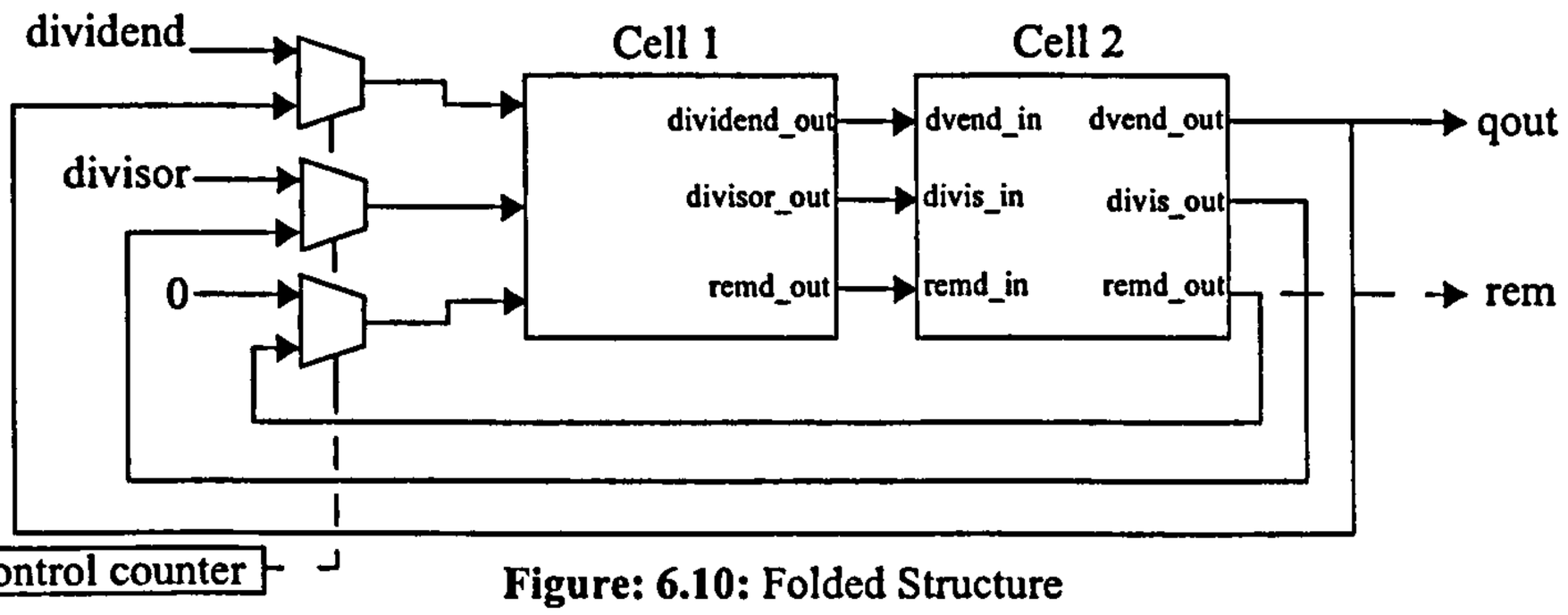


Figure: 6.10: Folded Structure

Folding is not straightforward though. A situation can occur where the feed

back data arrives at the same time as new data. Hence, a rule must be followed to avoid this situation occurring. The number of delays through the feedback section can be denoted as x and the number of clock cycles between arriving samples as y, then to avoid data collision, x and y must be coprime. This means that x and y must have no common factor other than 1 and -1. To illustrate what happens when this rule is broken, consider the following scenario where $x = 4$ and $y = 6$, which are clearly not coprime. As can be seen in Figure 6.11, data sample 3 (denoted by triangle) arrives at the multiplexer feeding the quotient generator at the same time as data sample 1 (denoted by circle) is fed back for the third time. Clearly this causes a problem as only one of these samples can enter the quotient generator, which means that one piece of data will be lost. By obeying the coprime rule this situation is avoided.

Unfortunately due to limitations with SystemVue the Folding option was never fully implemented. The problem with SystemVue was that it was not possible to have a token with an input rate of x Hz and an internal rate higher than this, which is required for such an architecture.

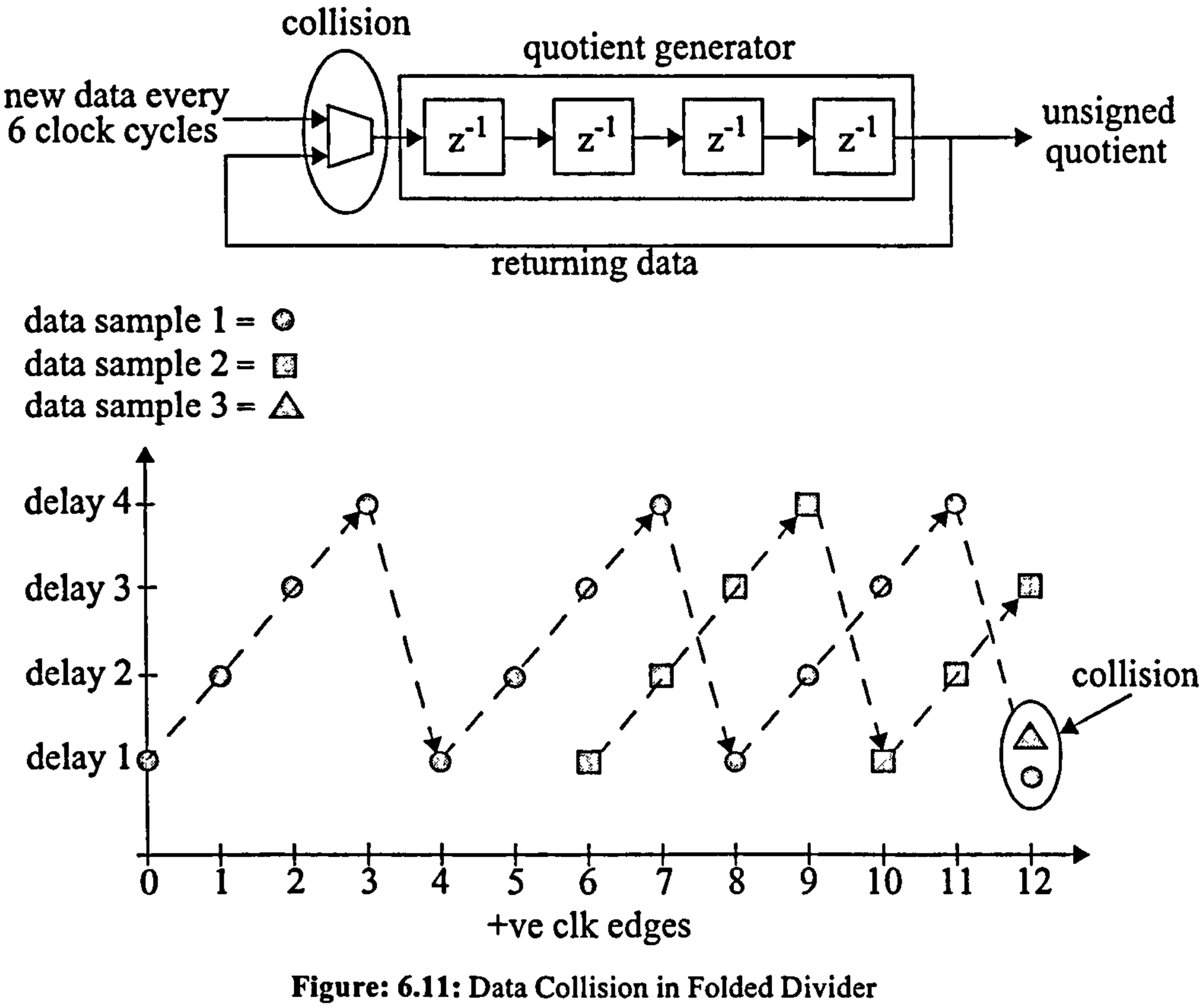


Figure: 6.11: Data Collision in Folded Divider

6.4 The Software Implementation

In this section the software model that was written in C++ to simulate the hardware divider is discussed. The code is used within a SystemVue token, which can be seen in

Figure 6.12 along with the token parameters window:

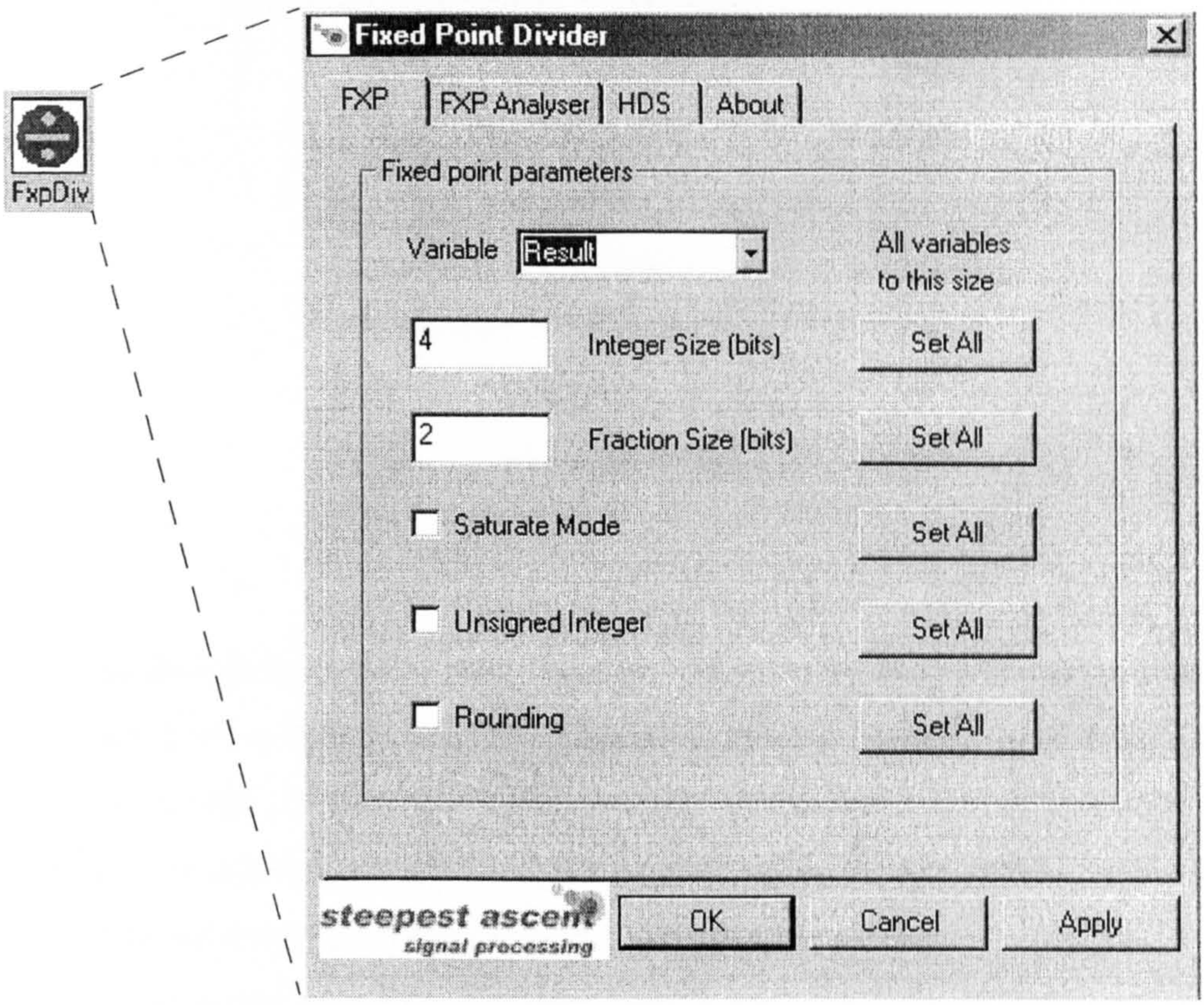


Figure: 6.12: Divider Token & Token Parameters

6.4.1 Pseudo Code for Software Division

To compute binary long division in software, the following pseudo code was developed. As was illustrated in the decimal and binary long division examples given earlier, the number of cycles through the algorithm is determined by the width of the dividend.

The remainder variable is used to build the partial remainders at each stage of the algorithm and the dividend is used to store the quotient as it is generated. The divisor does not change during the algorithm. By shifting the remainder variable left 1 position (step 1) room is created in the LSB to add a 1 (or not) depending on what the

```
initially remainder = 0
for(i = 0; i < dividend_width; i ++)
{       remainder = remainder << 1;      //step 1
        if (dividend msb = 1)            //step 2
                remainder ++;
        dividend = dividend << 1;        //step 3
        if (divisor <= remainder)        //step 4
        {       dividend ++;
                remainder = remainder - divisor;
        }
}
```

MSB of the divisor is at that point in the algorithm (step 2). Step 3 is used to shift the dividend left 1 bit so that the MSB is updated for the next iteration. Step 4 deals with whether or not the divisor will divide into the partial remainder. If it will, a 1 is generated in the quotient, which the dividend holds. Also, the new partial remainder is calculated by subtracting the divisor from the current remainder.

To illustrate how this algorithm maps to the long division technique shown earlier requires stepping through an example. For this, consider the example 30/4.

$$30/4 = 11110/100$$

$$
\begin{array}{r}
00111 = 7 \\
100\overline{)\,11110} \\
\underline{0} \\
11 \\
\underline{00} \\
111 \\
\underline{100} \\
0111 \\
\underline{0100} \\
00110 \\
\underline{00100} \\
00010
\end{array}
$$

1st cycle: step1: remainder starts at 0

step2: dividend msb = 1 => remainder = 1

step3: dividend = 11100

step4: does divisor (100) divide into remainder (1)? No

2nd cycle: step1: remainder = 10

step2: dividend msb = 1 => remainder = 11

step3: dividend = 11000

step4: does divisor (100) divide into remainder (11)? No

3rd cycle: step1: remainder = 110

step2: dividend msb = 1 => remainder = 111

step3: dividend = 10000

step4: does divisor (100) divide into remainder (111)? Yes

dividend = 10001 (use dividend to store result)

remainder = 111 - 100 = 011

4th cycle: step1: remainder = 0110

step2: dividend msb = 1 => remainder = 0111

step3: dividend = 00010

step4: does divisor (100) divide into remainder (0111)? Yes

dividend = 00011 (use dividend to store result)

remainder = 0111 - 100 = 0011

5th cycle: step1: remainder = 00110

step2: dividend msb = 0 => remainder = 00110

step3: dividend = 00110

step4: does divisor (100) divide into remainder (00110)? Yes

dividend = 00111 (use dividend to store result)

remainder = 00110 - 100 = 00010

Hence, the result is correct:

- quotient = dividend = 00111 = 7, remainder = 00010 = 2

6.4.2 Full C++ Model

The pseudo code shown in Chapter 6.4.1 is used to compute unsigned division. Therefore, to use it with signed inputs requires the inputs to be converted into unsigned numbers and then the result corrected, similar to the way the VHDL works. There are several steps within the full model which are summarised below and in Figure 6.13.

- Step 1 - At this stage if the denominator is equal to 0, then a precomputed result is returned which matches with the VHDL output for this event.
- Step 2 - The numerator and denominator are now converted to unsigned numbers and the *isResultNegative* flag is set appropriately.
- Step 3 - The number of loops that are required to compute the result is covered in detail in the following subsection.
- Step 4 - The unsigned result is computed using a C++ version of the pseudo code shown earlier.
- Step 5: The quotient is converted back into signed (+ve/-ve) if necessary and then returned.

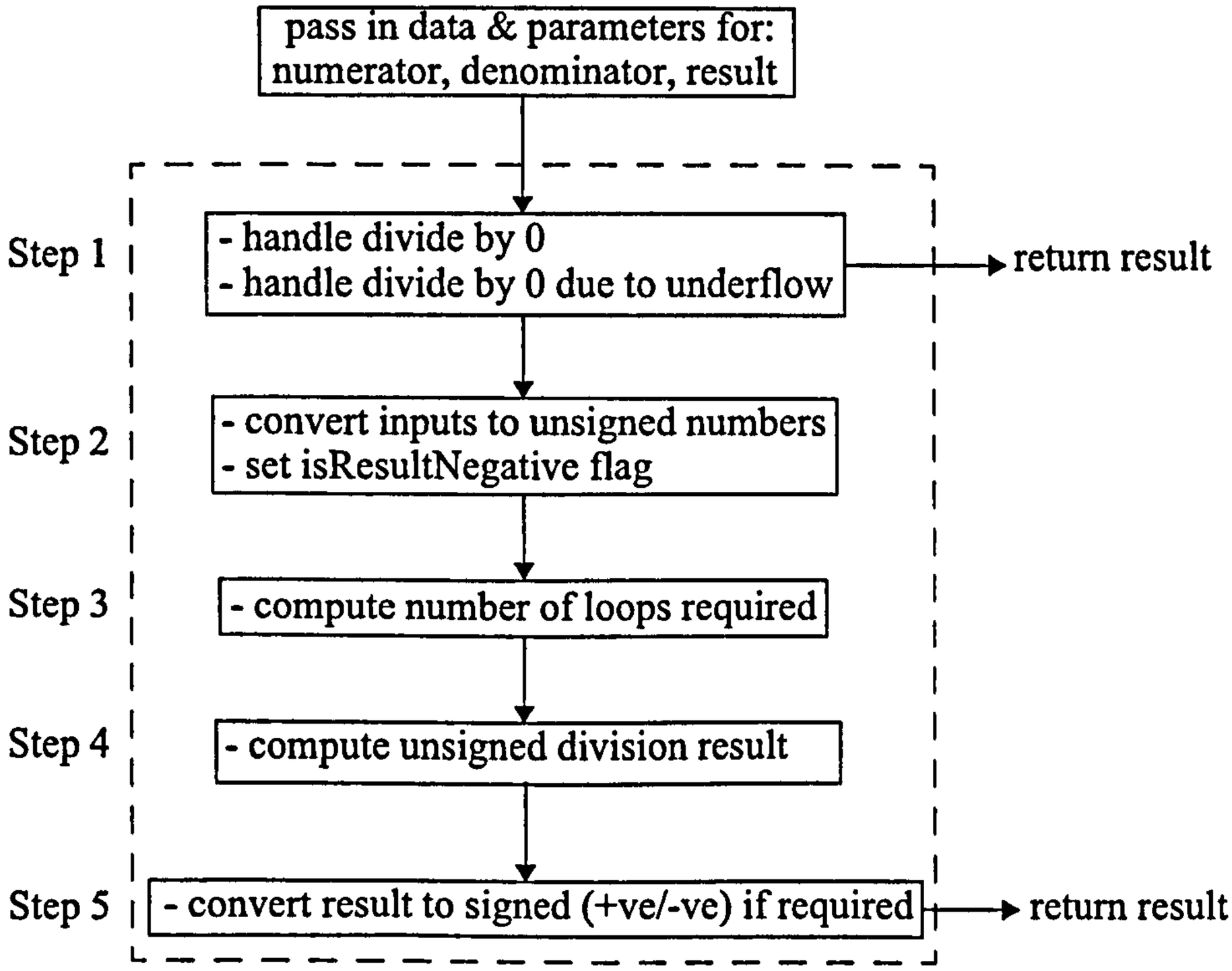


Figure: 6.13: DivideFXP Data Flow Diagram

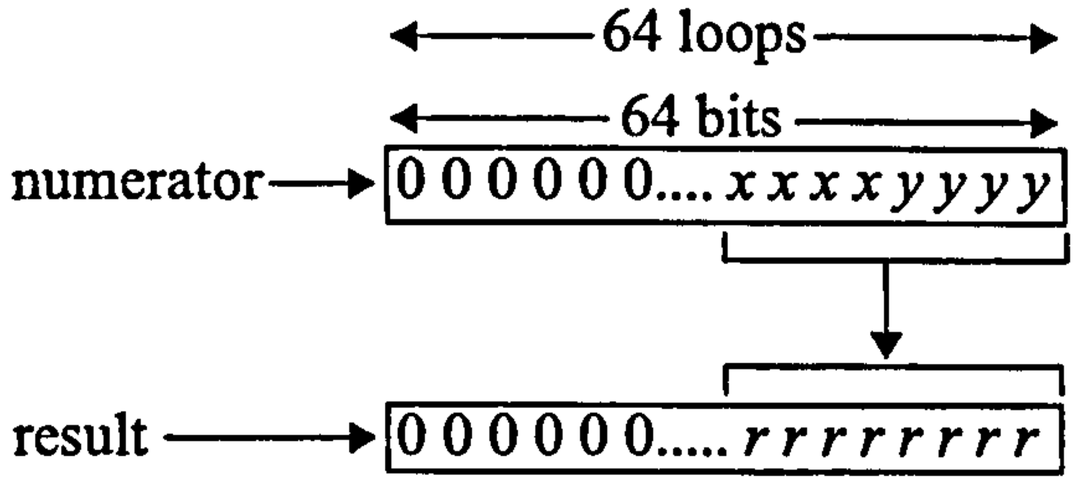
6.4.3 How Many Loops?

The algorithm shown earlier is used as the basis of the full software model. However, the number of times that the algorithm is looped around varies depending on the numerator and denominator widths and with the number of fractional bits in the output requested by the user, as can be seen in (6.3). Before discussing the derivation of the equation for the number of loops, it is important to realise that at this stage the numerator and denominator are stored in 64 bit unsigned integer variables.

For the case: $\frac{\pm\langle x, y\rangle}{\pm\langle a, b\rangle}$ where f fractional bits are required in the output

$$\text{loops} = 64 + b + f - y \tag{6.3}$$

The derivation of (6.3) is now given below:



For 64 loops of the divider algorithm, $x + y$ bits of the result will be generated.

However, the number of bits that are required from the result are: $x + b + f$ bits. Therefore, the difference that needs to be accounted for is:

$$\text{difference} = (x + b + f) - (x + y) = b + f - y$$

This means that the total number of loops required is:

$$\text{loops} = 64 + \text{difference} = 64 + b + f - y$$

6.4.4 Replacing The Binary Point

The output from the division algorithm is contained in an unsigned 64 bit variable. At this point it is converted into a signed format if either of the inputs was signed otherwise it is left unsigned. Before the result is returned it is vitally important that the binary point is inserted in the correct place. However, it has already been shown that for signed and unsigned division, the result takes the format given by (6.4).

$$\begin{aligned} \frac{\pm\langle x, y\rangle}{\pm\langle a, b\rangle} &= \pm\langle x + b + 1, f\rangle \\ \frac{\langle x, y\rangle}{\langle a, b\rangle} &= \langle x + b, f\rangle \end{aligned} \tag{6.4}$$

6.4.5 Simulating The Delay

As the C++ model must simulate the hardware divider, it is important to replicate the delay between the first sample entering the core and the corresponding result reaching the output. This delay is called latency. In this case the latency is a function of the width of the inputs and the desired fractional output. The relationship is given in (6.5) which is another way of writing (6.2).

$$\text{latency} = x + b + f + 3 \tag{6.5}$$

Again, if the user has enabled Rounding on the output, then an additional fractional bit ($f+1$) will be computed and consequently the latency will increase by 1.

6.4.6 Performance

To give an indication of how many resources and how fast the hardware Divider can operate at, several instances were implemented using the following:

- Target Technology: *Xilinx Virtex II Pro XC2VP30*
- Synthesis: *Synplify Pro 7.1*
- Map and Place & Route: *Xilinx ISE 8.1*

Table 4 and Table 5 show the results for the pipelined and non pipelined cores respectively. In both cases, dividers were implemented with 48, 32 and 16 bits in the inputs and the same number of bits on the output. As would be expected, the pipelined implementations are bigger and significantly faster than the corresponding non pipelined implementations.

Divider I/O	Slices	LUTs	Clock Speed
48 bits	6895	6930	>100 MHz
32 bits	3027	3066	>104 MHz
16 bits	746	769	>134 MHz

Table 4: Pipelined Divider Results

Divider Impl.	Slices	LUTs	Clock Speed
48 bits	2552	4882	>2 MHz
32 bits	1184	2224	>4 MHz
16 bits	315	579	>9 MHz

Table 5: Non Pipelined Divider Results

6.5 Direct Square Root

Performing a direct square root on an integer can be achieved using a pencil and paper technique. As fixed-point binary numbers can be considered as integers it is possible to use the paper and pencil method to calculate the square root of fixed-point binary numbers, although the binary point must be tracked and re-inserted to the result.

As with division there are fast iterative techniques that require an estimation of the root to begin the algorithm such as the Goldschmidt technique [10]. However, it is not easy to predict the accuracy of such techniques for a given number of iterations and hence a direct approach was taken.

Decimal Square Root

The paper and pencil technique for performing a direct square root can be summarised by the following steps:

1. split argument into pairs (starting from left hand side)
2. write down 1st square root by inspection
3. subtract its square from 1st two digits
4. draw down next two digits to obtain remainder
5. double square root and append 0 (= *approximate divisor*)
6. **estimate** next root digit by dividing remainder by approximate divisor
7. substitute next root digit for last digit of approximate divisor (= *divisor*)
8. **verify** next root digit by dividing remainder by divisor. If result = next root digit, ok, otherwise backtrack
9. multiply next root digit by divisor
10. subtract result from remainder
11. repeat from step 4 until done

The best way to illustrate the direct method for calculating a decimal square root is by considering an example. Consider the calculation $\sqrt{123456.0000}$:

```
      3  5  1. 3  6  <-- result
    +-------------
  3 | 12 34 56.00 00
      9
    --
 65   3 34
      3 25
      ----
701      9 56
         7 01
         ----
7023     2 55 00
         2 10 69
         -------
70266      44 31 00
           42 15 04
           --------
            2 15 04
```

Working through the steps it is clear how the above working is obtained:

1. split argument into pairs
2. $\sqrt{12} = 3$ (square root)
3. $12 - 3^2 = 3$
4. remainder = 3 34
5. 3*2 = 6, append 0 = 60 (= *approximate divisor*)
6. $\lfloor 334/60 \rfloor = 5$
7. substitute 5 into 60 = 65 (= *divisor*)
8. $\lfloor 334/65 \rfloor = 5$, therefore ok (no need to backtrack)
9. 65*5 = 325
10. 334 - 325 = 9
4. remainder = 9 56
5. 35*2 = 70, append 0 = 700
6. $\lfloor 956/700 \rfloor = 1$
7. substitute 1 into 700 = 701
8. $\lfloor 956/701 \rfloor = 1$, therefore ok (no need to backtrack)
9. 701*1 = 701
10. 956 - 701 = 255

etc.

One point to note from this working is that the number of cycles through the algorithm depends upon the number of digits within the operand. More specifically, it is clear that each pair of operand digits yields one digit of the root. Hence, the number of cycles

through the algorithm is equivalent to the number of pairs formed from the argument.

Another important point to note from the above example is that the accuracy of the result can be increased by padding the argument with zeroes. Without padding the argument, the integer result (351) would only have been obtained. Therefore, with every two zeroes that are padded to the argument, a further decimal point of accuracy is obtained.

Binary Square Root

The direct method for calculating a binary square root is easier than decimal. This is now explained via an example where $\sqrt{5.75}$ is computed:

```
          1   0 . 0   1 = 2.25
        ┌──────────────
        │ 01 01.11 00
        │ 01
    101 │ 00 01
   1001 │ 00 01 11
  10001 │ 00 01 11 00
        │ 00 01 00 01
        ├──────────────
        │ 00 00 10 11
```

The steps involved with this working are:

1: split the argument into pairs

2: as long as the first pair of bits ≠ 0, the first root is always = 1

3: subtract root from 1st pair (01 -01) to give partial remainder (0)

4. append next pair (01) to partial remainder to form remainder (0 01)

5. append 01 to the root (1) to form divisor (101)

6. divisor > remainder therefore root = 0

7. append next pair (11) to remainder to from new remainder (0 01 11)

8. append 01 to the root (10) to form divisor (10 01)

9. divisor > remainder therefore root = 0

10. append next pair (00) to remainder to from new remainder (0 01 11 00)

11. append 01 to the root (100) to form divisor (10001)

12. divisor < remainder therefore root = 1

13. subtract divisor from remainder to form partial remainder (= 10 11)

6.6 Specification

The specification that was developed for a Square Root core to be included within the HDL Design Studio library included the following:

Develop a parameterisable VHDL core able to:

- Compute the square root of either unsigned or signed fixed point inputs. Unsigned fixed-point numbers are not supported in HDS, thus to have fixed-point square root functionality meant that signed fixed-point numbers must be supported.
- Return the output in either signed fixed point or unsigned integer format at the request of the user.
- Allow the user to specify output parameters such as the integer and fractional widths of the output. Also, the option to Round and Saturate should be available.
- Be pipelined to increase throughput.

Develop a C++ model able to:

- Mimic the VHDL exactly, hence must be bit and cycle accurate.

6.7 The Hardware Implementation

In this section, the hardware implementation that was designed for computing a direct square root is presented. The architecture is derived from the illustrated paper and pencil technique.

6.7.1 The Top Level

The Square Root core has a fixed number of I/O ports which include input ports for the Input, Clock, Enable and Reset signals. The output ports include the Sqrt and Ready signals. Similar to the Divider, generic parameters are used within the VHDL to pass information on the width and data type of the input and output data signals as well as whether rounding, saturation or truncation is to be used. A summary of the information used to configure the top-level of the Square Root core can be seen in Figure 6.14.

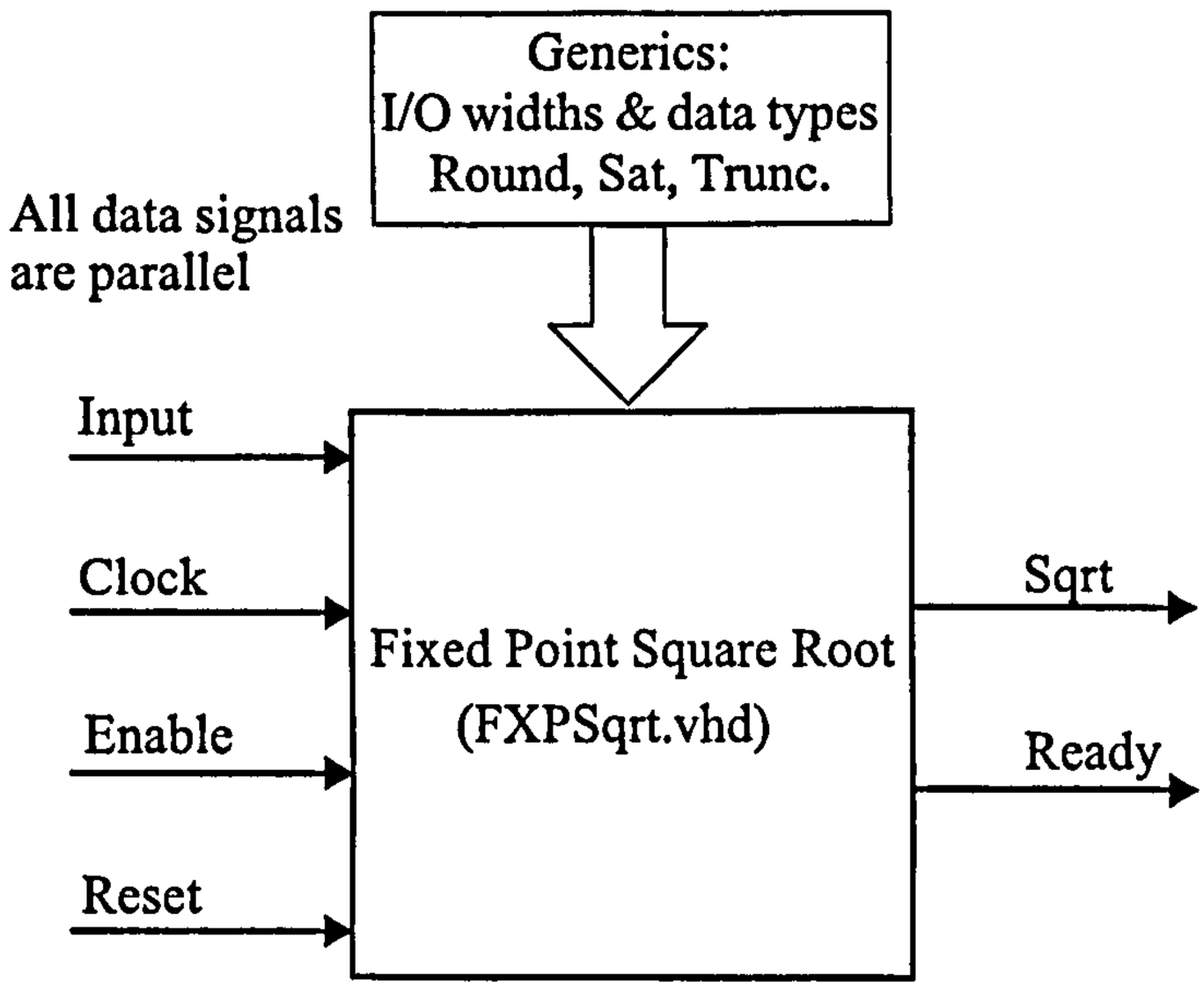


Figure: 6.14: Fixed Point Square Root Top Level

6.7.2 Inside The Fixed Point Square Root Core

Inside the Square Root core there are several steps that are carried out to allow the result to be generated correctly. In Figure 6.15 the sequence of operations are shown.

- Step 1: At this point the input is either truncated or padded with zeroes to produce an input that has twice as many bits as the desired output. This is discussed in more detail in the next subsection.
- Step 2: Here the square root is computed using an algorithm based on the paper

and pencil technique shown earlier.

- Step 3: As with the Divider, a final stage exists to convert the output to the desired format which may include Saturation and/or Rounding.

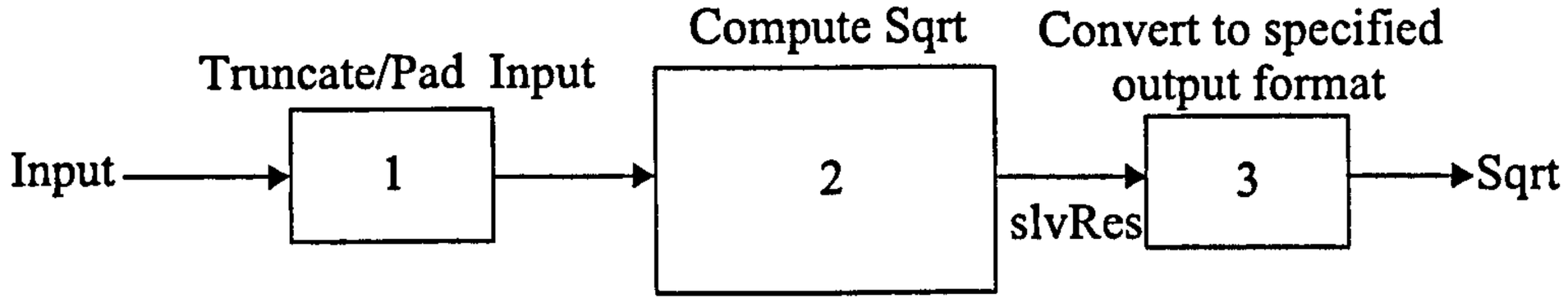


Figure: 6.15: Square Root Sequence Of Operations

6.7.3 Truncate/Pad Input

Depending on the width of the input fractional part and the desired output fractional width, the input may be either truncated or padded with zeroes. Truncation of the input fractional bits will occur if there are more fractional bits than are needed to produce the required level of accuracy. So for example, if the input has 10 fractional bits then the output will have 5 fractional bits. But if only 2 fractional bits are required from the output, then only 4 fractional bits are required at the input, not 10. Thus by truncating the input, computing unnecessary bits is avoided. Conversely, if the input has only 4 fractional bits but the user requires 10 fractional bits from the output, then the input needs padded with 16 zeroes so that 20 fractional bits exist in the input. By doing this the output will have 10 fractional bits of accuracy.

6.7.4 Computing The Square Root

Unlike division, it is impossible for the integer width of the output of a square root to be greater than the integer width of the input. The integer output width obeys the following for signed and unsigned inputs:

$$\text{for signed input:} \quad \sqrt{\pm\langle x, y\rangle} = \pm\langle \frac{x}{2} + 1, ? \rangle$$
$$\text{for unsigned input:} \quad \sqrt{\langle x, y\rangle} = \langle \frac{x}{2}, ? \rangle \tag{6.6}$$

This can be shown by considering the following example. Consider the following division:

The number of fractional bits cannot be worked out in the same way, as for many inputs an infinite amount of fractional bits may be required. Hence, the number of fractional bits to be calculated must be specified by the user.

