No access before Apr 98

High Sample-Rate Givens Rotations for Recursive Least Squares

Richard Lewis Walke

A thesis submitted to The University of Warwick for the degree of

DOCTOR OF PHILOSOPHY

Faculty of Science

Department of Computer Science

July 1997

To Valerie

Summary

The design of an application-specific integrated circuit of a parallel array processor is considered for recursive least squares by QR decomposition using Givens rotations, applicable in adaptive filtering and beamforming applications. Emphasis is on high sample-rate operation, which, for this recursive algorithm, means that the time to perform arithmetic operations is critical. The algorithm, architecture and arithmetic are considered in a single integrated design procedure to achieve optimum results.

A realisation approach using standard arithmetic operators, add, multiply and divide is adopted. The design of high-throughput operators with low delay is addressed for fixed- and floating-point number formats, and the application of redundant arithmetic considered. New redundant multiplier architectures are presented enabling reductions in area of up to 25%, whilst maintaining low delay. A technique is presented enabling the use of a conventional tree multiplier in recursive applications, allowing savings in area and delay. Two new divider architectures are presented showing benefits compared with the radix-2 modified SRT algorithm.

Givens rotation algorithms are examined to determine their suitability for VLSI implementation. A novel algorithm, based on the Squared Givens Rotation (SGR) algorithm, is developed enabling the sample-rate to be increased by a factor of approximately 6 and offering area reductions up to a factor of 2 over previous approaches. An estimated sample-rate of 136 MHz could be achieved using a standard cell approach and 0.35μm CMOS technology.

The enhanced SGR algorithm has been compared with a CORDIC approach and shown to benefit by a factor of 3 in area and over 11 in sample-rate. When compared with a recent implementation on a parallel array of general purpose (GP) DSP chips, it is estimated that a single application specific chip could offer up to 1,500 times the computation obtained from a single GP DSP chip.

Contents

List of Figures

List of Tables

Acknowledgements

I have been fortunate to receive guidance and supervision from a number of knowledgeable and generous people. In particular, I thank my supervisors Prof. John McWhirter and Prof. Graham Nudd for their encouragement and help. I also thank Dr. Richard Evans for his encouragement, the helpful discussions on arithmetic, his comments on this thesis, and enabling the supporting research to be performed. I am very grateful to Dr. Ian Proudler for his explanations and guidance on the adaptive filtering and numerical simulation aspects of this research.

The results and conclusions of this thesis have relied on circuit synthesis results produced by Integrated Silicon Systems Ltd. An enormous effort has been made by Dr. Yi Hu to code and synthesise these circuits which has been far and beyond the call of duty. I am very grateful for his efforts.

I would also like to thank Prof. John McCanny for allowing me to use the VLSI design tools in August 1995 at The Queen's University of Belfast to produce circuit layout.

My thanks goes to Dr. Robert Hamill for the CORDIC results, Dr. David Trainor for the use of the IRIS tools and help with the other VLSI design tools, and Simon Knowles for discussions on arithmetic and for the information on the power consumption of circuits. My thanks to Dr. Neil Burgess for his helpful comments on the thesis.

This work has been funded by the Ministry of Defence, through the Defence Evaluation and Research Agency, and I thank those who have enabled this to happen.

I thank Rev. Frank Holt for proof reading the thesis.

Lastly, I would like to thank my wife Valerie for being patient and supportive during this challenging period.

Declaration

Unless otherwise stated, the work in this thesis is the author's own original research work.

Some of the work has been published previously:

R. L. Walke and R. A. Evans, "A Minimally Redundant Radix-4 Systolic Array for High Performance IIR Filtering", VLSI Signal Processing, VI, ISBN 0-7803-0996-0, pp. 168-178, 1993.

J. G. McWhirter, R. L. Walke and J. Kadlec, "Normalised Givens Rotations for Recursive Least Squares Processing", VLSI Signal Processing, VIII, ISBN 0-7803-2612-1, pp. 323-332, 1995.

Abbreviations

lsb	least significant bit
lsd	least significant digit
msb	most significant bit
msd	most significant digit
msdf	most significant digit first
ASIC	Application specific integrated circuit
CAD	Computer aided design
CLA	Carry look ahead (adder)
CMOS	Complementary metal oxide semiconductor
CORDIC	CoOrdinate Rotation by DIgital Computer
CSA	Carry save adder
DFE	Decision feedback equaliser
DG	Dependence graph
DSF	Divide-and-square-root-free (Givens rotation algorithm)
DSP	Digital signal processing
GP DSP	General purpose digital signal processor
FFT	Fast fourier transform
FIR	Finite impulse response (filter)
FPU	Floating-point unit
IEEE	Institute of Electronic and Electrical Engineers
IIR	Infinite impulse response (filter)
LPGS	Locally parallel globally sequential
LSGP	Locally sequential globally parallel
LUT	Look-up table
MFLOPS	Million floating-point operations per second

MinR4	Minimally redundant radix-4
MaxR4	Maximally redundant radix-4
RAM	Random access memory
QR-Algorithm	QR decomposition by Givens rotations
QR-Array	Triangular array of processors for performing the QR-algorithm
SFG	Signal flow graph
SGR	Squared Givens rotation
SNDR	Signed digit number representation
SNR	Signal-to-noise ratio
SQF-XFB	Square-root-free X-feedback (Givens rotation algorithm)
SRT	Sweeney, Robertson and Tocher (divider algorithm)
VHDL	VHSIC hardware description language
VLSI	Very large scale integration

Chapter 1 Introduction

1.1 Background

The rapid evolution of integrated circuit technology into the realms of *very large scale integration* (VLSI) has enabled the realisation of the complex circuit structures necessary to process real-time signals in a digital manner. Processing a signal in this way offers considerable advantages over the real-world analogue representation, as it simplifies the storage of signals and allows functions to be performed on them that would otherwise be impractical. Consequently, the application of *digital signal processing* (DSP) has grown, and become an important enabling technology in many new applications of digital electronics. For example, the development of digital audio, multimedia and mobile communications have relied heavily on DSP.

There is now a wide variety of programmable general purpose digital signal processors commercially available with which to implement DSP. These offer a rapid and low cost development route, but currently only achieve computation rates of the order of 50 million floating-point operations per second (MFLOPS) as only one arithmetic unit is generally integrated. Also, for some simple functions they are inefficient, and use large amounts of silicon area and power as a consequence of the generality of their architecture and the overheads of programmability. In some instances, as little as 5% of the chip area may be occupied in useful operations on a signal.

More efficient use of the silicon resource can be achieved using an *application specific integrated circuit* (ASIC), as the processor architecture and arithmetic unit may be tailored to the specific requirements of the algorithm. In this way, significant reductions in area and power are possible, which enables either die sizes to be reduced, or higher levels of computation to be achieved by integrating more arithmetic units on a chip.

When adopting an ASIC approach, the optimum design is achieved in a process of developing the algorithm, the processor architecture and arithmetic in a way which accommodates the benefits and limitations of VLSI technology. Parallelism, regularity and local interconnect are features that are sought to achieve high levels of computation from circuits which are easy to realise.

Parallelism is perhaps the most important feature of any algorithm, as it allows results to be produced at a greater rate by using more than one arithmetic unit simultaneously to process a signal. With current technology it is possible to integrate large numbers of arithmetic units on a single chip and so achieve computation rates in excess of 10,000 MFLOPS. If the algorithm and architecture are both parallel and scalable, then it is possible to apply the increasing level of circuit integration to improve the rate at which signals can be processed (i.e. the *sample-rate*).

Recursive algorithms are very restrictive, as they have a sequential dependence between operations, as by their definition they use previous outputs to generate the next. In this case, the maximum sample-rate achievable is governed by the time it takes to perform the sequence of operations. In which case, improvements can only be made by either transforming the algorithm to reduce the number of operations, or implementing the operators in a way which reduces their delay, referred to as *latency*.

Most-significant-digit-first (msdf) arithmetic is a technique which has been used to minimise the latency of an operator, and has been applied in the design of a multiplier-adder for a high-sample-rate infinite impulse response (IIR) filter. However, this is a relatively simple application, and so the general applicability of this technique has not yet been fully established.

1.2 Objective of Research

The objective of this research has been to examine the VLSI implementation of an array processor for adaptive filtering. Currently, ASIC solutions employing parallel architectures are highly relevant due to the enormous levels of computation required by applications such as radar, sonar and communications. For example, radar applications can require computation levels of 50,000 MFLOPS. One adaptive algorithm which is of particular interest is *recursive least squares*, as this requires only low wordlength arithmetic when implemented using *QR decomposition* by *Givens rotations*, and has a parallel architecture for high-throughput implementations. However, this algorithm, which henceforth shall be referred to as the *QR-algorithm*, is recursive and this can limit sample-rate at which an implementation may operate. In certain radar and communications applications the sample-rate can be very high (in excess of 100MHz). Therefore, a specific objective of this work has been to examine techniques to maximise the sample-rate of the QR-algorithm. Before considering the implementation issues any further, it is worth introducing the concepts of adaptive filtering, and presenting applications in which it could be applied.

1.3 Adaptive Filtering

A filter is a device for extracting information from a noisy signal, where the noise is considered to be any unwanted component of that signal. For example, the noise could be interference introduced into sensors or the circuits of a system, or be due to distortions or signal echoes resulting from imperfections in a transmission channel.

When it is not possible to determine the coefficients of a suitable filter because there is insufficient *a priori* knowledge of the signal and noise, or if the filter characteristics vary in an unknown way, it is necessary to use an adaptive filter. In this event an *adaptive algorithm* is used to update the filter parameters as new samples of the signal become available.

1.3.1 Adaptive Linear Combiner

At the heart of most adaptive filters is the *adaptive linear combiner* shown in Figure 1.1.

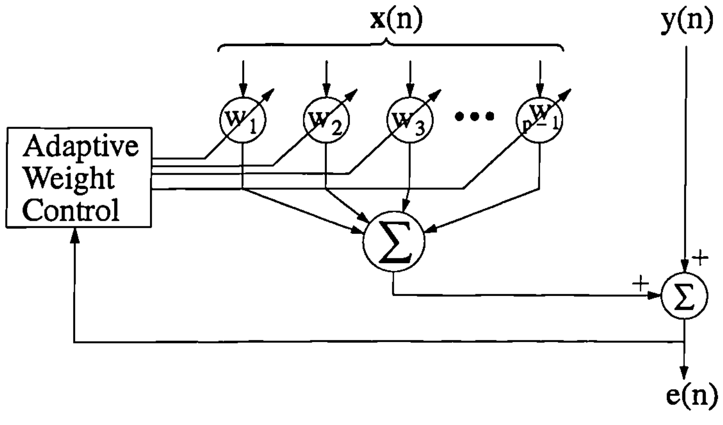


Figure 1.1 Adaptive linear combiner

The combiner has $(p-1)$ *auxiliary inputs* and a single *primary input*, which are given in a discrete time form at time t_n by $\mathbf{x}(n)$ and $y(n)$ respectively. (Note that bold lower-case lettering has been used to denote a vector quantity, and bold upper-case will be used to denote a matrix quantity). The combiner forms a sum of the auxiliary inputs weighted by parameters w_1 to w_p, which is added to the primary input to form an *error signal* $e(n)$. These parameters, which are referred to as weights, are controlled by an adaptive algorithm to minimise a measure of the error signal, such as its *mean squared value*. The error is also, on occasions, referred to as the *residual*, and is given by

$$e(n) = \mathbf{x}^T(n)\mathbf{w} + y(n) \tag{1.1}$$

where $\mathbf{w}$ denotes the optimum weights in vector form, and $\mathbf{x}^T$ is the transpose of auxiliary input data vector.

1.3.2 Applications of Adaptive Filtering

Two important applications of adaptive filtering are *channel equalisation* and *adaptive beamforming*. The channel equaliser is used in Chapter 7 to evaluate the numerical properties of algorithms for performing Givens rotations, whereas adaptive beamforming is considered to be the primary application for the work of this thesis.

1.3.2.1 Channel Equalisation

If the bandwidth of a transmission channel is restricted, then dispersion of the conveyed signal will occur resulting in intersymbol interference. If severe, this will cause the signal to be incorrectly detected at the receiver. The channel equaliser, shown in Figure 1.2, aims to undo the distortion of an imperfect channel by implementing its inverse (as described in [Hayk91], p. 492). This is done using an adaptive transversal filter, which is constructed from an adaptive linear combiner fed by a tapped delay line. A predetermined sequence of data is generated at the transmitter and receiver to provide the adaptive filter with an uncorrupted version of the transmitted signal from which to adapt its weights.

An adaptive equaliser is useful when the channel characteristics are unknown, or vary with time.

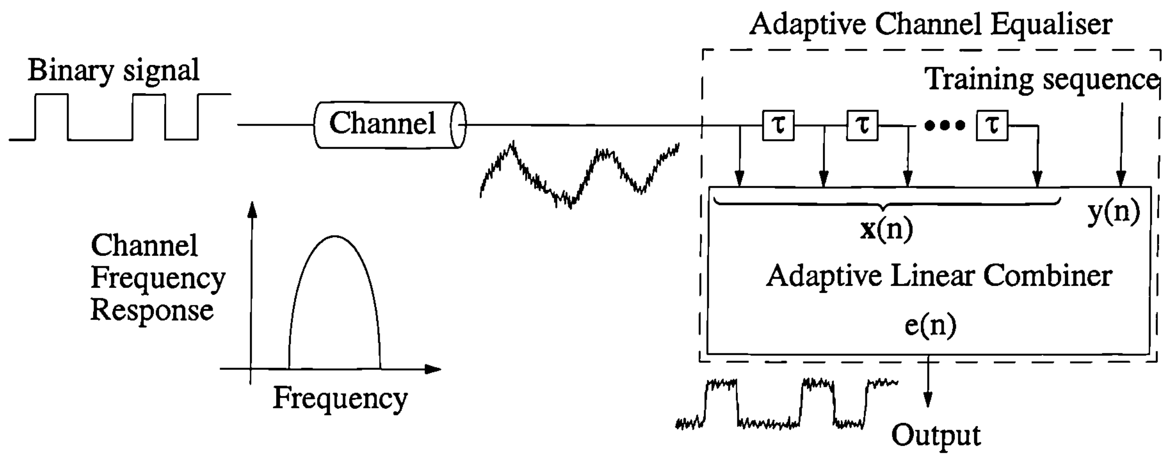


Figure 1.2 Channel equalisation using an adaptive filter

1.3.2.2 Adaptive Beamforming

In the adaptive beamforming application, the adaptive linear combiner is used to provide spatial filtering to reduce the effects of interference impinging upon an antenna array from a direction other than the direction of interest (i.e. the *look direction*). Figure 1.3 depicts an *adaptive sidelobe canceller.*

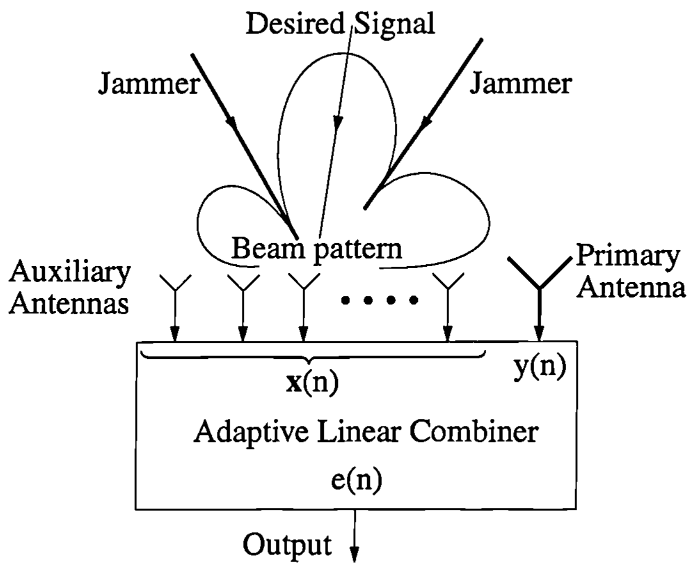


Figure 1.3 Diagram of an adaptive sidelobe canceller

The primary signal constitutes the input from a main antenna which has high directivity, but, due to theoretical and practical design constraints, has a small but significant gain in other directions (i.e. it has sidelobes). If there is a strong unwanted signal incident on the array in the direction of a sidelobe, e.g. due to jamming, it is possible that enough interference will be received to swamp a signal of interest in the look direction. To overcome this, a number of omnidirectional auxiliary antennae are used to sample the interference. The adaptive combiner is then used to remove any signal common with the main antenna. At this stage in the processing, it is assumed that the signal of interest is hidden by noise, so only interference is suppressed. The effect of the combiner on the beam pattern of the main antenna is to introduce deep nulls in the direction of the interference as shown in Figure 1.3.

1.4 The QR-Algorithm

The adaptive linear combiner residual can be minimised in a least mean squares sense using the *QR-algorithm*. This algorithm is of interest to an ASIC implementation as it has good numerical properties. In particular Ward *et al.*[Ward86a] have shown its use can reduce the wordlength requirement over a matrix inversion approach from 24-bits to 16-bits in their particular adaptive beamforming example.

The residuals up to time n can be expressed collectively in matrix form as

$$\begin{bmatrix} e(1) \\ e(2) \\ \vdots \\ e(n) \end{bmatrix} = \begin{bmatrix} x^T(1) \\ x^T(2) \\ \vdots \\ x^T(n) \end{bmatrix} w(n) + \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(n) \end{bmatrix} \qquad (1.2)$$

A more compact representation is

$$e(n) = X(n)w(n) + y(n) \qquad (1.3)$$

where the square error is simply $E(n) = |e(1)|^2 + |e(2)|^2 + \ldots + |e(n)|^2 = ||e(n)||^2$.

A very numerically stable method of solving equation (1.3) is to use orthogonal triangularisation by QR-decomposition to obtain

$$\mathbf{R}(n)\mathbf{w}(n) + \mathbf{u}(n) = \mathbf{0} \tag{1.4}$$

where $\mathbf{R}(n)$ is upper-triangular matrix obtained by applying a unitary matrix $\mathbf{Q}(n)$ to the input data $\mathbf{X}(n)$ i.e.

$$\mathbf{Q}(n)\mathbf{X}(n) = \begin{bmatrix} \mathbf{R}(n) \\ \mathbf{0} \end{bmatrix} \quad \text{and} \quad \mathbf{Q}(n)\mathbf{y}(n) = \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{v}(n) \end{bmatrix} \tag{1.5}$$

The matrix $\mathbf{Q}(n)$ can be applied recursively using a simpler matrix $\hat{\mathbf{Q}}(n)$ such that

$$\hat{\mathbf{Q}}(n)\mathbf{Q}(n-1) = \mathbf{Q}(n) \quad \hat{\mathbf{Q}}(n)\begin{bmatrix} \mathbf{R}(n-1) \\ \mathbf{x}^T(n) \end{bmatrix} = \begin{bmatrix} \mathbf{R}(n) \\ \mathbf{0} \end{bmatrix} \quad \hat{\mathbf{Q}}(n)\begin{bmatrix} \mathbf{u}(n-1) \\ y(n) \end{bmatrix} = \begin{bmatrix} \mathbf{u}(n) \\ \alpha(n) \end{bmatrix} \tag{1.6}$$

where $\hat{\mathbf{Q}}(n)$ is also a unitary matrix which is chosen to set the elements of $\mathbf{x}^T(n)$ to zero in the middle equation. The term $\alpha(n)$ is used to obtain the residual and is discussed later. The transformation $\hat{\mathbf{Q}}(n)$ can be constructed from a sequence of 2-dimensional rotations, known as Givens rotations, which progressively set, from left to right, each element of $\mathbf{x}^T(n)$ to zero. To avoid undoing the effect of previous rotations, the Givens rotation is performed between the elements of $\mathbf{x}^T(n)$ and the row of $\mathbf{R}(n-1)$ which has the same number of leading zeros. For example, to zero the i^{th} element in $\mathbf{x}^T(n)$, we use the i^{th} row of $\mathbf{R}(n-1)$ and the rotation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} 0 & \dots & 0 & r_{i,i} & r_{i,i+1} & \dots & r_{i,k} \\ 0 & \dots & 0 & x_i & x_{i+1} & \dots & x_k \end{bmatrix} = \begin{bmatrix} 0 & \dots & 0 & r'_{i,i} & r'_{i,i+1} & \dots & r'_{i,k} \\ 0 & \dots & 0 & 0 & x'_{i+1} & \dots & x'_k \end{bmatrix} \tag{1.7}$$

The 2-by-2 rotation matrix consists of the sine and cosine of the rotation angle obtained from the leading terms x_i and $r_{i,i}$ using

$$c = \frac{r_{i,i}}{\sqrt{x_i^2 + r_{i,i}^2}} \qquad s = \frac{x_i}{\sqrt{x_i^2 + r_{i,i}^2}} \tag{1.8}$$

A complete derivation of the QR-algorithm can be found in Shepherd[Shep93].

1.5 The QR-Array

One benefit of performing QR-decomposition by Givens rotations is that it can be implemented using a highly parallel array processor, known as the *QR-array,* and shown in Figure 1.4. The array was originally proposed by Gentleman and Kung[Gent81], but contains important modifications made by McWhirter[McWh83] to generate the residual directly from the array. It presents a good starting point for the design of a parallel implementation of the adaptive linear combiner, as considered in Chapter 8.

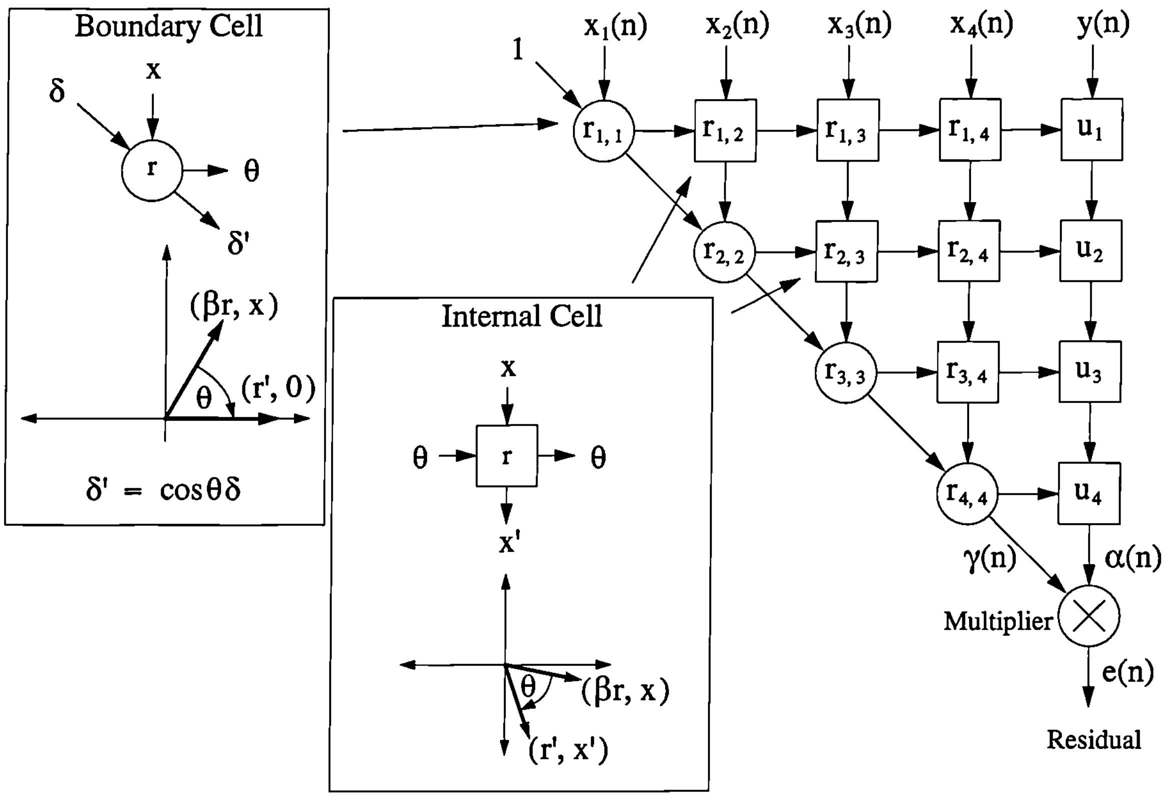


Figure 1.4 The QR-array architecture for performing the QR-algorithm

The array is composed from *boundary cells* and *internal cells* which perform the Givens rotation as indicated in the corresponding insets. The term β is generally referred to as the *forget-factor*, and is simply a constant, which is close to, but less than 1 (e.g. $\beta = 0.996$ would be a typical value in adaptive beamforming).

The elements of $\mathbf{R}(n)$ and $\mathbf{u}(n)$ are stored within the cells of the array, and are initialised to zero. Each combiner input $\mathbf{x}(n)$ enters the top of the array, and propagates down from row to row. On each row of the array the leading non-zero element of the vector is set to zero by the rotation performed in the boundary cell on its input and the stored element r. The same angle of rotation is applied to the remainder of the vector by the internal cells within the same row. The rotation in each cell is performed on a vector, formed from the cell input x and the stored variable, r or u. As a consequence of the rotation process, the stored variable is updated to r' and u' respectively. The term β provides a forget-factor, such that the stored variables decay over time.

The weights of the adaptive linear combiner can be obtained by extracting $\mathbf{R}(n)$ and $\mathbf{u}(n)$ from the array and solving equation (1.4) by back-substitution. However, in some applications, such as adaptive beamforming, it is the residual which is of primary interest. Direct residual extraction has been incorporated into the QR-array by McWhirter[McWh83] by noting that the residual can be obtained by multiplying the output of the right-hand column, denoted $\alpha(n)$ in Figure 1.4, by the product of the rotation cosines, formed down the diagonal of the array (and denoted $\gamma(n)$ in Figure 1.4).