When the Square Root algorithm was discussed earlier, it was clear that for every 2 bits of the input, an output bit was generated. This is an important point to remember when calculating the number of iterations required to generate a result with a particular resolution. The algorithm for computing the number of iterations is given below:

$$\text{Iterations} = \underbrace{\frac{\text{Input Int. Width}}{2}}_{\text{Iterations required to generate the integer part of the result}} + \underbrace{\text{Desired Frac. Width}}_{\text{Iterations required to generate the fractional part of the result}}$$

Note: if Rounding is enabled, an extra fractional bit is computed

Figure: 6.16: Required Iterations for Division Calculation

So that there are twice as many bits in the input as there are iterations, the input may be either truncated or padded with zeroes. So for example, if the input has the format $\pm\langle 4, 10\rangle$, then the natural output would be $\pm\langle 3, 5\rangle$ which would require 7 iterations. However, the user may only want 2 fractional bits from the output, thus the input would be truncated to a $\pm\langle 4, 4\rangle$ format to cut down the number of iterations to 4 and save computing unnecessary bits. Conversely, the user may require 10 fractional bits from

the output, thus the input would have to be padded to a $\pm\langle 4, 20\rangle$ format and 12 iterations would be required. If the user does not require any fractional bits, then at least the full integer result will be computed. Also, if the user specifies that the output is to be Rounded, then an additional fractional bit will be computed, thus one extra iteration will be performed.

Like the divider, the hardware used to compute the unsigned quotient is made up of a series of cells. Each cell computes one iteration of the algorithm and thus generates one bit of the result. Hence, the number of cells is equal to the number of iterations. So for example, if the input format was $\pm\langle 4, 2\rangle$ then 2 cells would be required to compute the integer part of the result. For this example, also consider that the user wants 1 fractional bit of accuracy and has enabled Rounding on the output. This means that 2 fractional bit will need to be computed so that Rounding can be performed, although the final result will have 1 fractional bit. Thus, a total of 4 cells are required to compute the result. This can be seen in Figure 6.17. Note that this figure has omitted the control signals (clock, reset, enable, ready) for clarity.

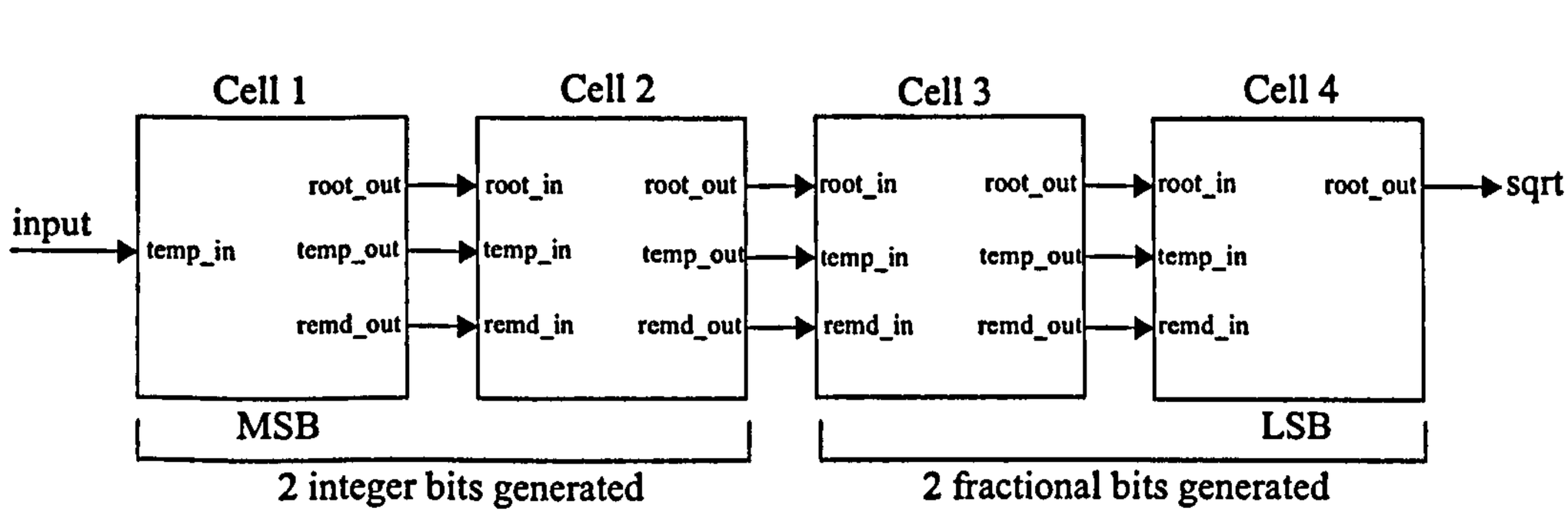


Figure: 6.17: 4 Cells Required to Compute 2 Integer Bits and 2 Fractional Bits

The signals passing through each cell are used as follows:

- *root_out:* is used to store the developing result as each bit is generated.
- *temp_out:* is simply the input passing through each iteration. Each time it is shifted left two places thus the 2 MSBs are ready for the next iteration.
- *remd_out:* is used to store the partial remainder as it is generated at each

iteration.

The actual hardware within a cell is shown in Figure 6.18. Note how similar it is to the hardware used in the Divider cell. It is made up of an adder, which is actually set up to subtract (same as divider), a comparator, a multiplexer and registers.

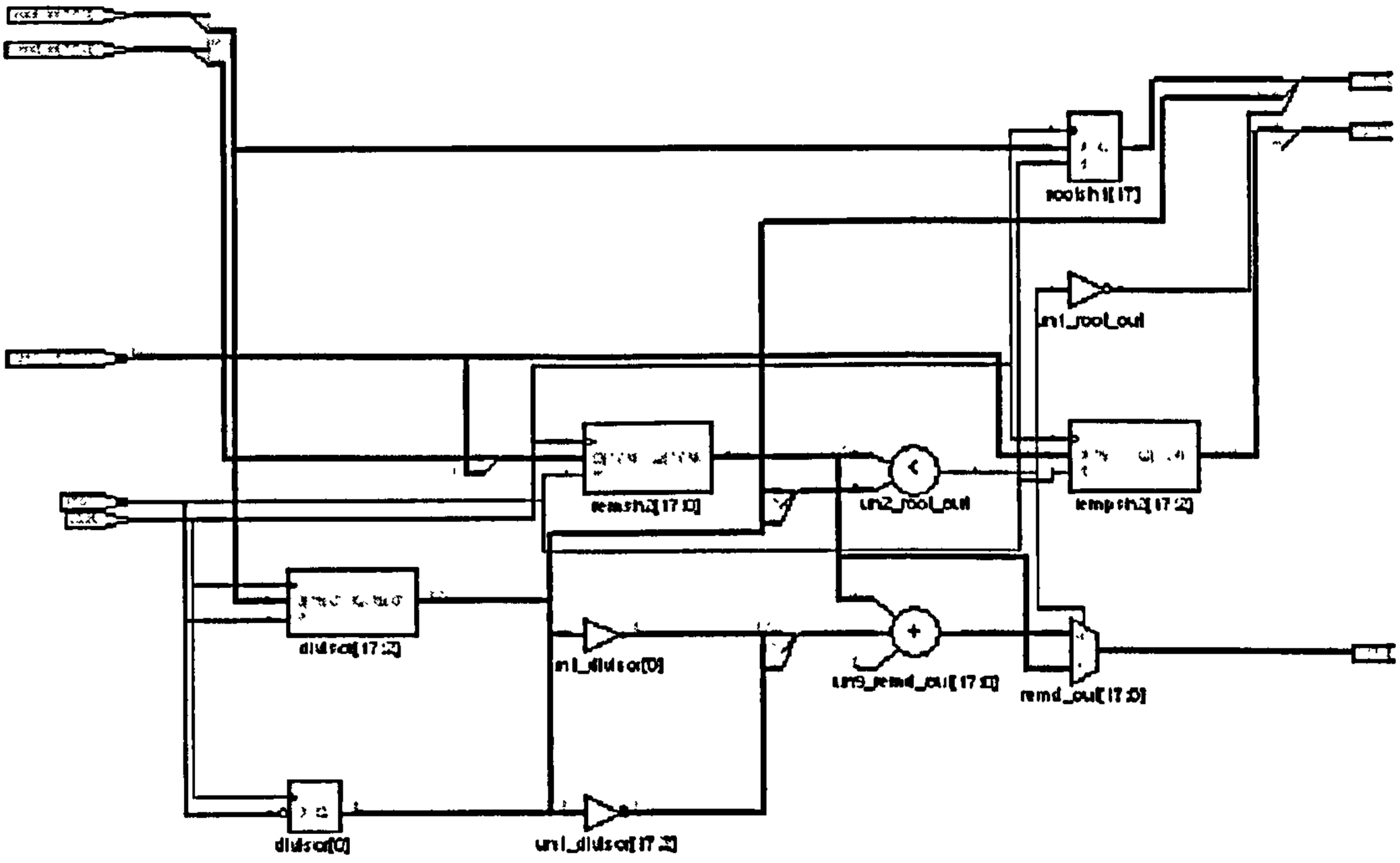


Figure: 6.18: Hardware Within A Square Root Cell

Two versions have been written, one fully pipelined and one fully combinatorial. Figure 6.18 shows the hardware within a pipelined cell. The combinatorial version is exactly the same although there are no registers within the cells.

6.7.5 Pipelined Latency

The latency through the pipelined design is dependent on the width of the input and on the number of fractional bits in the result requested by the user. There are two registers, each with a single delay, that exist in the data path before the input gets to the stage where the square root is computed. These registers break up the data path between the

input being truncated/padded and entering the square root generator. Within the square root calculator there is a single delay in each cell. Hence, the total delay here is equal to the number of cells. Finally, there is a single delay after the result is converted to the final output format. The latency through the data path can be seen in Figure 6.19.

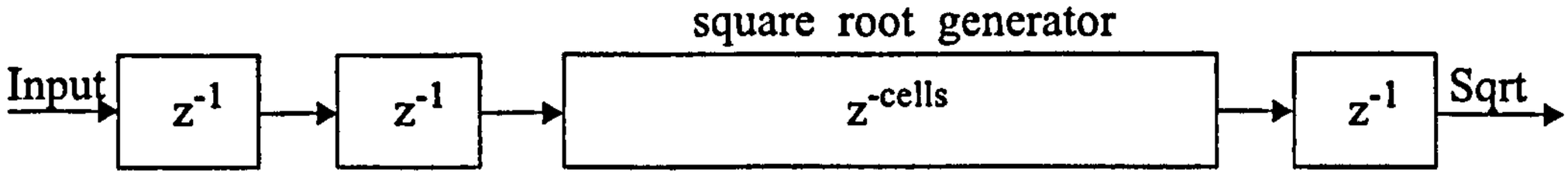


Figure: 6.19: Hardware Latency

Thus the latency can be computed according to:

$$\text{latency} = 2 + \text{number of cells} + 1 \tag{6.7}$$

6.8 The Software Implementation

This section presents the software version of the Square Root core, which was written in C++ to model the hardware design. This implementation must be cycle and bit accurate when compared to the hardware. The code is included within a SystemVue token, which can be seen in Figure 6.20.

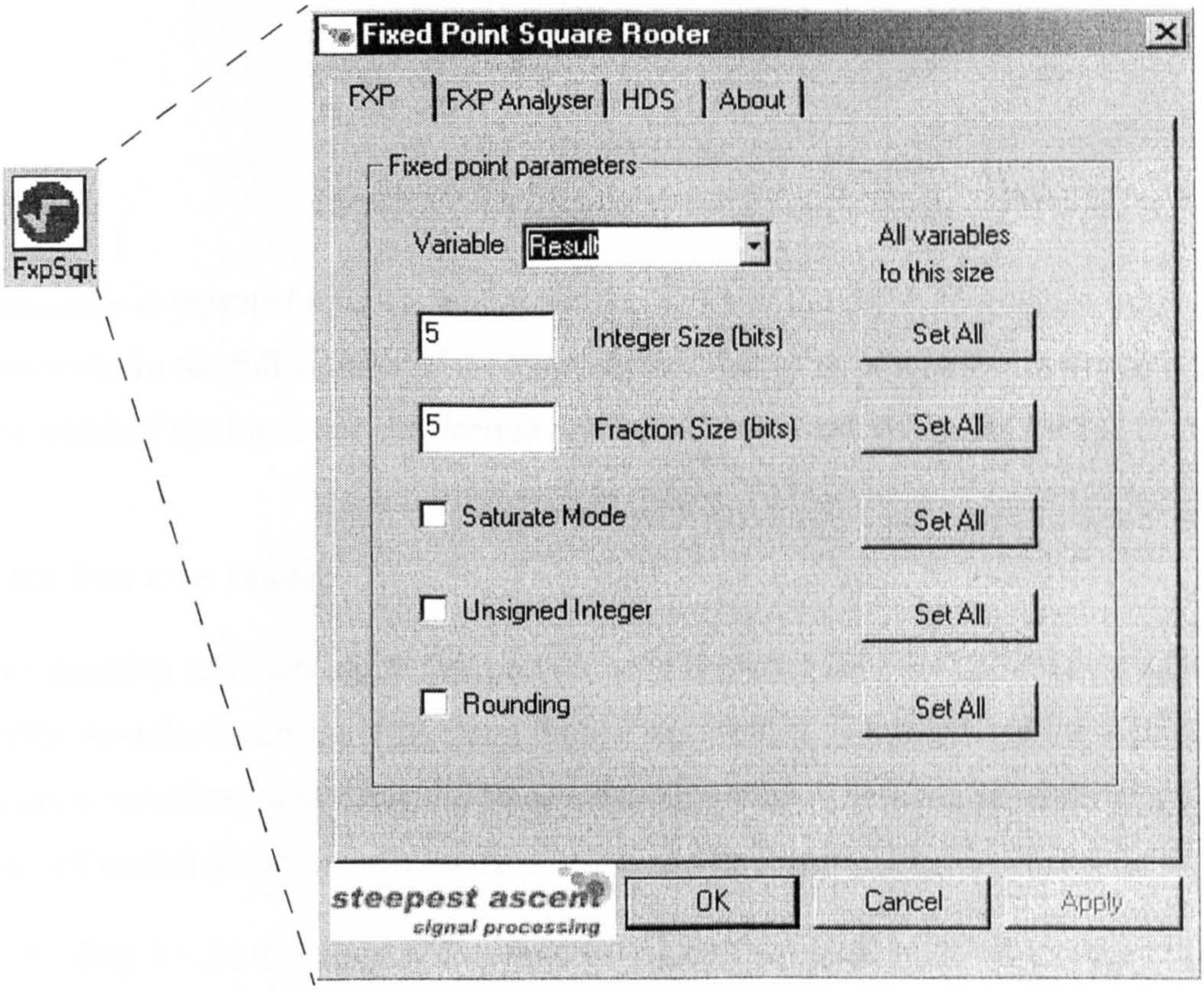


Figure: 6.20: Square Root Token & Token Parameters

6.8.1 The Basic Algorithm

Pseudo code representing an algorithm for calculating direct square roots is now given:

Initially the variables *rem*, *root* and *divisor* = 0.

```
for (i=0; i < (input_width); i++)
{
    root = root << 1;
    rem = ((rem << 2) + (input >> (input_width - 2)));
    input = input << 2;
    divisor = (root<<1) + 1;
    if (divisor <= rem)
```

```
        {
                rem = rem - divisor;
                root++;
        }
    }
```

This code is repeated *n* times, where *n* is the width of the input. Hence, the output is *n* bits wide. In the full version of the code, the number of iterations varies depending on the width of the input and the desired number of fractional bits in the output.

6.8.2 Full C++ Model

For negative and zero inputs the pseudo code shown earlier will produce results that differ from the hardware. Therefore, these cases must be identified and the appropriate result returned without using the square root algorithm. There are several steps within the full model which are summarised below and in Figure 6.21.

- Step 1 - At this stage if the input is negative or equal to 0, then a precomputed result is returned which matches the VHDL output for these events.
- Step 2 - The number of loops that are required to compute the result is covered in detail in the following subsection.
- Step 3 - The square root is computed using a C++ version of the pseudo code shown earlier.
- Step 4 - The result is converted to a signed format if the input was signed. Also, Saturation and/or Rounding are performed if requested. Note also that the integer result may be extended of truncated depending on the number of integer bits requested by the user. Hence, it is entirely down to the user to request enough bits to fully represent the integer part of the result.

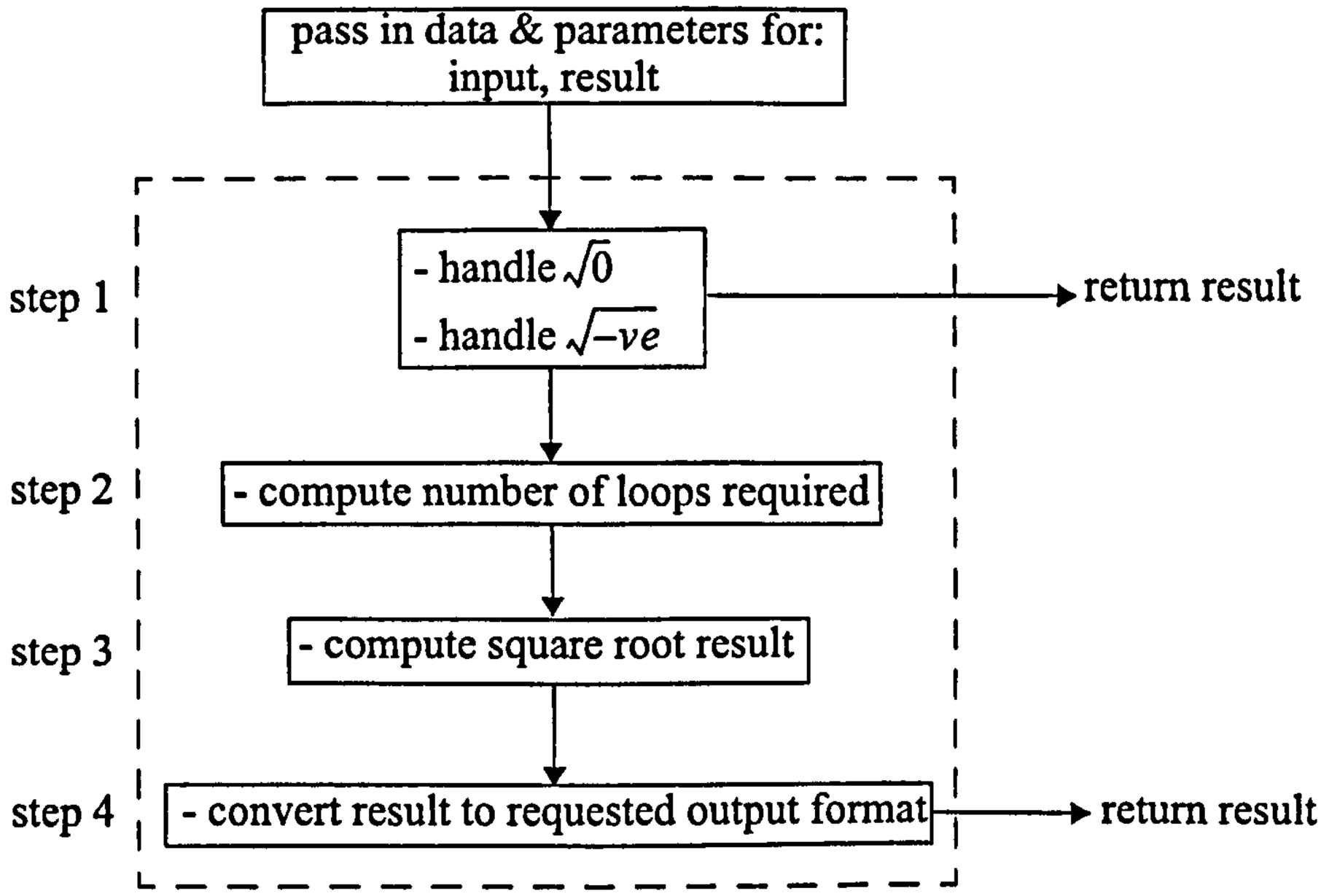


Figure: 6.21: SqrtFXP Data Flow Diagram

6.8.3 How Many Loops?

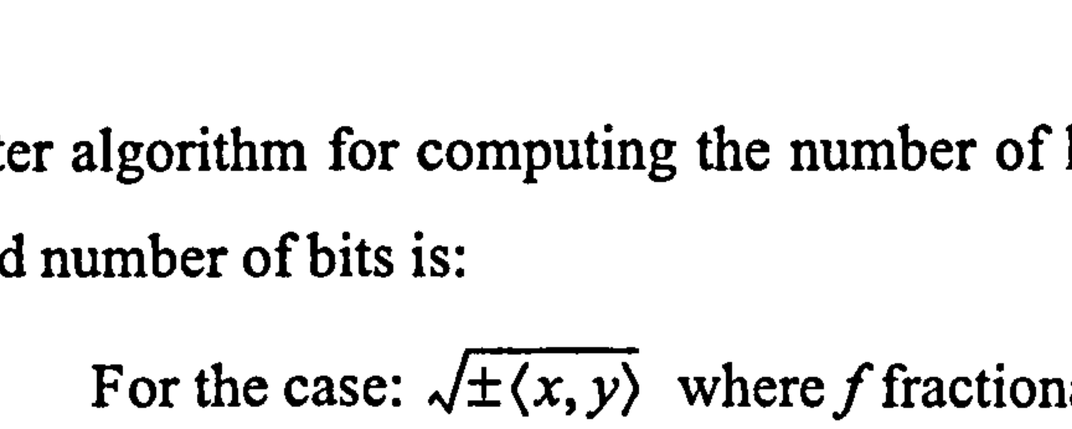
The number of times that the square root algorithm is repeated varies depending on the input width and the number of fractional bits requested by the user from the output. Before discussing the derivation of the equation for the number of loops, it is important to realise that at this stage the input is stored in a 64 bit unsigned integer variable. Also, SystemVue signals are stored in double precision floating point format, which means that the mantissa is 53 bits wide. Hence, the maximum number of bits that can be stored for a fixed point result is 53. The approach that was taken for the square root token was to always compute a 53 bit result and then truncate accordingly after. This was supposed to be a temporary measure, which would be replaced by an algorithm that computed only the desired number of bits (full integer width + desired fractional width) as is done in the divider. However, due to time constraints this was never carried out.

The number of loops required to compute 53 bit results is given by

For the case: $\sqrt{\pm\langle x, y\rangle}$ 53 bit results are computed using:

$$\text{loops} = 32 + \text{mantissa width} - \left(\frac{x+y}{2}\right) \tag{6.8}$$

The derivation of (6.8) is now given below:



For 32 loops of the sqrt algorithm, $(x+y)/2$ bits of the result will be generated.

However, the number of bits that are required from the result are 53 bits.
Therefore, the difference that needs to be made up is:

$$\text{difference} = 53 - \left(\frac{x+y}{2}\right)$$

This means that the total number of loops required are:

$$\text{loops} = 32 + \text{difference} = 32 + 53 - \left(\frac{x+y}{2}\right)$$

A better algorithm for computing the number of loops required to only generate the desired number of bits is:

For the case: $\sqrt{\pm\langle x, y\rangle}$ where f fractional bits are requested:

$$\text{loops} = 32 + f - \frac{y}{2} \tag{6.9}$$

The derivation of (6.9) is now given below:

32 loops of the sqrt algorithm, gives $(x+y)/2$ bits of the result.
However, the number of bits that are required from the result are $(x/2) + f$ bits.
Therefore, the difference that needs to be made up is:

$$\text{difference} = \frac{x}{2} + f - \left(\frac{x+y}{2}\right) = f - \frac{y}{2}$$

This means that the total number of loops required are:

$$\text{loops} = 32 + \text{difference} = 32 + f - \frac{y}{2}$$

6.8.4 Replacing The Binary Point

The result from the square root algorithm is contained in an unsigned 64 bit variable. At this point it is converted into a signed format if the input was signed otherwise it is left unsigned. Before the result is returned it is vitally important that the binary point is inserted in the correct place. However, it has already been shown that for signed and unsigned inputs, the result takes the following respective formats, thus the binary point location can be found accordingly:

$$\begin{aligned} \sqrt{\pm \langle x, y \rangle} &= \pm \langle \frac{x}{2} + 1,\ f \rangle \\ \sqrt{\langle x, y \rangle} &= \langle \frac{x}{2},\ f \rangle \end{aligned} \tag{6.10}$$

6.8.5 Simulating The Delay

As the C++ model must simulate the hardware square rooter, it is important to replicate the delay between the first sample entering the core and the corresponding result reaching the output. In this case the latency is a function of the width of the input and the desired fractional output. The relationship is given in (6.11) (see Figure 6.19).

$$\text{latency} = \frac{x}{2} + f + 3 \tag{6.11}$$

Again, it worth remembering that if the user has enabled Rounding on the output, then an additional fractional bit ($f+1$) will be computed in the divider and consequently the latency will increase by 1.

6.8.6 Performance

To give an indication of how many resources and how fast the hardware Square Rooter can operate at, several instances were implemented using the following:

- Target Technology: *Xilinx Virtex II Pro XC2VP30*

- Synthesis: *Synplify Pro 7.1*
- Map and Place & Route: *Xilinx ISE 8.1*

Table 6 and Table 7 show the results for the pipelined and non pipelined cores respectively. In both cases, square root cores were implemented with 48, 32 and 16 bits in the input and the same number of bits on the output..

Sqrt I/O	Slices	LUTs	Clock Speed
48 bits	5835	8192	>100 MHz
32 bits	2477	3623	>101MHz
16 bits	610	930	>104 MHz

Table 6: Pipelined Square Root Core Results

Sqrt I/O	Slices	LUTs	Clock Speed
48 bits	3719	7171	>1 MHz
32 bits	1606	3073	>3 MHz
16 bits	397	777	>9 MHz

Table 7: Non Pipelined Square Root Results

A significant observation regarding the above results is that they are similar to that of the Divider, both in terms of the number of slices used and the maximum clock speed available. This is a surprising result as it has been shown that a square root can be carried out using approximately half the logic that a divider requires [31][20]. However, the reason that the square rooter is of similar size to the divider in this case is because of the use of generic parameters within the VHDL. Generics are used to parameterise the square root core. For example, they specify the width of the signals within each square root cell. The problem is that they must be set to be a fixed value and cannot vary for each cell. This means that the widths used for each cell must be set to the maximum that will be required by any particular cell (the final cell). Obviously this leads to a very inefficient design but as yet a solution has not been found.

Consequently, the square root core uses nearly the same amount of hardware as the divider, when in actual fact it should use approximately half the logic.

6.9 Verification Of Cores

The verification process for both cores involved two stages. The first was to make sure that the output from the cores was actually correct. Secondly, the software and hardware for each core had to be examined to make sure that the output from both implementations were bit and cycle identical.

To verify that the output from the Divider and Square Rooter were correct, a floating point version of both functions was used. First of all the difference between the floating point reference solution and the fixed point solution is computed for each input. Then, for the case of a Truncated output, the magnitude of the difference is checked to see if it is ever greater than or equal to one LSB. If so, the fixed point core has failed otherwise it has passed. For the case of a Rounded output, the magnitude of the difference is checked to see if it is ever greater than one LSB/2. If so, the fixed point core has failed otherwise it has passed. This process was carried out for many cases where the full range of inputs and outputs were tested. Specific cases such as division by zero and the square root of negative numbers generated different outputs from the floating point design but this was expected.

To verify that the software and hardware were bit and cycle identical required running many cases where the output from one implementation was subtracted from the other. If the result was zero for each sample then the latency and output for both implementation was identical, otherwise a problem would be identified. This process was automated via scripts due to the huge amount of testing required. It should be noted that this was not carried out as part of the EngD project and that other engineers involved in the development of HDS performed this task.

Chapter 7

CORDIC

7.1 Introduction

The CORDIC (COordinate Rotational DIgital Computer) algorithm is an iterative technique based on the rotation of a vector which allows many trigonometric and algebraic functions to be calculated. The key aspect of this method is that it is achieved using only shifts, additions/subtractions and table look-ups which map well into hardware and are ideal for FPGA implementation. The work presented in this chapter focusses on the effort to develop a closed form equation for analysing the error in fixed point CORDIC systems computing vector magnitudes and cosines/sines. By doing this, the most efficient parameters required to produce a desired level of accuracy from such CORDIC systems could be found. One goal of this work was to compare CORDIC implementations to direct implementations computing the same functions to see which were the most efficient. Hence, the division and square root cores developed for HDS were used as part of this assessment.

7.2 COordinate Rotational DIgital Computer (CORDIC)

The original work on CORDIC was done by Jack Volder [33] in the 1950's although this was limited to computing trigonometric functions with the purpose of developing

a digital solution to real-time navigation problems. This work was then extended by John Walther [34] to provide solutions to a broader range of functions. Since then, much research has been carried out on the algorithm, with a thorough survey of this work with respect to FPGAs being published by Andraka [2]. The CORDIC algorithm has been used in applications such as calculators, math-coprocessors, radar signal processors and robotics.

In this chapter the CORDIC algorithm is introduced as well as the problems associated with its use. These problems relate to finding the key parameters required to guarantee a desired level of accuracy from the output of CORDIC systems. However, for the cases of Vector Magnitude calculations and Sine/Cosine calculations, a technique has been developed to solve the aforementioned problems. The technique for both types of calculation is presented in sections 7.4 and 7.5 respectively.

7.2.1 Givens Rotations

The CORDIC method is based on the rotation of a vector from position [x(0),y(0)] to [x(1),y(1)] as shown in Figure 7.1.

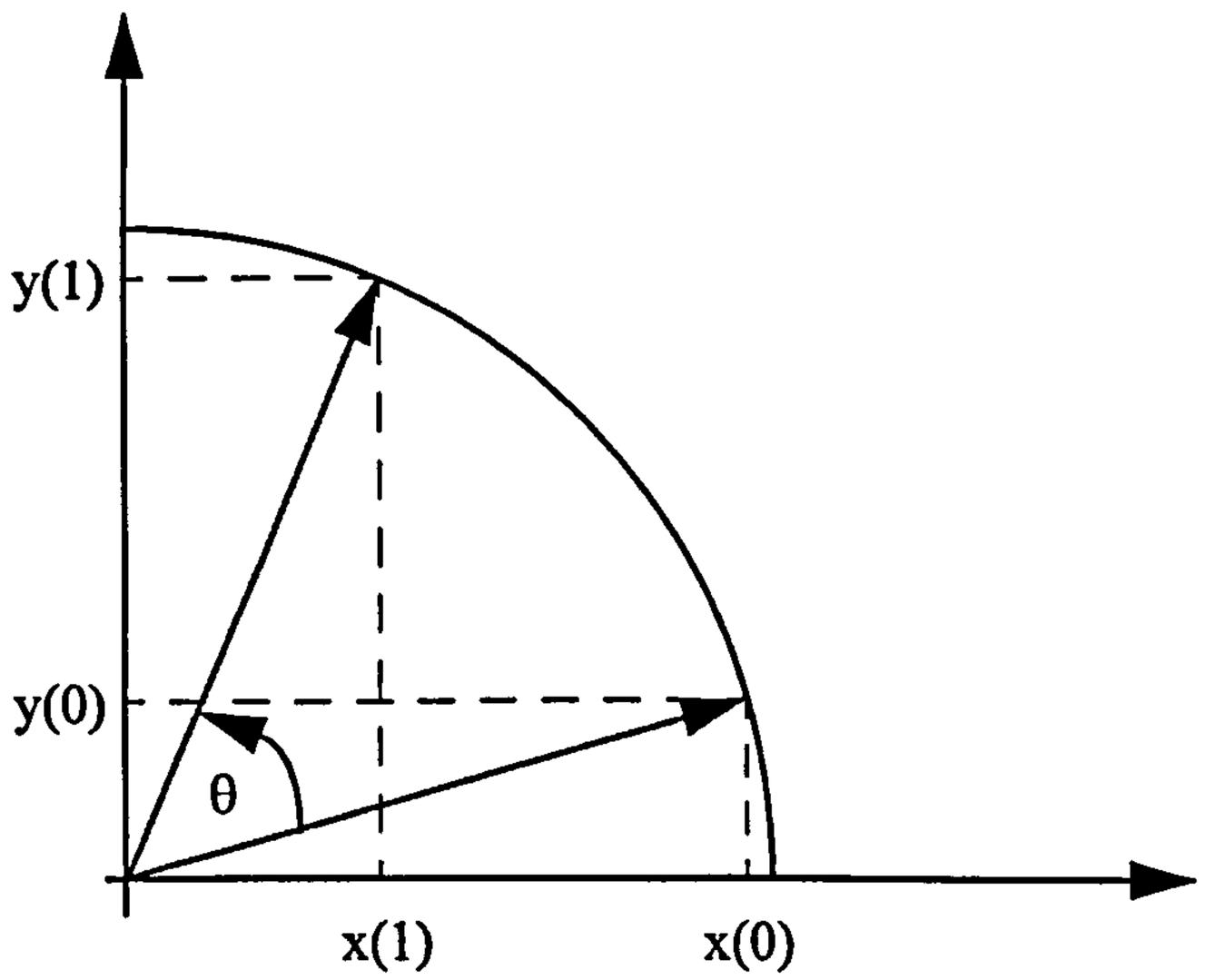


Figure: 7.1: Vector Rotation

The new position can be calculated using the Givens rotation given in (7.1).

$$\begin{aligned} x(1) &= x(0)\cos\theta - y(0)\sin\theta \\ y(1) &= x(0)\sin\theta + y(0)\cos\theta \end{aligned} \tag{7.1}$$

Note that equation (7.1) gives the new position for an anticlockwise rotation of the initial vector. To rotate in a clockwise direction, equation (7.2) should be used.

$$\begin{aligned} x(1) &= x(0)\cos\theta + y(0)\sin\theta \\ y(1) &= y(0)\cos\theta - x(0)\sin\theta \end{aligned} \tag{7.2}$$

7.2.2 Pseudo-Rotations

With some manipulation, (7.1) becomes,

$$\begin{aligned} x(1) &= \cos\theta(x(0) - y(0)\tan\theta) \\ y(1) &= \cos\theta(y(0) + x(0)\tan\theta) \end{aligned} \tag{7.3}$$

which, through dropping the $\cos\theta$ term can be reduced to,

$$\begin{aligned} x(1) &= x(0) - y(0)\tan\theta \\ y(1) &= y(0) + x(0)\tan\theta \end{aligned} \tag{7.4}$$

By dropping the $\cos\theta$ term the rotation that is achieved is no longer a true rotation and is referred to as a pseudo-rotation which can be seen in Figure 7.2.