1.6 Performing Rotations in VLSI

There are two principal approaches to performing the rotations required by the QR-array. CORDIC[Vold59][Walt71] (COordinate Rotation by Digital Computer) has been a popular approach as it implements the rotations directly using a single arithmetic component. Whether CORDIC can offer high sample-rate implementations was a question proposed earlier in this work, and has subsequently has been addressed by Hamill in [Hami95a].

In this thesis, the alternative approach based on the standard arithmetic operators, add, multiply, divide and square-root has been addressed. Figure 1.5 shows a *signal flow graph* (SFG) of the QR-array cells implemented using standard arithmetic operators. The conventional Givens rotation algorithm has been used, and the equations defining the operation of the cells are shown in the figure.

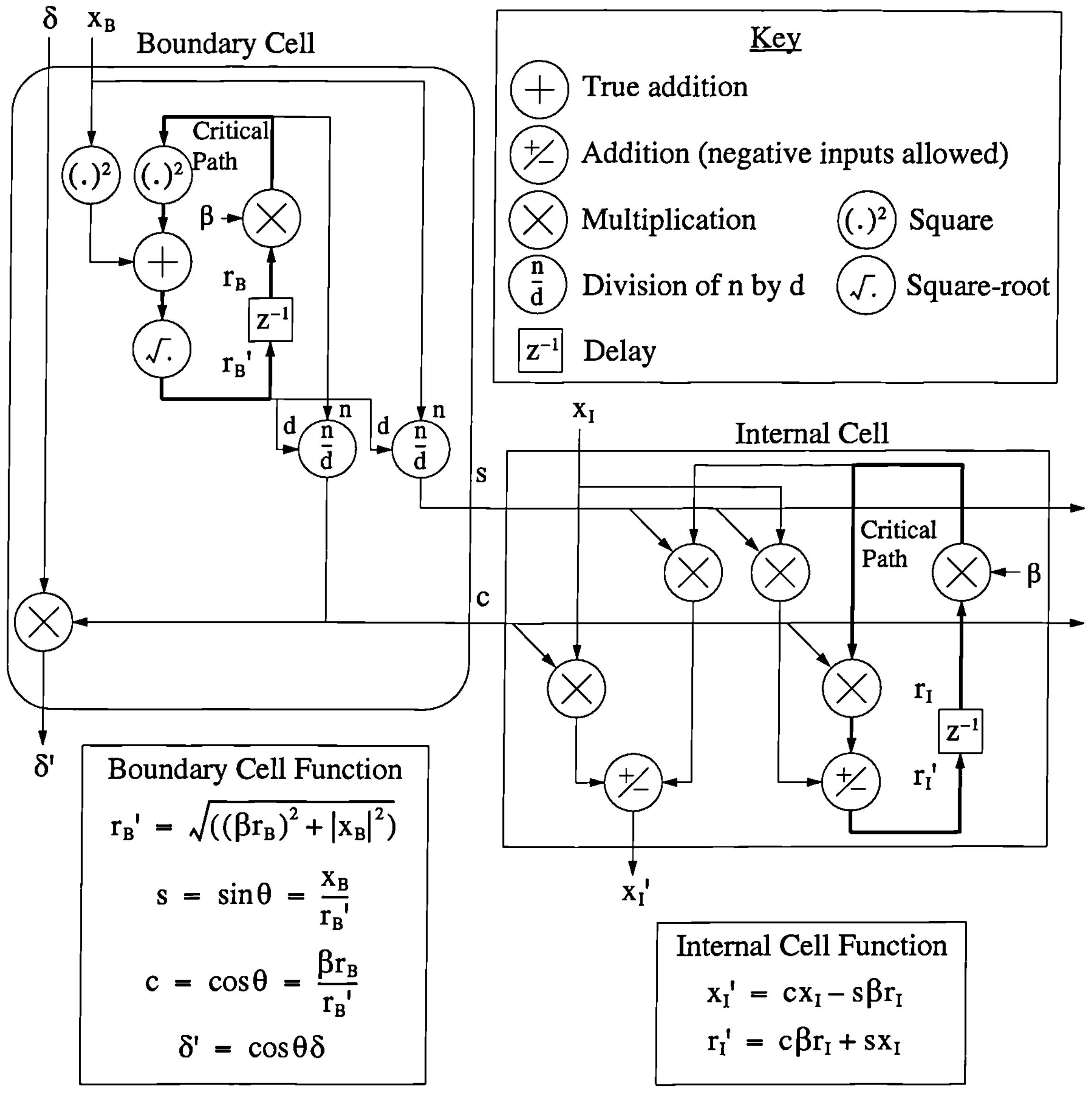


Figure 1.5 Signal flow graph of conventional rotation algorithm

Several observations can be made about the algorithm:

- division and square-root operations are required and these are conventionally high-latency operations,
- loops exist in the boundary and internal cells to update the quantities r_B and r_I, and the latency of these will limit the sample-rate at which the QR-array may operate.

Increased sample-rate can be achieved by moving the square-root operation out of the loop in the boundary cell by maintaining r_B in a squared form. However, there are other transformations which can be performed to the conventional algorithm which will completely remove the need for a square-root operation and also reduce the number of multipliers. As a consequence, there is a wide variety of algorithm possibilities for implementing the QR-array. To determine the best choice for a particular application, it is essential to consider the requirements of each algorithm for wordlength and fixed- and floating-point arithmetic as well the number of operations, their type and topology.

Fixed-point implementations of a DSP algorithm are often sought, as this simplifies the design of the arithmetic operators. For many conventional DSP operations, such as FFT and FIR filtering, this can be achieved with ease, and the solutions are generally fast and efficient. However, the QR-algorithm contains a greater range of arithmetic operations, and has many variants, and it is not obvious what fixed-point solutions are possible, and whether they would offer lower area, higher sample-rate or greater throughput. Therefore, both fixed- and floating-point solutions need to be considered.

Another important aspect of the arithmetic, irrespective of whether fixed- or floating-point arithmetic is used, is the type of rounding used. It is apparent from the SFG of the rotation algorithm that r and u are accumulated quantities. When a number is accumulated, so are the errors. If the errors are unbiased, i.e. their mean is zero, the error grows as $\sqrt{n}$, where n is the number of additions performed (as a consequence of the central limit theorem). If there is a bias, then the error will grow with n. Arithmetic errors are introduced in finite-precision arithmetic in the process of returning the wordlength of the result of an arithmetic operation to that of its input[Wilk63]. Unbiased arithmetic errors can be achieved by rounding the result to the nearest valid number. As will be shown later, this form of rounding takes more time than a simpler scheme, but is worth adopting because the accumulated numerical errors are significantly lower.

1.7 VLSI Design Methodology

The task of implementing VLSI circuits has been greatly simplified over recent years. It is now possible to use CAD tools to synthesise circuits from VHDL (Very high speed integrated circuit Hardware Description Language) into optimised circuits composed of standard cells (such as gates and adders). Tools also exist which will generate circuit layout by automatic placement-and-routing of the standard cell circuit.

This automated design route provides a very rapid path into silicon. For the relatively low wordlengths required by the QR-algorithm applications considered in this thesis (i.e. less than 20-bits), it is possible to synthesise the operators as one complete entity with very good speed and area results. For wordlengths greater than this, the synthesis times become large, the results are not so good, and it is necessary to break down the operators into smaller parts.

Within the thesis, the synthesis tool Synopsys has been used to generate estimates of the circuit delay and number of gates required by arithmetic operators. VHDL descriptions have been produced, which have been parameterised in terms of wordlength and level of pipelining, and so specific circuits can be realised relatively easily (although synthesis times can be long). A 0.35μm standard cell CMOS process has been used as the target technology, and represents the minimum commercially available circuit geometry available at the time of writing. Some arithmetic operators have also been taken through to layout using automatic place-and-route tools, and this has provided a useful guide of how the number of gates translates into circuit area.

The synthesis tool Synopsys is able to identify full-adder components within a circuit description and employ an optimised standard cell for its realisation (if supplied within the standard cell library). This can lead to significant performance improvements over a realisation from basic gates. Hence, there is a strong motivation to use full-adder descriptions of circuits where possible. Within the thesis, full-adders are used almost exclusively to realise arithmetic circuits.

1.8 Overview of Thesis

The QR-algorithm has been the subject of a number of implementation studies and designs, most notably by Rader[Rade96] and McWhirter *et al.*[Ward86b] for applications such as adaptive beamforming in radar systems. None of these, however, has directly addressed the issue of achieving high-sample-rate operation.

The thesis is in two parts. The first part (Chapters 3 to 5) considers the design of arithmetic operators with low-latency and high-throughput. Although this research was motivated by the needs of the QR-algorithm, the results are more generally applicable. The second part (Chapters 6 to 8) specifically addresses the issues of determining the algorithm, architecture and arithmetic implementation of a VLSI implementation of the QR-algorithm. A detailed overview by chapter now follows.

In Chapter 2 the important concept of redundant arithmetic is introduced, as this provides a means of avoiding carry-propagation, which enables the latency of an arithmetic operation to be reduced. On-line and msdf arithmetic techniques are presented as ways in which redundant arithmetic has been exploited to reduce the latency of operators. The extension of the msdf operators to floating-point arithmetic is discussed, as this is of some importance to modern DSP algorithms.

In Chapter 3 two schemes for achieving low-latency, high-throughput multiplication are investigated. Firstly, msdf approaches are analysed for a range of digit-sets, and a new architecture based on the minimally-redundant, radix-4 digit-set is presented as offering the best compromise between latency, area and redundancy. Circuit delay and gate numbers are presented here and later in the thesis to support the research. Secondly, low-latency multipliers based on trees of adders are presented as an alternative which does not require the data skewing of an msdf format. A multiplier with a single redundant input is developed as a replacement to the msdf multipliers considered, which offers single-cycle latency with an area less than a conventional tree-multiplier. For multipliers with two redundant inputs, a new architecture is presented, which uses a minimally redundant radix-n recoding on both multiplier and multiplicand inputs, to reduce the cost of the redundant representation by up to 25%. The application of this technique to a squarer is also addressed, as a dedicated squarer circuit offers significantly reduced area over a multiplier used for the purpose.

Division is a requirement of all but one of the Givens rotation variants examined, and is important in achieving an efficient processor implementation. In chapter 4 the various approaches to achieving low-latency, high-throughput division are examined. Two new architectures are presented, one a modification of a speculative SRT (Sweeney, Robertson and Tocher) divider, made to obtain an acceptable gate count, and the other a high-throughput implementation of the convergence approach.

Floating-point operators are required by the Givens rotation algorithms considered later in the thesis; a range of these has been developed and is presented in Chapter 5.

In Chapter 6 an overview of the Givens rotation variants is presented, and a generalised set of equations is given from which a number of relevant algorithms are derived. Also, normalisation of the Givens rotation is described, as a means of allowing implementation using fixed-point arithmetic.

The suitability of a range of Givens rotation variants for VLSI implementation is examined in Chapter 7. Their numerical performance and subsequent wordlength requirements have been investigated, together with the type, order and number of arithmetic operations used. An enhanced version of the *Squared Givens Rotation* (SGR) algorithm is developed, which obtains extremely high sample-rate with good numerical performance and low circuit area. A detailed comparison with other Givens rotation variants shows that it offers the highest sample-rate and lowest area of the algorithms investigated.

Developing an architecture for the QR-algorithm which can be mapped onto an integrated circuit is addressed in Chapter 8. A number of array architectures are presented that would enable a range of problem sizes and sample-rates to be implemented. The circuit layout, for one of these architectures, is presented using the enhanced-SGR algorithm, and the area and speed has been derived and compared with a solution based on CORDIC and an array of programmable DSP chips.

The research has produced a range of results in the areas of arithmetic, algorithm and architecture from which conclusions have been drawn. These are presented in Chapter 9 and followed by a summary of topics on which further research could be productive.

Chapter 2 Redundant Arithmetic

2.1 Introduction

The representation of a number plays a vital role in achieving low-latency arithmetic. In this chapter the important concept of a *redundant number representation* is introduced, and it is shown how it may be used to obtain fast addition, an operation fundamental to the implementation of all arithmetic operations. A simple, but effective, technique is presented to realise redundant adders using standard full-adders, and a number of examples are given. This technique is used extensively in later chapters to obtain low-latency arithmetic operators.

Redundant arithmetic also enables arithmetic operations to be performed in a most-significant-digit-first (msd-first) manner. This offers an approach for achieving arithmetic with latency which is low and independent of wordlength, and is considered later in the chapter.

2.2 Redundant Number Systems

2.2.1 Redundant Representation

Consider the representation of a number using a fixed-positional number system with radix-r

$$\begin{aligned} X &= x_a x_{a+1} \ldots x_{-1} x_0 \cdot x_1 x_2 \ldots x_b \\ &= \sum_{i=a}^{b} x_i r^{-i} \qquad x_i \in \{\rho_{min}, \ldots, \rho_{max}\} \end{aligned} \tag{2.1}$$

Here and later in the thesis upper case letters are used to represent a complete word and lower case letter to represent individual digits.

In a conventional number representation each digit is allowed to assume one of r values. In a redundant number representation each digit assumes more than r values i.e. $\rho_{max} - \rho_{min} \geq r$. More formally, the cardinality of a digit set is $C_r = \rho_{max} - \rho_{min} + 1$, and by definition the digit-set is redundant if $C_r > r$ and non-redundant if $C_r = r$. If there is only one additional value in the digit-set (i.e. $C_r = r + 1$) then the digit-set is *minimally redundant.* If $C_r = 2r - 1$ then it is referred to as *maximally redundant* and *over redundant* if $C_r \geq 2r$. If negative digits are allowed, the representation is signed. If the digit set is symmetric (i.e. $-\rho_{min} = \rho_{max}$) then it is a *Signed Digit Number Representation (SDNR)*, otherwise it is *asymmetric* and a *Generalised Signed-Digit Number Representation*[Parh90].

Table 2.1 [McQu92] presents a summary of the number systems, and for each gives examples of how the number 528_{10} could be represented. An overbar is used as a more succinct representation of negative digit values.

Table 2.1 Representations of the number 528_{10}

Number system	Digit-set	Cardinality	Representation of 528_{10}
Conventional	$[0...9]$	10	528
Non-redundant, signed	$[\bar{3}...6]$	10	$53\bar{2}$
Minimally-redundant, symmetric	$[\bar{5}...5]$	11	$53\bar{2}$, $1\bar{5}3\bar{2}$
Maximally-redundant, symmetric	$[\bar{9}...9]$	19	$53\bar{2}$, $1\bar{5}3\bar{2}$, $1\bar{4}\bar{8}8$
Over-redundant, symmetric	$[\bar{F}...F]$[a]	30	$53\bar{2}$, $1\bar{5}3\bar{2}$, $1\bar{4}\bar{8}8$, $4C8$
Asymmetric	$[\bar{7}...8]$	16	528, $53\bar{2}$, $1\bar{5}3\bar{2}$

a. Hexadecimal digit coding adopted (i.e. A=10, B=11 etc.)

One advantage of a redundant representation is that the need for carry-propagation is reduced or completely eliminated when two numbers are added. This reduces the time to perform addition and also makes it independent of wordlength. Due to the fundamental importance of addition this has implications for the realisation of all arithmetic operators.

The *carry-save* representation constitutes a redundant number system, since each digit comprises a carry and a sum bit, and so may take values $[0...2]_2$ (where the subscript is used to signify the radix of the digit-set). The carry-save representation has been used for many years to avoid carry-propagation, and so reduce the time to perform repeated additions in an accumulator or multiplier. On completion of either operation, the result is converted to conventional, non-redundant binary by adding the sum and carry bits together using an adder which propagates the carries. In the worst case, the conversion takes as long as a carry takes to propagate from the least to the most significant bit of the result. However, it is performed only once, whereas the number of redundant additions is far greater.

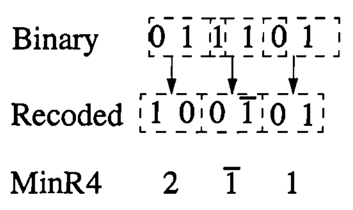
SDNRs have the advantage that the truncation error is almost unbiased[Priv90], whereas asymmetric digit-sets, such as carry-save, require rounding to achieve similar low-levels of bias in truncation error. A particularly useful SDNR is the *minimally redundant radix-4 representation (MinR4)*, which uses the digit-set $[\bar{2}...2]_4$. This does not contain the digit 3 or $\bar{3}$, so it is possible to form products between its digits and a binary number using only shift and complement operations. More specifically, multiplies by 3 are not required, which avoids the need for an adder to produce them.

The MinR4 number representation has been used to reduce the number of partial products generated in multiplication, and is obtained from the conventional binary representation using *modified Booth's coding*. Both the original[Boot51] and the modified Booth's coding schemes[Rubi75] represent important arithmetic techniques, which are considered further in the next section.

2.2.2 Booth's Coding

The original Booth's algorithm replaces sequences of 1's in a number with a 1 preceding the sequence and a $\bar{1}$ at the end, e.g. 1111 would be represented by $1000\bar{1}$. In multiplication, this recoding operation can be applied to increases the number of 0s in a multiplier, and so reduce the number of partial-products which must be formed and added. The recoding of digits can be performed without carry-propagation. However, on occasions where there are few runs of 1s or 0s, this form of recoding can *increase* the number of non-zero digits.

The number of non-zero digits can be minimised by using an alternative recoding. However, it is time consuming as the whole word must be examined to determine each bit. Furthermore, the result is of little benefit, as in a synchronous parallel multiplier it is necessary to design for the worst-case number of non-zero bits, which is approximately half the total number of bits. Therefore, a modified form of Booth's coding is generally used, which achieves the same worst-case reduction of partial-products, but in much less time. The modified recoding ensures that there is at most one non-zero digit in each pair of output digits. Figure 2.1 presents an example. Each pair of digits can be obtained in parallel from 3-bits of the binary using the truth-table shown in the figure. When interpreted as a radix-4 number the result is in a MinR4 representation, as shown in the figure.



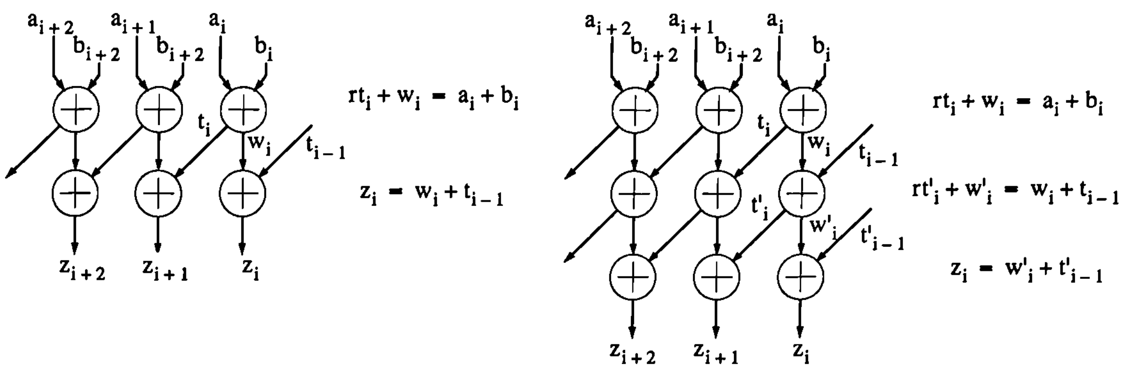
Input Triplet	Output	Radix-4
000 or 111	00	0
001 or 010	01	1
011	10	2
100	$\bar{1}0$	$\bar{2}$
101 or 110	$0\bar{1}$	$\bar{1}$

Figure 2.1 An example of modified Booth's recoding

2.3 Addition of Redundant Numbers

2.3.1 Redundant Adders

Avizienis[Aviz61] showed that two SDNR numbers could be added using one or two transfer digits as shown in Figure 2.2. In the figure, the digits to be added have been denoted a_i and b_i. Their sum has a greater range than can be represented by the output digit z_i. Therefore, the objective of the adders is to reduce the range of the input digits to that of the output using a number of stages of addition.



(a) Adder with single transfer digit (b) Adder with two transfer digits

Figure 2.2 Redundant number adders

Figure 2.2 (a) shows this being achieved in radix-r arithmetic using only two stages of addition and one transfer-digit. The first stage generates the transfer digit t_i and an intermediate sum w_i from the input digits a_i and b_i. The second stage adds the intermediate sum to the transfer digit from the digit-slice to the right. This two-stage addition process ensures that the transfer digit propagates only one digit-slice to the left. For this to occur correctly, the two relationships presented in the figure, defining the operation of the adders, must be satisfied. This is not possible for a radix-2 representation when adding two or more redundant numbers, and it is necessary to use three stages of addition and two transfer digits, as shown in Figure 2.2 (b). It is also necessary to use three stages when the output digit-set has reduced redundancy, as will be demonstrated later in the chapter by an adder design with a MinR4 output. In either case, the extra level of adders means that the transfer digits may now propagate two digits to the left.

The number of transfer digits required depends upon the radix, the number of inputs, and the level of redundancy in the input and output of the adder. The design of redundant adders has also been considered by Parhami[Parh90] for generalised signed digit numbers. The logical design of the adders is not considered, yet is important, as the efficiency of the adder is dependent upon the representation used for intermediate digits. Carter and Robertson[Cart90] have simplified the design of adder logic by decomposing higher-radix adders into a number of stages of lower-radix addition. If decomposed into radix-2, then only 3 primitive operators are required to construct an adder. However, the logical implementation of the adders must be chosen to accommodate the digit-sets used within the adder and these digit-sets are identified by an exhaustive search.

In this thesis, it is proposed that the adder design process be further simplified by basing the adder design on a single primitive component, which consists of a generalised interpretation of the full-adder. Also, and perhaps more significantly, the use of a single full-adder primitive offers high speed and low area when an optimised full-adder standard cell is available. This strategy is supported by recent work by Oklobdzija *et al.*[Oklo96] which demonstrates that, for the case of an unsigned adder for multipliers, an approach using full-adders with a careful choice of inputs and outputs to accommodate their differences in timing, can be an effective approach for obtaining high-speed implementations of large adders. In this thesis, SDNR adders are required. To accomplish this, it is necessary to encode signed-digits using a binary representation so that they may be added using full-adders. This is discussed in the following sections.

2.3.2 Encoding of Redundant Numbers into Binary

Signed-digits can be encoded into binary in various ways. Signed magnitude and 2's complement are two possible approaches. Another method is to use a convenient choice of positively or negatively weighted bits to represent each digit. For example, the MinR4 digit can be represented by three bits: one bit with weight $\bar{2}$ and two bits with weight 1. The advantage of this representation is that a generalised version of the full-adder can be used to add digits.

2.3.3 Generalised Full-Adders

The addition of positively- and negatively-weighted bits can be achieved using a full-adder, providing that the coding presented in Table 2.2 is adopted. As given, the usual logical encoding is used for positively-weighted bits, but an inverted coding is used for negatively-weighted bits. The conversion from a positively-weighted to a negatively-weighted bit is obtained by inverting it.

Table 2.2 Encoding of signed bits

Arithmetic Value of Bit	Logical Value
0	0
+1	1

(a) Encoding of a positively-weighted bit

Arithmetic Value of Bit	Logical Value
0	1
-1	0

(b) Encoding of a negatively-weighted bit

By adopting this coding, it is possible to realise the four 3-input adder combinations of the *generalised adder* shown in Figure 2.3 using the conventional full-adder. It is no longer necessary, as suggested by Hwang[Hwan79] p. 173 and others, to change the logic itself. Extensive use is made of this fact in this and the next chapter.

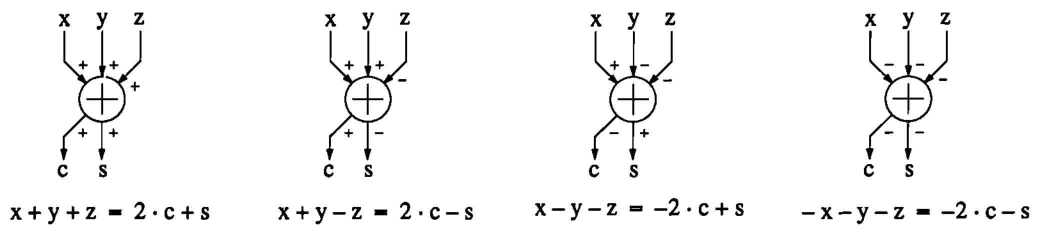
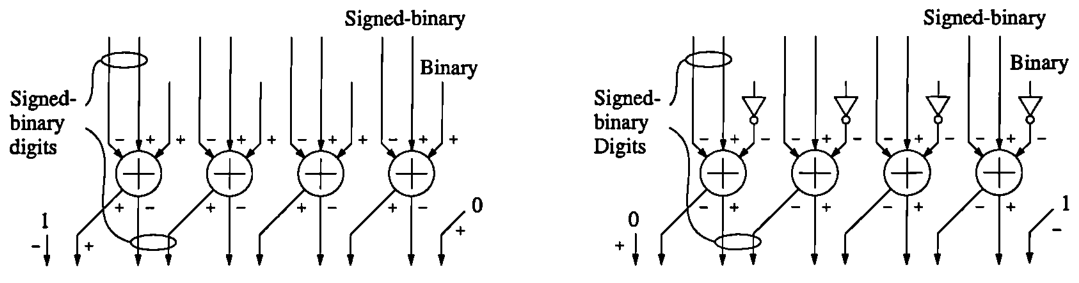


Figure 2.3 The generalised full-adder

2.3.4 Redundant Adders Based on Generalised Full-Adders

Redundant number adders can be constructed from generalised full-adders. For example, Figure 2.4 (a) shows the addition of a signed-binary number and a binary number.