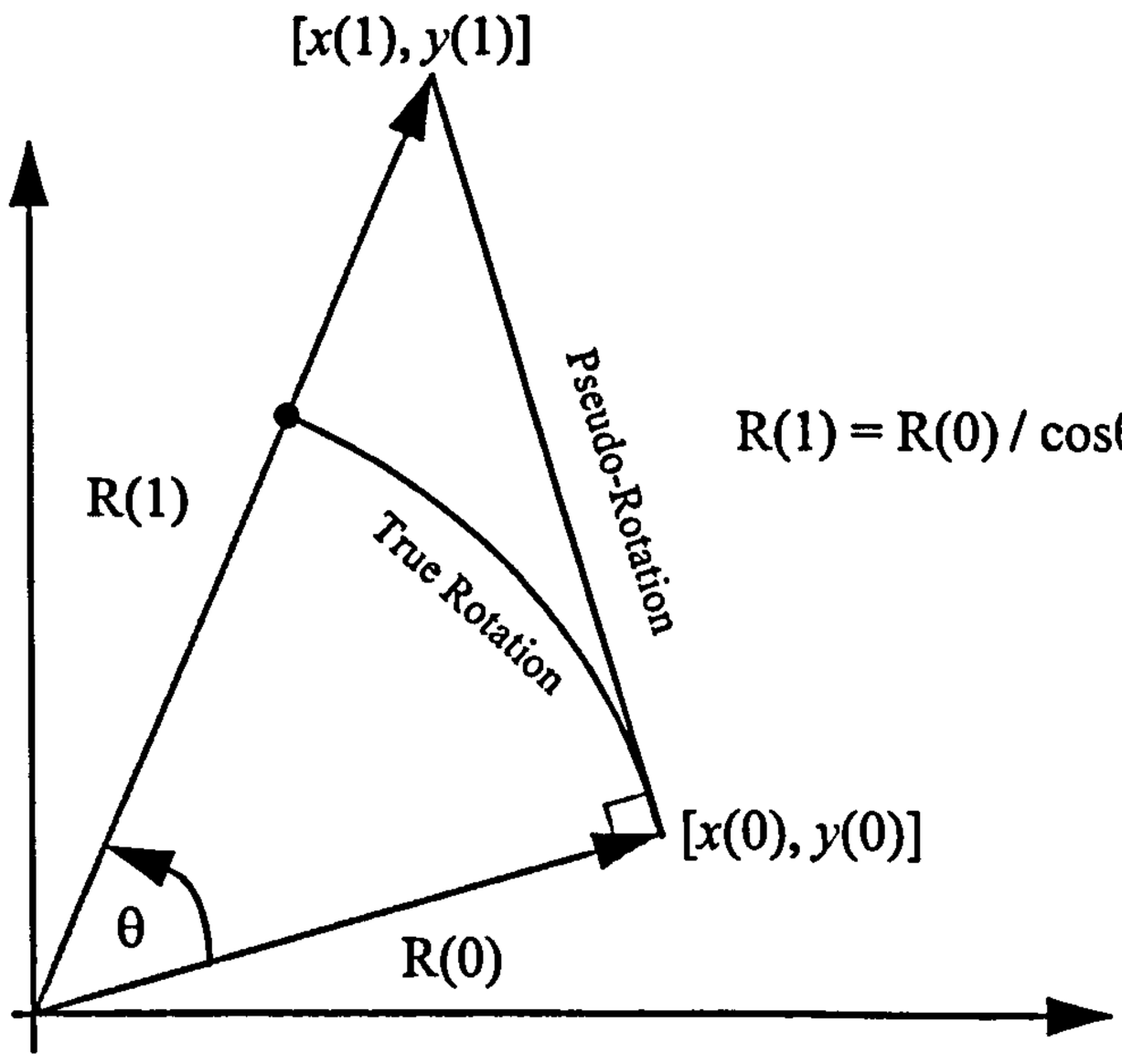


Figure: 7.2: Pseudo Rotation

The effect of the pseudo-rotation is that the length of the vector R(0) is increased to R(1). However, the values of $x(1)$ and $y(1)$ for a true rotation are still desired and clearly a pseudo-rotation does not give this.

7.2.3 Basic Iterations

From equation (7.4), it can be observed that addition, subtraction and multiplication operations are required to achieve a pseudo-rotation. However, the key to the CORDIC algorithm is that the multiplication term can be simplified to a shift operation using the following technique.

The CORDIC algorithm is altered such that rotating by an angle θ is now achieved by performing a series of iterations i, which represent successively smaller rotations θ_i that accumulate to approximate θ. However, the key is that each rotation step is chosen such that $\tan\theta_i = 2^{-i}$ at the i^{th} iteration. Therefore, multiplication by $\tan\theta_i$ reduces to a shift operation, which is much cheaper to implement in hardware. Table 8 illustrates the first few angles that **must** be used to achieve θ. The direction of each rotation obviously affects the accumulative angle. With this iterative scheme arbitrary angles can be approximated within the range $-99.7 \leq \theta \leq 99.7$ as the sum of all angles obeying the law $\tan\theta^i = 2^{-i}$ is 99.7. For angles outside this range, trigonometric identities can be used to convert the desired angle into one within the range.

iteration (i)	$\tan\theta_i = 2^{-i}$	θ_i
0	1	45
1	0.5	26.565
2	0.25	14.036
3	0.125	7.125
4	0.0625	3.576

Table 8: CORDIC Rotation Angles

7.2.4 Angle Accumulator

The simplified Givens rotation given in (7.4) can now be expressed in the following format, which gives the new position of the vector after each iteration.

$$x(i+1) = x(i) - d_i(2^{-i}y(i))$$
$$y(i+1) = y(i) + d_i(2^{-i}x(i)) \quad (7.5)$$
$$\text{where } d_i = +/-1$$

The variable d_i is called the decision operator which is used to decide which direction to rotate. At this stage a 3rd equation is introduced called the Angle Accumulator which is used to keep track of the accumulative angle rotated at each iteration.

$$z(i+1) = z(i) - d_i\theta_i \quad (7.6)$$

The conditions of d_i depend on the **mode** of operation which shall be discussed shortly. These equations now represent the CORDIC algorithm for rotations in a Circular Coordinate System. It will be shown later that there are other coordinate systems that can be used with the CORDIC method to calculate a greater range of functions.

7.2.5 Shift-Add Algorithm

The original Givens rotation has now been reduced to an iterative *shift-add* algorithm where pseudo-rotations are made rather than true rotations. The algorithm is now comprised of the following 3 equations.

$$x(i+1) = x(i) - d_i(2^{-i}y(i))$$
$$y(i+1) = y(i) + d_i(2^{-i}x(i))$$
$$z(i+1) = z(i) - d_i\theta_i$$

Looking at these equations it is clear that this algorithm requires the following operations per iteration,

- 2 shifts, 1 table look-up (θ_i values) and 3 additions/subtractions

Each of these operations is easy to implement on an FPGA and hence the interest in the

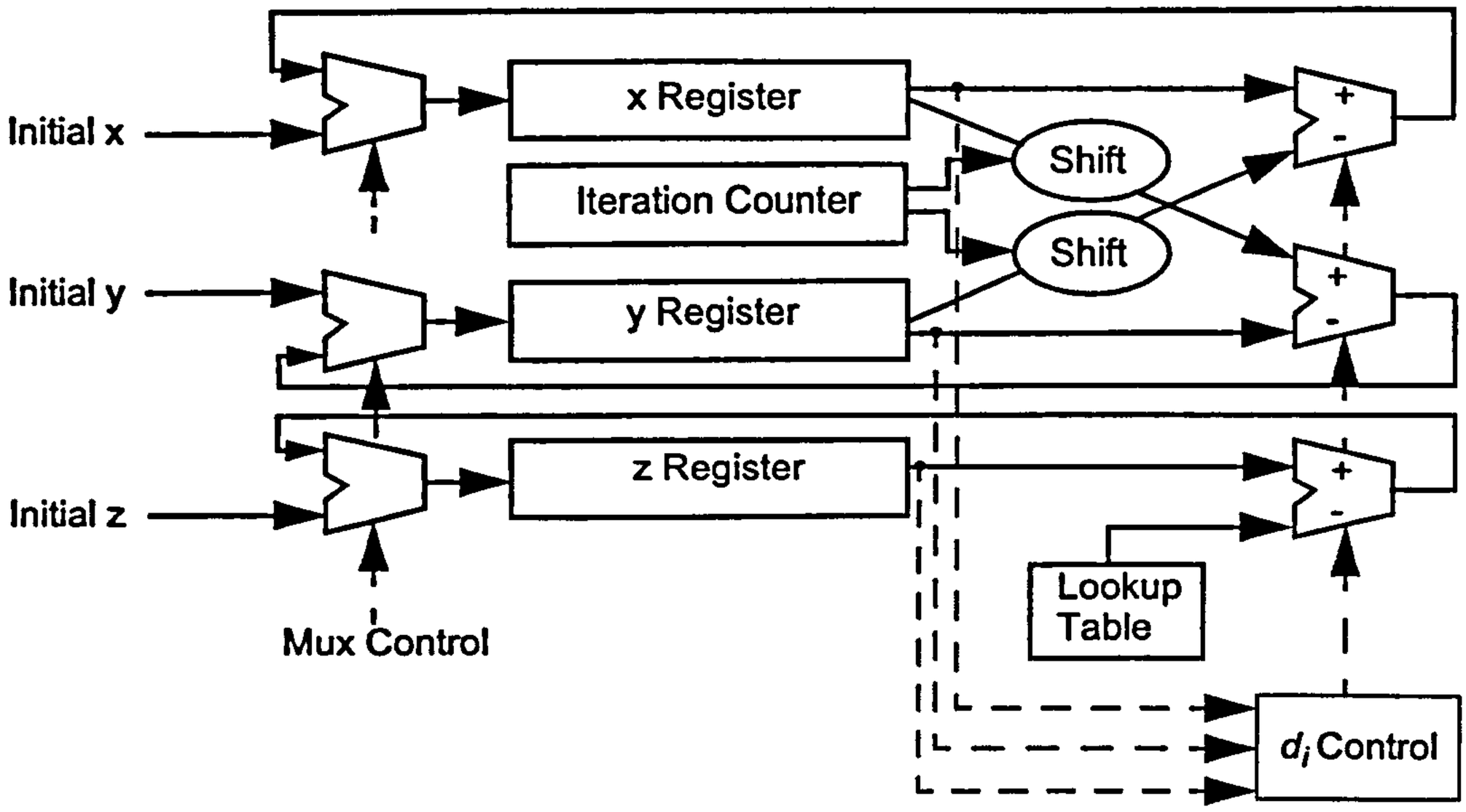


Figure: 7.3: A CORDIC Architecture Using Feedback

CORDIC algorithm. One potential architecture for this algorithm [2] can be seen in Figure 7.3. Note, that this architecture uses feedback although another approach is to unroll this structure so that all data paths are feedforward. This type of design decision depends on how fast and how small the hardware has to be.

7.2.6 The Scaling Factor

Previously, the $\cos\theta$ term was removed from the Givens rotation equations and this resulted in pseudo-rotations instead of true rotations. In Figure 7.2 it is clear that a pseudo-rotation results in the magnitude of the initial vector growing once rotated. Hence, the coordinates of the new vector are not the true values and are in fact scaled. However, the scaling can be measured and removed.

Looking back at Figure 7.2, the new vector is seen to be related to the initial vector by the following equation.

$$R(1) = R(0)/(\cos\theta) \tag{7.7}$$

Applying this to the iterative scheme that has now been derived gives the scaling factor, K_n

$$K_n = \prod_{i=0}^{n-1} 1/(\cos\theta_i) = \prod_{i=0}^{n-1} (\sqrt{1+\tan^2\theta_i}) = \prod_{i=0}^{n-1} (\sqrt{1+2^{(-2i)}}) \qquad (7.8)$$

As the number of iterations are usually known when using CORDIC, K_n and consequently $1/K_n$ can be precomputed. Therefore, the scaling factor can be removed by multiplying with its inverse. Note that:

$$K_n \to 1.6476 \ as \ n \to \infty$$
$$1/K_n \to 0.607 \ as \ n \to \infty$$

7.2.7 Modes Of Operation

The CORDIC algorithm can be used in 2 modes of operation. Each mode determines the condition of the decision operator, d_i, and is selected depending on the particular function to be computed.

Rotation Mode

In *Rotation Mode*, d_i is given by:

$$d_i = sign\ z(i) \qquad (7.9)$$

In this mode the input vector is rotated by a specified angle, which is given as the argument $z(0)$. The aim then is to reduce the angle accumulator to zero although in a real system this is unlikely to occur and a small amount will remain. In an ideal system, after n iterations, the CORDIC equations give,

$$x(n) = K_n(x(0)\cos z(0) - y(0)\sin z(0))$$
$$y(n) = K_n(y(0)\cos z(0) + x(0)\sin z(0)) \qquad (7.10)$$
$$z(n) = 0$$

From equation (7.10) it can be seen that $\cos\theta$ and $\sin\theta$ can be computed by setting $x(0) = 1/K_n, y(0) = 0$ and $z(0) = \theta$. Thus, $x(n) = \cos\theta$ and $y(n) = \sin\theta$.

By considering an example it is clear to see what is happening at each iteration of the algorithm. In this example $\cos\theta$ and $\sin\theta$ shall be computed for $\theta = 30$. Looking at Figure 7.4, the first 3 pseudo-rotations can be seen. To accompany this figure, Table 9 shows the values of each of the parameters as the algorithm converges on the solution.

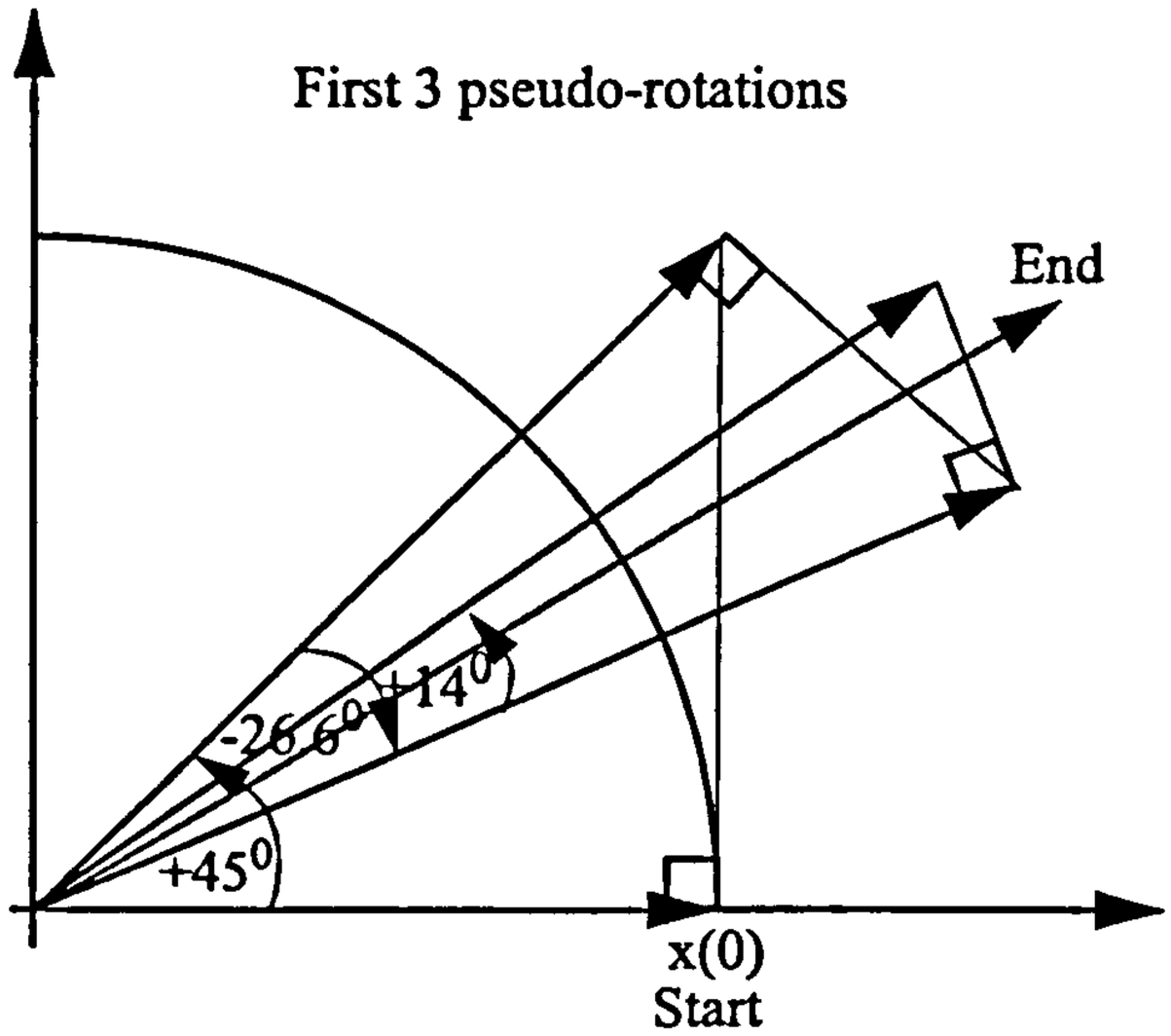


Figure: 7.4: Rotation Mode Example

i	d_i	θ_i	z(i)	y(i)	x(i)
0	+1	45	+30	0	0.607
1	-1	26.6	-15	0.607	0.607
2	+1	14	+11.6	0.303	0.910
3	-1	7.1	-2.4	0.531	0.835
4	+1	3.6	+4.7	0.427	0.901
5	+1	1.8	+1.1	0.483	0.875
6	-1	0.9	-0.7	0.510	0.859
7	+1	0.4	+0.2	0.497	0.867
8	-1	0.2	-0.2	**0.504**	**0.863**

Table 9: Rotation Mode Example

Vectoring Mode

With *Vectoring Mode*, d_i is given by:

$$d_i = -sign\ (x(i)y(i)) \tag{7.11}$$

In this mode the input vector is rotated onto the x-axis and the angle required to achieve this is accumulated in the angle accumulator. Hence, the aim here is to drive $y(n)$ to zero. After n iterations the CORDIC equations ideally give,

$$\begin{aligned} x(n) &= K_n(\sqrt{(x(0))^2 + (y(0))^2}) \\ y(n) &= 0 \\ z(n) &= z(0) + \tan^{-1}\left(\frac{y(0)}{x(0)}\right) \end{aligned} \tag{7.12}$$

Equation (7.12) shows that $\tan^{-1}y(0)$ can be computed by setting $x(0) = 1$ and $z(0) = 0$. Using an example it is clear to see how this is achieved. In this example, $y(0) = 2$. Figure 7.5, shows the first 3 pseudo-rotations and it is clear that the rotated vector is converging on the x-axis as desired. Table 10 accompanies this example to show the values of the variables at each iteration of the algorithm.

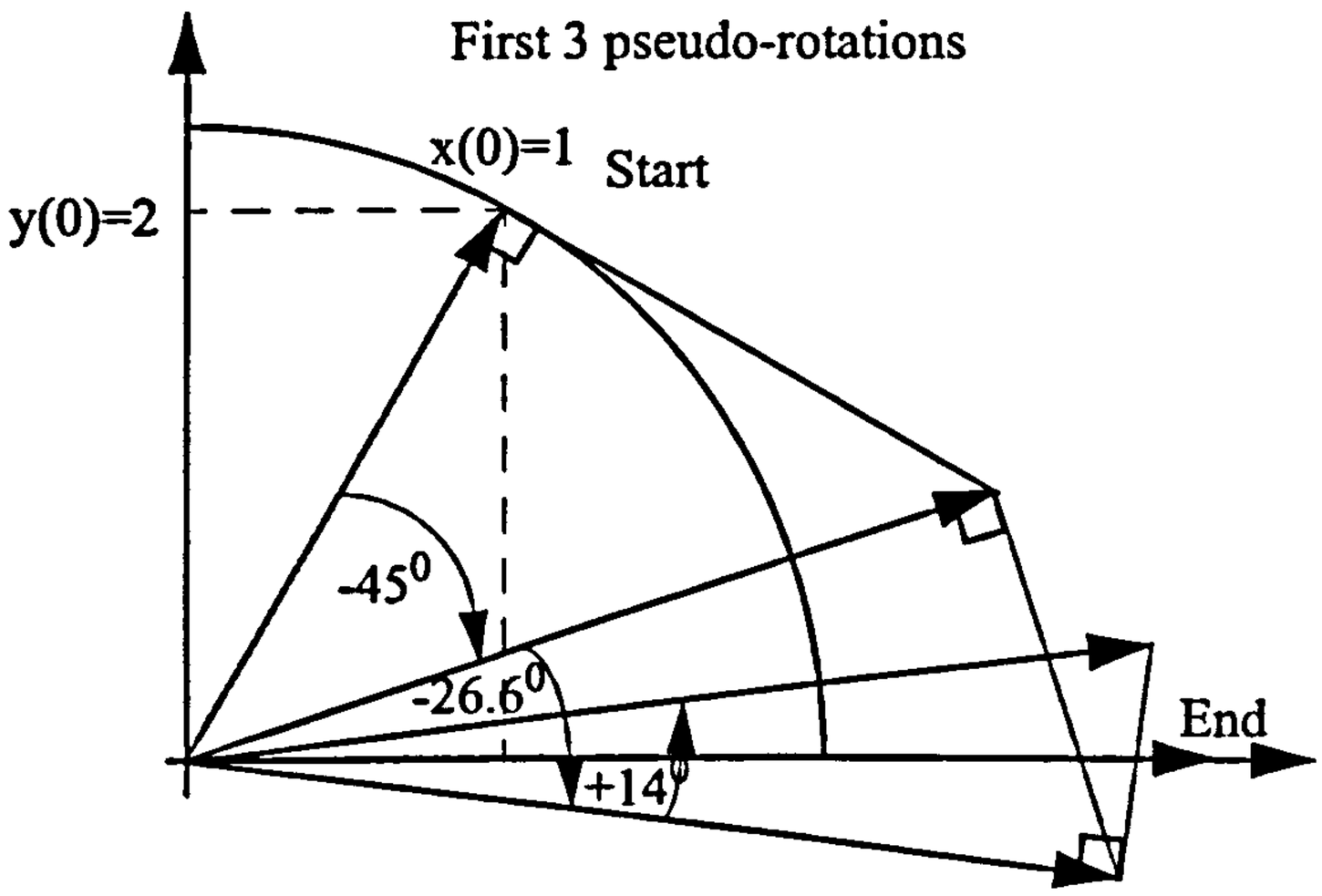


Figure: 7.5: Vectoring Mode Example

i	z(i)	θ_i	y(i)
0	0	45	2
1	45	26.6	1
2	71.6	14	-0.5
3	57.6	7.1	0.375
4	**64.7**	3.6	-0.078

Table 10: Vectoring Mode Example

7.2.8 Coordinate Systems

So far, rotations in a *Circular* coordinate system have been considered. The functions that can be computed using this system are summarised in Figure 7.6.

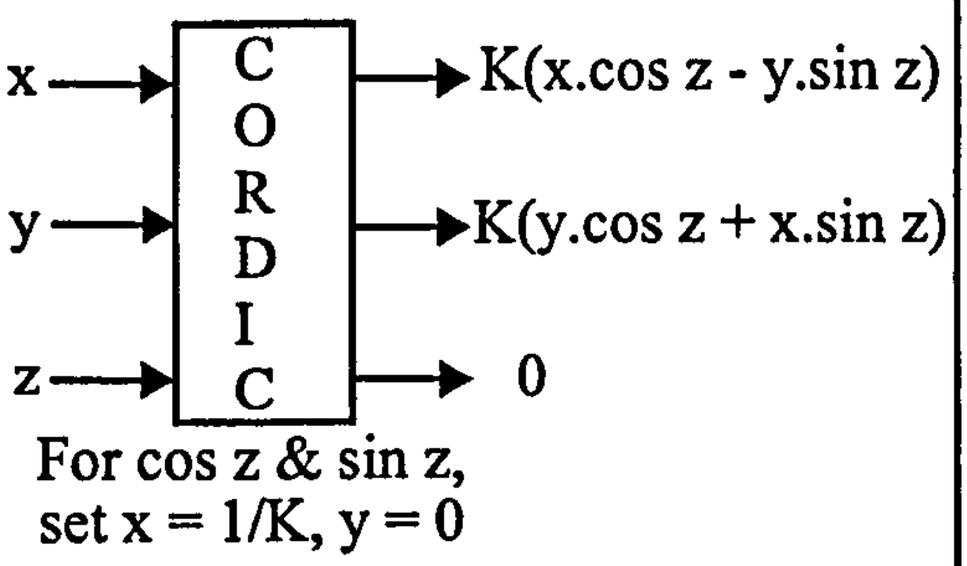
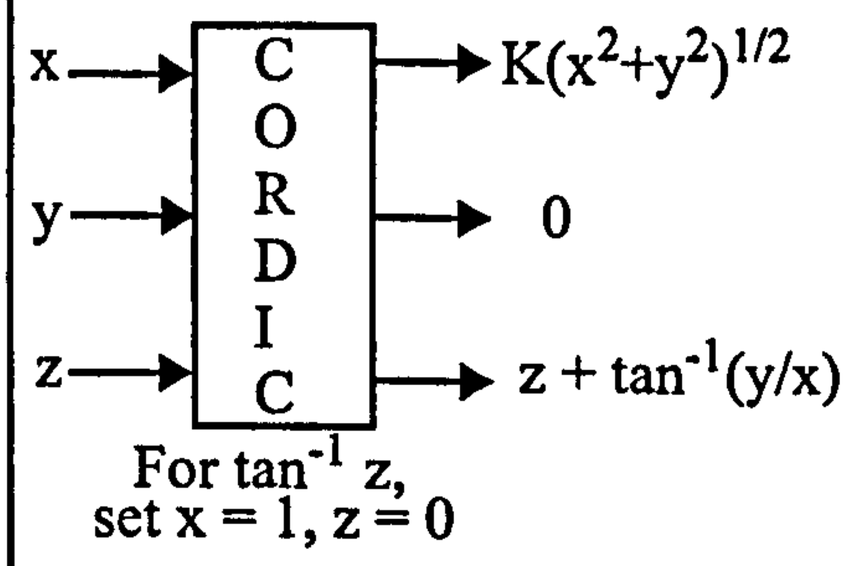
Coordinate System	Rotation Mode $z(i) \longrightarrow 0; d_i = \text{sign}(z(i))$	Vectoring Mode $y(i) \longrightarrow 0; d_i = -\text{sign}(x(i)y(i))$
Circular		

Figure: 7.6: Circular Coordinate System Summary

However, by using other coordinate systems, the set of functions that can be computed using the CORDIC algorithm can be extended. There are two other coordinate systems that can be used with CORDIC. These are *Linear* and *Hyperbolic* which can be seen in Figure 7.7 and Figure 7.8 respectively.

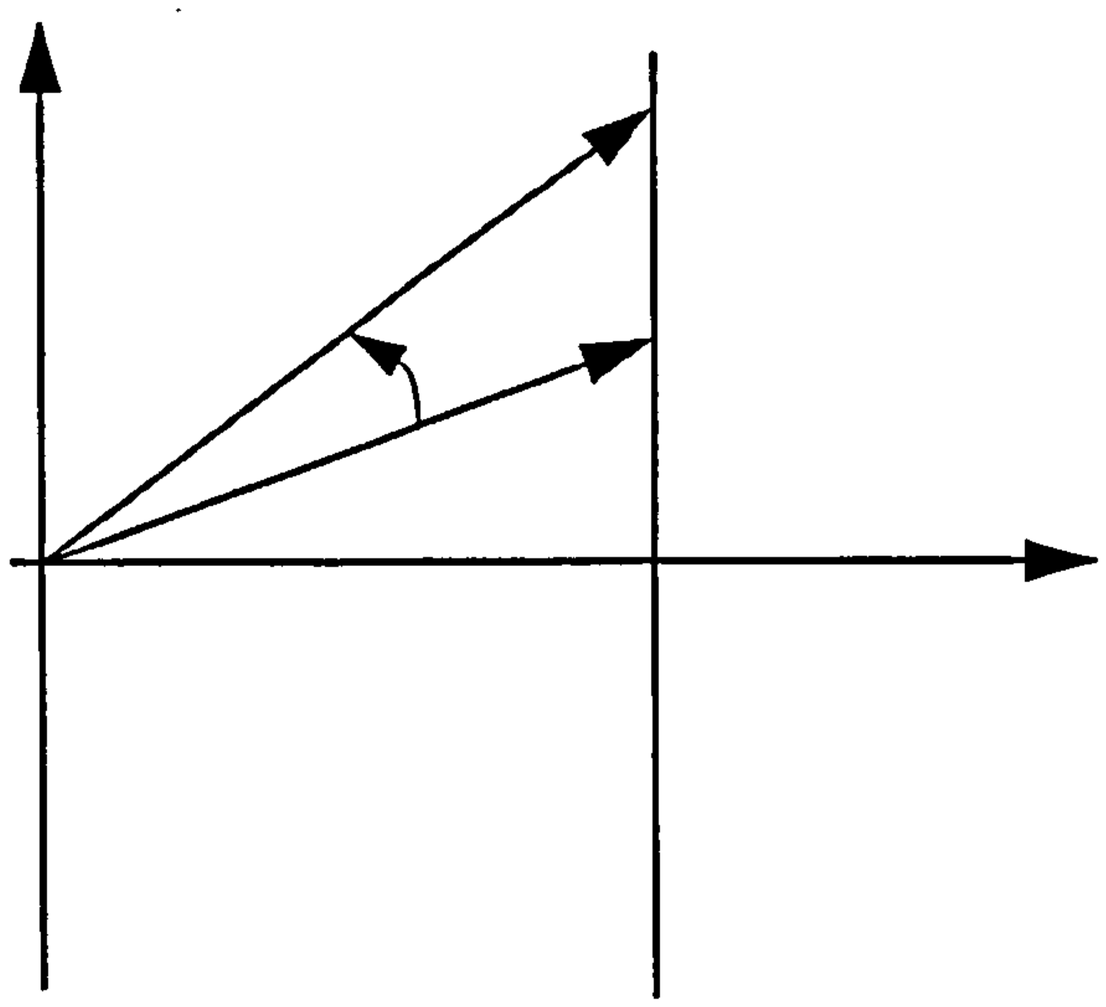


Figure: 7.7: Rotations In A Linear Coordinate System

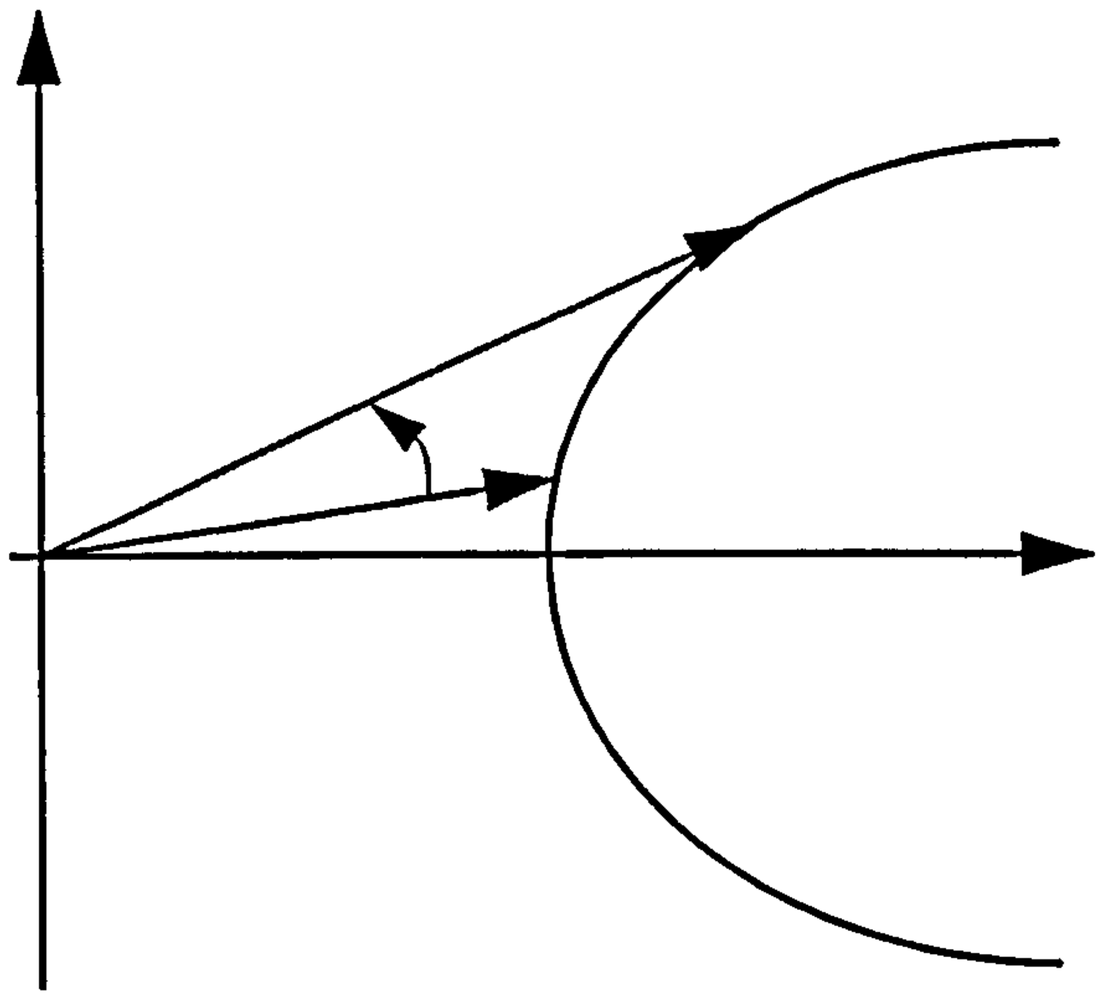


Figure: 7.8: Rotations In A Hyperbolic Coordinate System

The CORDIC equations can be altered to give the Unified CORDIC Equations which can be applied to any of the 3 coordinate systems discussed. These are given by:

$$\begin{aligned} x(i+1) &= x(i) - \mu d_i(2^{-i}y(i)) \\ y(i+1) &= y(i) + d_i(2^{-i}x(i)) \\ z(i+1) &= z(i) - d_i e^{(i)} \end{aligned} \tag{7.13}$$

where the values of μ and $e^{(i)}$ depend on the coordinate system according to,

- Circular System: $\mu = 1, e^{(i)} = \tan^{-1}2^{-i}$
- Linear System: $\mu = 0, e^{(i)} = 2^{-i}$
- Hyperbolic System: $\mu = -1, e^{(i)} = \tanh^{-1}2^{-i}$

7.2.9 Convergence

With 2 modes of operation and 3 coordinate systems, there are several ways in which the CORDIC algorithm can be used. Convergence is guaranteed for Circular and Linear coordinate systems where rotations are constrained to the range [-99.7°, 99.7°]. For rotations outside this range, the angle can be pre-processed to move it into the range and then the output can be post-processed accordingly to give the correct result. However, convergence is not guaranteed using elemental rotations in a Hyperbolic coordinate system. To guarantee convergence certain iterations must be repeated. The sequence of rotations is [1,2,3,**4**,**4**,5,6,7,8,9,10,11,12,**13**,**13**,14,15,...n,..3n+1,..]. Hence, iterations {4,13,40,..n,..3n+1,..} should be repeated [34].

Little work has been carried out to assess the precision of CORDIC across the different functions that it can be used for except for [18][21] which specifically focusses on circular co-ordinate systems. However, a significant part of the work presented in this thesis is in this area and shall be discussed in Section 7.3 with references to [18] and [21].

7.2.10CORDIC Summary

The functions that can be computed directly using CORDIC with either of the 3 coordinate systems that have been discussed in this document are summarised in Figure 7.9. Note that many more functions can be computed indirectly by combining several CORDIC implementations. A list of some of the potential functions that can be computed indirectly are given in Figure 7.10.