(a) Addition of signed-binary and binary (b) Subtraction of binary from signed-binary

Figure 2.4 Using full-adders to add a signed-binary and a binary number

A single redundant input means that only one transfer digit is required in this addition, and this is added to the output by appending the digit, avoiding a second row of adders. Figure 2.4

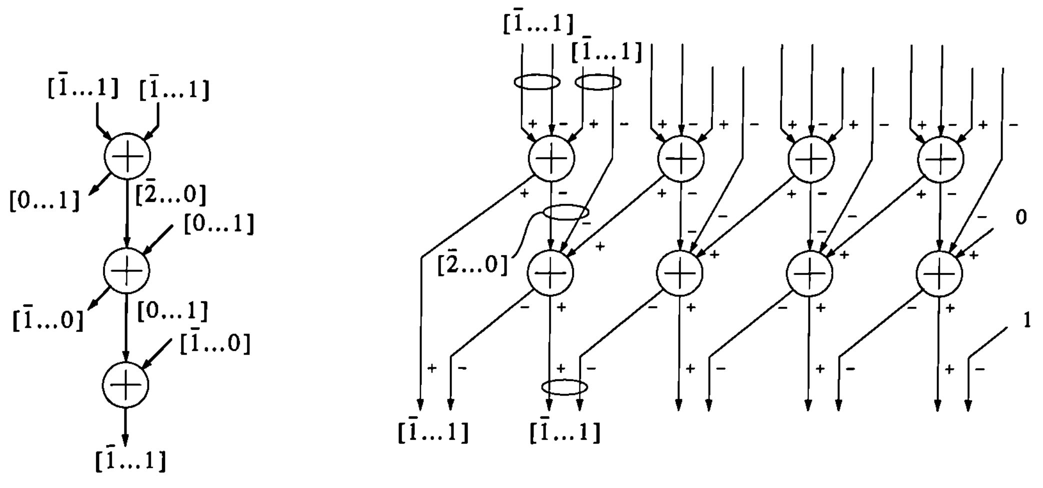
(b) shows that subtraction of the binary input can be obtained by adding a negated version of it, obtained by inversion. The sign of the full-adder output must change to accommodate the two negative input bits. Also, it is necessary to ensure that arithmetically-zero signals are set to the correct logical values. This means that the transfer digit entering the least-significant-digit of the result should be set to logic one. Note that obtaining subtraction in this way (i.e. by inverting bits and adding 1) is the same process as taking the 2's complement. However, allowing individual bits to take on positive or negative signs offers greater flexibility, as demonstrated by the following examples.

2.3.5 Further Examples of Redundant Adders

Two further examples of redundant adders constructed using full-adders are presented in this section. In these and other designs developed in the thesis, the specification of the coding and the use of full-adders has so constrained the design space that an optimum solution is simply found by trial and error. When the number of inputs is large, a more systematic design procedure can be used, and an adder structure based on an adder-tree can be adopted.

As mentioned earlier, the addition of two signed-binary numbers requires at least two transfer digits. Figure 2.5 (a) shows one digit-slice of the generic adder with digit bounds specified for signed-binary addition. Its implementation using full-adders is shown in Figure 2.5 (b). As in the previous example, the final row of addition is achieved without the need for logic circuits. Furthermore, the first transfer digit is determined using three of the four input bits, so there is a degree of flexibility in its calculation which has been exploited to simplify its generation.

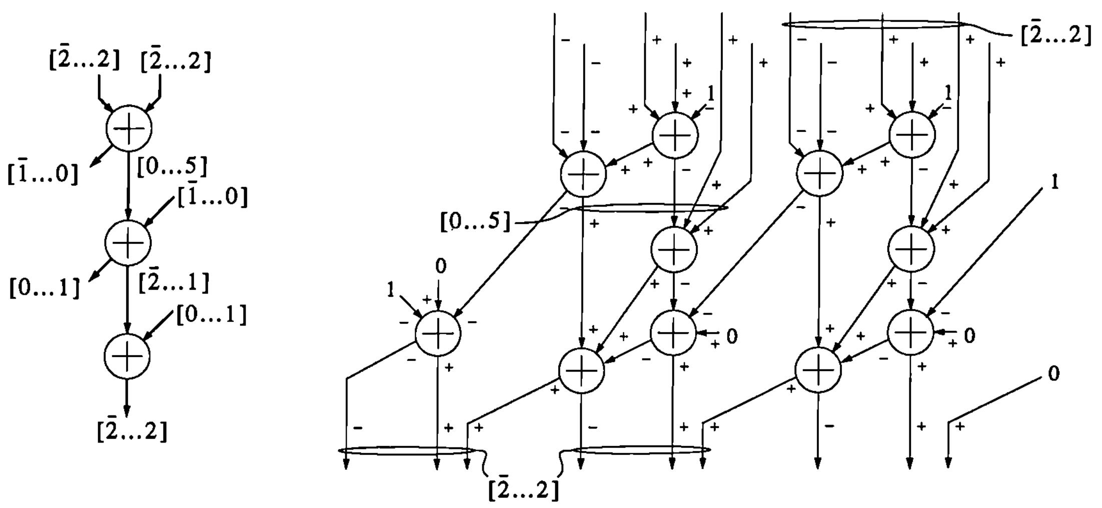
In Figure 2.5 (a) the first intermediate sum is a redundant number (i.e. $(w_i)_{max} - (w_i)_{min} > r$). This avoids a transfer digit from the first adder with the bounds $[\bar{1} \ldots 1]$, requiring two bits. This would have been a valid solution, but would have required an additional half-adder to implement the first row of adders.



(a) Two transfer digit adder-slice (b) Adder implemented using full-adders

Figure 2.5 Using full-adders to add two signed-binary numbers

As a final example, consider the implementation of the MinR4 adder mentioned briefly in section 2.3. At least two transfer digits are required, but these can be coded using single bits by allowing a redundant intermediate sum (as done in Figure 2.5 (a)). This implementation using full-adders is shown in Figure 2.6 (b).



(a) MinR4 adder-slice (b) Adder implemented using full-adders

Figure 2.6 Using full-adders to add two MinR4 numbers

In conclusion, in this section it has been shown how full-adders may be used to implement SBNR adders by a suitable binary encoding of positively- and negatively-weighted bits. The output representation of an adder, or converter, can be controlled by a suitable choice of output coding. The design process then becomes a matter of constructing a network of full-adders which compresses the adder inputs into the bit representation of the output.

Conversion from a redundant representation into a non-redundant one, such as a 2's complement representation, can also be performed by generalised full-adders. Alternatively, the negatively-weighted bits can be converted to a negative 2's complement number and added to the positively-weighted bits using a conventional 2's complement adder. In either case, carry-propagation the full length of the adder will be required. To obtain high-throughput, the addition should be pipelined. If low-latency is required, then a fast carry-propagation technique can be used, such as carry-look-ahead (CLA)[Wein56] or carry-select addition[Bedr62].

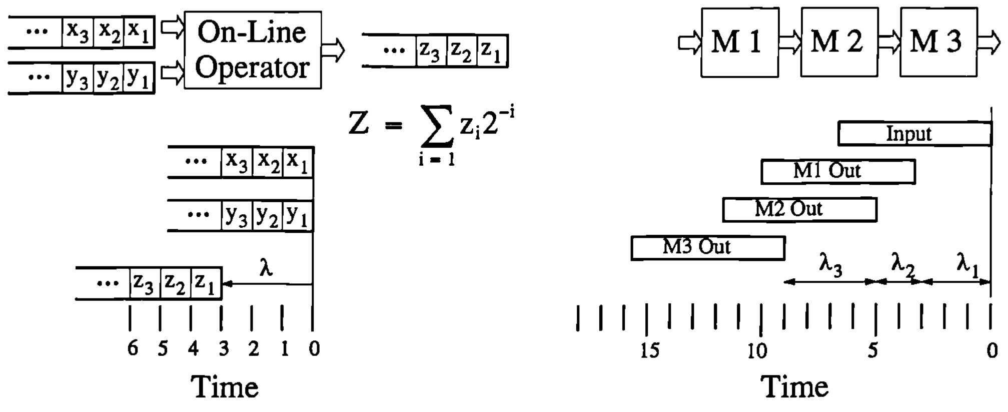
2.4 Most-Significant-Digit-First Arithmetic

Employing redundant arithmetic to reduce carry-propagation time is one effective method of reducing operator latency. However, another application, which aims to offer further reductions, is to allow arithmetic to be performed in a most-significant-digit-first (msd-first) manner. There are serial and parallel forms of msd-first arithmetic. The parallel form offers high-throughput and is of particular relevance in this thesis, but by way of introduction, the serial form, referred to as *on-line arithmetic*, is considered first.

On-line arithmetic was proposed by Ercegovac [Erce77], and since then there has been considerable development of the approach[Triv77][Irwi87]. Figure 2.7 (a) shows how numbers are represented by a serial sequence of digits presented in a msd-first order. Unlike conventional arithmetic, the arithmetic calculation starts with the msd and generates the result in the same order. The benefits of performing the operation this way are that it offers lower latency, and the calculation can be stopped as soon as the required number of significant digits has been calculated.

In contrast, using a conventional least-significant-bit-first serial approach, as employed by

Denyer[Deny85], it would be necessary in multiplication to calculate all the least significant digits of the result in the process of calculating the most significant ones - even if they are not required. This is particularly true in repeated multiplications, since the lower half of each intermediate product must be discarded to avoid wordlength growth.



(a) Digit serial on-line computation (b) Timing of cascaded operators

Figure 2.7 Digit-serial, on-line computation

As shown in Figure 2.7 (b) the on-line operators can be cascaded to obtain the result of a whole sequence of operations with low-latency. The latency can sometimes be reduced further by merging two or more operations into a single combined operator. The most relevant example of this is by Ercegovac[Erce88] who computes the function $z = \frac{1}{\sqrt{x^2 + y^2}}$, which is a requirement of the conventional Givens rotation.

The serial format of on-line arithmetic minimises the number of wires required to communicate a number, which has advantages in chip design. However, it requires that one calculation be completed before another starts and so provides low throughput. To obtain higher throughput, Woods and Knowles[Wood88] proposed the extension of on-line arithmetic to parallel architectures (generally referred to as *msdf arithmetic*). They also generalised the approach to allow inputs in both msdf and parallel formats, as shown in Figure 2.8.

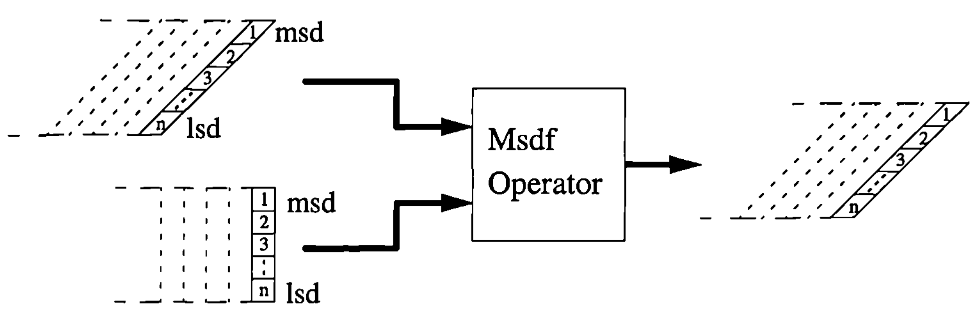


Figure 2.8 Msdf arithmetic

An application which has benefited from msdf arithmetic is recursive filtering, in particular, the infinite impulse response (IIR) filter. In this application, the filter output is generated by summing weighted versions of previous outputs. High sample-rate operation is obtained by generating and feeding back the output as quickly as possible, using an architecture which can sustain the rate of computation. An msdf multiply-add operator offers both the high-throughput and low-latency necessary for high sample-rate operation to be achieved[Wood95].

2.4.1 Converting To and From the Msdf Format

Converting a number into an msdf representation is trivial. However, converting back the skewed representation of msdf into a parallel non-redundant representation incurs a delay while all the digits are aligned, and requires a carry-propagate addition. The addition can be performed as the digits are being aligned using *'on-the-fly'* conversion. This was originally proposed for radix-2 by Majerski[Maje85] and extended to higher radices by Ercegovac and Lang[Erce87], and is achieved in the following way.

For each input digit the sum up to and including digit d_j is obtained recursively by $A_j = A_{j-1} + 2^{-j}d_j$. If carry-propagation is to be avoided, then the result must be assembled by only *appending* bits to the binary result. If the signed-binary digit, d_j, is 1 or 0 the output is updated by simply appending a 1 or 0 to A_{j-1}. If the digit is $\bar{1}$ then a subtraction is required, which will result in a borrow. This can be avoided by maintaining a second term in which the borrow has already been performed, i.e. $B_{j-1} = A_{j-1} - 2.2^{-j}$. The result for

$d_j = \bar{1}$ is then $B_{j-1} + 2^{-j}$. The B_j term can be updated in a similar fashion to A_j.

Knowles[Know89b] produced an alternative form of the signed-binary to binary converter by noting that the addition of the digit $\bar{1}$ had the effect of inverting the preceding string of bits of the form 10...0 in A_{j-1} (where the number of logic 0s may be zero). For example $101\mathbf{1000}\bar{1} = 101\mathbf{01111}$. Now only A_j need be maintained, although a second term is not avoided as a flag must be included for each bit to indicate whether it should be inverted when a $\bar{1}$ occurs. However, this solution does offer simplification of the hardware and is used later in the thesis when conversion is required (e.g. at the output of the modified-SRT divider).