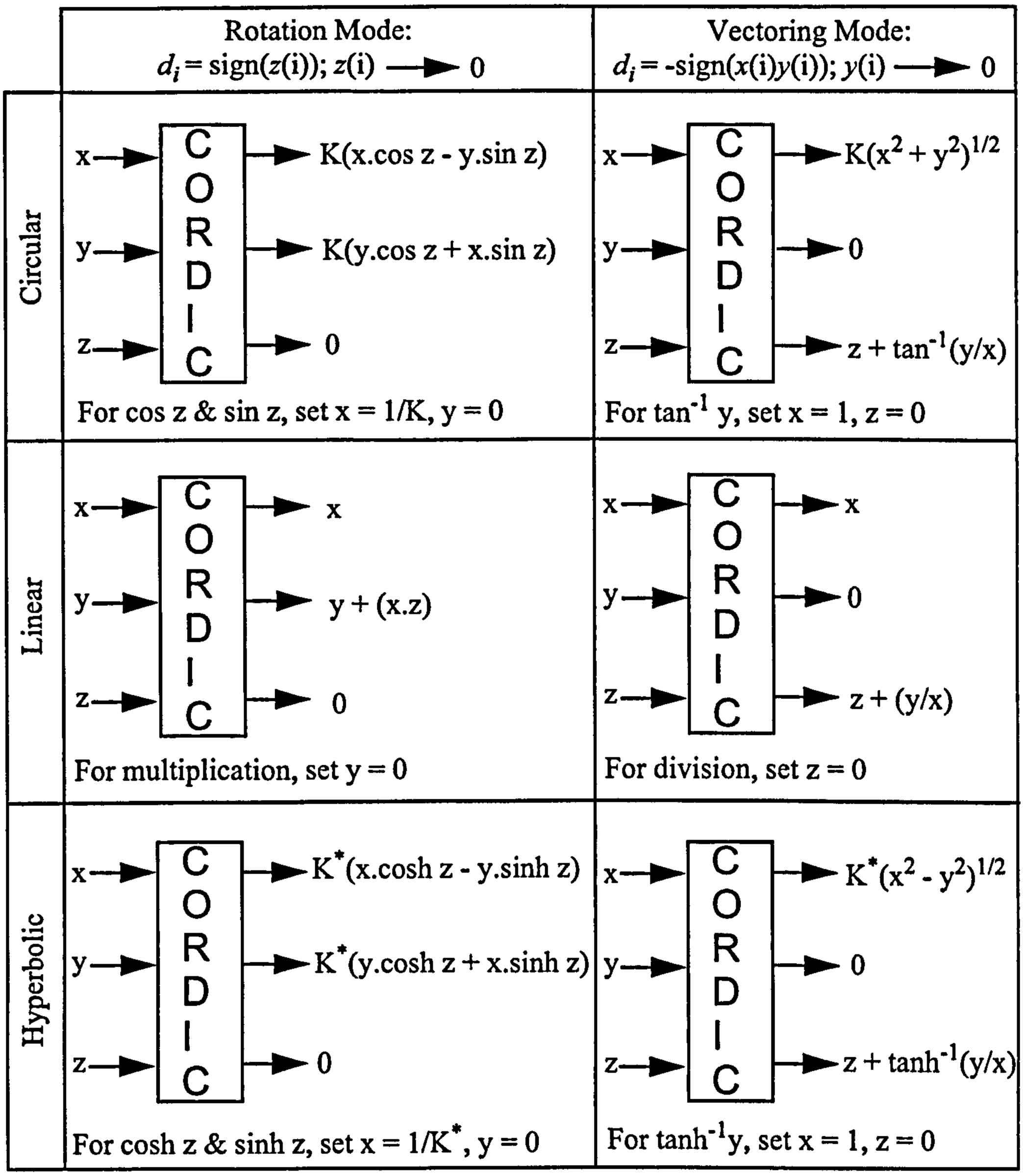


Figure: 7.9: CORDIC Output

K^* is defined as,

$$K^*_n = \sum_{i=0}^{n-1} \sqrt{1 - 2^{(-2i)}}$$

$$K^*_n \to 0.82816 \text{ as } n \to \infty$$

$$\tan z = \frac{\sin z}{\cos z} \qquad \tan^{-1} w = \tan^{-1} \frac{\sqrt{1-w^2}}{w}$$

$$\tanh z = \frac{\sinh z}{\cosh z} \qquad \sin^{-1} w = \tan^{-1} \frac{w}{\sqrt{1-w^2}}$$

$$\ln w = 2\tanh^{-1} \left| \frac{w-1}{w+1} \right| \qquad \cosh^{-1} w = \ln\left(w + \sqrt{1-w^2}\right)$$

$$e^z = \sinh z + \cosh z \qquad \sinh^{-1} w = \ln\left(w + \sqrt{1+w^2}\right)$$

$$w^t = e^{t \ln w} \qquad \sqrt{w} = \sqrt{(w+(1/4))^2 - (w-(1/4))^2}$$

Figure: 7.10: Possible CORDIC Functions

7.3 CORDIC Precision

Traditionally, fixed point CORDIC systems have been designed using a trial and error approach where a system is designed and then the accuracy of the output is assessed via lengthy simulations. If the desired accuracy is not observed then the number of iterations and/or the number of bits in the data path is altered before re-running the simulations. This is a very inefficient way of designing CORDIC systems.

Little work has been carried out to try and quantify the error in fixed point CORDIC systems so that the error can be predicted thus avoiding lengthy simulations. However, Yu Hen Hu [18] developed an algorithm to compute the Overall Quantisation Error in CORDIC systems using vectoring mode with a circular coordinate system. After assessing the work done in this paper it was found that the algorithm that was developed was not very accurate. Further, it was also found that this work could be extended and applied to other CORDIC systems using, for example, rotation mode with a circular coordinate system. Hence, the following sections present the work that was carried out to improve, extend and verify the original work published

in [18].

7.4 Predicting The Accuracy Of Vector Magnitude Calculations

This section highlights the work that was carried out to predict the accuracy of CORDIC systems computing a Vector Magnitude.

7.4.1 Using CORDIC To Compute The Magnitude Of A Vector

By designing a CORDIC system using *vectoring mode* with a *circular coordinate* system it is possible to calculate the magnitude of a vector. As is the case with some CORDIC functions, the output is scaled by a factor K which must be removed to obtain the true result. This can be confirmed by consulting the top right entry in Figure 7.9. A common method for removing the scaling factor is to multiply the result by $1/K$.

The Generalised CORDIC equations are given by:

$$\begin{aligned} x^{(i+1)} &= (x^{(i)} - \mu d_i(2^{-i}y^{(i)})) \\ y^{(i+1)} &= (y^{(i)} + d_i(2^{-i}x^{(i)})) \\ z^{(i+1)} &= z^{(i)} - d_i e^{(i)} \end{aligned} \tag{7.14}$$

where i is the i^{th} iteration, and μ and $e^{(i)}$ are defined depending on the rotation system used:

- Circular Rotations: $\mu = 1$, $e^{(i)} = \arctan(2^{-i})$
- Linear Rotations: $\mu = 0$, $e^{(i)} = 2^{-i}$
- Hyperbolic Rotations: $\mu = -1$, $e^{(i)} = \text{arctanh}(2^{-i})$

Also, d_i is known as the *decision operator* and is used to control the direction in which the vector is rotated at each iteration. Thus, it takes the value of ± 1. Note that when computing the magnitude of a vector $[x, y]$, the expression for $z^{(i+1)}$ is redundant and can be ignored. Further, when using circular rotations, (7.14) can be simplified to the

form shown in (7.15).

$$\begin{aligned} x^{(i+1)} &= (x^{(i)} - d_i(2^{-i}y^{(i)})) \\ y^{(i+1)} &= (y^{(i)} + d_i(2^{-i}x^{(i)})) \end{aligned} \tag{7.15}$$

This is the starting point for designing a CORDIC system to compute the magnitude of a vector. Clearly the operations that are required are addition/subtraction and shifting. Also, the decision operator must be handled. To do this, the MSB of $x^{(i)}$ and $y^{(i)}$ must be determined to allow d_i to be computed with the use of an XOR gate. Finally, an inverter is required to ensure that when one expression adds the other subtracts and vice versa. This can been seen in Figure 7.11 which illustrates a single CORDIC cell (representing one iteration) designed using HDL Design Studio (HDS). Note that a combined adder/subtractor token is not yet available in HDS, thus an adder, subtractor and switch combination was used instead.

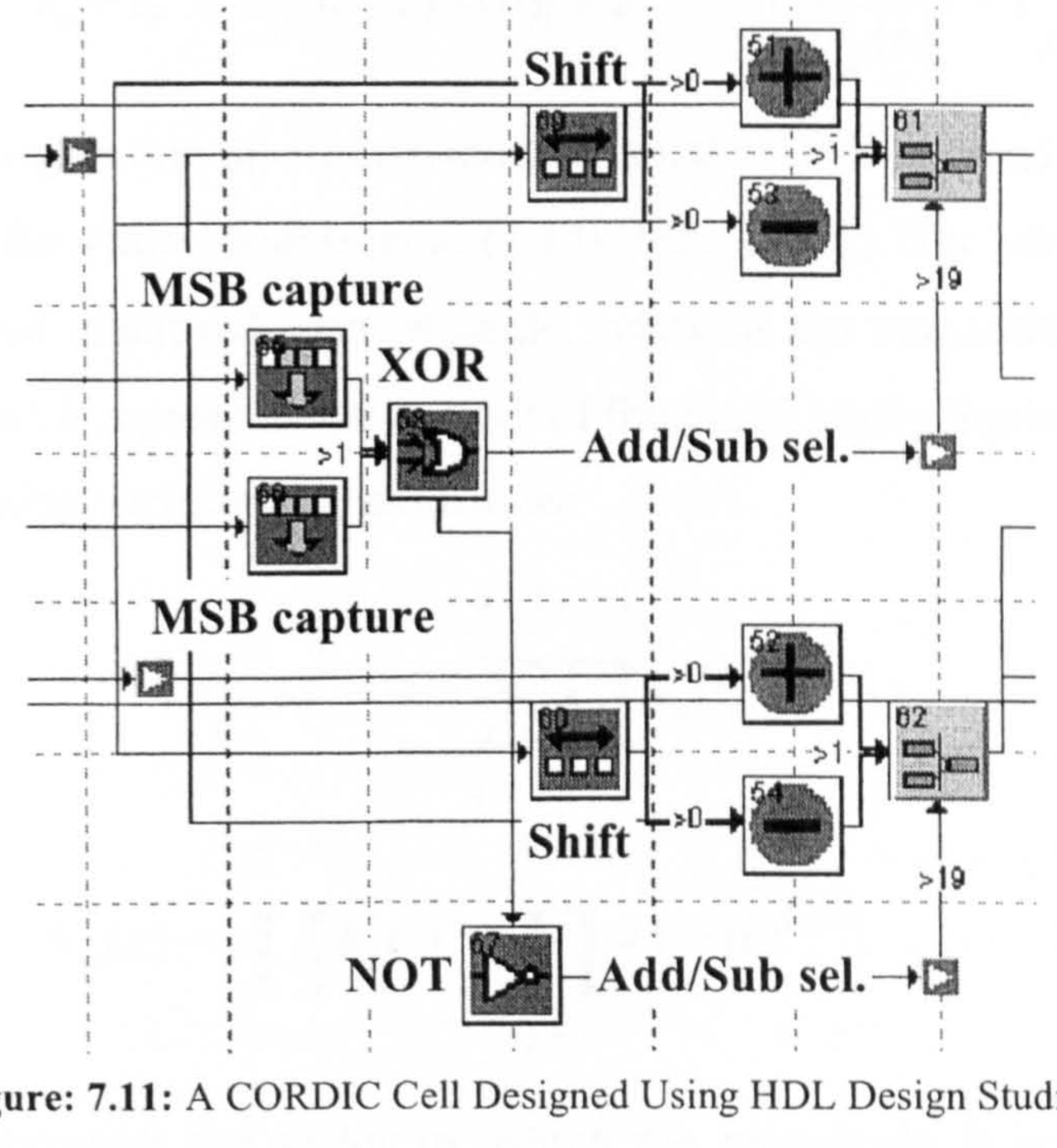


Figure: 7.11: A CORDIC Cell Designed Using HDL Design Studio

7.4.2 Assessing The Overall Quantisation Error

Choosing the number of iterations n and deciding on the number of fractional bits to be used in the data path b so that the magnitude is computed to a desired accuracy is not trivial. However, an equation for computing the Overall Quantisation Error (OQE) using these parameters was developed by Yu Hen Hu. Examining (7.16), it is cleat that the OQE is made up of two distinct errors. The first part is the Approximation Error ε_a, which relates to the error due to the quantised representation of a CORDIC rotation angle by finite numbers of elementary angles. The second part is the Rounding Error ε_r, which is due to the finite precision arithmetic used in a practical implementation.

$$\varepsilon_a = a_\mu(n-1)|v(0)|$$

$$\varepsilon_r = 2^{-b-0.5}\left[\frac{G(\mu, n)}{K_\mu(n)} + 1\right] \tag{7.16}$$

$$\text{OQE} = \varepsilon_a + \varepsilon_r = a_\mu(n-1)|v(0)| + 2^{-b-0.5}\left[\frac{G(\mu, n)}{K_\mu(n)} + 1\right]$$

To investigate this equation in more detail, the variables need to be explained. First of all, μ is the same variable as in (7.14) (here $\mu = 1$). The variable $a_\mu(n-1)$ represents the final quantised rotation angle, $|v(0)|$ is the magnitude of the largest vector possible and b represents the number of fractional binary digits used in the data paths. The remaining variables are defined as:

$$G(\mu, n) = 1 + \sum_{j=1}^{n-1}\prod_{i=j}^{n-1} k_\mu(i)$$

$$K_\mu(n) = \prod_{i=0}^{n-1} k_\mu(i) = \prod_{i=0}^{n-1}\sqrt{1+\mu 2^{(-2i)}} \tag{7.17}$$

where $K_\mu(n)$ represents the factor by which the magnitude is scaled by after n iterations, and $G(\mu, n)$ is used to compute the worst case rounding error.

By computing the OQE it is possible to determine the number of *effective fractional bits* d_{eff} that a CORDIC system generates using:

$$d_{eff} = -(\log_2 \mathrm{OQE}) - 1 \tag{7.18}$$

In [18], Yu Hen Hu computed d_{eff} using (7.18) for a set of n and b. To verify the results, simulations were also carried out for the same set of n and b. The two sets of results were very close for certain n and b, which suggested that these equations could be used to estimate these parameters for a desired accuracy. From here a search could begin to find the optimum combination of these parameters which would lead to the most efficient design for a specific accuracy.

To illustrate the tables that Yu Hen Hu developed, a small section has been given in Table 11 below. It is clear from this section that to obtain 6 effective (fractional) bits of accuracy requires either 9 iterations and 10 fractional binary bits in the data paths or 8 iterations with 11 fractional bits.

n / b	8	9	10	11	12
3	1.43	1.47	1.48	1.49	1.5
4	2.35	2.42	2.46	2.48	2.49
5	3.17	3.32	3.41	3.45	3.48
6	3.82	4.12	4.3	4.4	4.45
7	4.27	4.76	5.08	5.28	5.38
8	4.5	5.18	5.69	**6.04**	6.25
9	4.57	5.4	**6.09**	6.63	7.0
10	4.55	5.46	6.3	7.01	7.57

Table 11: Section of Yu Hen Hu Tables

However, an issue with these tables was discovered when, after confirming via a SystemVue simulation that 9 iterations with 10 fractional bits actually did result in 6 effective fractional bits, further simulations showed that 6 effective fractional bits could be obtained with as few as 4 iterations and 10 fractional bits. According to the table, only 2 effective bits should be obtainable. The test results showed that the equations developed by Yu Hen Hu were consistently underestimating the number of effective fractional bits. Ideally the equations should allow the user to select the most efficient architecture for a desired accuracy and clearly they did not. Hence, a further

examination of the OQE was required, and this work is presented in the next section.

7.4.3 Taking The OQE Further

To verify the work presented in [18] and to allow further analysis of this work, MATLAB code was written to compute the OQE and consequently d_{eff} for a set of n and b. First of all functions for computing $K_1(n)$ and $G(1,n)$ were written. These functions can be seen below in Figure 7.12 and Figure 7.13 respectively.

```
function K = Kgen(n)
%Computes CORDIC scale factor K(n-1)
%Kgen returns the CORDIC scale factor for a given number of iterations n
K = 1;
for i = 0:n-1
    k(i+1) = (1 + 2^(-2*(i)))^0.5;
    K = K*k(i+1);
end
```

Figure: 7.12: MATLAB Function for Computing $K_1(n)$

```
function G = Ggen(n)
%Computes G(n)
%Ggen returns G(n) which is used to compute the worst case rounding error in
G = 0;
for j = 1:n-1
    K = 1;
    for i = j:n-1
        k(i) = (1 + 2^(-2*(i)))^0.5;
        K = K*k(i);
    end
    G = G + K;
end
G = G+1;
```

Figure: 7.13: MATLAB Function for Computing G(1,n)

A function to compute d_{eff} was also written. This function calls the two previous functions and uses them to compute the Approximation Error and Rounding Error. The total error (OQE) is then computed before the number of effective bits is determined and returned. This function takes in the number of iterations n, the fractional bit width b and the maximum magnitude of the vector m, as can be seen in

Figure 7.14.

```
function Deff = EffBits(n,b,m)
%EffBits returns the number of effective fractional bits computed in
%a fixed point CORDIC system relative to a floating point CORDIC system
%EffBits inputs: n = iterations, b = bits used, m = max. magnitude of
%input vector; Returns the number of effective bits computed.
G = Ggen(n);
K = Kgen(n);
approx_error = 2^(-n+1)*m;
roumding_error = 2^(-b-0.5)*((G/K)+1);
OQE = approx_error + rounding_error;
Deff = -(log2(OQE))-1;
```

Figure: 7.14: MATLAB Function for Computing d_{eff}

To allow d_{eff} to be computed for a set of n and b, a final function was written as in Figure 7.15 below. This function computes all values of d_{eff} for $1 \le n \le 40$, $1 \le b \le 40$ and $m = \sqrt{0.5}$

```
function bits = error_test()
for b = 1:40
    for n = 1:40
        bits(n,b) = EffBits(n,b,0.5^0.5);
    end
end
```

Figure: 7.15: MATLAB Code for Computing d_{eff} for a set of n & b

The results that were produced for d_{eff} have been verified against the small set of n and b presented in [18] and can be found in Appendix A. As mentioned already, simulations showed that the equations used here consistently underestimate the number of effective fractional bits that are achievable for a given n and b. Hence, it was important to find out the reason for this behaviour and to try and improve on this work.

7.4.4 The Approximation Error

By considering the OQE in more detail it was found that the Approximation Error is more dominant than the Rounding Error, especially for small n. Hence, an initial assumption was that this error was being overestimated thus causing the number of

effective bits to be underestimated.

When using CORDIC to compute the magnitude of a vector, the aim is to drive the initial vector onto the x-axis. This mode of operation is known as Vectoring or sometimes Backward rotation mode. However, due to finite rotations, a small angle δ is usually left between the rotated vector and the x-axis thus resulting in the Approximation Error ε_a. This angle is known as the Angle Approximation Error. Figure 7.16 shows the effect that δ has on ε_a for an exaggerated angle δ to illustrate the problem. It is clear that as δ is reduced the rotated vector gets closer to the x-axis and the smaller the ε_a will be. Clearly an accurate estimation of δ is required to obtain an accurate estimation of the Approximation Error.

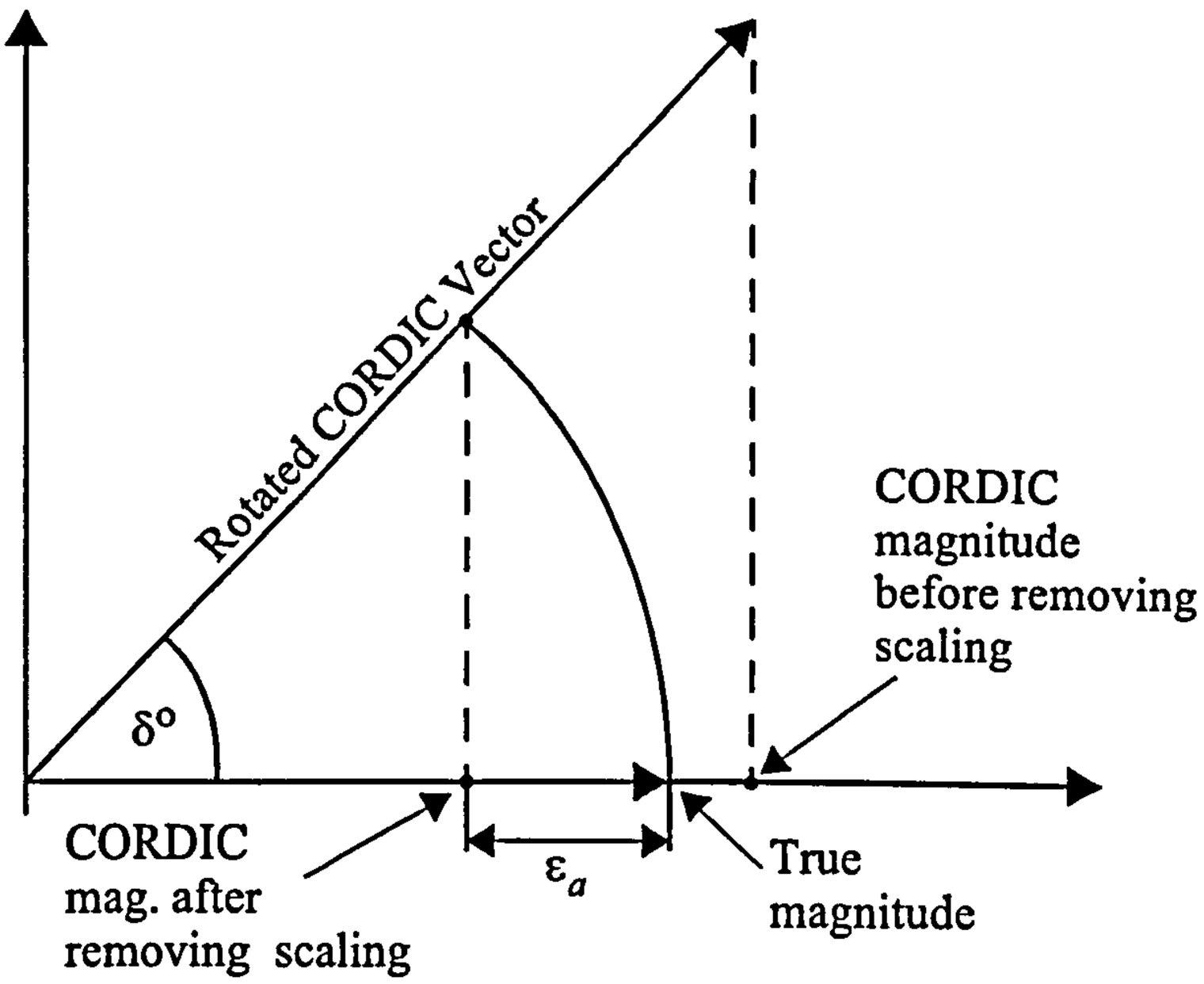


Figure: 7.16: The Angle Approximation Error δ

In Yu Hen Hu's paper, the upper bound on δ is given as $\delta \leq a_1(n-1)$, where $a_1(n-1)$ is the final rotation angle. Obviously the more iterations there are, the smaller ε_a will be. For n sufficiently large, $a_1(n-1)$ can be approximated by:

$$a_1(n-1) \approx 2^{-n+1} \quad \text{(in radians)} \tag{7.19}$$

Yu Hen Hu used this simplification to compute the Approximation Error as:

$$\varepsilon_a = 2^{-n+1}|v(0)| \tag{7.20}$$

where $|v(0)|$ is the magnitude of the largest vector that can be represented using the chosen fixed point format. It is this simplification that results in the underestimation of the number of effective fractional bits. Both n and $|v(0)|$ are required to compute the Approximation Error but as will be shown shortly, (7.20) is simply not accurate.

7.4.5 Improving The Approximation Error Estimate

A more accurate equation for calculating the Approximation Error was developed by considering the diagram shown in Figure 7.16. A simplified version of this diagram is given below in Figure 7.17.

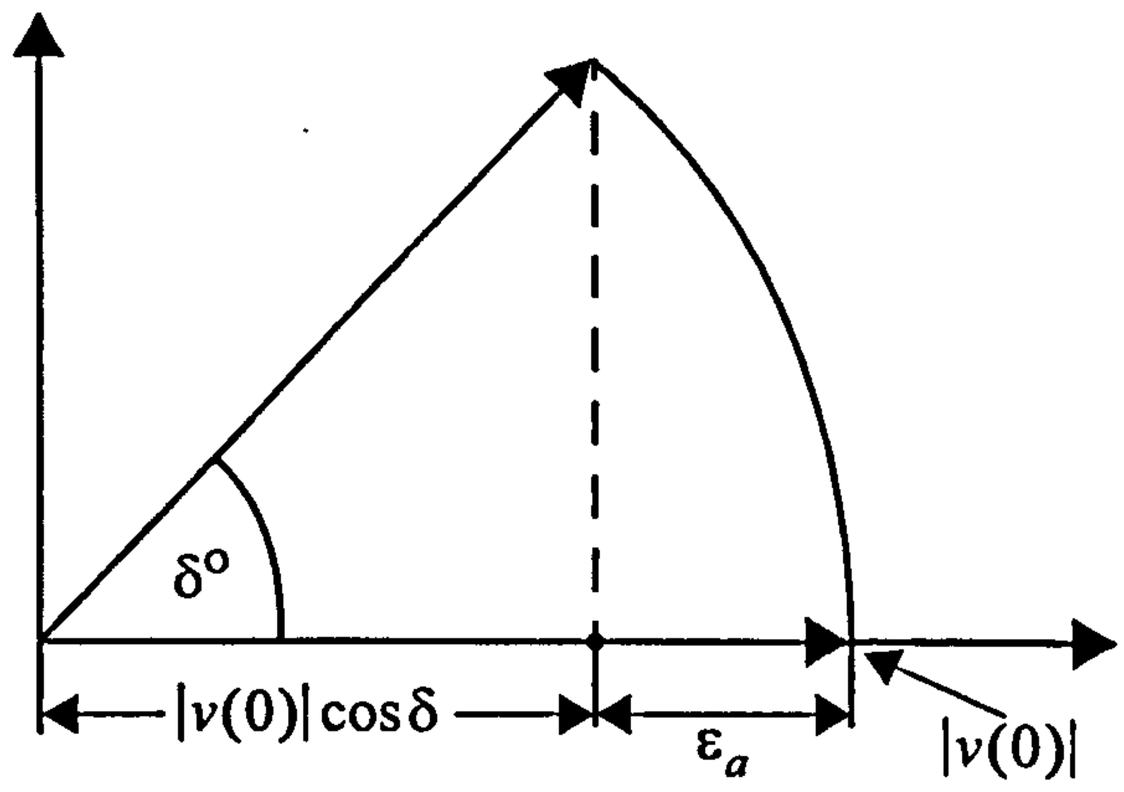


Figure: 7.17: CORDIC Approximation Error

It can be observed in Figure 7.17 that the CORDIC magnitude, after removing the scaling factor, is equal to:

$$\text{CORDIC Magnitude} = x(n-1) = |v(0)|\cos\delta \tag{7.21}$$

Then, clearly the error between the true magnitude and the CORDIC magnitude is the Approximation Error which is computed via:

$$\varepsilon_a = |v(0)| - |v(0)|\cos\delta \tag{7.22}$$

Finally, if the upper bound for δ is applied, the equation becomes:

$$\varepsilon_a = |v(0)| - |v(0)|\cos(a_1(n-1)) \tag{7.23}$$

where

$$a_1(n-1) = \arctan(2^{-n+1}) \tag{7.24}$$

Note that (7.24) is preferred to (7.19) when computing $a_1(n-1)$ as it is more accurate for small n.

To see the difference between this new algorithm for computing the Approximation Error and the one given in Yu Hen Hu's paper, both were plotted for a set of n and can be seen in Figure 7.18. The graph clearly shows that Yu Hen Hu's algorithm generates a larger error which consequently causes the number of effective fractional bits to be underestimated. Another way to view the Approximation Error is as the number of effective fractional bits that it represents. This can be seen in Figure 7.19 which clearly shows that as n increases the approximation error predicted by the new algorithm is much smaller than that given by the original algorithm. However, if we look at Appendices A and B it can be seen that the OQE in both tables converge to the same results as n increases. So for example, looking at the $b = 5$ column in

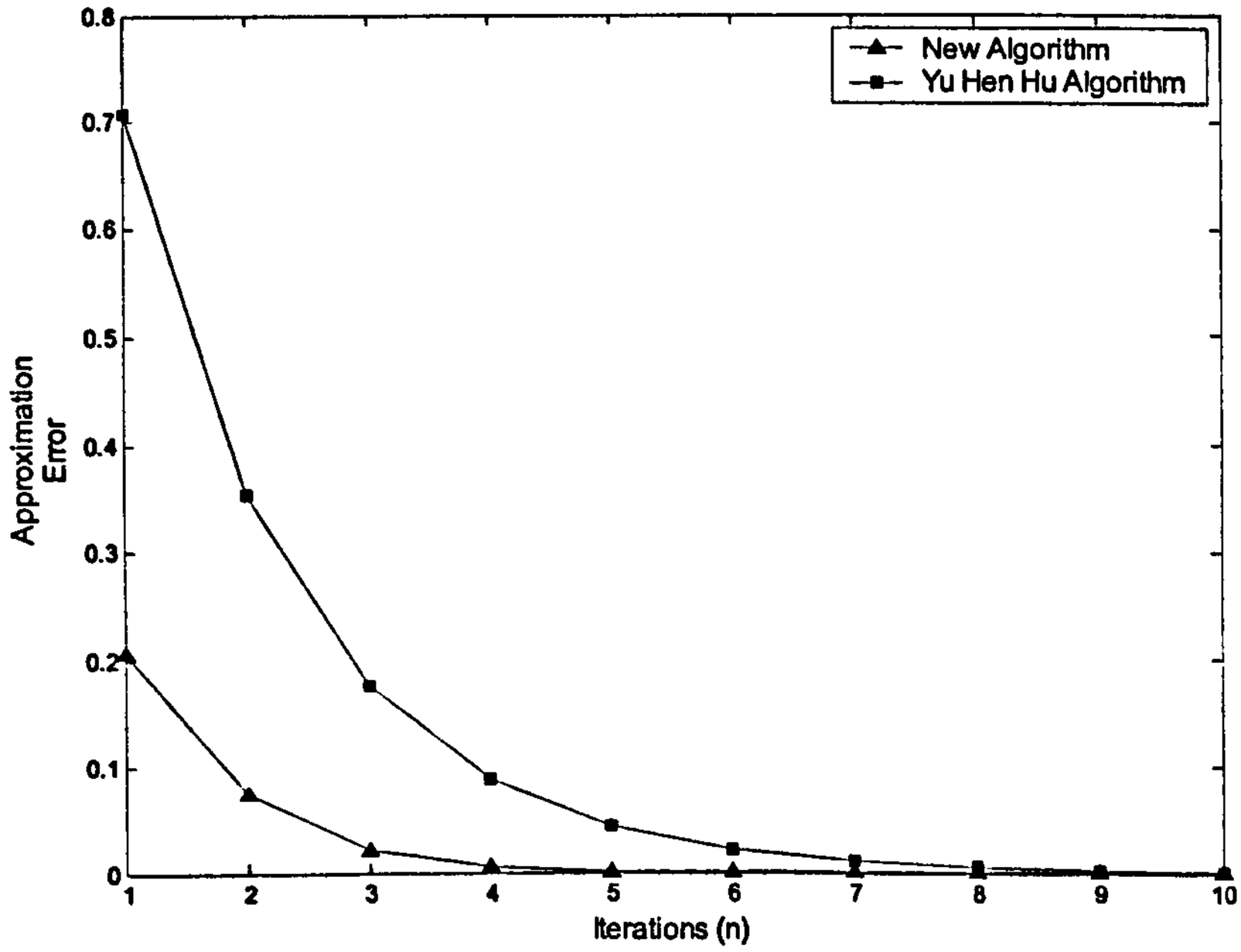


Figure: 7.18: Approximation Error vs Iterations (n)

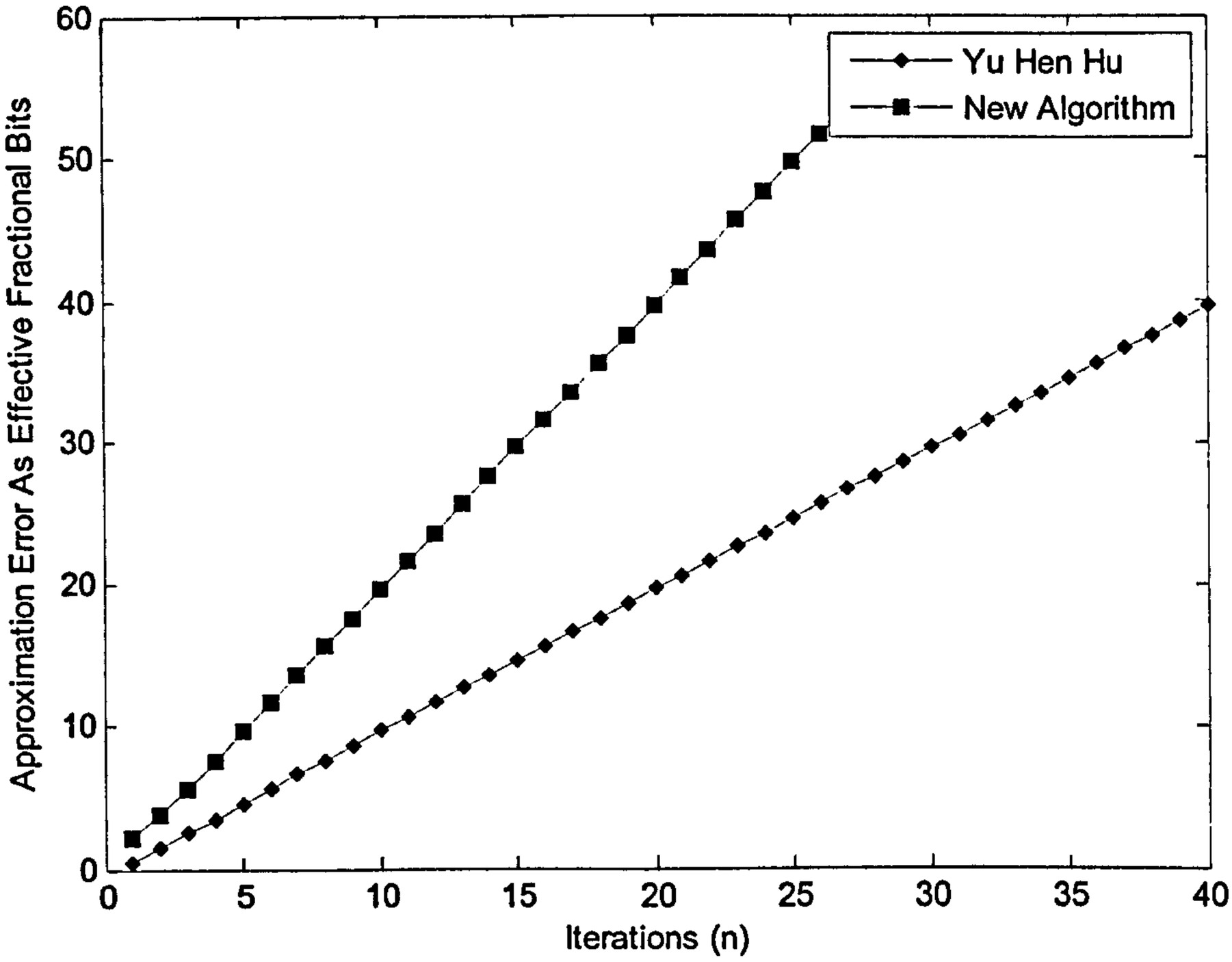


Figure: 7.19: Approximation Error As Effective Fractional Bits vs Iterations (n)

Appendices A and B, the results in both columns are the same from $n = 11$ onwards if we add 1 bit onto the results in Appendix A due to the fact that (7.18) was used instead of (7.25). The reason for this is discussed in the next section. This trend in both tables shows that as n increases the Approximation Error becomes less relevant and the Rounding Error starts to dominate. Hence, the reason for both tables converging. Clearly the new algorithm had to be verified and this work is covered in the next section.

7.4.6 Verifying The New Algorithm

To verify (7.23), the Matlab code shown earlier in Figure 7.14 was altered to use the new algorithm as can be seen in Figure 7.20. Note also that (7.25) was used to compute d_{eff} rather than (7.18).

$$d_{eff} = -(\log_2 \mathrm{OQE}) \tag{7.25}$$

```
function Deff = EffBits2(n,b,m)
%EffBits returns the number of effective fractional bits computed in
%a fixed point CORDIC system relative to a floating point CORDIC system
%EffBits inputs: n = iterations, b = bits used, m = max. magnitude of
%input vector; Returns the number of effective bits computed.
G = Ggen(n);
K = Kgen(n);
Z = atan(2^(-n+1));
approx_error = m - (m*(cos(Z)));
roumding_error = 2^(-b-0.5)*((G/K)+1);
OQE = approx_error + rounding_error;
Deff = -(log2(OQE));
```

Figure: 7.20: New MATLAB Function for Computing d_{eff}

This is because, in [18] subtracting a bit from d_{eff} is justified to counter growth of the wordlength by 1 integer bit due to the scaling factor $K(n)$. However, as it is fractional bits that this analysis is interested in, there is no need to subtract a bit from d_{eff} to counter the integer bit growth. It will be shown later that when comparisons were made between simulations and predicted values, that this correction was justified.