2.5 DSP Using Msdf Arithmetic

A wide range of arithmetic operators can be performed using msdf parallel structures, and it is possible to implement complete DSP algorithms using a combination of msdf operators to achieve low-latency, high-throughput implementations. For algorithms with feedback this offers the potential for high sample-rate operation.

DSP system implementation using on-line arithmetic has already been widely addressed, and the principles can be directly extended to the msdf approach. With a view to exploiting this work for implementing msdf systems, McQuillan reviewed the on-line arithmetic literature and considered its application in areas other than IIR filtering[McQu92]. He also examined the extension of on-line fixed-point operators to floating-point arithmetic. This work is of particular relevance to this thesis and identifies a serious limitation of the msdf approach, so is considered next.

Floating-point requires the number representation to be composed from an exponent and a normalised mantissa. Exponents can be processed very rapidly using conventional arithmetic as they generally have short wordlength. The mantissas can be processed using fixed-point msdf operators providing that *normalisation* and *alignment* operations are used to convert to and from the normalised representation (where normalisation is a shift-left and alignment a shift-right operation).

Conventionally, a mantissa of a floating-point number, M, is normalised if $r^{-1} \leq |M| < 1$ [IEEE85]. To determine if this is true for a redundant representation, it is necessary to examine all digits (e.g the signed-binary number $0.1000000\bar{1}$ is unnormalised). Therefore, *quasi-normalised* numbers have been proposed (by Watanuki[Wata81]), where the mantissa is quasi-normalised if $r^{-2} \leq |M| < 1$. To determine if this is the case only the first two digits of a number need be examined. However, the truncation error is greater and the wordlength must be increased by one bit to maintain the same worst-case relative error.

McQuillan showed that all the basic arithmetic floating-point operators (i.e. multiply, divide, square-root and addition) offer fixed low-latency, apart from true subtraction which could require a latency of $n + 3$ (where n is the mantissa wordlength). In subtraction, the problem arises from the large number of leading zeros which occur when two numbers which have similar values are subtracted. Consequently, to normalise the result a wordlength dependent delay can be incurred. In applications where the latency of the subtraction operation is critical, for example when it is performed within a loop, floating-point msdf arithmetic in its current form, will be of little benefit. McQuillan identified three approaches to tackling the problem.

Variable Precision Arithmetic: Unnormalised numbers which require more than m-cycles to normalise are assumed to be zero. Hence, only a delay of $m + 3$-cycles is introduced where $m < n$. The smaller m is, the greater the numerical implications of this approach will be. Unless the system is trivial, it will be necessary to establish the impact on the quality of the result using numerical simulations. In the QR-algorithm, the implications of this approach are likely to be severe.

Unnormalised Arithmetic: Floating-point arithmetic, in which the mantissas are not normalised, has been considered by Metropolis in [Metr63]; although normalisation before division was still proposed. A limitation of this approach is a lack of error and stability analysis for unnormalised arithmetic. Owens considers the normalisation problem in detail in [Owen83] and suggests that certain algorithms can be coerced into generating numbers that are normalised or nearly so. Generating normalised values has the added benefit that classical error and stability analysis can be applied.

Asynchronous Systems: The average normalisation delay will be considerably less than the worst-case delay. If a system is designed to operate in an asynchronous manner, the variable normalisation delay can be accommodated. Such a system will need buffers to queue data and a greater level of control to manage its operation, so the extra complexity of implementing these would need to be assessed and compared with the savings offered by msdf arithmetic.

All of the above approaches will require extensive computer simulations to ensure that the numerical performance and, in the asynchronous case, the throughput are acceptable. This can be a computer-intensive task, particularly if very detailed modelling of the operators is required to faithfully generate numerical errors.

Chapter 3 High-Throughput, Low-Latency Multipliers

3.1 Introduction

3.1.1 Importance of Multiplication

Many DSP algorithms are dominated by multiply-add operations. Multipliers can also be used to realise other operations, such as divides, square-roots and transcendental functions, using convergence or series expansion approaches. For these reasons high-throughput, low-latency multipliers have been the primary focus of modern computer arithmetic research.

High-throughput multipliers can be achieved by pipelining. This divides the operation into stages using latches, so that only the smaller delay of a single stage determines the rate at which inputs may be applied. Unfortunately, this does not reduce the latency of the result, as this is generated by the combination of all stages, irrespective of how they are partitioned. In microprocessors, latency complicates operation and programming, and in the implementation of recursive systems, limits the sample-rate. One approach to reducing latency is to use redundant arithmetic, as considered in this chapter.

3.1.2 Low-Latency, Redundant Multiplication

Digital multiplication is essentially a series of additions, which can be performed with a reduced level of carry-propagation using a redundant representation (as discussed in Chapter 2). Indeed, the redundant carry-save representation has been used extensively for this purpose in conventional multipliers, where the final sum, in its carry-save form, is converted to non-redundant binary using a carry-propagate adder. The duration of the latter step is similar to that of the partial-product addition, but can be avoided by allowing a redundant output representation from the multiplier. To enable the output to be fed back into the input, or when cascaded, into another multiplier input, the multiplier must also be designed to accept a redundant input. This shall be referred to as a *redundant multiplier*.

The redundant multiplier output can be viewed as a result of not completing the partial-product accumulation process. As a consequence, more bits are required to represent the output, and when used as an input, the number of partial-product bits which must be formed and added is increased. In this chapter, ways of minimising the cost of a redundant representation in multiplication are presented. In particular, it is shown that a multiplier can be designed with one redundant input, which is suitable for applications such as high sample-rate IIR filters and the Givens rotation, with *less* area than a conventional multiplier.

Two architectures are considered for obtaining low-latency redundant multiplication:

- Array multipliers employing msdf arithmetic
- Tree multipliers using a parallel number format

Using the msdf approach, multipliers with only one redundant input are presented. For the tree-based approach, multipliers with one and two redundant inputs are considered, as the latter is pertinent to the implementation of high-throughput, multiplicative elementary function generators, an example of which, is the reciprocal circuit presented in the next chapter.

3.2 Msdf Multipliers

3.2.1 Background

As discussed in Chapter 2, msdf arithmetic aims to reduce the latency of the multiply operation by obtaining the result in an msd-first, skewed manner, so that the result digits can be produced as quickly as possible, before all terms of the partial-products are formed and summed. This is made possible by the reduced level of carry-propagation that occurs using redundant addition, which enables digits of the result to be determined from only a few of the most significant digits of the incomplete partial-product sum.

Msdf arithmetic was proposed by Woods *et al.* [Wood88] as a means of directly achieving high-throughput, multiply-add operations with low-latency. The architectures were aimed at implementing high sample-rate IIR filters, and have the advantages of being regular and simply extensible for increased wordlength with little degradation in speed.

A number of radix-2 architectures were proposed by Woods *et al.* to perform the computation $M = YX + A$, where X is a coefficient, A an additive input, and Y and M the msdf input and output respectively. The work culminated in an architecture which was based on a simple and regular carry-save array with a signed-binary, msdf representation for the input and output, and a 2's complement parallel representation for the coefficient[Know89a]. This multiplier-adder was used to implement a high sample-rate programmable IIR filter chip[Wood95]. Its implementation required similar area to that of a conventional array multiplier, yet offered a significant reduction in latency. However, when it was compared with a conventional Booth's recoded multiplier the area was found to be considerably greater.

This difference was, in part, due to the use of modified Booth's recoding in the conventional multiplier to almost halve the number of partial-products. This is achieved, as discussed in Chapter 2, by recoding the multiplier input into the MinR4 digit-set $[\bar{2}...2]_4$. The radix-4 digit-set halves the number of multiplier digits and associated partial-products, yet only slightly increases the complexity of forming the partial-products. Consequently, recoding almost halves the area of a multiplier.

Lapoint *et al.* in [Lapo90] aimed to obtain a similar reduction for an msdf multiplier by using the radix-4 digit-set $[\bar{3}...3]_4$ and avoided the complexity of calculating the partial-product with the digit 3 by precalculating $3X$. However, the saving in area is restricted by the extra pipelining latches required to distribute $3X$ throughout the array and the more complex partial-product logic.

Therefore, the question arose as to whether it was possible to use a redundant representation other than signed-binary, such as MinR4, directly within the multiplier to reduce its area or latency. In answer to this, a study of msdf multipliers using a range of digit-sets was undertaken, and the results are presented in the first half of this chapter.

To provide an algorithmic basis for this research, the analysis of radix-2 msdf multipliers performed by McQuillan[McQu95] has been extended to higher radices. This has enabled the minimum latency to be determined for a range of digit-sets and radices. By considering these results and the complexity of the partial-product generation for each multiplier a number of the more promising cases have been identified and architectures suitable for IIR filtering produced. In particular, a new architecture has been developed and published[Walk93] using the MinR4 digit-set, which has been shown by circuit synthesis to offer significant savings in area over the signed-binary approach.

A similar study was performed by Brackert and Ercegovac [Brac89a] for on-line multipliers, and they have also considered the application of IIR filtering, but suggest a solution based on a number of on-line modules[Brac89b]. Here, a more direct approach is used to achieve high-throughput, based on a parallel msdf implementation. As mentioned in section 2.4, there is actually little difference in the underlying algorithms used by on-line and msdf multipliers. However, the architectures are quite different, and the msdf parallel architectures, presented later in the chapter, and the circuit synthesis results are of sufficient interest for presentation in this thesis.

3.2.2 Msdf Multiplier Algorithm

The aim of the msdf multiplier-adder is to compute the function

$$M = XY + A \tag{3.1}$$

The product M is computed, most significant digit first, with Y and A supplied in a digit-by-digit, msd-first manner. The multiplicand X is known at the start of the calculation and is presented in a 2's complement representation.

For the general radix case, the partial multiplier Y_j and partial addend A_j at the j^{th} iteration are given by

$$\begin{aligned} Y_j &= Y_{j-1} + y_j r^{-j} = \sum_{i=1}^{j} y_i r^{-i} \qquad y_i \in \{\pi_{min}, \ldots, \pi_{max}\} \\ A_j &= A_{j-1} + a_j r^{-j} = \sum_{i=1}^{j} a_i r^{-i} \qquad a_i \in \{\pi_{min}, \ldots, \pi_{max}\} \end{aligned} \tag{3.2}$$

Where π_{min} and π_{max} represent the two extreme values of the digit range.

On each iteration, a digit is added to the partial terms Y_j and A_j, starting with the most significant digit and finishing with the least significant one. The final result M is also compiled in this manner, but is delayed by the time required to perform the computation. This latency shall be denoted δ and defined to be the number of iterations between a multiplier digit entering the computation and a result digit of the same significance being computed. Hence on the j^{th} iteration, only $M_{j-\delta}$ is available, where

$$M_{j-\delta} = M_{j-\delta-1} + m_{j-\delta} r^{-j+\delta} = \sum_{i=1}^{j-\delta} m_i r^{-i} \qquad m_i \in \{\pi_{min}, \ldots, \pi_{max}\} \tag{3.3}$$

To compute the result in this manner a residual can be defined as:

$$Z_j = r^{j-\delta}(XY_j + A_j - M_{j-\delta}) \tag{3.4}$$

This residual represents the multiply-add operation performed with all available digits of the input minus the partial result $M_{j-\delta}$. The scaling factor $r^{j-\delta}$ has been introduced for convenience only, and ensures that the residual is maintained within a fixed range.

A recurrence equation for Z_j can be developed to compute the next residual from the previous one. That is,

$$Z_j = rZ_{j-1} + r^{-\delta}(Xy_j + a_j) - m_{j-\delta} \tag{3.5}$$

Where $Z_0 = 0$.

As formulated, a digit of the result (i.e. $m_{j-\delta}$) is determined on each iteration of the recurrence. As indicated by equation (3.3) the partial result $M_{j-\delta}$ is obtained by simply appending successive result digits, and no modification of previous digits occurs.

The minimum latency of the result is derived in Appendix A, and is given by

$$\delta \geq \left\lceil \frac{\log\left(\frac{R((|X|)_{max}+1)}{R-1-\Delta}\right)}{\log r} + 1 \right\rceil \tag{3.6}$$

where

- $(|X|)_{max}$ is the maximum absolute value of the multiplier coefficient.
- Δ is the overlap and represents the degree of choice in the output digit. The greater the overlap the fewer digits of the partial-product sum which must be examined to generate an output digit. If there is no overlap (i.e. $\Delta = 0$) all digits must be examined (i.e. carry-propagate addition of the partial-products is required).
- R represents the redundancy in digit-set of the input and output, where $R = \frac{\pi_{max} - \pi_{min}}{r-1}$ for the digit-set $[\pi_{min} \ldots \pi_{max}]_r$.