Using the code in Figure 7.20 with the other Matlab functions shown earlier, a new table was generated for d_{eff}, for $1 \le n \le 40$, $1 \le b \le 40$ and $m = \sqrt{0.5}$. This table can be seen in Appendix B. A quick comparison of the table in Appendix A with the new table shows that for a given b, the values for d_{eff} become very close as n increases. It should be kept in mind when making this comparison that the values in Appendix A were computed using (7.18) and effectively represent $d_{eff} - 1$ when compared to Appendix B. However, the major difference between the two tables can be seen as n decreases. This is because, as was shown in Figure 7.18, the equation developed by Yu Hen Hu overestimates the Approximation Error, particularly for small n, causing the number of effective fractional bits to be underestimated.

7.4.7 SystemVue (HDS) Simulations

To verify the accuracy of the new algorithm, a number of simulations were carried out. Figure 7.21 shows the top-level of the SystemVue model that was used for the

simulations. A floating point Golden Reference Design was used with a fixed point CORDIC design. Both designs were driven by the same fixed point input data generated by two Uniform Noise tokens set to produce data in the range ± 0.5. With this range the limit for $|v(0)|$ is $\sqrt{0.5}$, which matches with the parameter $m = \sqrt{0.5}$ which was used when producing the tables. The fixed point format for the input data was set to $\pm<1, b>$ where b is the number of fractional bits used in the data paths.

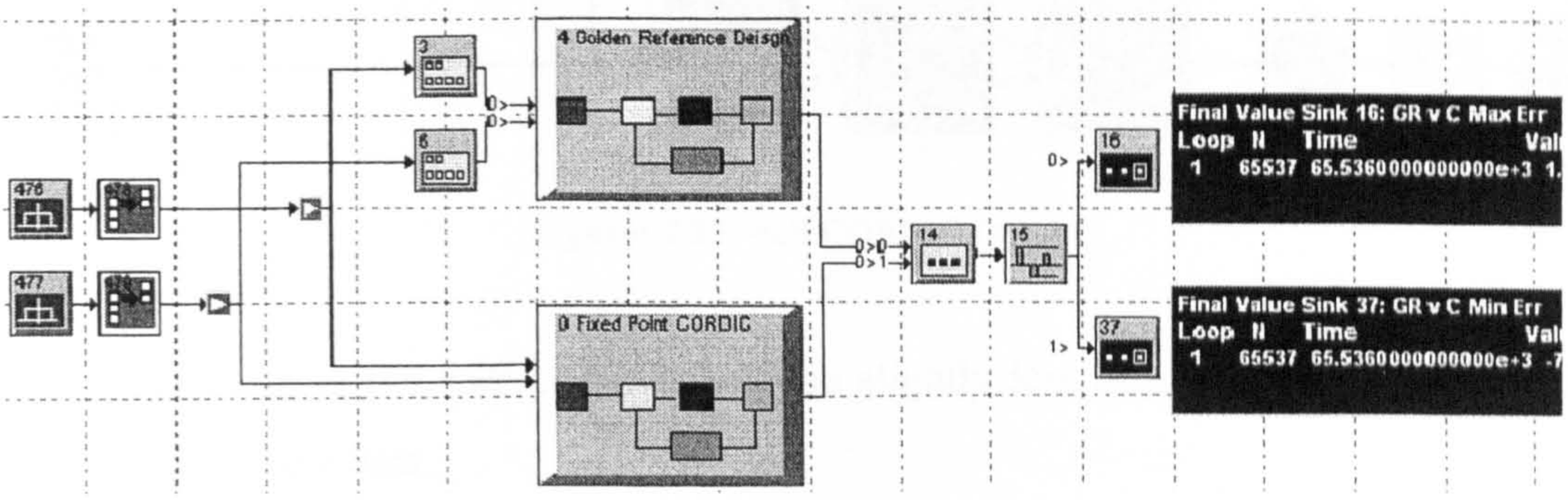


Figure: 7.21: Top-Level Of Simulation

To compute d_{eff} from the simulations, the output from the fixed point CORDIC design is subtracted from the output of the Golden Reference design. The maximum positive error and the maximum negative error are then captured before (7.26) is applied to the error with the greatest magnitude.

$$d_{eff} = -(\log_2 \text{error}) \tag{7.26}$$

Before discussing the results from the simulations, the full fixed point CORDIC design should be discussed in more detail. Earlier, the design of an individual CORDIC cell was discussed. Figure 7.22 shows a full design using 3 cells which corresponds to the number of iterations n. The design is fully parallel and is unrolled as presented in [19]. Before the x data reaches the first cell it must be converted to be positive, if it is negative, which is achieved via a negator. A switch token is used to control whether or not the negated version or the original is allowed to pass through. The control signal for the switch is generated via a compare to zero token. If negative values for the x data are allowed to pass, the computed magnitude will be negative and

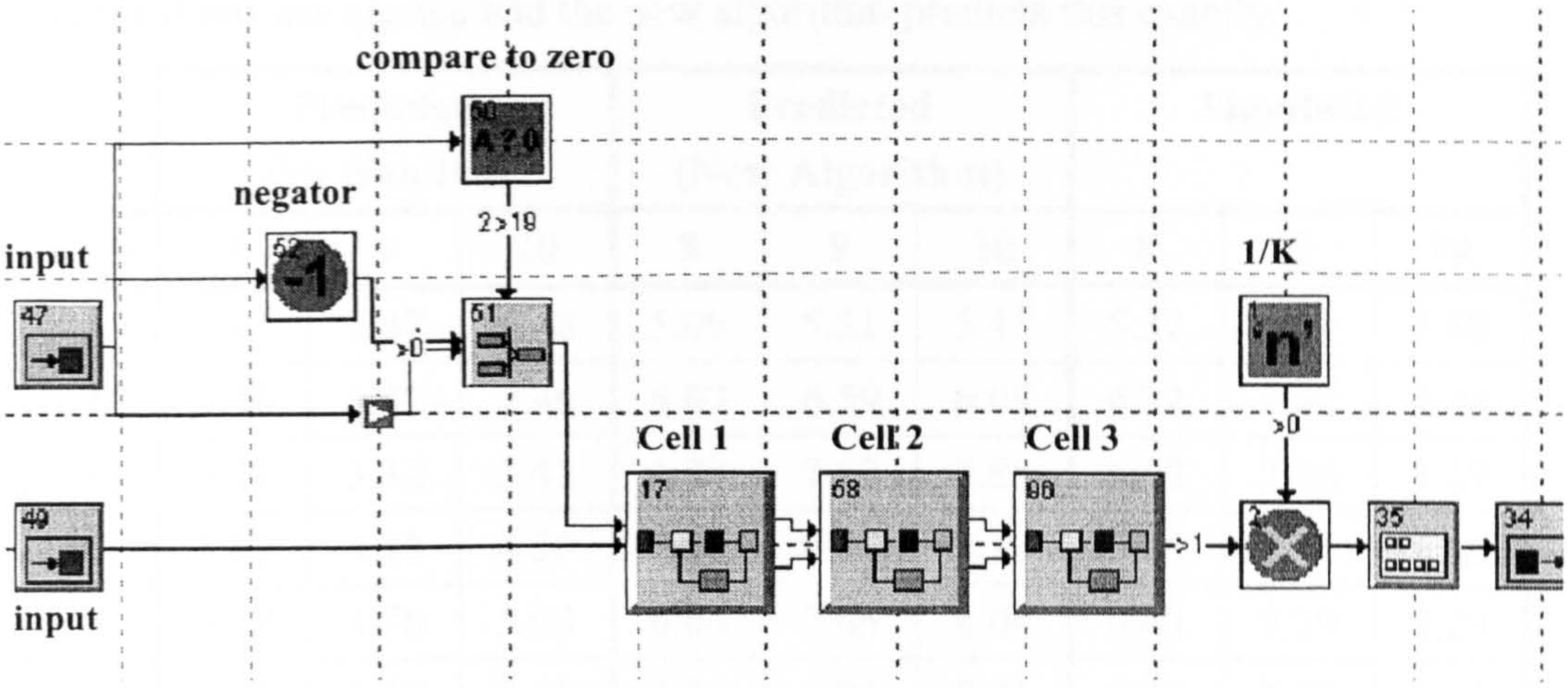


Figure: 7.22: Full CORDIC Design

although close to the correct result, will be slightly less accurate. There is no such concern for the *y* data.

To remove the scaling factor, *K*, a multiplier is used which is fed the input 1/*K* via a constant token. The Matlab code shown in Figure 7.12 was used to compute this value before each simulation. The format of the constant token is set to ±<1, b> as is the initial output of the multiplier, although this can be reduced later to ±<1, d_{eff}> once d_{eff} has been verified. Note that Rounding must be enabled on the multiplier once the output is reduced to the ±<1, d_{eff}> format. Only then will d_{eff} bits of accuracy be maintained. Within each cell, the adders/subtractor and shift tokens take the format ±<2, b> where the extra integer bit is required to allow for the growth due to the scaling factor, *K*.

A sub-set of the results for these simulations are given in Table 12 along with the predicted values generated using the Yu Hen Hu approach and the new algorithm. It is clear that the new algorithm predicts the simulated values far more accurately than the original algorithm. For example, examining the case where 6 effective fractional bits are required. The Yu Hen Hu algorithm predicts that at least 9 iterations with 10 fractional bits are required. However, the simulation shows that only 4 iterations with

8 fractional bits are needed and the new algorithm predicts this exactly.

	Predicted (Yu Hen Hu)			Predicted (New Algorithm)			Simulated		
n / b	**8**	**9**	**10**	**8**	**9**	**10**	**8**	**9**	**10**
3	1.43	1.47	1.48	5.09	5.31	5.43	5.32	5.71	5.80
4	2.35	2.42	2.46	**6.03**	6.59	6.98	**6.22**	6.88	7.34
5	3.17	3.32	3.41	6.28	7.13	7.88	6.52	7.56	8.27
6	3.82	4.12	4.30	6.21	7.17	8.10	6.55	7.11	8.12
7	4.27	4.76	5.08	6.06	7.05	8.04	6.42	7.29	8.29
8	4.50	5.18	5.69	5.92	6.91	7.91	6.33	7.29	8.31
9	4.57	5.40	**6.09**	5.78	6.78	7.78	6.00	7.00	8.00
10	4.55	5.46	6.30	5.65	6.65	7.65	5.68	6.68	7.68

Table 12: Predicted vs Simulated Values Of d_{eff}

7.4.8 Hardware Comparison

To assess how efficient the CORDIC algorithm is, comparisons were made between CORDIC systems found using both versions of the OQE algorithm and Direct systems computing the same level of accuracy but performing $\sqrt{x^2+y^2}$ directly using two multipliers, an adder and the square root operator developed for HDS. A comparison was made in terms of gate count to give an estimation of the resources required by each design.

Gate Count

To assess the gate count of each design, the case where 16 effective fractional bits are required from the vector magnitude calculation was considered. The CORDIC design found using the new OQE, requires $n=9$ and $b=20$. The number of gates in this design can be assessed by breaking it down into individual components. First of all the circuit that converts any negative values to equivalent positive ones is considered. This

requires one XOR gate and one half adder per input bit, therefore 3 gates/bit are required.

Now the CORDIC cells must be examined. Each cell uses one NOT gate and two adder/subtractors. The shift operations are all achieved using wiring and do not contribute to any hardware consumption. An adder/subtractor requires one XOR gate and a full adder per bit, which means there is a total of 6 gates/bit required for each one. It is important to note that the final cell (cell 9) uses only 1 adder/subtractor as it is only the x output that is of interest. Therefore, cell 9 uses only half the logic relative to the other cells.

Finally, the parallel multiplier used to remove the scaling factor requires a x b cells where a and b are the width of the inputs to the multiplier. Each cell in the multiplier represents an AND gate and a full adder, so there are 6 gates/cell.

This means that with an input width of ±<1,20>, the total number of gates for this CORDIC design is:

$$|x| = 21 \times 3 = 63$$
$$\text{cells} = 8.5 \times ((22 \times 6 \times 2) + 1) = 2252.5$$
$$\text{mult} = 22 \times 22 \times 6 = 2904$$
$$\text{Total} \approx 5220$$

Note that the width of the data paths through the cells is 22 bits, as 2 integer bits are required plus 20 fractional bits.

If a similar analysis is made to the design that was found using the original OQE algorithm ($n = 17$ and $b = 21$), the number of gates is assessed as:

$$|x| = 22 \times 3 = 66$$
$$\text{cells} = 16.5 \times ((23 \times 6 \times 2) + 1) = 4570.5$$
$$\text{mult} = 23 \times 23 \times 6 = 3174$$
$$\text{Total} \approx 7810$$

A similar approach was taken with a Direct design which can be seen in Figure 7.23.

To assess the number of gates in this design first remember that multipliers require 6 gates/cell as described previously. However, in this case the multipliers are computing

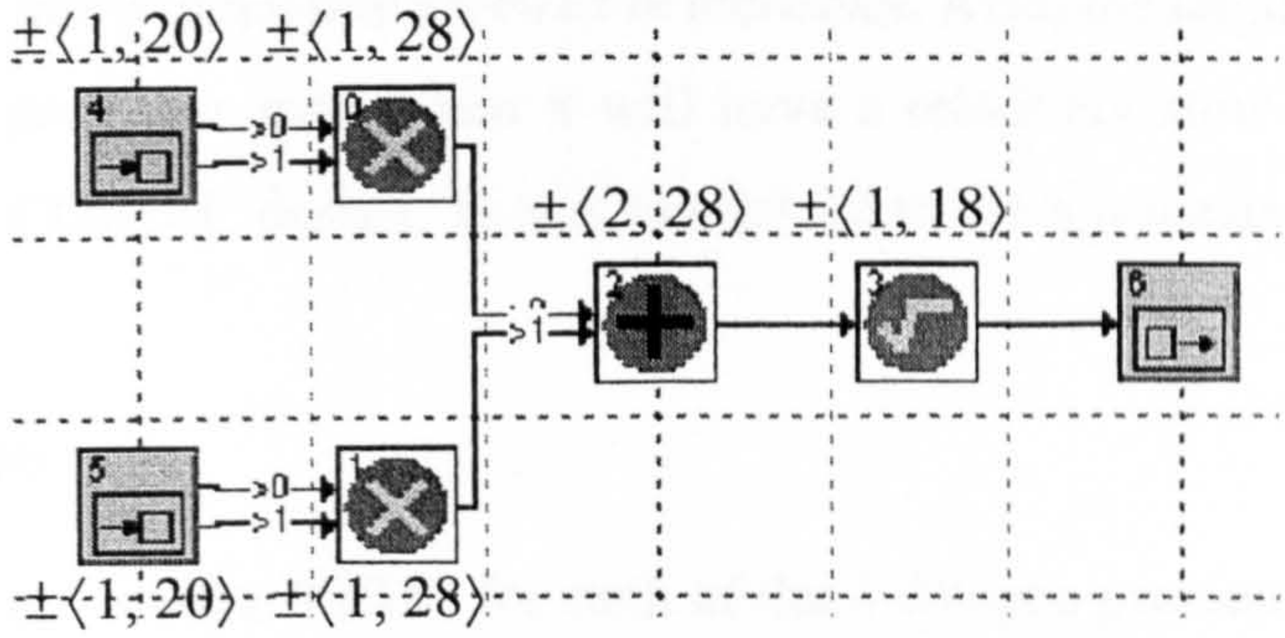


Figure: 7.23: Direct Design

squares and therefore the number of cells required is approximately half that required by a normal multiplication [27]. This has been taken into consideration when estimating the gate count. An adder requires 5 gates/bit as each full adder uses 5 gates. Finally, a square root function can be viewed as half the logic required for a divider array [31] which requires m x m cells where m is the width of the numerator and the denominator. Each cell is made up of an XOR gate and a full adder, hence 6 gates are required per cell. This means that if the inputs are ±<1,20>, and the number of fractional bits in the multipliers and the adder are allowed to only grow to 28, which is required to achieve 16 *effective* fractional bits from the square rooter, the number of gates is assessed as:

$$\text{multipliers} = 21 \times 21 \times 6 = 2646$$

$$\text{adder} = 29 \times 5 = 145$$

$$\text{square root} = (30 \times 30 \times 6)/2 = 2700$$

$$\text{Total} \approx 5491$$

Clearly the CORDIC design found using the new OQE equation is the most efficient in terms of the number of gates required. Relative to the CORDIC system found using the original OQE equation, a saving of 33% is achieved. The saving relative to the Direct design is not quite as impressive but is still significant at 5%. It should be noted that the gate count for the Direct design is an optimistic estimate and that in reality it would be greater. Also, this saving will increase with larger designs as the Direct design will not scale well due to the design of the HDS square rooter. The structure of the square root core requires $m^2/2$ cells where m the width of the input to

the core. Clearly this will not scale well as m increases. Also, the large critical path that this structure experiences means that it will have a relatively slow clock rate when compared to the CORDIC design. This is confirmed in the Synthesis results.

Synthesis Results

Using HDL Design Studio, VHDL for each of the 3 designs previously analysed was automatically generated. Xilinx ISE v8.1 was then used to implement the designs on a Virtex-II Pro device. The results are given in Table 13, which illustrates the efficiency of the CORDIC design found using the new OQE algorithm relative to the other two designs. Not only does it use fewer slices but it also permits a higher clock frequency. It should be noted that both CORDIC designs have been implemented using only slice logic, therefore the post scaling multiplier has been generated using the FPGA fabric rather than a dedicated multiplier. This means that the slice count for the CORDIC designs could be reduced even further by forcing the synthesis tool to use a dedicated multiplier instead.

The results for the Direct design are slightly unfair as the VHDL for the square root design is not as efficient as it could be. It uses twice the logic actually required. Hence, the slice count for the Direct design should be approximately 700. Also, it is possible that the synthesis tool has not optimised the multipliers for computing squares.

	CORDIC *n*9 *b*20	CORDIC *n*17 *b*21	Direct
Slices	364	670	1234 (700)
18x18 Mults	0	0	0
MHz	>20	>10	>10

Table 13: Post Implementation Results For Each Design

7.5 Predicting The Accuracy Of Sine/Cosine Calculations

In this section the work that has been carried out to allow the accuracy of CORDIC systems computing sine and cosine to be predicted is presented.

7.5.1 The Algorithm

To compute the cosine and sine of an angle z using CORDIC, a Circular coordinate system must be used in Rotation mode. The output that is produced from the CORDIC algorithm when operated in this mode is shown in Figure 7.24.

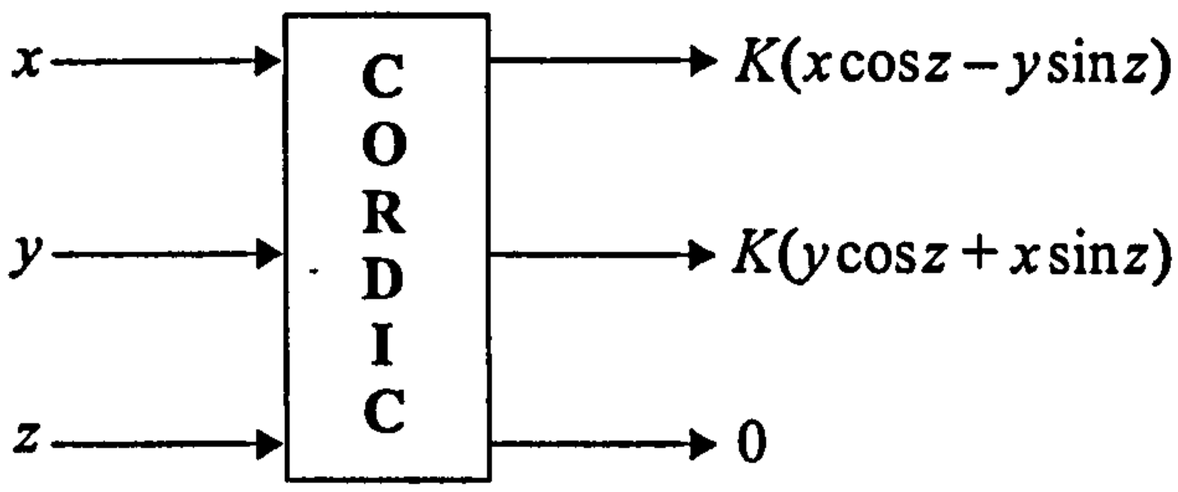


Figure: 7.24: CORDIC Output For Circular Rotations In Rotation Mode

However, by setting the initial values of x and y to:

$$x = 1/K, \quad y = 0 \tag{7.27}$$

the output from the x and y equations can be forced to equal $\cos z$ and $\sin z$ respectively. Thus, the scaling factor K is removed without having to use a post scaling multiplier.

The CORDIC algorithm is given in (7.28) where, $e^{(i)} = \arctan(2^{-i})$ is used due to the Circular rotations that are being implemented.

$$\begin{aligned} x^{(i+1)} &= (x^{(i)} - d_i(2^{-i}y^{(i)})) \\ y^{(i+1)} &= (y^{(i)} + d_i(2^{-i}x^{(i)})) \\ z^{(i+1)} &= z^{(i)} - d_i e^{(i)} \\ \text{where, } d_i &= \text{sign}(z^{(i)}) \end{aligned} \tag{7.28}$$

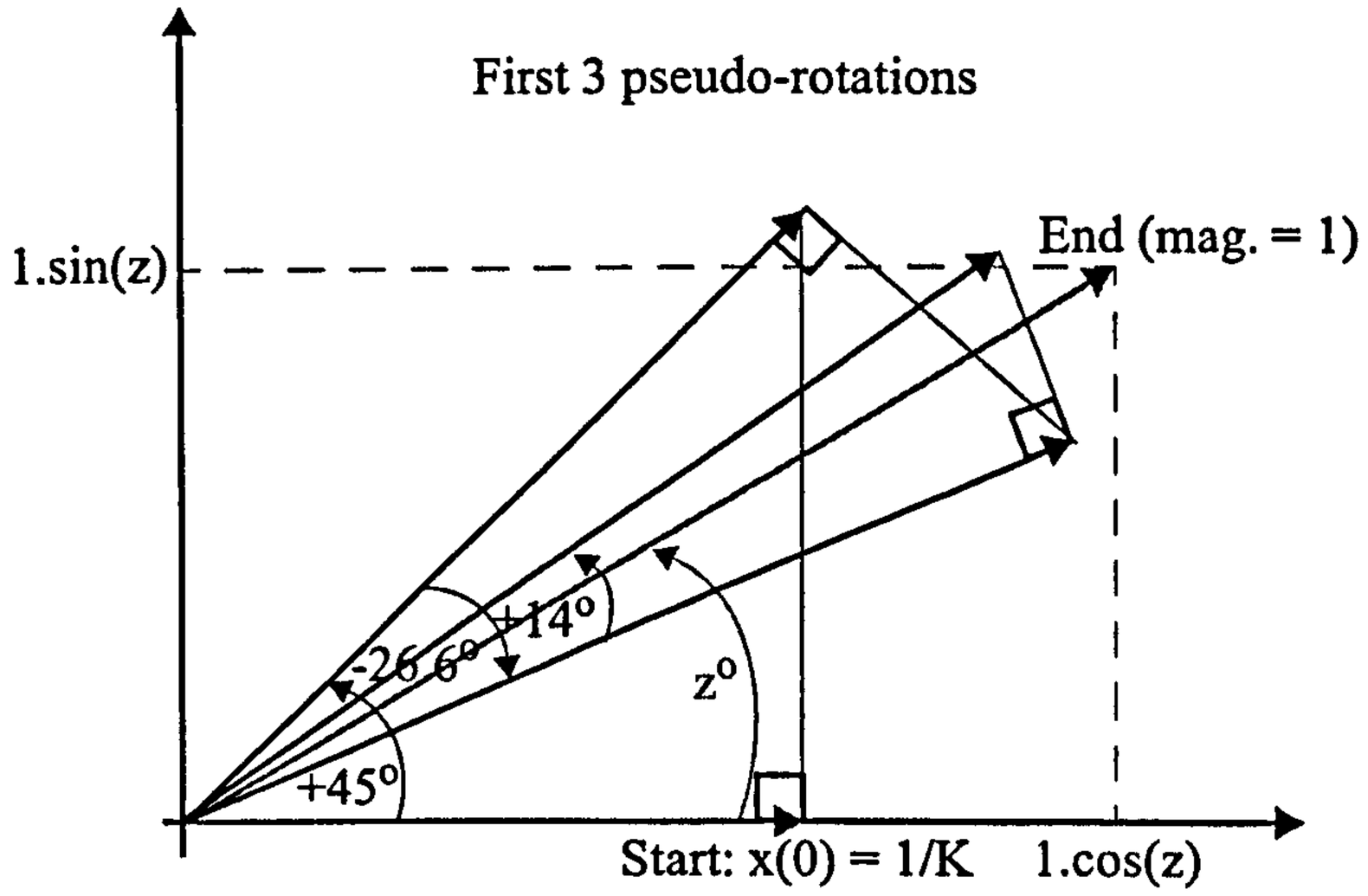


Figure: 7.25: Rotation Mode Example

By considering Figure 7.25, the first 3 rotations for some arbitrary angle z can be seen. It is clear that by starting with a magnitude of $1/K$, then after n iterations, because the magnitude has been scaled by K, the magnitude of the final vector is equal to 1. Hence, the x component and y component are equal to $\cos z$ and $\sin z$ respectively.

7.5.2 The Overall Quantisation Error (OQE)

The Overall Quantisation Error (OQE) of a CORDIC system is presented in [18] and has already been presented as consisting of two parts:

- The Approximation Error ε_a : the error due to the quantised representation of a CORDIC rotation angle by finite numbers of elementary angles.
- The Rounding Error ε_r : the error due to the finite precision arithmetic used in a practical implementation.

In [18], an equation for the Rounding error was developed and is defined in terms of the number of iterations n and the number of fractional bits in the data path b. This equation is given as:

$$\varepsilon_r = 2^{-b-0.5}\left[\frac{G(\mu, n)}{K_\mu(n)} + 1\right]$$

$$\text{where, } G(\mu, n) = 1 + \sum_{j=1}^{n-1}\prod_{i=j}^{n-1} k_\mu(i) \tag{7.29}$$

$$\text{and, } K_\mu(n) = \prod_{i=0}^{n-1} k_\mu(i) = \prod_{i=0}^{n-1}\sqrt{1+\mu 2^{(-2i)}}$$

The Rounding Error can apply to any CORDIC system using Circular rotations. However, to predict the error in CORDIC systems computing sine and cosine, a new Approximation Error needed to be developed and this is presented next.

In [21] the work presented in [18] was extended to develop an equation for the precision of CORDIC systems operating in rotation mode. The equation is given as:

$$x = n + \log_2 n + 2 \tag{7.30}$$

where n is the number of iterations and x is the total number of bits in the data path. According to [21] this should be sufficient to obtain n bits of precision from the output. Note that the total number of bits uses only two for the integer part and the remaining $(n + \log_2 n)$ bits are for the factional part. However, by comparing floating point CORDIC simulations computing cosines and sines with a floating point reference design, it has been found that, at best, n - 1 bits of precision can be achieved for n iterations. Hence, (7.30) has been found to be inaccurate.

7.5.3 The Approximation Error

To analyse the Approximation Error of a CORDIC system computing cosines and sines a graphical approach was initially taken. An important parameter in predicting this error is to know the maximum angle that is left between the ideal finishing position at z and the actual CORDIC finishing position after a number of iterations n. This parameter is known as the Angle Approximation Error δ. This error at the nth iteration

is given in [18] as $\delta \leq a(n-1)$ where $a(n-1)$ represents the rotation angle at the $n-1$ iteration. This can be illustrated by considering the simple case of $n = 3$. Figure 7.26 shows all the possible finishing angles for this case. By assuming that the input $z \leq 90^{\circ}$

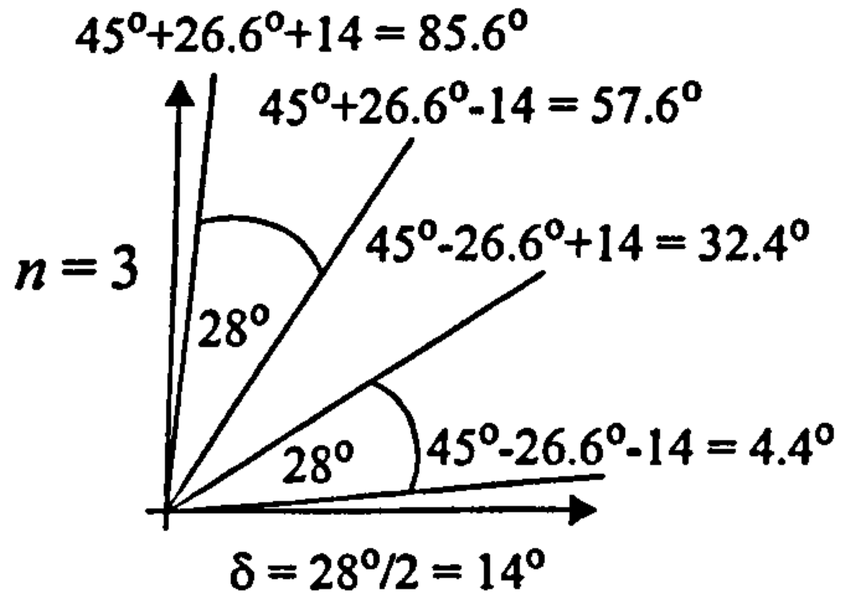


Figure: 7.26: The Approximation Error δ

then it is clear that the greatest value that δ can achieve is 14° $(a(n-1))$ which would occur if the input angle was either $z = 71.6^{\circ}$ or $z = 18.4^{\circ}$.

With this limit in mind, the relation that the Angle Approximation Error δ has on the error in the $\cos z$ and $\sin z$ calculation must be examined. For this, consider the case of $n = 2$. In Figure 7.27 the worst case input, which for this case is 45°, has been drawn along with the two nearest CORDIC outputs, 71.6° and 18.4°. Clearly the upper limit to the worst case error for the x output ($\cos z$) can be seen to be 0.39146 which is

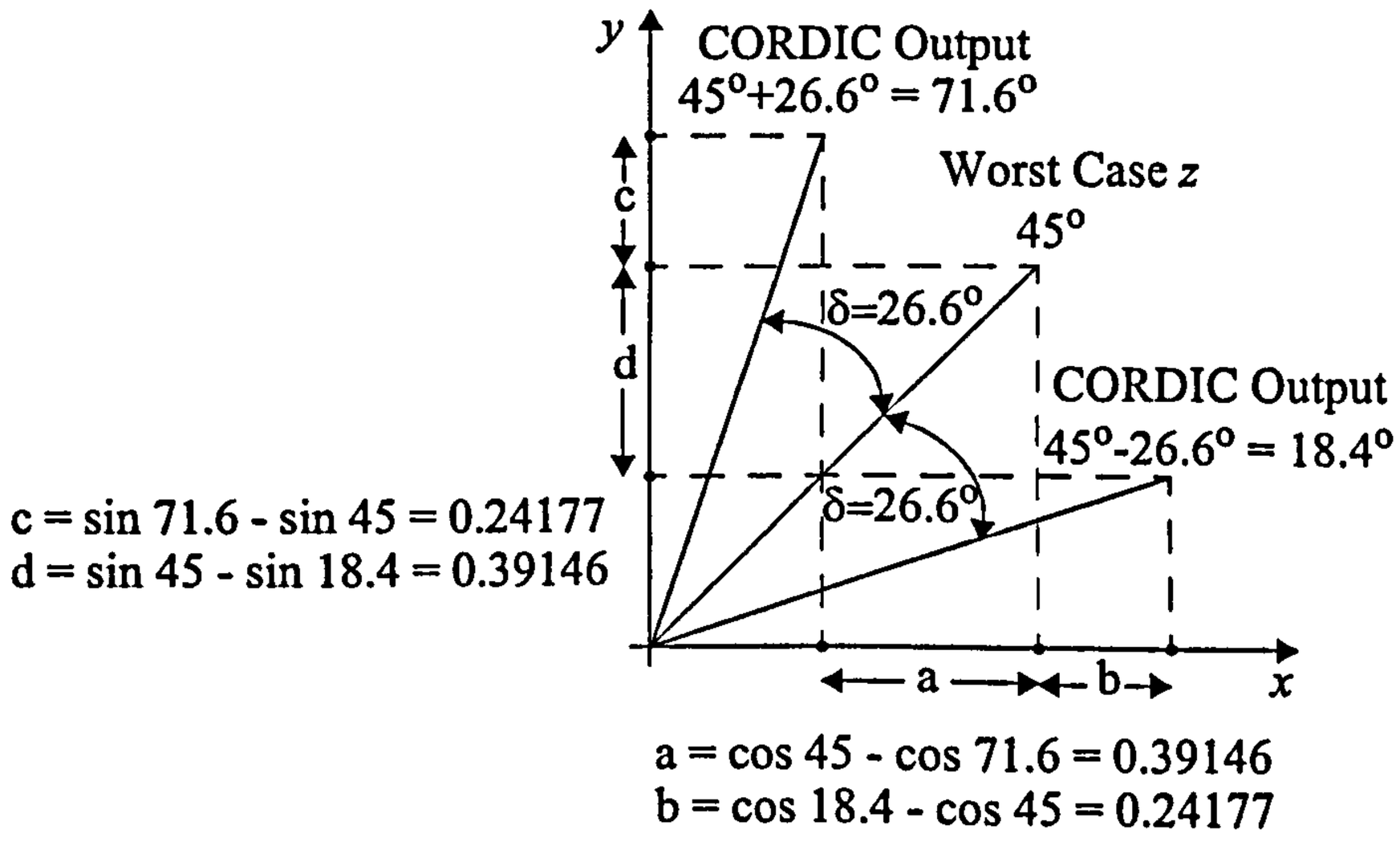


Figure: 7.27: Assessing The Approximation Error

also the limit for the y output ($\sin z$).