The minimum value of latency can be calculated by setting the overlap to zero (i.e. $\Delta = 0$). With $(|X|)_{max} = 2$, Figure 3.1 shows the minimum latency as a function of the redundancy R for a range of radices. Also shown as points on the graph is the latency obtained for a range of digit-sets. In practice, a value of overlap greater than zero is required to avoid carry-propagate additions, and the greater it is, the faster the result digit selection. For many of the digit-sets shown in Figure 3.1 the latency is obtained by rounding up to the nearest integer. Consequently, the overlap is usually greater than zero. If not, then it is necessary to increase the latency by one to make it so.

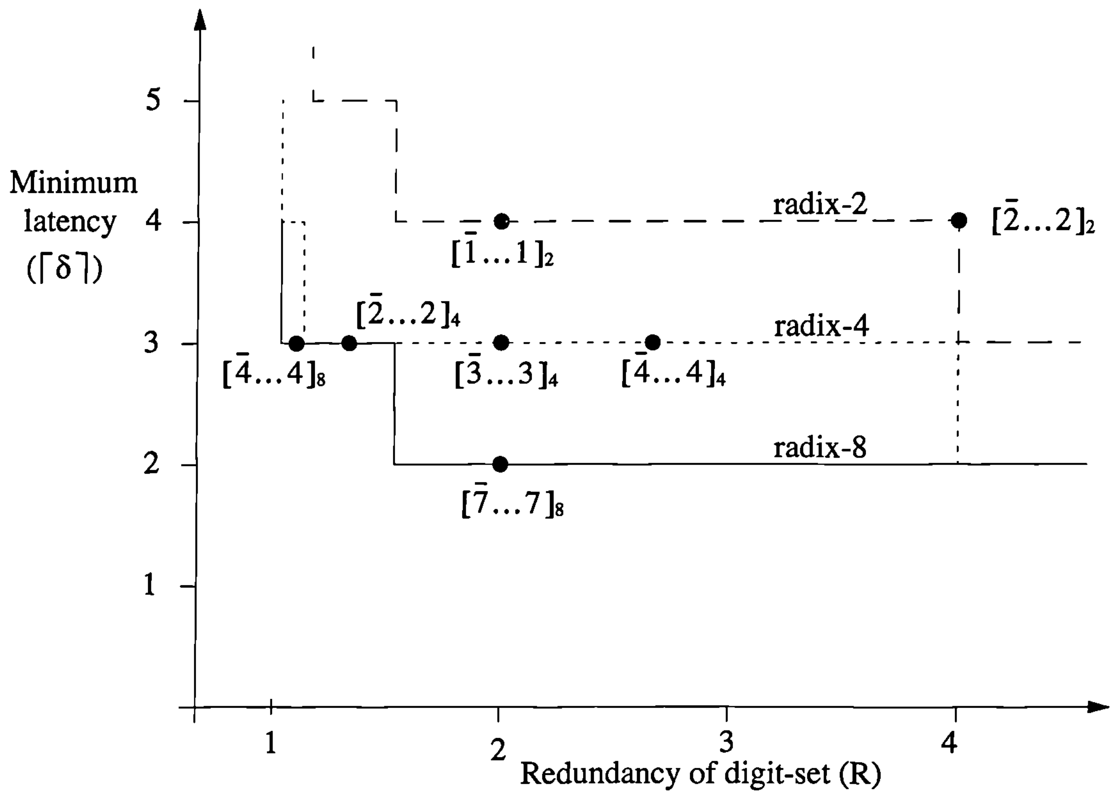


Figure 3.1 Minimum latency for a range of digit redundancy and radix

3.2.3 Msdf Multiplier Options

In this section, msdf multiplier-adders based on a range of digit-sets are explored with the objective of identifying an optimum choice. This is undertaken using IIR filtering as the target application as described by Woods *et al.*[Wood95], and requires that multiplier-adders be designed with a coefficient range of $|X| \le 2$. The sample-rate of the IIR filter depends upon the latency of the multiplier-adder and the complexity of the circuits to form the partial-products and select the result digits. With this in mind, the benefits of each digit-set are discussed below.

Using radix-8 arithmetic and the maximally redundant digit-set $[\bar{7}...7]_8$ provides the lowest latency. However, the generation of the partial-products will be relatively complex due to the size and number of digit values, which will offset any benefit of lower latency. Simplifying the digit-set to $[\bar{4}...4]_8$ increases the latency to that of radix-4 arithmetic, at which point there are simpler radix-4 digits sets which may be used. Therefore, the radix-8 options are considered no further.

All three of the radix-4 digit-sets offer a latency of 3. The over-redundant case will not be considered further as the digit values will add an excessive amount of further complexity, but with no saving in latency.

The signed-binary case presents a high latency of 4, but offers simple partial-product generation. The over-redundant digit-set $[\bar{2}\ldots2]_2$ has a minimum latency of 3 but there is no overlap at this point, so the latency will need to be increased to 4. However, if $(|X|)_{max}$ is reduced then the overlap will be increased and allow a latency of 3. This is no less than the MinR4 digit-set, yet there will be twice as many digits in the radix-2 case and the multiplier will be twice as large. Hence, this option will also be discarded at this point.

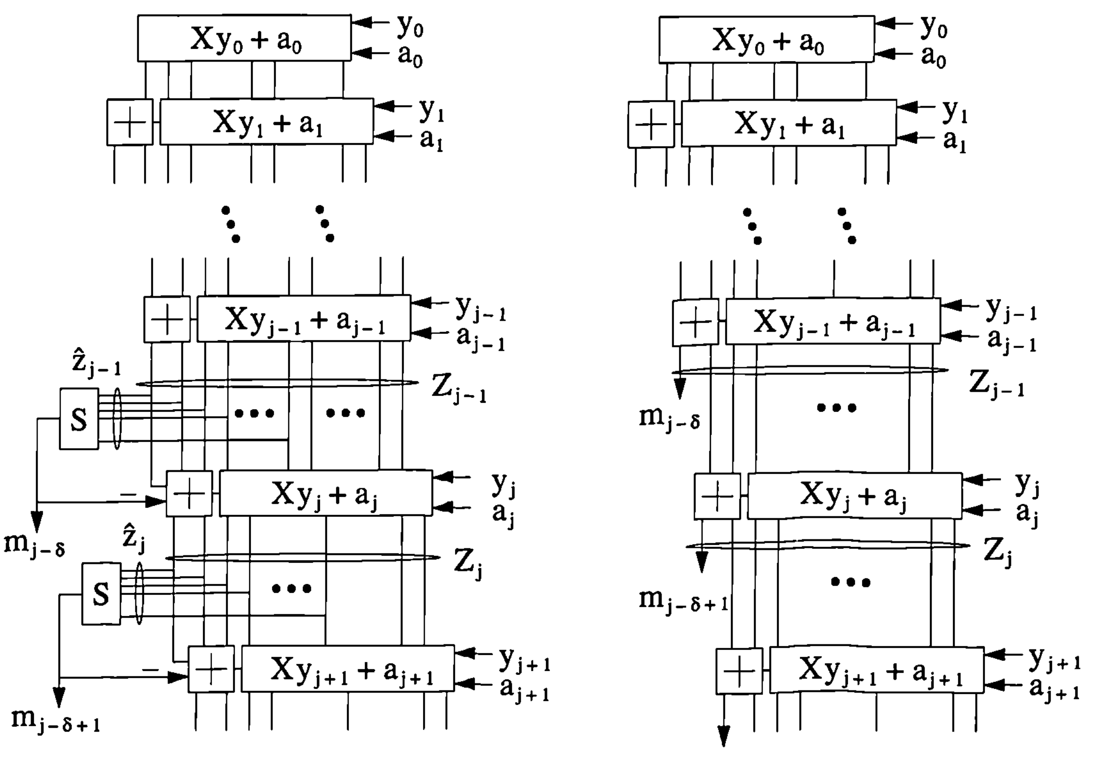
Having eliminated those digit-sets which are clearly of little interest, only three remain. These are:

- Radix-2 signed-binary $[\bar{1}\ldots1]_2$
- Radix-4 minimally redundant (MinR4) $[\bar{2}\ldots2]_4$
- Radix-4 maximally redundant (MaxR4) $[\bar{3}\ldots3]_4$

The most promising digit-set is that of MinR4 i.e. $[\bar{2}\ldots2]_4$, as it offers the same latency as the other radix-4 digit-sets, yet with very simple partial-product generation requirements.

3.2.4 Msdf Multiplier Architecture Types

To implement the msdf multiplier-adder two types of architecture are possible, and representative portions of both are shown in Figure 3.2.



(a) Type 1 SFG (b) Type 2 SFG

Figure 3.2 Msdf multiplier-adder architectures

The type 1 architecture is a direct mapping of recurrence equation (3.5) on to an array of cells, and is useful for explaining the operation of the multiplier. The architecture is like that of an array multiplier, apart from the fact that the partial-products are added most significant first, and the result digits are generated before the product sum is complete. Each row of the array forms and accumulates the partial-product between the coefficient X and a digit of the multiplier y_j. The digits of the additive input are also accumulated, one at a time, most significant first. The result digits are generated msd-first, one by each row of the array. In the type 1 architecture the result digits are obtained in a two step process in which a selection cell (denoted S in the figure) determines the correct digit value from a truncated partial remainder, and a subtractor in the next row removes it from the partial-product sum.

In the type 2 architecture, shown in Figure 3.2(b), the result digit subtraction is avoided by using adders to partition the partial-product sum into two parts: a leading digit, which forms the result digit, and remainder, which is the partial-product sum with the result digit removed. In effect, the selection cell is implemented using the same adders which calculate the remainder of the partial-product sum.

3.2.5 Msdf Multiplier Architectures

Only type 2 architectures are considered for implementation, as these are constructed from simple full-adder circuits, and should offer reduced result digit selection time over the type 1 architectures.

3.2.5.1 Radix-2 Multiplier-Adder

Figure 3.3 shows the signed-binary multiplier-adder developed by Woods *et al.* [Wood95].