Taking a mathematical view of the situation in Figure 7.27 for the Approximation Error in both the cosine and sine components gives:

$$\begin{aligned} \varepsilon_a^{\cos} &= \cos(z \pm \delta) - \cos z \\ \varepsilon_a^{\sin} &= \sin(z \pm \delta) - \sin z \end{aligned} \tag{7.31}$$

Expanding both equations in (7.31) gives:

$$\begin{aligned} \varepsilon_a^{\cos} &= (\cos z \cos\delta \mp \sin z \sin\delta) - \cos z \\ \varepsilon_a^{\sin} &= (\sin z \cos\delta \pm \cos z \sin\delta) - \sin z \end{aligned} \tag{7.32}$$

Now, as n increases $\cos\delta \to 1$. This allows (7.32) to be simplified to:

$$\begin{aligned} \varepsilon_a^{\cos} &= \mp \sin z \sin\delta \\ \varepsilon_a^{\sin} &= \pm \cos z \sin\delta \end{aligned} \tag{7.33}$$

Finally, as it is the worst case error that is of interest, it is possible to reduce (7.33) further. Clearly the upper limit for $\sin z$ and $\cos z$ is 1. Hence, both equations reduce to:

$$\left|\varepsilon_a\right|_{max} = \sin\delta \tag{7.34}$$

To confirm (7.34), simulations were run for a set of n using the DSP package SystemVue. The simulations involved comparing the output of two systems. The first system was a double precision floating point CORDIC design computing cosines and sines for inputs in the range $-90^\circ \le z \le 90^\circ$. The second system also computed these functions using a floating point direct computation and was considered as the reference output. Both sets of output data were then compared and the worst case CORDIC error tracked. As double precision floating point data was used, the Rounding Error can be neglected and consequently any difference between the two simulations can be attributed to the Approximation Error ε_a. This was repeated for several n and the results can be seen in Table 14. Clearly as n increases, the predicted results get very

close to the simulated results thus verifying (7.34).

n	δ	ε_a (sim.)	$\varepsilon_a = \sin\delta$
1	45°	0.7071	0.7071
2	26.6°	0.3909	0.4472
3	14°	0.2395	0.2425
4	7.1°	0.1242	0.1240
5	3.58°	0.0619	0.0624
6	1.79°	0.0312	0.0312
7	0.90°	0.0156	0.0156
8	0.45°	0.0078	0.0078

Table 14: Simulation Results for ε_a Versus Predicted Results

To further confirm the simplification of the Approximation Error in (7.34), a fuller examination was also carried out. This began by considering the $\varepsilon_a^{\cos}$ equation in (7.32), which can be rewritten as:

$$\varepsilon_a^{\cos} = \mp \sin z \sin\delta + \cos z(\cos\delta - 1) \tag{7.35}$$

If the values of z and δ are in the range:

$$0 \le z \le \pi/2$$
$$0 \le \delta \le \pi/2$$

then the maximum Approximation Error is given with:

$$\left|\varepsilon_a^{\cos}\right|_{max} = max_z\left|-\sin z \sin\delta + \cos z(\cos\delta - 1)\right| \tag{7.36}$$

To find the z_{max} which causes $\left|\varepsilon_a^{\cos}\right|_{max}$ for each δ requires (7.36) to be differentiated with respect to z. This gives:

$$\frac{d}{dz}\varepsilon_a^{\cos}(z) = -\cos z \sin\delta - \sin z(\cos\delta - 1) \tag{7.37}$$

Then, the value of z where the maximum occurs (z_{max}) can be found from:

$$-\cos z_{max} \sin\delta - \sin z_{max}(\cos\delta - 1) = 0 \tag{7.38}$$

Rearranging (7.38) leads to:

$$z_{max} = \text{acot}\left(-\frac{\cos\delta - 1}{\sin\delta}\right) \tag{7.39}$$

Now, with some manipulation it can be found that (7.39) can be written as:

$$z_{max} = \frac{\pi}{2} - \frac{\delta}{2} = \frac{\pi - \delta}{2} \tag{7.40}$$

To find $\left|\varepsilon_a^{\cos}\right|_{max}$ for each δ, (7.40) can be used with (7.36) to give:

$$\left|\varepsilon_a^{\cos}\right|_{max} = -\sin\left(\frac{\pi - \delta}{2}\right)\sin\delta + \cos\left(\frac{\pi - \delta}{2}\right)(\cos\delta - 1) \tag{7.41}$$

This function can then be plotted against δ. This is shown in Figure 7.28 where the simplified equation for $|\varepsilon_a|_{max}$ (7.34) has also been plotted. It is clear from this plot that both equations give almost identical results for $\delta < 0.2$ rads ($< 11^{\circ}$). The difference between the two algorithms can be seen in Table 15. The number of effective fractional bits that each error represents is also shown. It can be seen that as

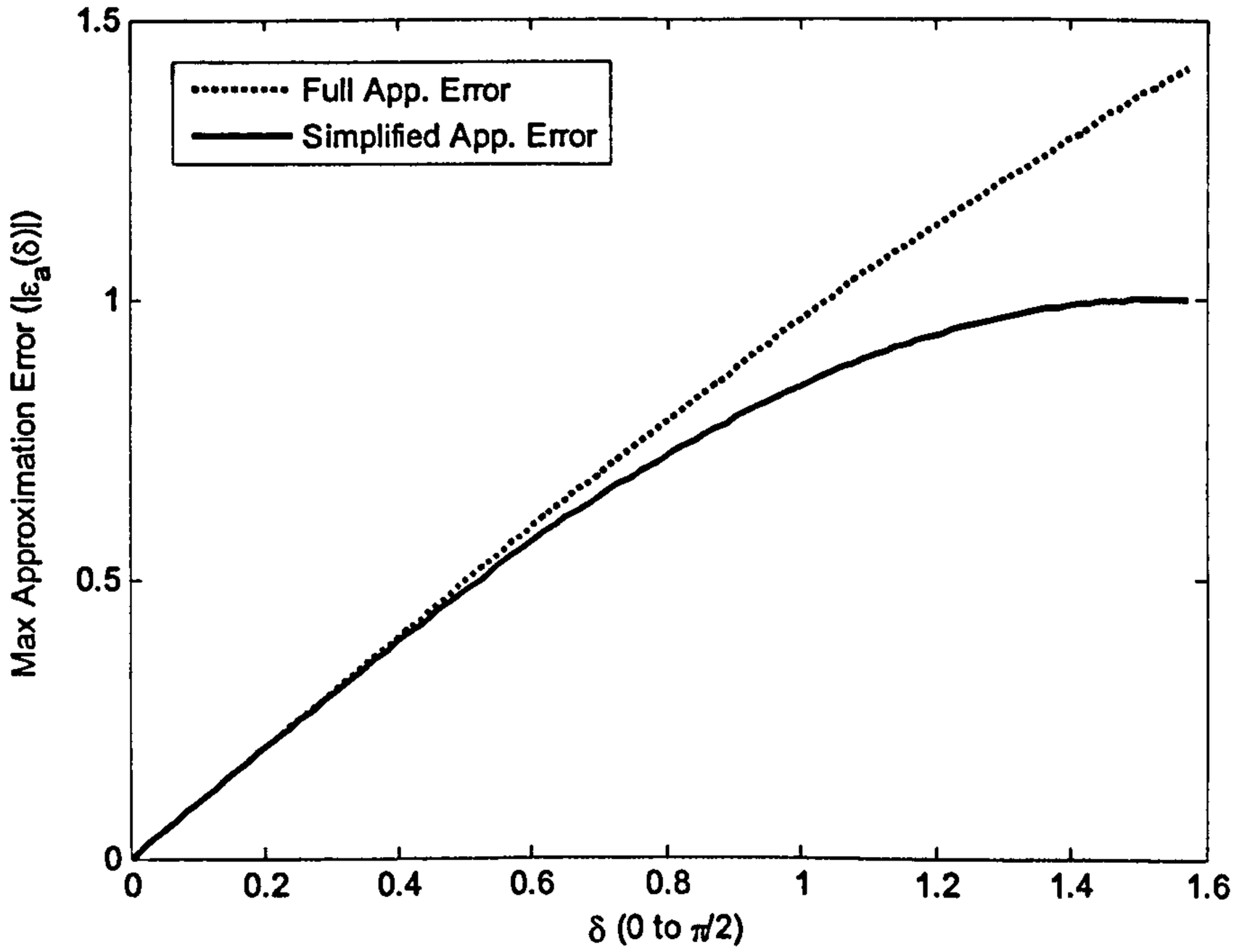


Figure: 7.28: $|\varepsilon_a|_{max}$ versus δ for the Full and Simplified Approximation Error Equations

n increases from 4 to 7 both equations start to give almost exactly the same results and hence it can be concluded that the simplified equation is a relatively accurate approximation for a real system.

	Full App. Error Result		Simplified App. Error Results	
n	Error	Eff. Frac. Bits	Error	Eff. Frac. Bits
4	0.1242748831	3.0083933485	0.1240347345	3.0111839065
5	0.0624086775	4.0021095485	0.0623782861	4.0028122745
6	0.0312385631	5.0005280943	0.0312347523	5.0007040971
7	0.0156235697	6.0001320678	0.0156230930	6.0001760887

Table 15: Comparison of Full versus Simplified Approximation Error Equations

As mentioned already, the OQE is made up of two parts, the Approximation Error and the Rounding Error. Having developed an expression for ε_a, the complete OQE equation, in terms of *n* and *b*, is given as:

$$\text{OQE} = \varepsilon_a + \varepsilon_r = \sin\delta + 2^{-b-0.5}\left[\frac{G(n)}{K(n)} + 1\right] \tag{7.42}$$

$$\text{where } \delta = a(n-1) = \arctan(2^{-n+1})$$

7.5.4 Simulations

To verify the OQE given in (7.42), two steps were taken:

- Matlab code was written to compute the OQE for various *n* and *b* and then convert it into the number of effective fractional bits that it represents.
- Fixed point CORDIC simulations for various *n* and *b* were carried out and the number of effective fractional bits that were produced relative to a direct floating point cosine and sine calculation were measured.

With two sets of effective fractional bits (d_{eff}) it was then possible to verify how accurate the OQE is at predicting the accuracy of fixed point CORDIC systems

computing cosine and sines.

7.5.5 Predicting The Accuracy

To allow the OQE to be assessed for many combinations of n and b, Matlab code was written which allowed this to be done quickly and accurately. First of all functions to compute K and G were written (see Figure 7.12 and Figure 7.13). These functions were then called and used to compute the Rounding Error according to (7.29). The Approximation Error ($\sin\delta$) was also computed before both errors were added together to form the OQE. Finally, using (7.43), the OQE was converted to represent the number of effective fractional bits (d_{eff}) that are achievable. The Matlab code can be seen in Figure 7.29.

$$d_{eff} = -(\log_2 OQE) \tag{7.43}$$

```
function Deff = EffBits2(n,b)
%EffBits returns the number of effective fractional bits computed in
%a fixed point CORDIC system relative to a floating point CORDIC sys
%EffBits inputs: n = iterations, b = bits used
%input vector; Returns the number of effective bits computed.
G = Ggen(n);
K = Kgen(n);
Z = atan(2^(-n+1));
approx_error = sin(Z);
rounding_error = 2^(-b-0.5)*((G/K)+1);
OQE = approx_error+rounding_error;
Deff = -(log2(OQE));
```

Figure: 7.29: Matlab Code For Computing The OQE

Using this code, a table listing d_{eff} values for all combinations of $1 \le n \le 40$ and $1 \le b \le 40$ was produced. This table can be seen in Appendix C.

7.5.6 Fixed Point Simulations

To verify the predicted d_{eff} values, a number of fixed point CORDIC simulations were carried out for various n and b. In Figure 7.30 a fixed point CORDIC system computes

cos$z$ and sin$z$. At the same time, a Sin token computes the floating point solution to cos$z$ and sin$z$ and this is used as the reference solution. The corresponding fixed and floating point versions are then subtracted from each other yielding the fixed point error for each specific z. The worst case error for both cosine and sine computations is tracked for the duration of the simulation. Finally the d_{eff} value is computed and displayed.

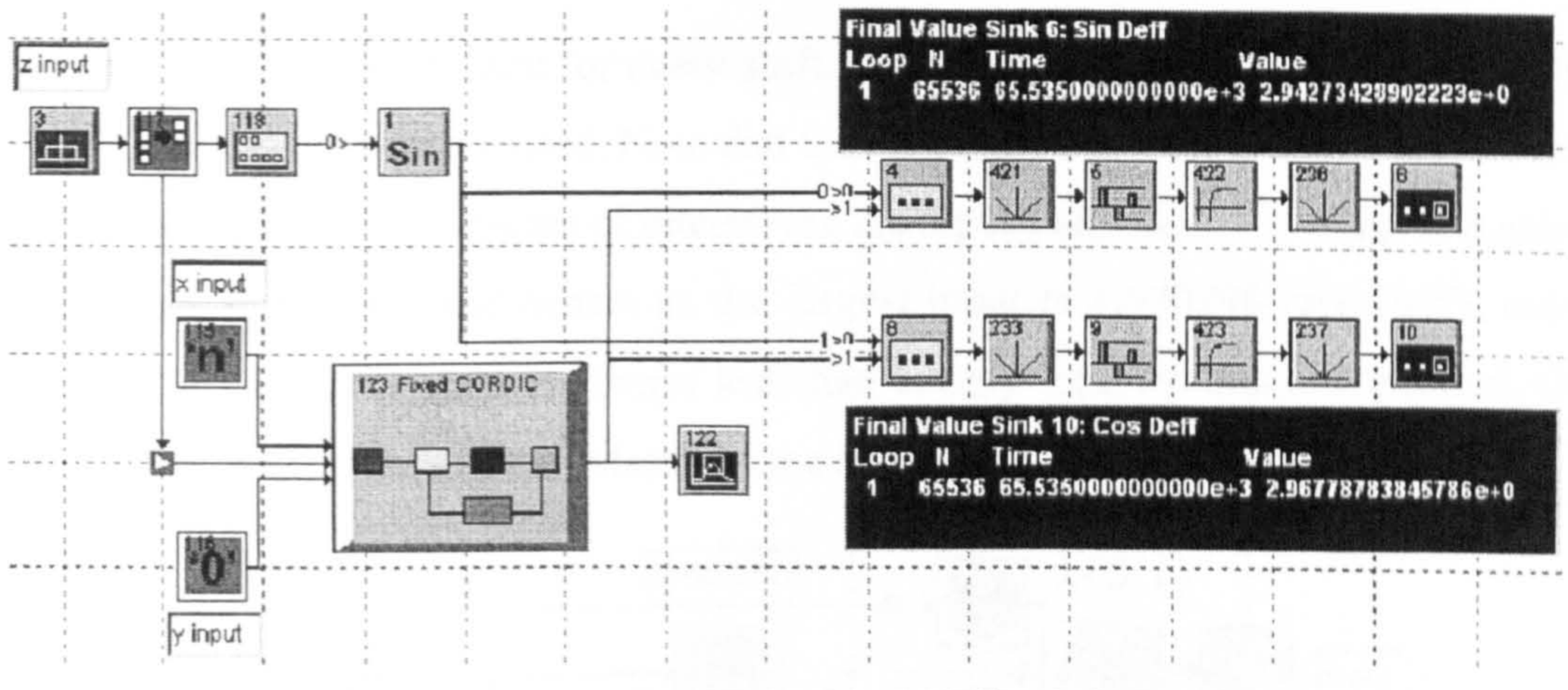


Figure: 7.30: Fixed Point CORDIC Simulation

Figure 7.30 shows the top level of the simulation but to understand what is happening within the CORDIC system, it is helpful to drop down a level and look inside the CORDIC box. Figure 7.31 shows that the CORDIC algorithm is made up of several cells indicating that a fully unrolled implementation has been selected as shown in [19]. Each cell computes one iteration. In the example shown, $n = 4$. To further understand the algorithm, the logic within one of these cells must be viewed. Figure 7.32 shows the logic that makes up each cell and clearly reflects the algorithm according to (7.28). It should be noted that every cell is almost identical, differing only in the size of the shift that is made and the angle that is rotated. The same bit widths are used in each cell. So for example, every addition/subtraction uses $\pm\langle 2, b\rangle$ format.

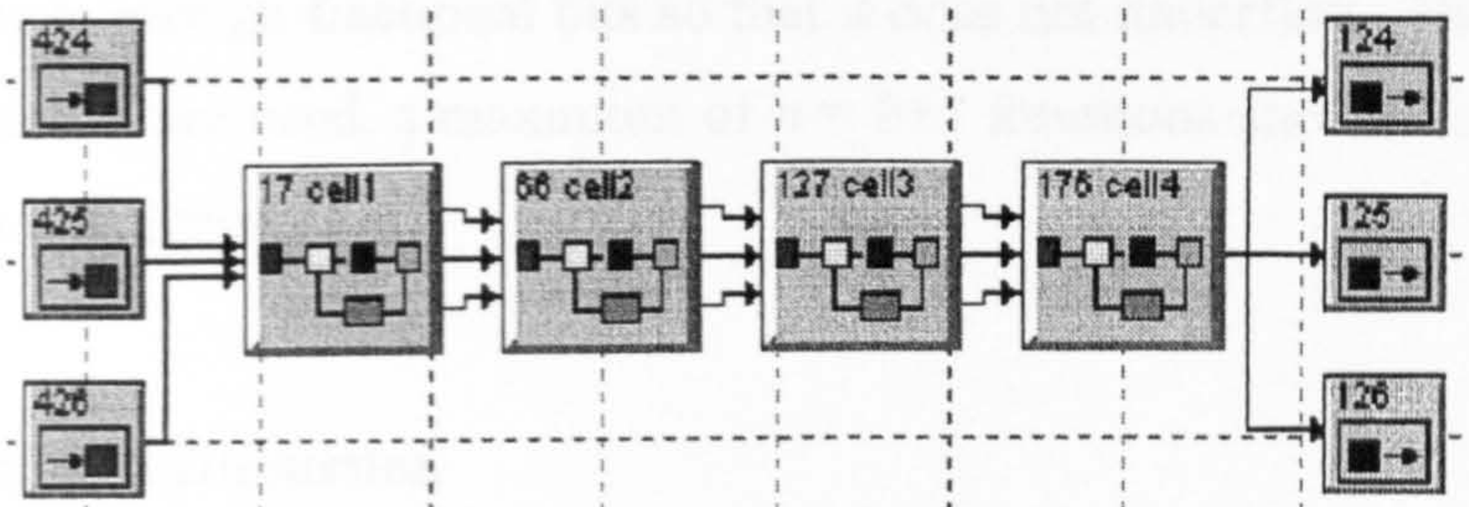


Figure: 7.31: CORDIC Cells

This is also the output format for every shift. However, each rotation angle uses only $\pm\langle 1, b\rangle$ bits, as radians are used. Note that the biggest rotation angle is 0.785398 rads (45°). Other parameters that are of interest are the input z and the $x = 1/K$ input. Again, radians are used for z and hence, as the largest input is 1.570796 rads (90°), only $\pm\langle 2, b\rangle$ bits are used. As x is always less than 1, only $\pm\langle 1, b\rangle$ bits are required. Of course, $1/K$ is computed (using Matlab) for every simulation as it varies for each n.

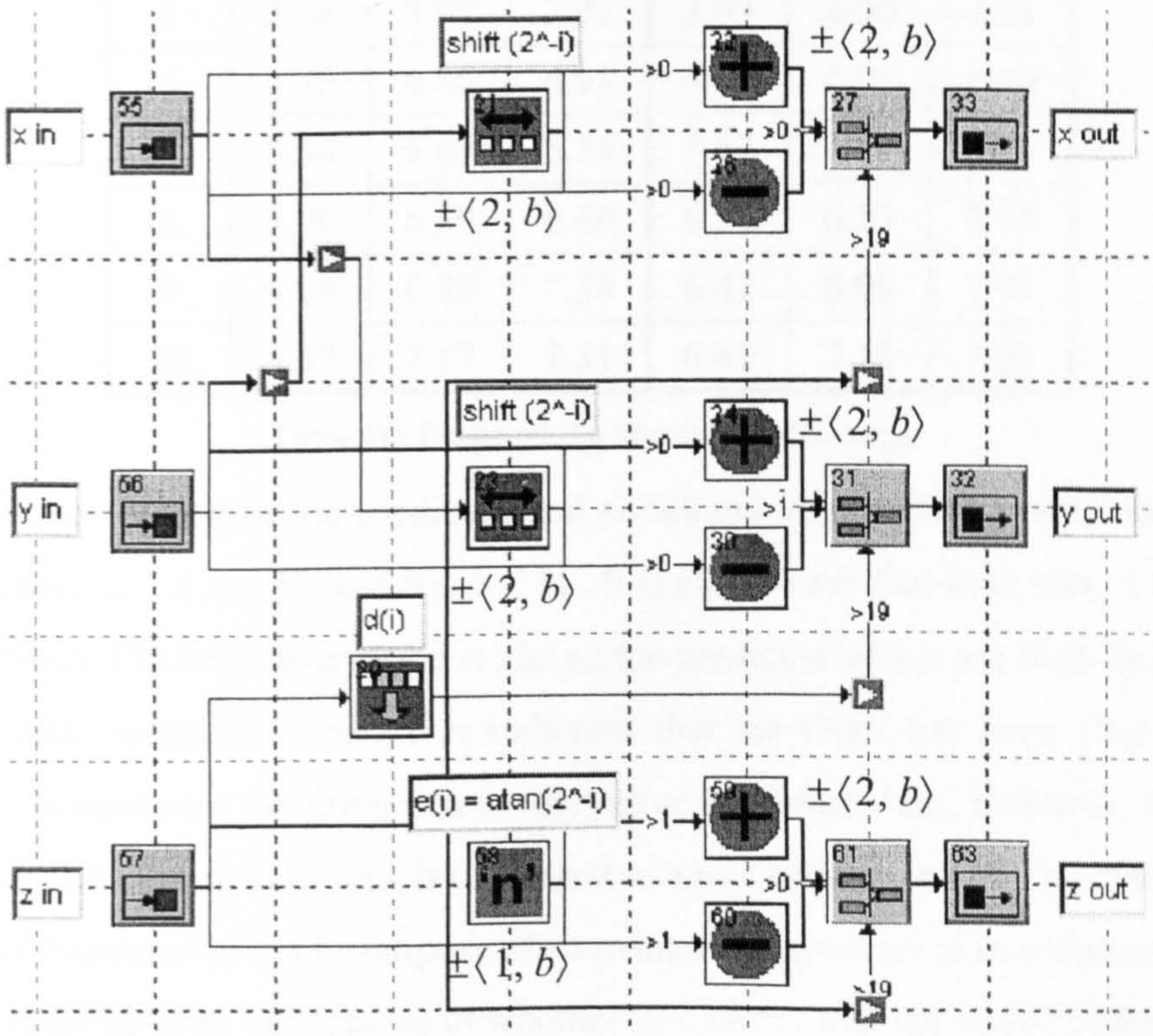


Figure: 7.32: Inside A CORDIC Cell

A final point to note on bit widths regards the rotation angle $e^{(i)}$. The rotation

angle must have enough fractional bits so that it does not underflow. This means that if b fractional bits are used, a maximum of $n = b+1$ iterations should be used or else this situation will occur.

7.5.7 Results And Discussion

To assess how accurate the OQE algorithm is at predicting the actual accuracy of fixed point CORDIC systems computing cosines and sines, both sets of d_{eff} values must be compared.

	Predicted			Simulated		
n / b	**9**	**10**	**11**	**9**	**10**	**11**
3	2.02	2.03	2.04	2.06	2.06	2.06
4	2.96	2.98	3.00	3.00	3.01	3.01
5	3.88	3.94	3.97	3.99	4.00	4.01
6	4.72	4.86	4.93	4.91	4.95	4.98
7	5.44	5.69	5.84	5.67	5.82	5.91
8	5.96	6.39	6.66	6.18	6.53	6.75
9	6.26	6.88	7.34	6.41	6.99	7.41
10	6.39	7.17	7.81	6.41	7.18	7.83

Table 16: Predicted d_{eff} Versus Simulated d_{eff}

Table 16 shows the predicted and simulated values for d_{eff} obtained for all combinations of $3 \le n \le 10$ and $9 \le b \le 11$. It is clear to see that both sets of numbers are very close. The first observation is that all the predicted values are slightly less than the equivalent simulated ones. This indicates that the OQE has been slightly over estimated in each case resulting in a conservative approximation. However, it is only the integer part of each value that is of interest as a real system can only have an integer number of fractional bits. The purpose of showing the d_{eff} values to two decimal places allows a judgement to be made as to whether it is likely that the actual system might be slightly more accurate than the predicted value suggests. For example, by considering the case of $n = 4$, $b = 9$ the predicted value is 2.96. As the OQE seems to

slightly over estimate each error, it is likely that 3 fractional bits of accuracy are obtainable. Indeed, by looking at the actual simulated result for this scenario, it is clear that this is the case.

7.5.8 Design Example

It is useful to work through a design example to illustrate one way of using the d_{eff} table. In this case $\cos z$ and $\sin z$ must be computed with an accuracy of 10 fractional bits. The first step is to look at the table and find the first location that yields 10 fractional bits. A useful rule to remember is that by ignoring ε_r, b fractional bits of accuracy require a minimum of $b + 1$ iterations. This can be verified by computing d_{eff} for each ε_a in Table 14. However, when ε_r is added $b + 2$ iterations are required to achieve b fractional bits of accuracy. Looking at the table in Appendix C, 10.2 fractional bits are achievable with $n = 12$ and $b = 14$. A simulation using these parameters confirms that this is actually the case. This illustrates the ease with which a CORDIC system can now be designed to guarantee a desired level of accuracy when computing cosines and sines.

Chapter 8

Pipelined Feedback Loops

8.1 Introduction

This chapter presents the work that was carried out to investigate the effects of pipelining a feedback loop. Much of the work presented in this thesis can be related to least squares adaptive equalisation techniques such as QR-RLS systems. Such systems have feedback within the systolic array as illustrated in Figure 5.11.

Chapter 8.2 starts by considering the issues involved with pipelining a feedback loop. In Chapter 8.3 the consequences of doing so are examined. Two hardware designs using feedback, one with pipelining and one without, which are algorithmically identical, are presented. It should be noted that the square root and divider cores developed for HDS are used in this analysis as they have the ability to have pipelining turned on and off. This comparison is used to show differences in speed, area and power consumption and ultimately to reveal which approach is best. In Chapter 8.4, a scenario is presented where pipelining a feedback loop can offer benefits. This is based on channel interleaving for a suitable multichannel scenario and offers a low cost hardware solution for low data-rate applications. Chapter 8.5 reviews the generic outcomes of the work before the conclusions are given in Chapter 8.6.

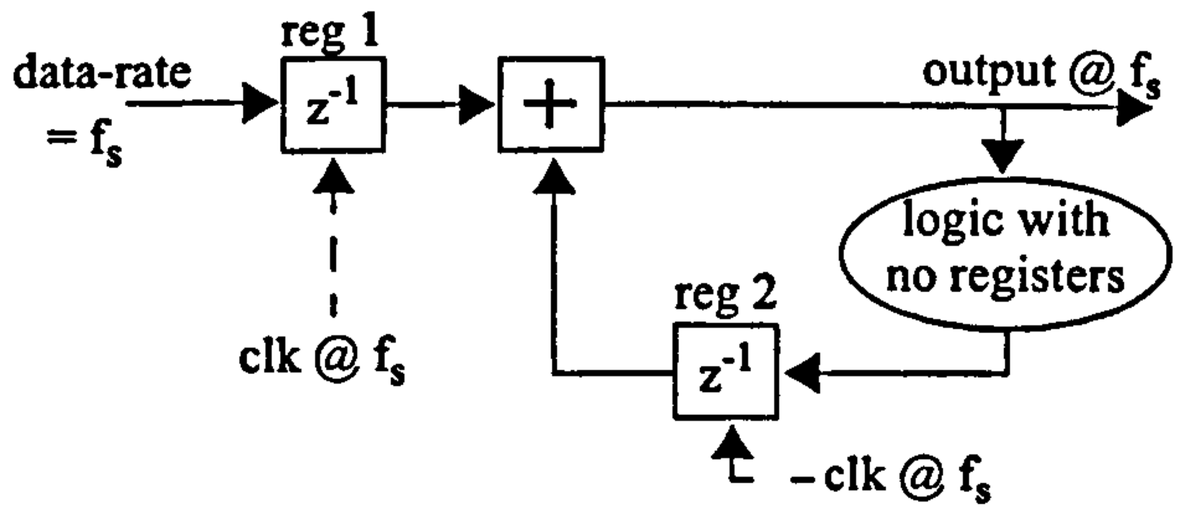


Figure: 8.1: Feedback loop

8.2 Pipelining A Feedback Loop

It is possible to pipeline a feedback loop while still preserving the original algorithm. Figure 8.1 shows a feedback loop with an adder and some logic containing no registers. A single delay (reg 2) is required to synchronise the feedback data with the new data as it arrives.

The maximum speed at which a system can be clocked at is determined by the *greatest delay* between any two connected registers. This delay represents the time it takes for data to travel between two registers and thus limits the clock speed. If the device is clocked faster than this limit, data to be captured by the receiving register will arrive after the clocking signal and so will not get registered. Thus, the longest delay between two registers in a design is known as the *critical path*. In Figure 8.1, the critical path is either the connection between reg 1 and reg 2 or it could be the feedback path connecting the output of reg 2 with its own input.

Figure 8.2 illustrates a pipelined version of the feedback loop shown in Figure 8.1, where it is assumed that pipelining registers in the feedback logic block will reduce the critical path. This circuit still represents the same algorithm as the non-pipelined design. To achieve this the registers in the feedback loop must be clocked at $n \times f_s$. The pipelined registers need to be clocked at this rate so that the feedback result arrives at the adder at the same time as the next sample. However, now the critical path will have been reduced as there are more registers in the data path. This means that the maximum

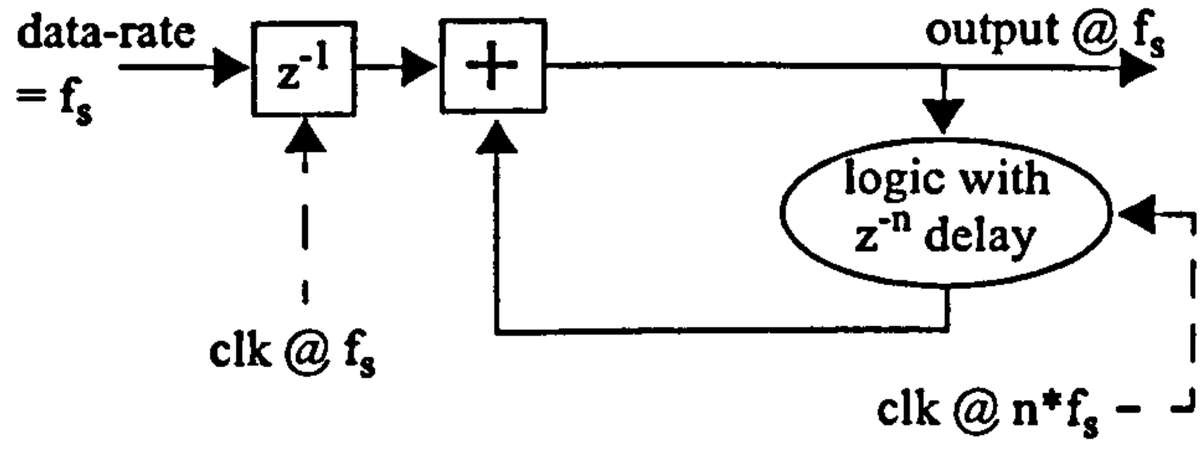


Figure: 8.2: Feedback loop with pipelining

clock speed has now increased. *To determine the maximum data-rate at which this circuit can operate with, the maximum clock rate must be obtained and divided by 'n'.* The question is then which of the two designs is faster and what are the respective power and logic resource requirements?

8.3 To Pipeline Or Not To Pipeline?

To try to answer these questions, two circuits were designed using HDL Design Studio. This tool is used for the design, simulation and implementation of DSP systems on FPGAs. The methodology is based on the professional DSP design software, SystemVue by Elanix, and uses a bit true fixed-point library (FXP-Lib) which maps directly to synthesisable HDL code.

8.3.1 Givens Rotation With Feedback

The circuits designed were based on logic found in a QR decomposition (QRD) using Givens rotations. This technique is used for QR-RLS optimisation and has a wide range of application in adaptive filtering. The QR algorithm can be implemented using a parallel array of cells as illustrated in Figure 8.3 and discussed in Section 5.4.3. Each cell in the array performs a Givens rotation according to (8.1). However, the boundary cell on each row differs from the other cells in that it must calculate θ_i and pass it along the row to the other cells (the Givens Generation), as well as perform a Givens rotation itself.

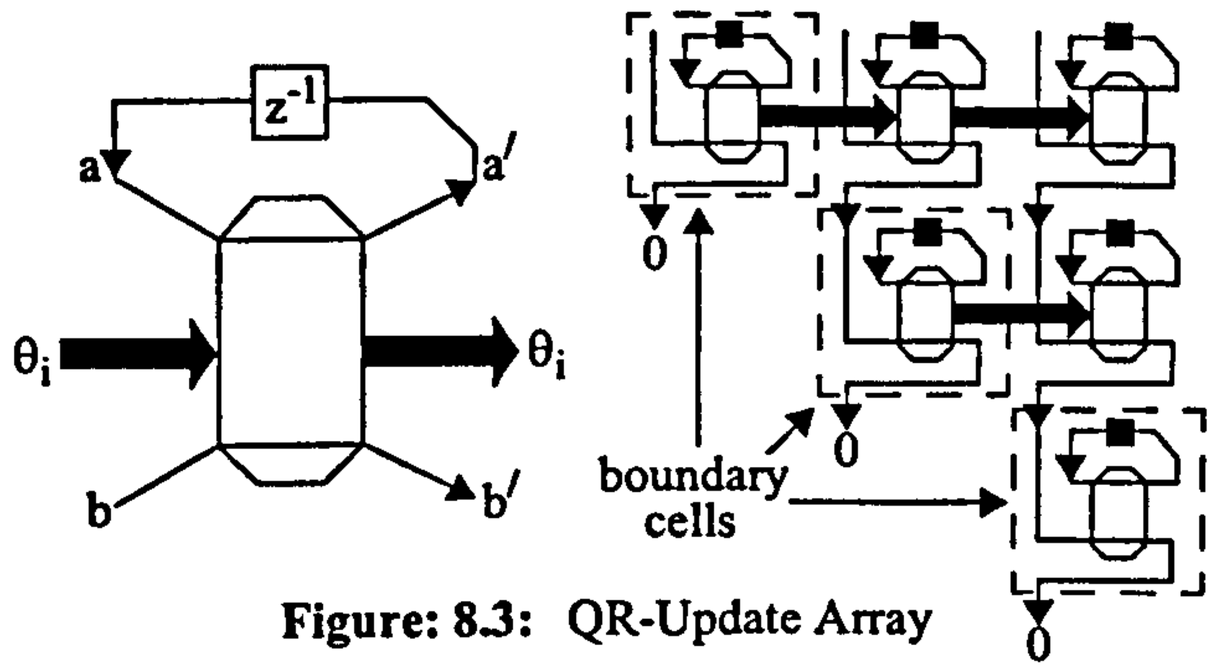


Figure: 8.3: QR-Update Array

$$\begin{aligned} a' &= a\cos\theta_i + b\sin\theta_i \\ b' &= b\cos\theta_i - a\sin\theta_i \end{aligned} \tag{8.1}$$

For the purpose of the experiment, a pipelined and a non-pipelined implementation of this particular cell was chosen. A schematic illustrating one implementation of a boundary cell is shown in Figure 8.4.

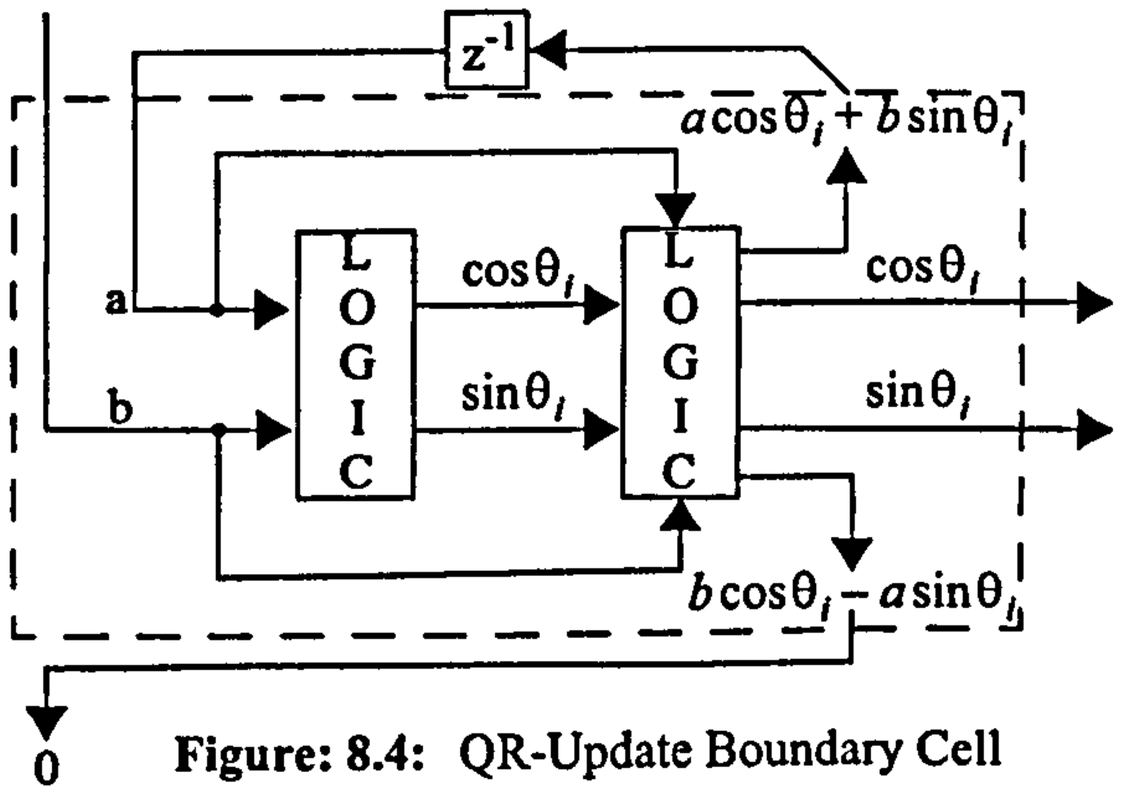


Figure: 8.4: QR-Update Boundary Cell

In this implementation, rather than compute θ_i to pass onto the other cells in the row, $\cos\theta_i$ and $\sin\theta_i$ are computed and passed on. The other cells must then perform 4 multiplies, 1 addition and 1 subtraction to complete a Givens rotation. (8.2) shows the relationship between the input vector $[a,b]$ and the calculated values $\cos\theta_i$ and $\sin\theta_i$.

$$\tan\theta_i = b/a$$

$$\cos\theta_i = \frac{1}{\sqrt{1+\tan^2\theta_i}} = \frac{1}{\sqrt{1+(b/a)^2}} \tag{8.2}$$

$$\sin\theta_i = \tan\theta_i \times \cos\theta_i = \frac{b/a}{\sqrt{1+(b/a)^2}}$$

8.3.2 Non-Pipelined Design

Figure 8.5 illustrates the non-pipelined boundary cell implemented using HDL Design Studio. The $\cos\theta_i$ and $\sin\theta_i$ components are computed using a combination of division, multiplication, addition and square root tokens in accordance with (8.2). The only register in the cell occurs in the feedback loop and is used to synchronise the feedback data with the next sample arriving. Note that the critical path is highlighted by dashed arrows and follows the path of the feedback loop.

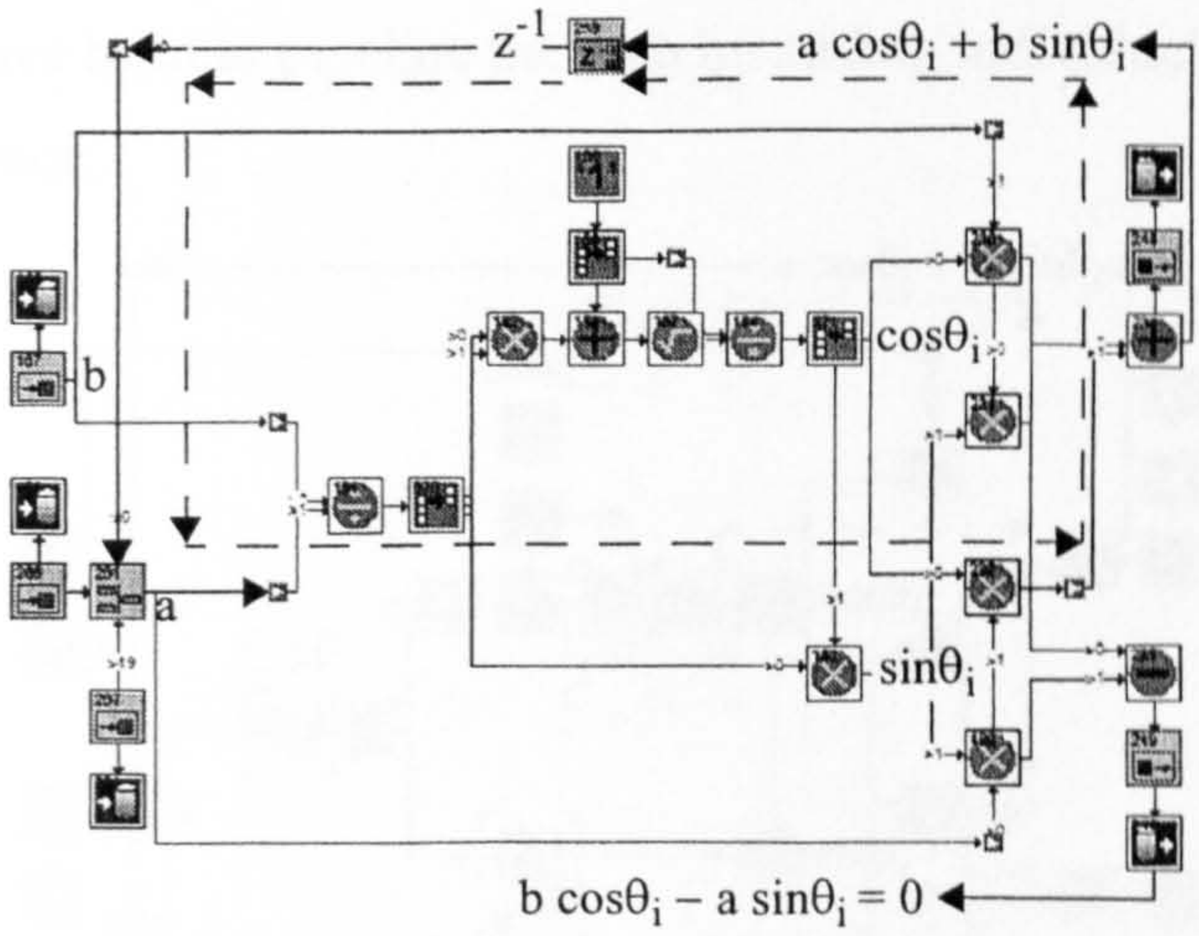


Figure: 8.5: Non-Pipelined Boundary Cell

8.3.3 Pipelined Design

The pipelined boundary cell is shown in Figure 8.6. Only the two dividers and the square root tokens are pipelined and combine to produce an overall delay of 86 samples. This delay is due to the fact that the divider and square root cores are fully

pipelined. Both dividers calculate signed 32 bit results. This requires 31 cells to generate the unsigned result. Looking at section 6.3.4 it can be seen that as well as having a delay in each cell, there is also 3 additional delays. Hence, a total of 34 delays is incurred in this case. The square rooter computes a signed 16 bit result although this only requires 15 cells. Just like the divider, there are 3 extra delays (section 6.7.5) in addition to the delay incurred by each cell. Hence, the square rooter incurs 18 delays. Thus, the pipelined stages are clocked at $(n = 86) \times f_s$. Clocking at this speed is required to synchronise the feedback result with the next sample arriving at the f_s data-rate thus maintaining the algorithm. The critical path in this design is also highlighted by dashed arrows. Note how much shorter it is compared to the non-pipelined design thus allowing the clock speed to be higher.

It should be noted that this is not the only pipelining option, there are many others that could have been employed. However, unfortunately the HDS square root and division cores only have the option to be fully pipelined or not at all. Another approach could have been to pipeline the path by adding individual delays in between each functional block.

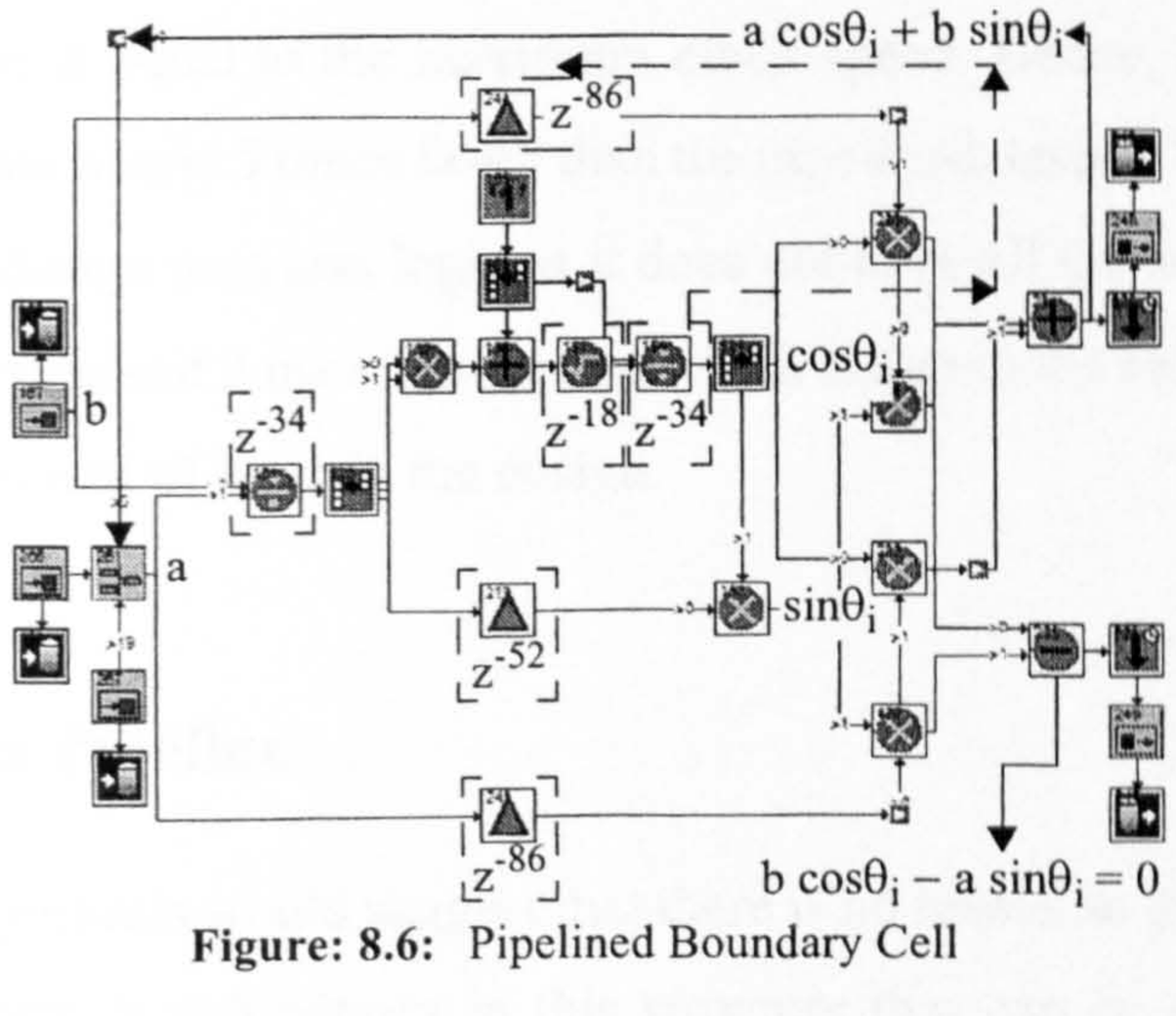


Figure: 8.6: Pipelined Boundary Cell

8.3.4 Synthesis Results

The results from synthesising both boundary cell designs are presented in Table 17. The results were obtained after completing the implementation process in ISE 6.2.03i which was used to target a Virtex-II XC2V8000 device. The power estimates were obtained using the XPower tool within ISE.

Design	Slices	Max. Clk Speed	Max. Data Rate	Total Est. Power Cons.
Pipelined	4672	45.55 MHz	530 KHz	902 mW
Non-Piped	2046	2.755 MHz	2.755 MHz	787 mW

Table 17: Synthesis Results

From these results it can be seen that the pipelined design can be clocked more than 16 times faster than the non-pipelined design. However, the maximum data rate for the pipelined design is one 86^{th} of the clock rate due to the fact that each data sample must wait 86 clock cycles before entering the design. This gives a maximum date rate of only 530 KHz. The non-pipelined design does not have this constraint and the maximum data rate is equal to the maximum clock speed. Hence, the non-pipelined design has a data rate nearly 5 times faster than the pipelined design. In addition to this, the non-pipelined design uses less logic as it does not have all the additional registers of the pipelined design and it uses less power, which is due to the reduced clock speed and the reduced amount of logic in the design.

8.4 Filling The Pipeline

The results from synthesis would suggest that there is no reason to pipeline a feedback loop. However, there is redundancy in this structure that can be exploited, making pipelining viable under certain conditions. The pipeline, in the situation considered so far, never fills up because a new sample must wait on the result of the previous one before it can enter the pipeline. Thus, for a single input data channel, a sample will

enter the pipeline and clock through each stage with nothing following it until it has passed through completely. This means that the majority of the logic is redundant for the majority of the time.

An approach that exploits this redundancy is to share the same structure with more than one input data channel (i.e channel interleaving as in Figure 8.7). Here, assume that the feedback loop has *n* pipeline stages. This means that up to *n* independent input channels can share this hardware. By multiplexing each input channel into the hardware, the pipeline can be filled. As soon as the first sample enters the pipeline and clears the first stage, a sample from another channel can enter.

This architecture offers a low cost hardware solution under the correct circumstances. Unless *n* input channels exist to fill *n* pipeline stages then there will still be some redundancy. Also, as *n* grows, the data-rate reduces, which means that this structure is only useful for low data-rate applications.

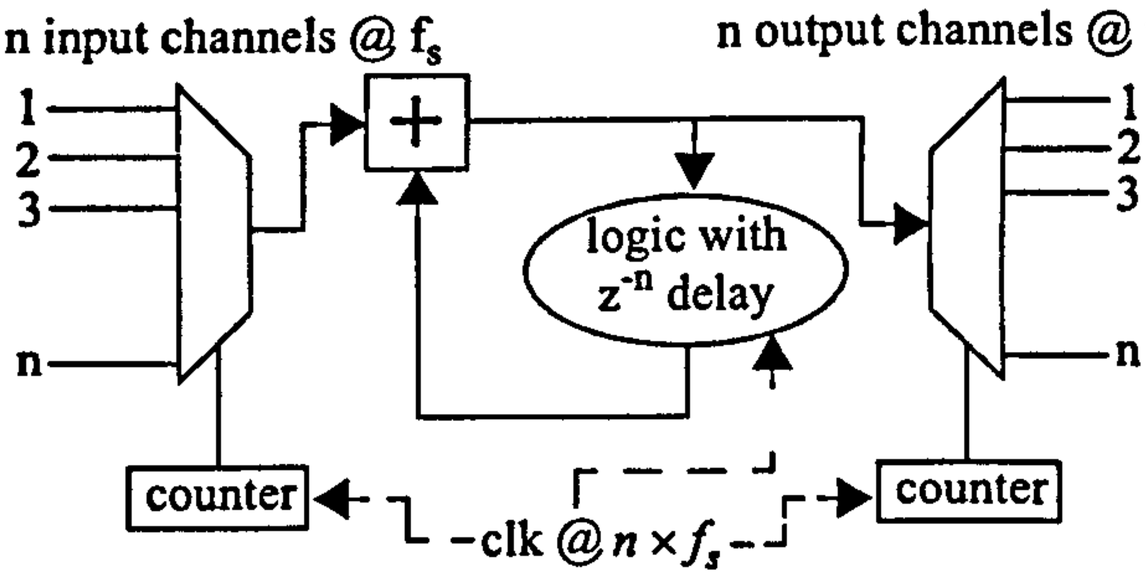


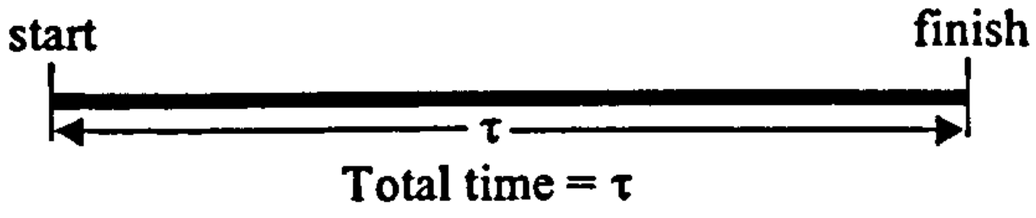
Figure: 8.7: Channel Interleaving

8.5 Discussion

The synthesis results have demonstrated that with a single data channel there is no reason to use pipelining in a feedback loop. The purpose of pipelining is to speed up data throughput, but clearly it has the opposite effect in this case. Not only is the pipelined version slower, but it uses more logic because of the extra registers and it consumes more power because of the extra clocking requirements.

These findings can be explained further by considering Figure 8.8. Here, a

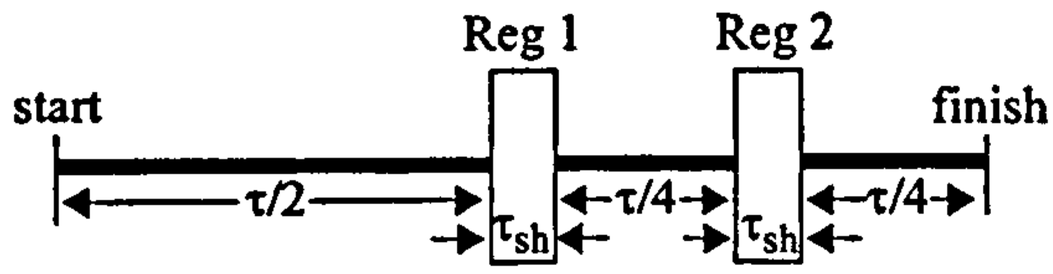
wire is shown, where τ represents the time it takes for a signal to travel from the start to the finish.



Total time = τ

Figure: 8.8: A simple wire

Figure 8.9 shows the same length of wire but this time it has been pipelined with two registers. The first register splits the wire in half so the time taken for the signal to travel from the start to Reg 1 is $\tau/2$. Reg 2 then splits the remainder of the wire in half, thus $\tau/4$ is the time taken for a signal to travel through each of the last two sections. So far, the time the signal takes to travel through the 3 sections of wire has been accounted for. However, there is also some additional delay that must be accounted for known as the setup and hold time. The setup time is the minimum amount of time that data must arrive at a register *before* the clock signal if it is to be successfully latched. Similarly, the hold time is the minimum amount of time that data must be held for *after* a clock signal has arrived. Thus, the minimum delay between data reaching a register and getting latched is the accumulation of the setup and hold time, known as τ_{sh}. If the setup or hold time is breached, then a situation known as metastability can occur where the latched value cannot be predicted.



Total time = $\tau/2 + 2\tau/4 + 2\tau_{sh} = \tau + 2\tau_{sh}$

Figure: 8.9: A pipelined wire

Accounting for τ_{sh} of both registers, the total time to travel along the wire becomes $\tau + 2\tau_{sh}$. Hence, it is clear that pipelining has only served to increase the travel time. This demonstrates that the shortest time for a signal to travel along a wire occurs without pipelining.

8.6 Conclusions

This chapter has shown that it is not always desirable to work with IP that has been pipelined. In situations involving a feedback loop it has been demonstrated that pipelined designs produce slower data throughput, use more logic and consume more power than a non-pipelined design. It has also shown that a non-pipelined design offers the fastest throughput possible and that a pipelined design cannot match this because of the additional delay due to the setup and hold time of each register in the pipeline.

A scenario that would benefit from using pipelined IP in a feedback loop was presented. This involved channel interleaving where the same pipelined feedback loop was shared with several data channels all running at the same data-rate. It was shown that for a pipeline with n stages, up to n independent data channels could be interleaved to share the same hardware and produce n independent output channels. This architecture offers a low cost hardware solution for low data-rate applications.

In summary, the conclusion that can be drawn from this work is that engineers using DSP algorithm design tools must fully understand the implications of working with pipelined IP blocks when implementing algorithms that require feedback.

Chapter 9

Conclusions

In this final chapter the work presented throughout the thesis is summarised and conclusions are made for each of the main research areas discussed within. Further to this, future research and development based on this work is proposed, particularly with regard to the CORDIC algorithm.

9.1 Core Development for HDS

HDL Design Studio (HDS) is a software package developed by EnTegra Ltd. to allow SystemVue simulations to be used to automatically generate equivalent bit and cycle accurate hardware designs. The advantage of this approach is that it significantly reduces design times by removing the need to hand code VHDL.

A major part of the EngD project was to develop two cores for inclusion in the HDS function library. These two cores were a Divider and a Square Rooter, both of which were based on direct methods of computation rather than an iterative approach. The advantage of using an algorithm based on a direct approach is that for a known number of iterations the accuracy of the output can be predetermined. This attribute is a major benefit in DSP. With an iterative technique such as Newton's method, the accuracy of the output for a given number of iterations is unknown unless lengthy simulations are run to track the worst case error for a particular scenario. Other

tools in the FPGA EDA tool market contain cores that will allow square roots and divisions to be computed. However, the cores tend to use CORDIC to compute these functions. It was shown within this thesis that the CORDIC algorithm is an iterative technique where determining the accuracy of the output it produces is not straight forward. Hence, for this reason and to differentiate HDS from its competitors a direct approach was taken for these functions.

Another reason for including Square Root and Division cores within HDS is that they are required to build Adaptive Equalisers using the QR-RLS update algorithm. This form of adaptive equalisation is in great demand at the moment due to its better performance relative to the LMS algorithm which has been traditionally used. However, increased performance comes at a cost and the QR-RLS uses far more resources than the LMS. This is why the LMS has been used so much until now as technology has limited the use of the RLS. Modern FPGAs now contain enough resources that this is no longer the case.

The importance of Square Root and Divide functionality is illustrated by the fact that one of Xilinx's latest FPGAs, the Virtex-4, contains logic known as the DSP48 slice which can be used to compute several math functions, two of which are Square Root and Divide [57]. Although there is no dedicated logic for these specific functions, it is the author's belief that this may well become a feature in future FPGAs, just like the embedded multiplier has become a regular feature since the early Virtex devices. Further to this, it is also clear that Xilinx see adaptive equalisation as a major area of interest to their customers as they recently acquired AccelChip [54] who have developed technology similar to SystemGenerator, but which includes IP for designing QR-RLS systems amongst other adaptive algorithms [39].

9.2 CORDIC Accuracy Research

The CORDIC algorithm has been shown to be a technique based on the rotation of a vector, which can be used to compute a large range of mathematical functions. It is

cheap to implement on FPGAs due to the fact that it requires mainly shifts and additions, which are in plentiful supply on todays devices. The problem with the CORDIC algorithm is that it is not easy to predict the accuracy of the output it produces. In many DSP algorithms it is vitally important to know the accuracy of each function in the algorithm to maintain numerical integrity. Obviously using CORDIC in such algorithms problematic.

The traditional approach to using the CORDIC algorithm in real systems has been to design the CORDIC component and then run lengthy simulations to test the output against some reference output. This allows the worst case error to be tracked thus yielding the accuracy of the CORDIC output. However, this approach takes a lot of time. If the desired level of accuracy is not found then the CORDIC design must be altered by adding more iterations and/or increasing the number of bits used in the data path. The simulations must then be rerun and the worst case error tracked again. This process is repeated until a CORDIC design with a satisfactory accuracy is found. Obviously this is not an ideal scenario as the process is lengthy and, importantly, the most efficient CORDIC design for a given level of accuracy may not be found even although the output is accurate enough.

To solve this problem, Yu Hen Hu [18] analysed the error in CORDIC systems computing vector magnitudes. By doing this he was able to develop a formula for the Overall Quantisation Error (OQE) in terms of the maximum magnitude of the rotated vector $|v(0)|$, the number of iterations n and the number of fractional bits in the data path b. These three parameters completely define the CORDIC system. To cut down the number of variables in the OQE, Yu Hen Hu proposed that the maximum value that $x(0)$ or $y(0)$ could take was 0.5, which meant that $|v(0)| \leq \sqrt{0.5}$. With this parameter fixed the OQE was computed for all combinations of $1 \leq n \leq 40$ and $1 \leq b \leq 40$. Finally each OQE was converted into the number of effective fractional bits that it represents and these values were entered into a table. This allowed someone wishing to design a CORDIC system computing a vector magnitude to scan the table and find the required level of accuracy for the calculation. Once found, the n and b

required to achieve this accuracy were simply read off the corresponding row and column respectively.

The OQE equation that was developed in [18] has since been found to be quite inaccurate for many cases. Hence, the author set out to build on and improve on this work. Consequently a new OQE was developed which has been shown to be far more accurate at predicting the accuracy of CORDIC systems computing vector magnitudes. The equation is used to generate a table, as before, which allows a desired level of accuracy to be searched for and the corresponding n and b are then given. Further to this, the work has been extended and an OQE equation has also been developed for CORDIC systems computing cosines and sines. This equation has also been proven to be extremely accurate in predicting the accuracy of such systems.

The development of accuracy tables has meant that the most efficient CORDIC design for a given level of accuracy can now be found quickly and easily without having to run lengthy simulations, as was the case with the traditional approach. This is a significant development for the CORDIC algorithm as it can now be assessed against other techniques for computing similar functions to evaluate which is best. This was not the case before as it was very difficult to find the most efficient design for a specified level of accuracy. Hence, a CORDIC design computing a vector magnitude with x fractional bits of accuracy could be found by trial and error but without a thorough search it would be very difficult to know whether or not the design used the very minimum hardware required to achieve this. Unless this is known with confidence, a fair comparison cannot be made with other techniques for computing the same function, such as a direct approach where the minimum hardware required is far easier to find. This type of examination has begun and for the case of vector magnitude calculations, it has been found that the CORDIC algorithm uses fewer resources.

Finally, this work can be extended in the future to include many more of the mathematical functions that CORDIC can compute. So far OQE equations for vector magnitude calculations and sine/cosine calculations have been developed but this is only the beginning. There are many more functions that can be computed, each of

which has its own specific OQE equation.

9.3 Adaptive Equalisation

Mobile communications require fast adaptive equalisers such as those that use the QR-RLS update algorithm. Such algorithms are extremely sensitive to numerical accuracy and hence it is vital that the error in each component is known with confidence. Further to this, such algorithms are also computationally intensive and are expensive in terms of the hardware required to implement them. Any research that leads towards minimising the hardware required in such systems is extremely valuable. Much of the work involved in this thesis can be related to the improvement of both of these areas of adaptive equalisation.

In the area of numerical accuracy, it has been shown that direct methods of computing division and square roots are ideal due to the fact that a solution with a guaranteed accuracy can be produced using a known number of iterations. This is not the case with the CORDIC algorithm, which can be used to compute vector magnitudes, a vital part of the QR-RLS algorithm. However, this thesis illustrates a new technique for finding the most efficient CORDIC design for such computations where the accuracy is known for a number of iterations. Thus, it is now possible that QR-RLS systems could be constructed using a variety of direct and CORDIC arithmetic. Where the optimal solution may lie is difficult to say although it is the RE's belief that a combination of CORDIC and direct arithmetic may be the answer, especially since a CORDIC system computing vector magnitudes has been shown to be smaller and faster than using two multipliers, an adder and a square rooter. Even if CORDIC was used to compute this part of the QR-RLS algorithm, multiplication and division are still required. These functions can be computed using CORDIC but it is not known yet whether this approach is better than direct arithmetic for these cases. Finally, it has been shown that the QR-RLS requires feedback within each of its cells. Often FPGA designs are pipelined to make use of the "free" registers throughout the

device with the benefit of increased throughput. However, it has been proven that this is not the best approach when feedback loops exist. The optimum solution occurs when no pipelining is used as this gives a design using the least resources yet with the highest data throughput.

Appendix A

Old Algorithm - Effective Fractional Bits Table

n / b	1	2	3	4	5	6	7	8	9	10
1	-1.39	-1.01	-0.78	-0.65	-0.57	-0.54	-0.52	-0.51	-0.50	-0.50
2	-1.24	-0.62	-0.16	0.13	0.30	0.40	0.45	0.47	0.49	0.49
3	-1.29	-0.48	0.19	0.70	1.05	1.26	1.37	1.43	1.47	1.48
4	-1.43	-0.52	0.31	1.02	1.58	1.97	2.21	2.35	2.42	2.46
5	-1.60	-0.64	0.28	1.13	1.87	2.47	2.89	3.17	3.32	3.41
6	-1.77	-0.79	0.17	1.10	1.97	2.74	3.37	3.82	4.12	4.30
7	-1.94	-0.95	0.04	1.01	1.94	2.83	3.62	4.27	4.76	5.08
8	-2.09	-1.09	-0.10	0.89	1.86	2.80	3.69	4.50	5.18	5.69
9	-2.22	-1.23	-0.23	0.76	1.75	2.72	3.67	4.57	5.40	6.09
10	-2.35	-1.35	-0.35	0.64	1.64	2.63	3.60	4.55	5.46	6.30
11	-2.47	-1.47	-0.47	0.53	1.53	2.52	3.51	4.49	5.44	6.36
12	-2.57	-1.57	-0.57	0.42	1.42	2.42	3.42	4.40	5.38	6.34
13	-2.67	-1.67	-0.67	0.33	1.32	2.32	3.32	4.32	5.31	6.29
14	-2.77	-1.77	-0.77	0.23	1.23	2.23	3.23	4.23	5.22	6.21
15	-2.86	-1.86	-0.86	0.14	1.14	2.14	3.14	4.14	5.14	6.13
16	-2.94	-1.94	-0.94	0.06	1.06	2.06	3.06	4.06	5.06	6.06
17	-3.02	-2.02	-1.02	-0.02	0.98	1.98	2.98	3.98	4.98	5.98
18	-3.09	-2.09	-1.09	-0.09	0.91	1.91	2.91	3.91	4.91	5.91
19	-3.16	-2.16	-1.16	-0.16	0.84	1.84	2.84	3.84	4.84	5.84
20	-3.23	-2.23	-1.23	-0.23	0.77	1.77	2.77	3.77	4.77	5.77
21	-3.30	-2.30	-1.30	-0.30	0.70	1.70	2.70	3.70	4.70	5.70
22	-3.36	-2.36	-1.36	-0.36	0.64	1.64	2.64	3.64	4.64	5.64
23	-3.42	-2.42	-1.42	-0.42	0.58	1.58	2.58	3.58	4.58	5.58
24	-3.47	-2.47	-1.47	-0.47	0.53	1.53	2.53	3.53	4.53	5.53
25	-3.53	-2.53	-1.53	-0.53	0.47	1.47	2.47	3.47	4.47	5.47
26	-3.58	-2.58	-1.58	-0.58	0.42	1.42	2.42	3.42	4.42	5.42
27	-3.63	-2.63	-1.63	-0.63	0.37	1.37	2.37	3.37	4.37	5.37
28	-3.68	-2.68	-1.68	-0.68	0.32	1.32	2.32	3.32	4.32	5.32

29	-3.73	-2.73	-1.73	-0.73	0.27	1.27	2.27	3.27	4.27	5.27
30	-3.77	-2.77	-1.77	-0.77	0.23	1.23	2.23	3.23	4.23	5.23
31	-3.82	-2.82	-1.82	-0.82	0.18	1.18	2.18	3.18	4.18	5.18
32	-3.86	-2.86	-1.86	-0.86	0.14	1.14	2.14	3.14	4.14	5.14
33	-3.90	-2.90	-1.90	-0.90	0.10	1.10	2.10	3.10	4.10	5.10
34	-3.94	-2.94	-1.94	-0.94	0.06	1.06	2.06	3.06	4.06	5.06
35	-3.98	-2.98	-1.98	-0.98	0.02	1.02	2.02	3.02	4.02	5.02
36	-4.02	-3.02	-2.02	-1.02	-0.02	0.98	1.98	2.98	3.98	4.98
37	-4.06	-3.06	-2.06	-1.06	-0.06	0.94	1.94	2.94	3.94	4.94
38	-4.10	-3.10	-2.10	-1.10	-0.10	0.90	1.90	2.90	3.90	4.90
39	-4.13	-3.13	-2.13	-1.13	-0.13	0.87	1.87	2.87	3.87	4.87
40	-4.17	-3.17	-2.17	-1.17	-0.17	0.83	1.83	2.83	3.83	4.83

n / b	11	12	13	14	15	16	17	18	19	20
1	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50
2	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
3	1.49	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50
4	2.48	2.49	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50
5	3.45	3.48	3.49	3.49	3.50	3.50	3.50	3.50	3.50	3.50
6	4.40	4.45	4.47	4.49	4.49	4.50	4.50	4.50	4.50	4.50
7	5.28	5.38	5.44	5.47	5.48	5.49	5.50	5.50	5.50	5.50
8	6.04	6.25	6.37	6.43	6.47	6.48	6.49	6.50	6.50	6.50
9	6.63	7.00	7.23	7.36	7.43	7.46	7.48	7.49	7.50	7.50
10	7.01	7.57	7.96	8.21	8.35	8.42	8.46	8.48	8.49	8.49
11	7.21	7.94	8.52	8.93	9.18	9.33	9.41	9.46	9.48	9.49
12	7.26	8.12	8.87	9.46	9.89	10.16	10.32	10.41	10.45	10.48
13	7.25	8.17	9.04	9.80	10.41	10.86	11.14	11.31	11.40	11.45
14	7.19	8.16	9.09	9.96	10.73	11.36	11.82	12.12	12.30	12.40
15	7.13	8.11	9.07	10.01	10.89	11.67	12.31	12.79	13.10	13.29
16	7.05	8.04	9.03	10.00	10.93	11.82	12.61	13.26	13.75	14.08
17	6.98	7.97	8.97	9.95	10.92	11.86	12.75	13.55	14.22	14.72
18	6.91	7.90	8.90	9.89	10.88	11.85	12.79	13.69	14.49	15.17
19	6.84	7.83	8.83	9.83	10.82	11.81	12.78	13.73	14.63	15.44
20	6.77	7.77	8.77	9.77	10.76	11.76	12.74	13.72	14.66	15.57
21	6.70	7.70	8.70	9.70	10.70	11.70	12.69	13.68	14.65	15.60
22	6.64	7.64	8.64	9.64	10.64	11.64	12.64	13.63	14.62	15.59
23	6.58	7.58	8.58	9.58	10.58	11.58	12.58	13.58	14.57	15.56
24	6.53	7.53	8.53	9.53	10.53	11.53	12.53	13.52	14.52	15.52
25	6.47	7.47	8.47	9.47	10.47	11.47	12.47	13.47	14.47	15.47
26	6.42	7.42	8.42	9.42	10.42	11.42	12.42	13.42	14.42	15.42
27	6.37	7.37	8.37	9.37	10.37	11.37	12.37	13.37	14.37	15.37
28	6.32	7.32	8.32	9.32	10.32	11.32	12.32	13.32	14.32	15.32
29	6.27	7.27	8.27	9.27	10.27	11.27	12.27	13.27	14.27	15.27
30	6.23	7.23	8.23	9.23	10.23	11.23	12.23	13.23	14.23	15.23
31	6.18	7.18	8.18	9.18	10.18	11.18	12.18	13.18	14.18	15.18
32	6.14	7.14	8.14	9.14	10.14	11.14	12.14	13.14	14.14	15.14
33	6.10	7.10	8.10	9.10	10.10	11.10	12.10	13.10	14.10	15.10
34	6.06	7.06	8.06	9.06	10.06	11.06	12.06	13.06	14.06	15.06
35	6.02	7.02	8.02	9.02	10.02	11.02	12.02	13.02	14.02	15.02
36	5.98	6.98	7.98	8.98	9.98	10.98	11.98	12.98	13.98	14.98
37	5.94	6.94	7.94	8.94	9.94	10.94	11.94	12.94	13.94	14.94
38	5.90	6.90	7.90	8.90	9.90	10.90	11.90	12.90	13.90	14.90
39	5.87	6.87	7.87	8.87	9.87	10.87	11.87	12.87	13.87	14.87
40	5.83	6.83	7.83	8.83	9.83	10.83	11.83	12.83	13.83	14.83

n / b	21	22	23	24	25	26	27	28	29	30
1	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50
2	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
3	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50
4	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50
5	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50
6	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50
7	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50
8	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50
9	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50
10	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50
11	9.49	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50
12	10.49	10.49	10.50	10.50	10.50	10.50	10.50	10.50	10.50	10.50
13	11.48	11.49	11.49	11.50	11.50	11.50	11.50	11.50	11.50	11.50
14	12.45	12.47	12.49	12.49	12.50	12.50	12.50	12.50	12.50	12.50
15	13.39	13.44	13.47	13.49	13.49	13.50	13.50	13.50	13.50	13.50
16	14.27	14.38	14.44	14.47	14.49	14.49	14.50	14.50	14.50	14.50
17	15.06	15.26	15.38	15.44	15.47	15.48	15.49	15.50	15.50	15.50
18	15.69	16.04	16.25	16.37	16.43	16.47	16.48	16.49	16.50	16.50
19	16.13	16.66	17.02	17.24	17.36	17.43	17.47	17.48	17.49	17.50
20	16.39	17.09	17.63	18.00	18.23	18.36	18.43	18.46	18.48	18.49
21	16.51	17.34	18.05	18.60	18.98	19.22	19.35	19.42	19.46	19.48
22	16.55	17.46	18.29	19.01	19.57	19.96	20.21	20.35	20.42	20.46
23	16.54	17.49	18.40	19.24	19.97	20.54	20.94	21.19	21.34	21.42
24	16.50	17.48	18.44	19.35	20.20	20.93	21.51	21.92	22.18	22.33
25	16.46	17.45	18.43	19.39	20.31	21.16	21.90	22.49	22.91	23.17
26	16.41	17.41	18.40	19.38	20.34	21.26	22.11	22.86	23.46	23.89
27	16.37	17.36	18.36	19.35	20.33	21.29	22.21	23.07	23.83	24.43
28	16.32	17.32	18.31	19.31	20.30	21.28	22.24	23.17	24.03	24.79
29	16.27	17.27	18.27	19.27	20.26	21.25	22.23	23.20	24.13	24.99
30	16.23	17.23	18.22	19.22	20.22	21.22	22.21	23.19	24.15	25.08
31	16.18	17.18	18.18	19.18	20.18	21.18	22.17	23.16	24.15	25.11
32	16.14	17.14	18.14	19.14	20.14	21.14	22.13	23.13	24.12	25.10
33	16.10	17.10	18.10	19.10	20.10	21.10	22.09	23.09	24.09	25.08
34	16.06	17.06	18.06	19.06	20.06	21.06	22.05	23.05	24.05	25.05
35	16.02	17.02	18.02	19.02	20.02	21.02	22.02	23.01	24.01	25.01
36	15.98	16.98	17.98	18.98	19.98	20.98	21.98	22.98	23.98	24.98
37	15.94	16.94	17.94	18.94	19.94	20.94	21.94	22.94	23.94	24.94
38	15.90	16.90	17.90	18.90	19.90	20.90	21.90	22.90	23.90	24.90
39	15.87	16.87	17.87	18.87	19.87	20.87	21.87	22.87	23.87	24.87
40	15.83	16.83	17.83	18.83	19.83	20.83	21.83	22.83	23.83	24.83

n / b	31	32	33	34	35	36	37	38	39	40
1	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50	-0.50
2	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
3	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50	1.50
4	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50	2.50
5	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50	3.50
6	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50	4.50
7	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50	5.50
8	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50	6.50
9	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50	7.50
10	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50	8.50
11	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50
12	10.50	10.50	10.50	10.50	10.50	10.50	10.50	10.50	10.50	10.50
13	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50
14	12.50	12.50	12.50	12.50	12.50	12.50	12.50	12.50	12.50	12.50
15	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50
16	14.50	14.50	14.50	14.50	14.50	14.50	14.50	14.50	14.50	14.50
17	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50
18	16.50	16.50	16.50	16.50	16.50	16.50	16.50	16.50	16.50	16.50
19	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50
20	18.50	18.50	18.50	18.50	18.50	18.50	18.50	18.50	18.50	18.50
21	19.49	19.50	19.50	19.50	19.50	19.50	19.50	19.50	19.50	19.50
22	20.48	20.49	20.50	20.50	20.50	20.50	20.50	20.50	20.50	20.50
23	21.46	21.48	21.49	21.50	21.50	21.50	21.50	21.50	21.50	21.50
24	22.41	22.46	22.48	22.49	22.49	22.50	22.50	22.50	22.50	22.50
25	23.33	23.41	23.46	23.48	23.49	23.49	23.50	23.50	23.50	23.50
26	24.16	24.32	24.41	24.45	24.48	24.49	24.49	24.50	24.50	24.50
27	24.87	25.15	25.32	25.40	25.45	25.48	25.49	25.49	25.50	25.50
28	25.41	25.85	26.14	26.31	26.40	26.45	26.48	26.49	26.49	26.50
29	25.76	26.38	26.84	27.13	27.30	27.40	27.45	27.47	27.49	27.49
30	25.96	26.73	27.36	27.82	28.12	28.30	28.40	28.45	28.47	28.49
31	26.04	26.92	27.70	28.33	28.80	29.11	29.29	29.39	29.45	29.47
32	26.07	27.00	27.88	28.66	29.31	29.78	30.10	30.29	30.39	30.44
33	26.06	27.03	27.97	28.85	29.63	30.28	30.77	31.09	31.28	31.39
34	26.04	27.02	27.99	28.93	29.81	30.60	31.26	31.75	32.08	32.27
35	26.01	27.00	27.98	28.95	29.89	30.78	31.58	32.24	32.74	33.07
36	25.97	26.97	27.96	28.95	29.92	30.86	31.75	32.55	33.22	33.72
37	25.94	26.94	27.93	28.92	29.91	30.88	31.82	32.71	33.52	34.19
38	25.90	26.90	27.90	28.90	29.89	30.87	31.84	32.79	33.68	34.49
39	25.87	26.87	27.87	28.86	29.86	30.85	31.84	32.81	33.76	34.65
40	25.83	26.83	27.83	28.83	29.83	30.83	31.82	32.80	33.78	34.72

Appendix B

New Algorithm - Effective Fractional Bits Table

n / b	1	2	3	4	5	6	7	8	9	10
1	0.30	0.97	1.48	1.82	2.03	2.15	2.21	2.24	2.26	2.26
2	0.15	1.03	1.83	2.49	2.98	3.31	3.51	3.62	3.68	3.71
3	-0.09	0.88	1.83	2.72	3.53	4.22	4.74	5.09	5.31	5.43
4	-0.34	0.65	1.64	2.62	3.57	4.48	5.31	6.03	6.59	6.98
5	-0.56	0.44	1.43	2.43	3.42	4.40	5.36	6.28	7.13	7.88
6	-0.76	0.24	1.24	2.24	3.24	4.23	5.23	6.21	7.17	8.10
7	-0.93	0.07	1.07	2.07	3.07	4.07	5.07	6.06	7.05	8.04
8	-1.08	-0.08	0.92	1.92	2.92	3.92	4.92	5.92	6.91	7.91
9	-1.22	-0.22	0.78	1.78	2.78	3.78	4.78	5.78	6.78	7.78
10	-1.35	-0.35	0.65	1.65	2.65	3.65	4.65	5.65	6.65	7.65
11	-1.47	-0.47	0.53	1.53	2.53	3.53	4.53	5.53	6.53	7.53
12	-1.57	-0.57	0.43	1.43	2.43	3.43	4.43	5.43	6.43	7.43
13	-1.67	-0.67	0.33	1.33	2.33	3.33	4.33	5.33	6.33	7.33
14	-1.77	-0.77	0.23	1.23	2.23	3.23	4.23	5.23	6.23	7.23
15	-1.86	-0.86	0.14	1.14	2.14	3.14	4.14	5.14	6.14	7.14
16	-1.94	-0.94	0.06	1.06	2.06	3.06	4.06	5.06	6.06	7.06
17	-2.02	-1.02	-0.02	0.98	1.98	2.98	3.98	4.98	5.98	6.98
18	-2.09	-1.09	-0.09	0.91	1.91	2.91	3.91	4.91	5.91	6.91
19	-2.16	-1.16	-0.16	0.84	1.84	2.84	3.84	4.84	5.84	6.84
20	-2.23	-1.23	-0.23	0.77	1.77	2.77	3.77	4.77	5.77	6.77
21	-2.30	-1.30	-0.30	0.70	1.70	2.70	3.70	4.70	5.70	6.70
22	-2.36	-1.36	-0.36	0.64	1.64	2.64	3.64	4.64	5.64	6.64
23	-2.42	-1.42	-0.42	0.58	1.58	2.58	3.58	4.58	5.58	6.58
24	-2.47	-1.47	-0.47	0.53	1.53	2.53	3.53	4.53	5.53	6.53
25	-2.53	-1.53	-0.53	0.47	1.47	2.47	3.47	4.47	5.47	6.47
26	-2.58	-1.58	-0.58	0.42	1.42	2.42	3.42	4.42	5.42	6.42
27	-2.63	-1.63	-0.63	0.37	1.37	2.37	3.37	4.37	5.37	6.37
28	-2.68	-1.68	-0.68	0.32	1.32	2.32	3.32	4.32	5.32	6.32

29	-2.73	-1.73	-0.73	0.27	1.27	2.27	3.27	4.27	5.27	6.27
30	-2.77	-1.77	-0.77	0.23	1.23	2.23	3.23	4.23	5.23	6.23
31	-2.82	-1.82	-0.82	0.18	1.18	2.18	3.18	4.18	5.18	6.18
32	-2.86	-1.86	-0.86	0.14	1.14	2.14	3.14	4.14	5.14	6.14
33	-2.90	-1.90	-0.90	0.10	1.10	2.10	3.10	4.10	5.10	6.10
34	-2.94	-1.94	-0.94	0.06	1.06	2.06	3.06	4.06	5.06	6.06
35	-2.98	-1.98	-0.98	0.02	1.02	2.02	3.02	4.02	5.02	6.02
36	-3.02	-2.02	-1.02	-0.02	0.98	1.98	2.98	3.98	4.98	5.98
37	-3.06	-2.06	-1.06	-0.06	0.94	1.94	2.94	3.94	4.94	5.94
38	-3.10	-2.10	-1.10	-0.10	0.90	1.90	2.90	3.90	4.90	5.90
39	-3.13	-2.13	-1.13	-0.13	0.87	1.87	2.87	3.87	4.87	5.87
40	-3.17	-2.17	-1.17	-0.17	0.83	1.83	2.83	3.83	4.83	5.83

n / b	11	12	13	14	15	16	17	18	19	20
1	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27
2	3.73	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74
3	5.50	5.53	5.55	5.56	5.56	5.56	5.56	5.57	5.57	5.57
4	7.22	7.36	7.44	7.48	7.50	7.51	7.51	7.51	7.52	7.52
5	8.47	8.90	9.17	9.33	9.41	9.46	9.48	9.49	9.50	9.50
6	8.97	9.74	10.37	10.83	11.12	11.30	11.40	11.45	11.47	11.49
7	9.01	9.94	10.83	11.62	12.27	12.76	13.08	13.28	13.38	13.44
8	8.90	9.89	10.86	11.80	12.69	13.50	14.18	14.69	15.04	15.25
9	8.77	9.77	10.76	11.75	12.72	13.67	14.57	15.40	16.09	16.63
10	8.65	9.65	10.65	11.65	12.64	13.63	14.60	15.55	16.46	17.30
11	8.53	9.53	10.53	11.53	12.53	13.53	14.52	15.51	16.49	17.45
12	8.43	9.43	10.43	11.43	12.43	13.43	14.42	15.42	16.42	17.41
13	8.33	9.33	10.33	11.33	12.33	13.33	14.33	15.32	16.32	17.32
14	8.23	9.23	10.23	11.23	12.23	13.23	14.23	15.23	16.23	17.23
15	8.14	9.14	10.14	11.14	12.14	13.14	14.14	15.14	16.14	17.14
16	8.06	9.06	10.06	11.06	12.06	13.06	14.06	15.06	16.06	17.06
17	7.98	8.98	9.98	10.98	11.98	12.98	13.98	14.98	15.98	16.98
18	7.91	8.91	9.91	10.91	11.91	12.91	13.91	14.91	15.91	16.91
19	7.84	8.84	9.84	10.84	11.84	12.84	13.84	14.84	15.84	16.84
20	7.77	8.77	9.77	10.77	11.77	12.77	13.77	14.77	15.77	16.77
21	7.70	8.70	9.70	10.70	11.70	12.70	13.70	14.70	15.70	16.70
22	7.64	8.64	9.64	10.64	11.64	12.64	13.64	14.64	15.64	16.64
23	7.58	8.58	9.58	10.58	11.58	12.58	13.58	14.58	15.58	16.58
24	7.53	8.53	9.53	10.53	11.53	12.53	13.53	14.53	15.53	16.53
25	7.47	8.47	9.47	10.47	11.47	12.47	13.47	14.47	15.47	16.47
26	7.42	8.42	9.42	10.42	11.42	12.42	13.42	14.42	15.42	16.42
27	7.37	8.37	9.37	10.37	11.37	12.37	13.37	14.37	15.37	16.37
28	7.32	8.32	9.32	10.32	11.32	12.32	13.32	14.32	15.32	16.32
29	7.27	8.27	9.27	10.27	11.27	12.27	13.27	14.27	15.27	16.27
30	7.23	8.23	9.23	10.23	11.23	12.23	13.23	14.23	15.23	16.23
31	7.18	8.18	9.18	10.18	11.18	12.18	13.18	14.18	15.18	16.18
32	7.14	8.14	9.14	10.14	11.14	12.14	13.14	14.14	15.14	16.14
33	7.10	8.10	9.10	10.10	11.10	12.10	13.10	14.10	15.10	16.10
34	7.06	8.06	9.06	10.06	11.06	12.06	13.06	14.06	15.06	16.06
35	7.02	8.02	9.02	10.02	11.02	12.02	13.02	14.02	15.02	16.02
36	6.98	7.98	8.98	9.98	10.98	11.98	12.98	13.98	14.98	15.98
37	6.94	7.94	8.94	9.94	10.94	11.94	12.94	13.94	14.94	15.94
38	6.90	7.90	8.90	9.90	10.90	11.90	12.90	13.90	14.90	15.90
39	6.87	7.87	8.87	9.87	10.87	11.87	12.87	13.87	14.87	15.87
40	6.83	7.83	8.83	9.83	10.83	11.83	12.83	13.83	14.83	15.83

n / b	21	22	23	24	25	26	27	28	29	30
1	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27
2	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74
3	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57
4	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52
5	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50
6	11.49	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50
7	13.47	13.49	13.49	13.50	13.50	13.50	13.50	13.50	13.50	13.50
8	15.37	15.43	15.47	15.48	15.49	15.50	15.50	15.50	15.50	15.50
9	17.00	17.23	17.36	17.43	17.46	17.48	17.49	17.50	17.50	17.50
10	18.01	18.57	18.96	19.21	19.35	19.42	19.46	19.48	19.49	19.50
11	18.36	19.21	19.94	20.52	20.93	21.19	21.33	21.42	21.46	21.48
12	18.38	19.34	20.26	21.12	21.87	22.46	22.89	23.16	23.32	23.41
13	18.32	19.31	20.29	21.25	22.17	23.04	23.80	24.41	24.86	25.14
14	18.23	19.23	20.22	21.21	22.20	23.16	24.09	24.96	25.73	26.36
15	18.14	19.14	20.14	21.14	22.14	23.13	24.11	25.08	26.01	26.89
16	18.06	19.06	20.06	21.06	22.06	23.06	24.05	25.04	26.03	27.00
17	17.98	18.98	19.98	20.98	21.98	22.98	23.98	24.98	25.97	26.97
18	17.91	18.91	19.91	20.91	21.91	22.91	23.91	24.91	25.91	26.90
19	17.84	18.84	19.84	20.84	21.84	22.84	23.84	24.84	25.84	26.84
20	17.77	18.77	19.77	20.77	21.77	22.77	23.77	24.77	25.77	26.77
21	17.70	18.70	19.70	20.70	21.70	22.70	23.70	24.70	25.70	26.70
22	17.64	18.64	19.64	20.64	21.64	22.64	23.64	24.64	25.64	26.64
23	17.58	18.58	19.58	20.58	21.58	22.58	23.58	24.58	25.58	26.58
24	17.53	18.53	19.53	20.53	21.53	22.53	23.53	24.53	25.53	26.53
25	17.47	18.47	19.47	20.47	21.47	22.47	23.47	24.47	25.47	26.47
26	17.42	18.42	19.42	20.42	21.42	22.42	23.42	24.42	25.42	26.42
27	17.37	18.37	19.37	20.37	21.37	22.37	23.37	24.37	25.37	26.37
28	17.32	18.32	19.32	20.32	21.32	22.32	23.32	24.32	25.32	26.32
29	17.27	18.27	19.27	20.27	21.27	22.27	23.27	24.27	25.27	26.27
30	17.23	18.23	19.23	20.23	21.23	22.23	23.23	24.23	25.23	26.23
31	17.18	18.18	19.18	20.18	21.18	22.18	23.18	24.18	25.18	26.18
32	17.14	18.14	19.14	20.14	21.14	22.14	23.14	24.14	25.14	26.14
33	17.10	18.10	19.10	20.10	21.10	22.10	23.10	24.10	25.10	26.10
34	17.06	18.06	19.06	20.06	21.06	22.06	23.06	24.06	25.06	26.06
35	17.02	18.02	19.02	20.02	21.02	22.02	23.02	24.02	25.02	26.02
36	16.98	17.98	18.98	19.98	20.98	21.98	22.98	23.98	24.98	25.98
37	16.94	17.94	18.94	19.94	20.94	21.94	22.94	23.94	24.94	25.94
38	16.90	17.90	18.90	19.90	20.90	21.90	22.90	23.90	24.90	25.90
39	16.87	17.87	18.87	19.87	20.87	21.87	22.87	23.87	24.87	25.87
40	16.83	17.83	18.83	19.83	20.83	21.83	22.83	23.83	24.83	25.83

n / b	31	32	33	34	35	36	37	38	39	40
1	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27	2.27
2	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74	3.74
3	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57	5.57
4	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52	7.52
5	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50	9.50
6	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50
7	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50	13.50
8	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50	15.50
9	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50	17.50
10	19.50	19.50	19.50	19.50	19.50	19.50	19.50	19.50	19.50	19.50
11	21.49	21.50	21.50	21.50	21.50	21.50	21.50	21.50	21.50	21.50
12	23.45	23.48	23.49	23.49	23.50	23.50	23.50	23.50	23.50	23.50
13	25.31	25.40	25.45	25.48	25.49	25.49	25.50	25.50	25.50	25.50
14	26.82	27.12	27.30	27.40	27.45	27.47	27.49	27.49	27.50	27.50
15	27.67	28.31	28.79	29.10	29.29	29.39	29.44	29.47	29.49	29.49
16	27.93	28.82	29.61	30.26	30.75	31.08	31.27	31.38	31.44	31.47
17	27.95	28.92	29.86	30.75	31.55	32.22	32.72	33.06	33.26	33.38
18	27.90	28.89	29.88	30.85	31.79	32.69	33.49	34.17	34.69	35.04
19	27.84	28.83	29.83	30.82	31.81	32.78	33.73	34.63	35.44	36.13
20	27.77	28.77	29.77	30.77	31.76	32.76	33.74	34.72	35.66	36.57
21	27.70	28.70	29.70	30.70	31.70	32.70	33.70	34.69	35.68	36.65
22	27.64	28.64	29.64	30.64	31.64	32.64	33.64	34.64	35.64	36.63
23	27.58	28.58	29.58	30.58	31.58	32.58	33.58	34.58	35.58	36.58
24	27.53	28.53	29.53	30.53	31.53	32.53	33.53	34.53	35.53	36.53
25	27.47	28.47	29.47	30.47	31.47	32.47	33.47	34.47	35.47	36.47
26	27.42	28.42	29.42	30.42	31.42	32.42	33.42	34.42	35.42	36.42
27	27.37	28.37	29.37	30.37	31.37	32.37	33.37	34.37	35.37	36.37
28	27.32	28.32	29.32	30.32	31.32	32.32	33.32	34.32	35.32	36.32
29	27.27	28.27	29.27	30.27	31.27	32.27	33.27	34.27	35.27	36.27
30	27.23	28.23	29.23	30.23	31.23	32.23	33.23	34.23	35.23	36.23
31	27.18	28.18	29.18	30.18	31.18	32.18	33.18	34.18	35.18	36.18
32	27.14	28.14	29.14	30.14	31.14	32.14	33.14	34.14	35.14	36.14
33	27.10	28.10	29.10	30.10	31.10	32.10	33.10	34.10	35.10	36.10
34	27.06	28.06	29.06	30.06	31.06	32.06	33.06	34.06	35.06	36.06
35	27.02	28.02	29.02	30.02	31.02	32.02	33.02	34.02	35.02	36.02
36	26.98	27.98	28.98	29.98	30.98	31.98	32.98	33.98	34.98	35.98
37	26.94	27.94	28.94	29.94	30.94	31.94	32.94	33.94	34.94	35.94
38	26.90	27.90	28.90	29.90	30.90	31.90	32.90	33.90	34.90	35.90
39	26.87	27.87	28.87	29.87	30.87	31.87	32.87	33.87	34.87	35.87
40	26.83	27.83	28.83	29.83	30.83	31.83	32.83	33.83	34.83	35.83

Appendix C

CORDIC Cos/Sin - Effective Fractional Bits Table

n / b	1	2	3	4	5	6	7	8	9	10
1	-0.39	-0.01	0.22	0.35	0.43	0.46	0.48	0.49	0.50	0.50
2	-0.35	0.22	0.61	0.86	1.00	1.08	1.12	1.14	1.15	1.16
3	-	0.39	0.99	1.42	1.70	1.86	1.95	2.00	2.02	2.03
4	-	-	1.19	1.83	2.30	2.61	2.80	2.90	2.96	2.98
5	-	-	-	2.02	2.69	3.20	3.55	3.76	3.88	3.94
6	-	-	-	-	2.87	3.57	4.12	4.49	4.72	4.86
7	-	-	-	-	-	3.73	4.46	5.04	5.44	5.69
8	-	-	-	-	-	-	4.61	5.36	5.96	6.39
9	-	-	-	-	-	-	-	5.50	6.26	6.88
10	-	-	-	-	-	-	-	-	6.39	7.17
11	-	-	-	-	-	-	-	-	-	7.29

n / b	11	12	13	14	15	16	17	18	19	20
1	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
2	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16
3	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04
4	3.00	3.00	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01
5	3.97	3.99	3.99	4.00	4.00	4.00	4.00	4.00	4.00	4.00
6	4.93	4.96	4.98	4.99	5.00	5.00	5.00	5.00	5.00	5.00
7	5.84	5.92	5.96	5.98	5.99	5.99	6.00	6.00	6.00	6.00
8	6.66	6.82	6.91	6.95	6.98	6.99	6.99	7.00	7.00	7.00
9	7.34	7.63	7.80	7.90	7.95	7.97	7.99	7.99	8.00	8.00
10	7.81	8.29	8.60	8.79	8.89	8.94	8.97	8.99	8.99	9.00
11	8.09	8.75	9.24	9.57	9.77	9.88	9.94	9.97	9.98	9.99
12	8.20	9.01	9.68	10.20	10.54	10.75	10.87	10.93	10.97	10.98
13	-	9.12	9.93	10.62	11.15	11.52	11.74	11.86	11.93	11.96
14	-	-	10.03	10.86	11.57	12.11	12.49	12.72	12.85	12.93

15	-	-	-	10.96	11.79	12.51	13.07	13.46	13.71	13.85
16	-	-	-	-	11.88	12.73	13.46	14.03	14.44	14.69
17	-	-	-	-	-	12.81	13.66	14.40	14.99	15.41
18	-	-	-	-	-	-	13.75	14.60	15.35	15.95
19	-	-	-	-	-	-	-	14.68	15.55	16.30
20	-	-	-	-	-	-	-	-	15.62	16.49
21	-	-	-	-	-	-	-	-	-	16.57

n / b	21	22	23	24	25	26	27	28	29	30
1	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
2	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16
3	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04
4	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01
5	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
6	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
7	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
8	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00
9	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00
10	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
11	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
12	10.99	11.00	11.00	11.00	11.00	11.00	11.00	11.00	11.00	11.00
13	11.98	11.99	12.00	12.00	12.00	12.00	12.00	12.00	12.00	12.00
14	12.96	12.98	12.99	13.00	13.00	13.00	13.00	13.00	13.00	13.00
15	13.92	13.96	13.98	13.99	14.00	14.00	14.00	14.00	14.00	14.00
16	14.84	14.92	14.96	14.98	14.99	15.00	15.00	15.00	15.00	15.00
17	15.67	15.83	15.91	15.96	15.98	15.99	15.99	16.00	16.00	16.00
18	16.38	16.66	16.82	16.91	16.95	16.98	16.99	16.99	17.00	17.00
19	16.92	17.36	17.64	17.81	17.90	17.95	17.98	17.99	17.99	18.00
20	17.26	17.88	18.33	18.63	18.80	18.90	18.95	18.97	18.99	18.99
21	17.44	18.21	18.85	19.31	19.61	19.79	19.89	19.95	19.97	19.99
22	17.51	18.39	19.17	19.81	20.29	20.60	20.79	20.89	20.94	20.97
23	-	18.45	19.34	20.12	20.78	21.26	21.59	21.78	21.88	21.94
24	-	-	19.40	20.29	21.08	21.74	22.24	22.57	22.77	22.88
25	-	-	-	20.35	21.24	22.04	22.71	23.22	23.56	23.76
26	-	-	-	-	21.30	22.20	23.00	23.68	24.19	24.54
27	-	-	-	-	-	22.26	23.15	23.97	24.65	25.17
28	-	-	-	-	-	-	23.21	24.11	24.93	25.62
29	-	-	-	-	-	-	-	24.17	25.07	25.89
30	-	-	-	-	-	-	-	-	25.12	26.03
31	-	-	-	-	-	-	-	-	-	26.08

n / b	31	32	33	34	35	36	37	38	39	40
1	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
2	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16	1.16
3	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04	2.04
4	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01	3.01
5	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
6	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
7	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
8	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00
9	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00
10	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
11	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
12	11.00	11.00	11.00	11.00	11.00	11.00	11.00	11.00	11.00	11.00
13	12.00	12.00	12.00	12.00	12.00	12.00	12.00	12.00	12.00	12.00
14	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00	13.00
15	14.00	14.00	14.00	14.00	14.00	14.00	14.00	14.00	14.00	14.00
16	15.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00
17	16.00	16.00	16.00	16.00	16.00	16.00	16.00	16.00	16.00	16.00
18	17.00	17.00	17.00	17.00	17.00	17.00	17.00	17.00	17.00	17.00
19	18.00	18.00	18.00	18.00	18.00	18.00	18.00	18.00	18.00	18.00
20	19.00	19.00	19.00	19.00	19.00	19.00	19.00	19.00	19.00	19.00
21	19.99	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00	20.00
22	20.99	20.99	21.00	21.00	21.00	21.00	21.00	21.00	21.00	21.00
23	21.97	21.99	21.99	22.00	22.00	22.00	22.00	22.00	22.00	22.00
24	22.94	22.97	22.98	22.99	23.00	23.00	23.00	23.00	23.00	23.00
25	23.88	23.94	23.97	23.98	23.99	24.00	24.00	24.00	24.00	24.00
26	24.75	24.87	24.93	24.97	24.98	24.99	25.00	25.00	25.00	25.00
27	25.53	25.75	25.87	25.93	25.97	25.98	25.99	26.00	26.00	26.00
28	26.15	26.51	26.74	26.86	26.93	26.96	26.98	26.99	27.00	27.00
29	26.59	27.13	27.50	27.73	27.86	27.93	27.96	27.98	27.99	28.00
30	26.86	27.56	28.11	28.49	28.72	28.85	28.93	28.96	28.98	28.99
31	26.99	27.82	28.53	29.09	29.47	29.71	29.85	29.92	29.96	29.98
32	27.04	27.95	28.79	29.51	30.07	30.46	30.71	30.85	30.92	30.96
33	-	28.00	28.92	29.75	30.48	31.05	31.45	31.70	31.84	31.92
34	-	-	28.96	29.88	30.72	31.45	32.03	32.43	32.69	32.84
35	-	-	-	29.93	30.84	31.69	32.43	33.01	33.42	33.68
36	-	-	-	-	30.89	31.81	32.66	33.40	33.99	34.41
37	-	-	-	-	-	31.86	32.78	33.63	34.37	34.97
38	-	-	-	-	-	-	32.82	33.74	34.60	35.35
39	-	-	-	-	-	-	-	33.79	34.71	35.57
40	-	-	-	-	-	-	-	-	34.75	35.68

References

[1] A.H. Abdullah, M.I. Yusof and S.R.M. Baki, "Adaptive noise cancellation: a practical study of the least-mean square (LMS) over recursive least-square (RLS) algorithm," *Student Conference on Research and Development 2002*, pp. 448-452, 16-17 July 2002.

[2] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field Programmable Gate Arrays*, pp. 191-200, Feb. 22-24, 1998.

[3] J.L. Barlow and I.C.F. Ipsen, "Scaled Givens rotations for the solution of linear least squares problems on systolic arrays", *SIAM Journal on Scientific and Statistical Computing*, 8: pp. 716-734, 1987.

[4] N. Bellas, S.M. Chai, M. Dwyer and D. Linzmeier, "FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators", *20th International Parallel and Distributed Processing Symposium 2006*, 25-29 April 2006.

[5] N. Bray, "Designing for the IP Supermarket," *Fall VIUF Workshop 1999*, pp. 8-13, 4-6 Oct. 1999.

[6] J. Canaris, "A High Speed Fixed Point Binary Divider", *In Proc. ICASSP-89*, pp. 2393-2396, Glasgow, Scotland, May 1989.

[7] T.A.C.M. Claasen, "An industry perspective on current and future state of the art in system-on-chip (SoC) technology," *Proc. of the IEEE*, pp. 1121 - 1137, Volume 94, Issue 6, June 2006.

[8] S. Dhanani, "FPGAs Enabling Consumer Electronics – A Growing Trend", *FPGA and Structured ASIC Journal*, http://www.fpgajournal.com/articles_2005/20050614_xilinx.htm.

[9] D. Denning, N. Harold, M. Devlin, J. Irvine, "Using System Generator To Design A Reconfigurable Video Encryption System", *Proc. 13th FPL Conference*, Sep. 1-3, 2003, Lecture Notes in Computer Science, Springer, Vol 2778, pp 980-983.

[10] M.D. Ercegovac, L. Imbert, D.W. Matula, J.-M. Muller and G. Wei, "Improving Goldschmidt division, square root, and square root reciprocal", *IEEE Transactions on Computers*, Vol. 49, Issue 7, pp. 759 - 763, July 2000.

[11] G. Even, P.-M. Seidel and W.E. Ferguson, "A Parametric Error Analysis of Goldschmidt's Division Algorithm," *Proc. 16th IEEE Symposium on Computer Arithmetic, 2003"*, pp. 165-171, June 2003.

[12] W.M. Gentleman, "Least Squares computations by Givens transformations without square-roots," *J. Inst. Math. Its Appl.*, vol. 12, pp. 329-336.

[13] W.M. Gentleman and H.T. Kung, "Matrix triangularization by systolic arrays", in *Proc. SPIE*, vol. 298, *Real Time Signal Processing* IV, pp. 298-303, 1981.

[14] S. Hammarling, "A note on modifications to the Givens plane rotation", *J. Inst. Maths Applics*, 13: pp. 215–218, 1974.

[15] S. Haykin. "Adaptive Filter Theory", 2nd ed., Prentice Hall, 1996.

[16] J. Henkel, "Closing The SOC Design Gap", *in Computer*, Vol 36, Issue 9, p119-121, Sept 2003.

[17] R.A. Horn and C.R. Johnson, "Matrix Analysis", Section 2.6, Cambridge University Press, 1985.

[18] Y. H. Hu, "The Quantization Effects of the CORDIC Algorithm", in IEEE Trans. On Signal Processing, Vol 40, No 4, pp. 834-844, 1992.

[19] Y. H. Hu, "CORDIC-based VLSI Architectures for Digital Signal Processing", *IEEE Signal Processing Magazine*, Vol. 9, Issue 3, July 1992, pp. 16-35.

[20] K. Hwang, "Computer Arithmetic: Principles, Architecture, And Design", John Wiley & Sons, 1979.

[21] K. Kota and J. R. Cavallaro, "Numerical Accuracy and Hardware Tradeoffs for CORDIC Arithmetic for Special-Purpose Processors", *IEEE Trans. Computers*, Vol. 42, No. 7, pp. 769-779, July 1993.

[22] W. P. Marnane, S. P. Bellis and P. Larsson-Edefors, "Bit Serial Interleaved High Speed Division", *Electronic Letters*, Vol. 33, No. 13, pp. 1124-1125, 1997.

[23] J.G. McWhirter, "Recursive least-squares minimization using systolic arrays", *Proc. SPIE, Real Time Signal Processing VI*, vol. 431, San Diego, pp. 105-112, 1983.

[24] J.G. McWhirter, R.L. Walke and J. Kadlec, "Normalised Givens rotations for recursive least squares processing", *IEEE Signal Processing Workshop on VLSI Signal Processing, VIII, 1995*, pp. 323-332, 16-18 Sept. 1995.

[25] G. Moore, "Cramming More Components Onto Integrated Circuits", *in Electronics Magazine*, Vol 38, No 8, 19 April 1965.

[26] D. Phanthavong, "Desigining With DSP48 Blocks Using Precision Synthesis", *Xilinx Xcell Journal*, 3rd Quarter 2005. http://www.xilinx.com/publications/xcellonline/xcell_54/xc_pdf/xc_dsp48-54.pdf

[27] J. Pihl, E.J Aas, "A multiplier and squarer generator for high performance DSP applications", *IEEE 39th Midwest symposium on Circuits and Systems*, 1996.

[28] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, Vol. EC-7, pp. 218-222, Sept. 1958.

[29] R. Sanz, P. Corral and A.C. de Castro Lima, "Adaptive beamforming techniques for OFDM based WLAN systems: a comparison between RLS and LMS," *Joint IST Workshop on Mobile Future 2004, SympoTIC '04*, pp. 29-32, 24-26 Oct. 2004.

[30] B. Sklar. *Digital Communications : Fundamenatls and Applications*, 2nd Edition, Prentice-Hall, ISBN 0-13-084788-7

[31] R.W. Stewart, R. Chapman, T.S. Durrani. "The Square Root in Signal Processing", *SPIE Real Time Signal Processing XII*, San Diego, USA, August 1989.

[32] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quart. J. Mech. Appl. Math.*, Vol. 11, pt. 3, pp. 364-384,1958.

[33] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans Electronic Computing*, Vol EC-8, pp330-334, Sept 1959.

[34] J.S. Walther, "A unified algorithm for elementary functions," *Spring Joint Computer Conf. proc.*, pp379-385, 1971

[35] Liang-Kai Wang and M. J. Schulte, "Decimal floating-point division using Newton-Raphson iteration," *Proc. 15th IEEE Conf. on Application-Specific Systems, Architectures and Processors*, pp. 84-95, 2004.

[36] C. R. Ward, P. J. Hargrave and J. G. McWhirter, "A Novel Algorithm and Architecture for Adaptive Digital Beamforming", *IEEE Trans. Antennas and Propagation*, Vol. AP-34, No. 3, 1986, pp. 338-346..

[37] B. Widrow and M. E. Hoff, Jr., "Adaptive switching circuits," in *IRE WESCONConv. Rec.*, pt. 4, 1960, pp. 96-104.

[38] B. Widrow et. al. "Adaptive Noise Cancelling: Principles and Applications", *Proc. IEEE, Vol. 53, No. 12,* December 1975.

[39] AccelWare DSP Intellectual Property (IP)
http://www.accelchip.com/accelware_dsp_block_libraries.html

[40] Altera DSP Builder Home Page:
http://www.altera.com/products/software/products/dsp/dsp-builder.html

[41] Altera Stratix II Architecture
http://www.altera.com/literature/hb/stx2/stx2_sii51002.pdf

[42] EnTegra Simulation Webpage:
http://www.entegra.co.uk/simulation.htm

[43] The Handel-C Reference Manual:
http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF

[44] HDL Design Studio information pages:
http://www.steepestascent.com

[45] Intersil Programmable Downconverter HSP50214B Data Sheet:
http://www.intersil.com/data/fn/fn4450.pdf

[46] LatticeXP Family Data Sheet
http://www.latticesemi.com/lit/docs/datasheets/fpga/DS1001.pdf

[47] Simulink Webpage:
http://www.mathworks.com/products/simulink/

[48] SpecC information pages:
http://www.ics.uci.edu/~specc/

[49] Steepest Ascent home page:
http://www.steepestascent.com/content/

[50] SystemVue information pages:
http://www.eagleware.com/products/genesys/listing.html#systemview

[51] SystemC Language Reference Manual:
http://www.systemc.org/web/sitedocs/lrm_2_1.html

[52] Xilinx CORDIC Core Data Sheet:
http://www.xilinx.com/bvdocs/ipcenter/data_sheet/cordic.pdf

[53] Xilinx Multiplier Core Data Sheet:
http://www.xilinx.com/bvdocs/ipcenter/data_sheet/mult_gen.pdf

[54] Xilinx Press Release - XILINX ACQUIRES DSP DESIGN TOOL LEADER ACCELCHIP
http://www.xilinx.com/prs_rls/xil_corp/0613xlnx_accelchip.htm

[55] Xilinx System Generator Home Page:
http://www.xilinx.com/ise/optional_prod/system_generator.htm
[56] Xilinx Virtex-II Pro Data Sheet:
http://direct.xilinx.com/bvdocs/publications/ds083.pdff
[57] Xilinx XtremeDSP for Virtex-4 FPGAs User Guide
http://www.xilinx.com/bvdocs/userguides/ug073.pdf