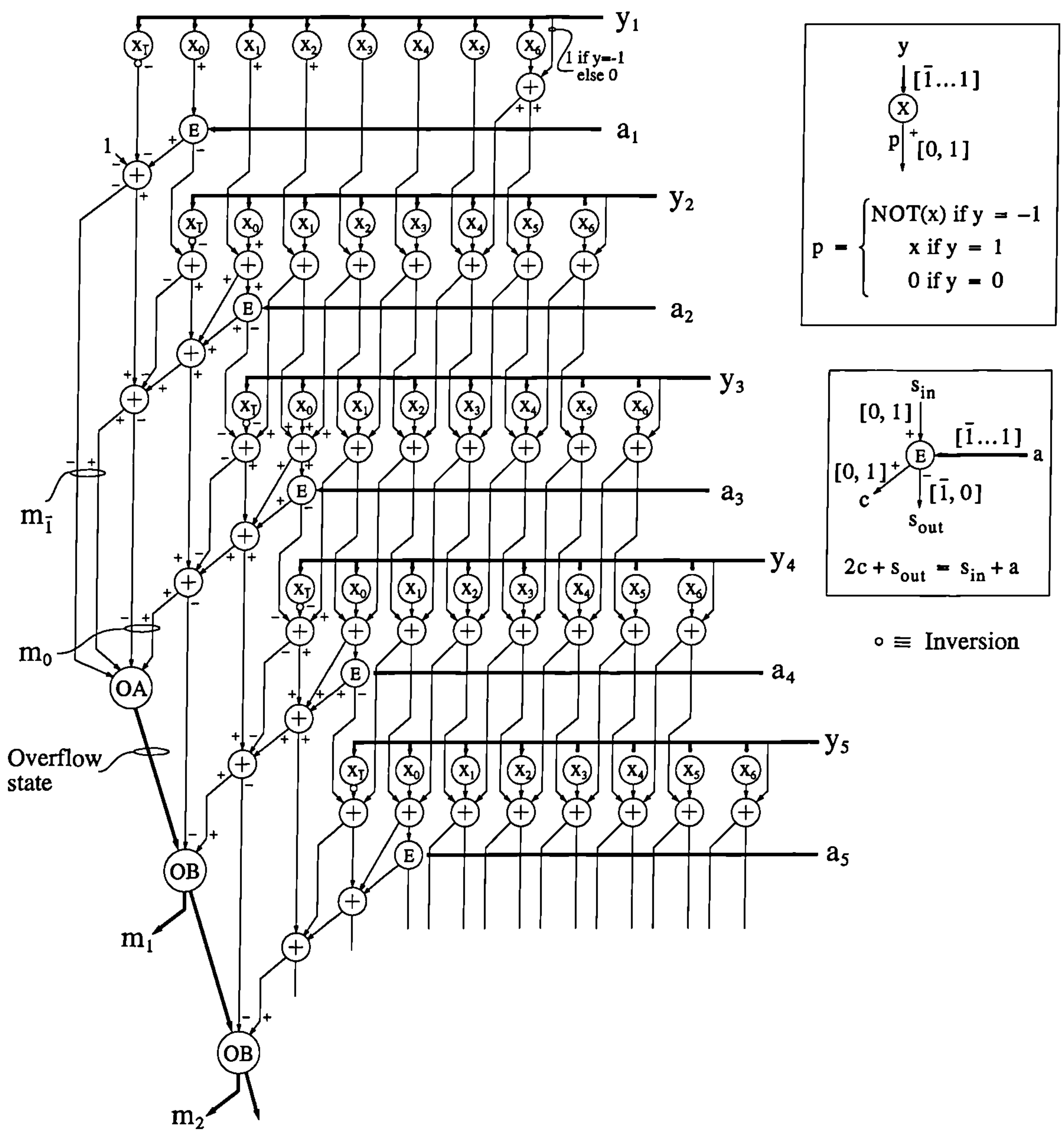
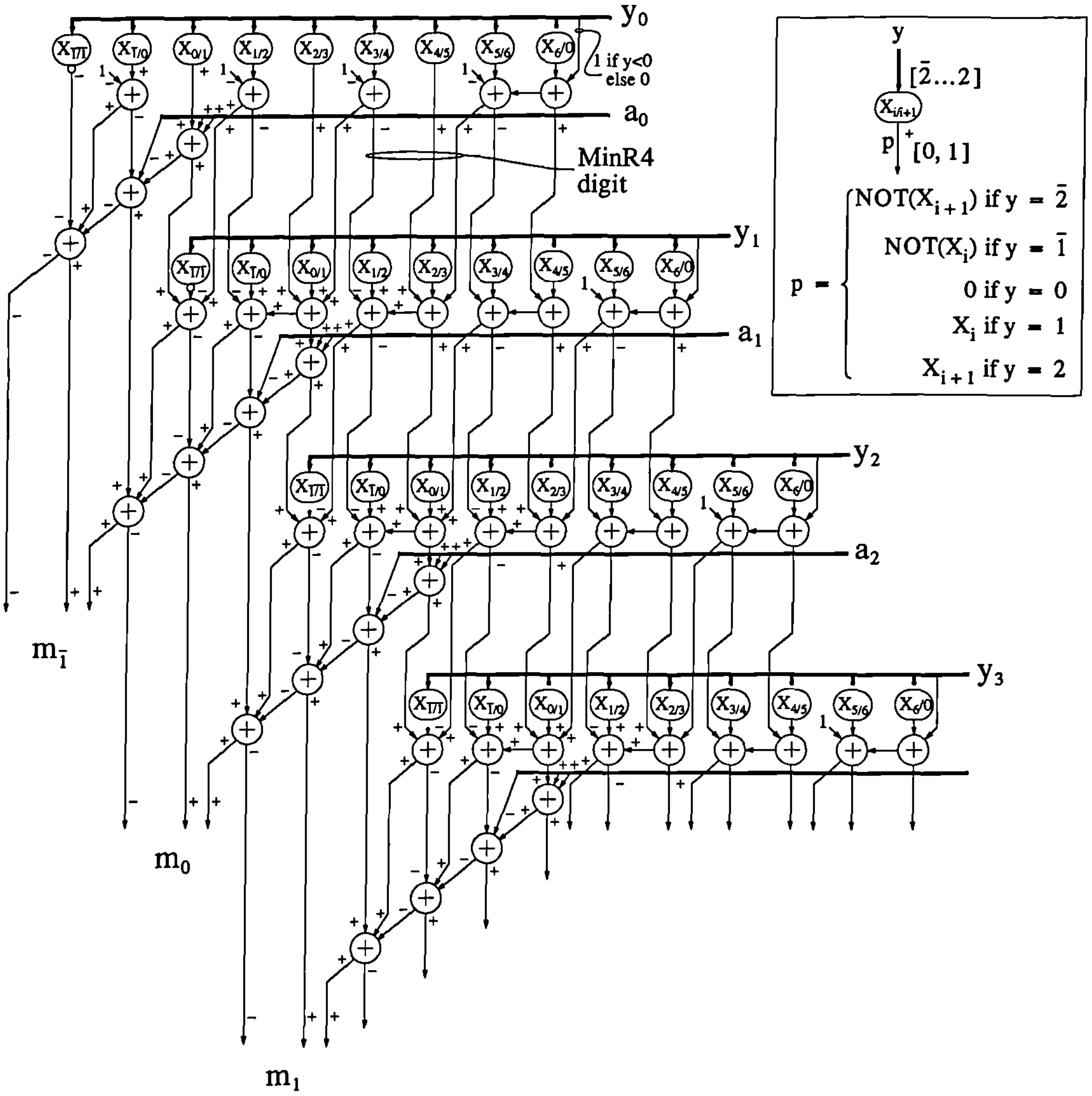


Figure 3.3 Type 2 architecture for a signed-binary, radix-2 multiplier-adder

Both the Y and A inputs are accepted in a signed-binary, msdf format, and a carry-save representation is used for the residual. The OA and OB cells provide compression to avoid increased wordlength due to redundancy overflow. They also provide saturation of the output to ensure that $M < 1$. This is essential in the IIR filter as a coefficient range of $(|X|)_{max} = 2$. can lead to overflow, causing gross errors in the output, and leading to oscillation of the filter. Oscillation can be avoided by saturating the output to a fixed limit when overflow occurs. Fortunately, saturation can be performed in an msdf manner, and the algorithm developed for the radix-2 IIR filter chip is described by Woods *et al.* in [Wood91].

3.2.5.2 MinR4 Multiplier-Adder

Figure 3.4 presents a type 2 architecture for a MinR4 multiplier-adder.



y_0
1 if y<0 else 0
a_0
MinR4 digit
y_1
a_1
y_2
a_2
y_3
$m_{\bar{1}}$
m_0
m_1
y
$[\bar{2}...2]$
$X_{i/i+1}$
p
$[0, 1]$
$p = \begin{cases} NOT(X_{i+1}) \text{ if } y = \bar{2} \\ NOT(X_i) \text{ if } y = \bar{1} \\ 0 \text{ if } y = 0 \\ X_i \text{ if } y = 1 \\ X_{i+1} \text{ if } y = 2 \end{cases}$

Figure 3.4 Type 2 architecture for a minimally-redundant, radix-4 multiplier-adder

The inputs A and Y and the output M are in a MinR4 representation. The MinR4 representation has also been used within the body of the array to simplify the result-digit selection adders. It also reduces the number of bits required to represent the residual, and hence decreases the number of pipelining latches by 25% over a carry-save representation. Further reductions in latches can be obtained by using a minimally-redundant digit-set of higher radix within the body of the array, although care must be taken to ensure that the resulting longer carry chains do not lengthen the critical path.

The MinR4 multiplier has only one row of adders for each radix-4 digit. This halves the number of adders over the signed-binary case. However, the partial-product cells are slightly more complex than for the signed-binary ones, and so the reduction in multiplier area is not quite one-half (as shown later).

3.2.5.3 Maximally Redundant Radix-4 Multiplier-Adder

The architecture for the MaxR4 multiplier-adder is shown in Figure 3.5. The radix-4 digits are represented using 4 bits, which are weighted $\bar{2}$, $\bar{1}$, 1 and 2. These can be grouped into two signed-binary digits b_1 and b_0 weighted by 2 and 1 respectively. Partial-products are formed for each MaxR4 digit by forming two separate products with each signed-binary digit and using an extra row of adders to add them. This results in twice the number of rows of adders over the MinR4 architecture, and as many as the signed-binary one.

Compression and saturation has been implemented in the MinR4 case using the function described in Table 3.1. The MaxR4 multiplier-adder uses a similar circuit to the signed-binary case.

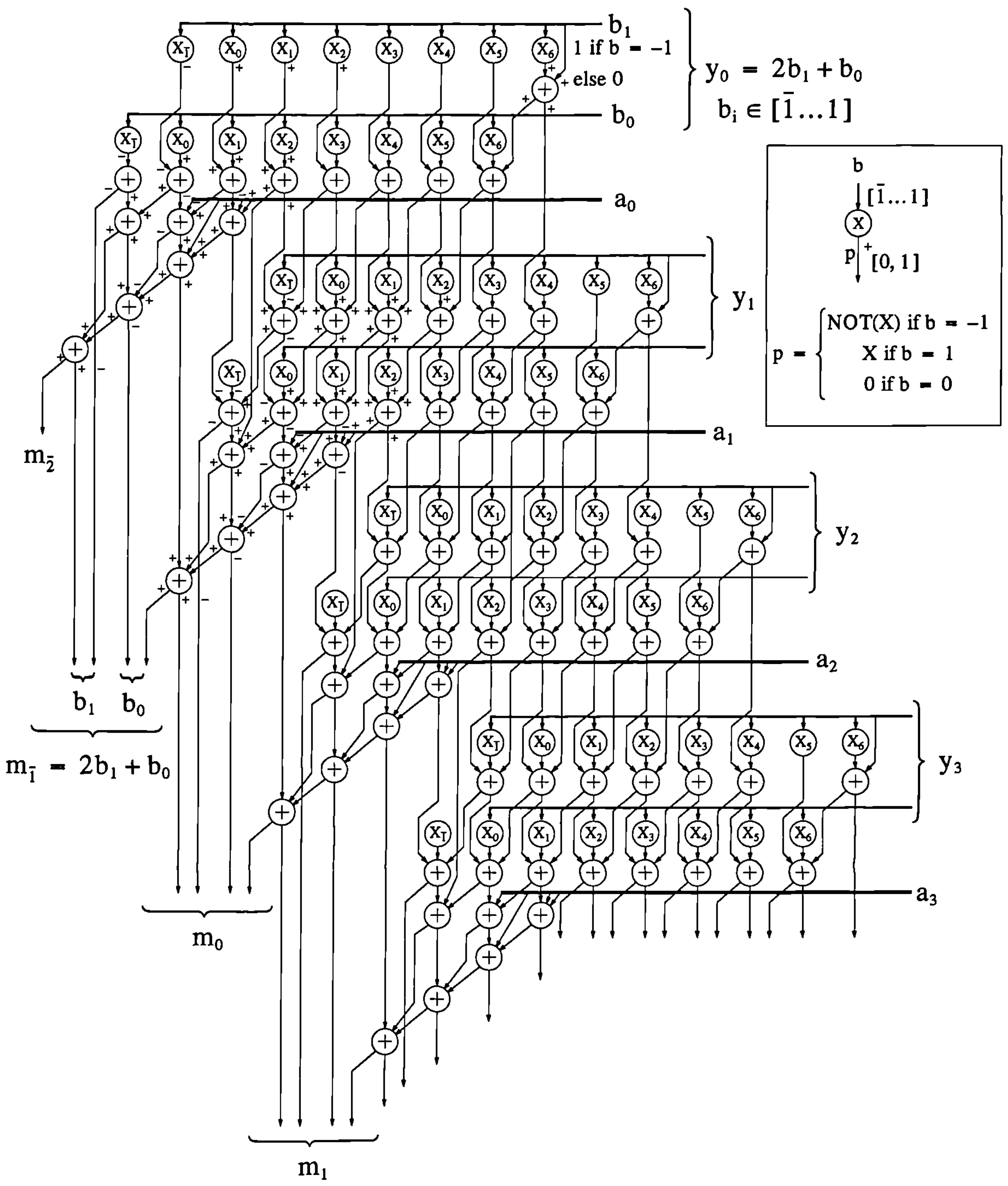


Figure 3.5 Type 2 architecture for a maximally redundant radix-4 multiplier-adder

Table 3.1 Compression and saturation cell functions

Input Overflow State	Input Digit	Output Digit	Output Overflow State
Positive Overflow	X	2	Positive Overflow
Possible Positive Overflow	0..2	$\bar{2}$	
	$\bar{1}$	$\bar{2}$	Possible Positive Overflow
	$\bar{2}$	$\bar{2}$	No Overflow
No Overflow	X	As Input	
Possible Negative Overflow	2	$\bar{2}$	
	1	$\bar{2}$	Possible Negative Overflow
	$\bar{2}$..0	$\bar{2}$	Negative Overflow
Negative Overflow	X	$\bar{2}$	

3.2.6 Comparison of Msdf Multipliers

Figure 3.6 shows the delay and area of the three msdf multiplier-adders as a function of wordlength. Figure 3.6 (a) shows the results when pipelining is applied between every row of the multiplier-adder, which results in a latency of 4 for the signed-binary case and 3 for the 2 radix-4 multiplier-adders. Figure 3.6 (b) shows the results when the level of pipelining is chosen to give a minimum latency of 2 clock cycles. In both cases the signed-binary multiplier-adder offers the lowest delay, but also the highest area. The MinR4 multiplier-adder is not much slower, but as expected achieves a much reduced area, giving a much better area-time product for both levels of pipelining.

Some of the area savings in the MinR4 have been achieved by the MinR4 coding of the partial sum and as a consequence a reduced number of latches are required to pipeline it over a carry-save representation. This technique of reducing the level of redundancy to reduce pipelining cost may be applied in areas of the circuit where speed is not critical, and may be of value in other operators such as digit recurrence dividers and CORDIC.

The area of all the msdf multiplier-adders is approximately dependent upon the square of the wordlength, and the delay is almost independent of it (note that the small but finite slope is due to limitations in the synthesis tool).

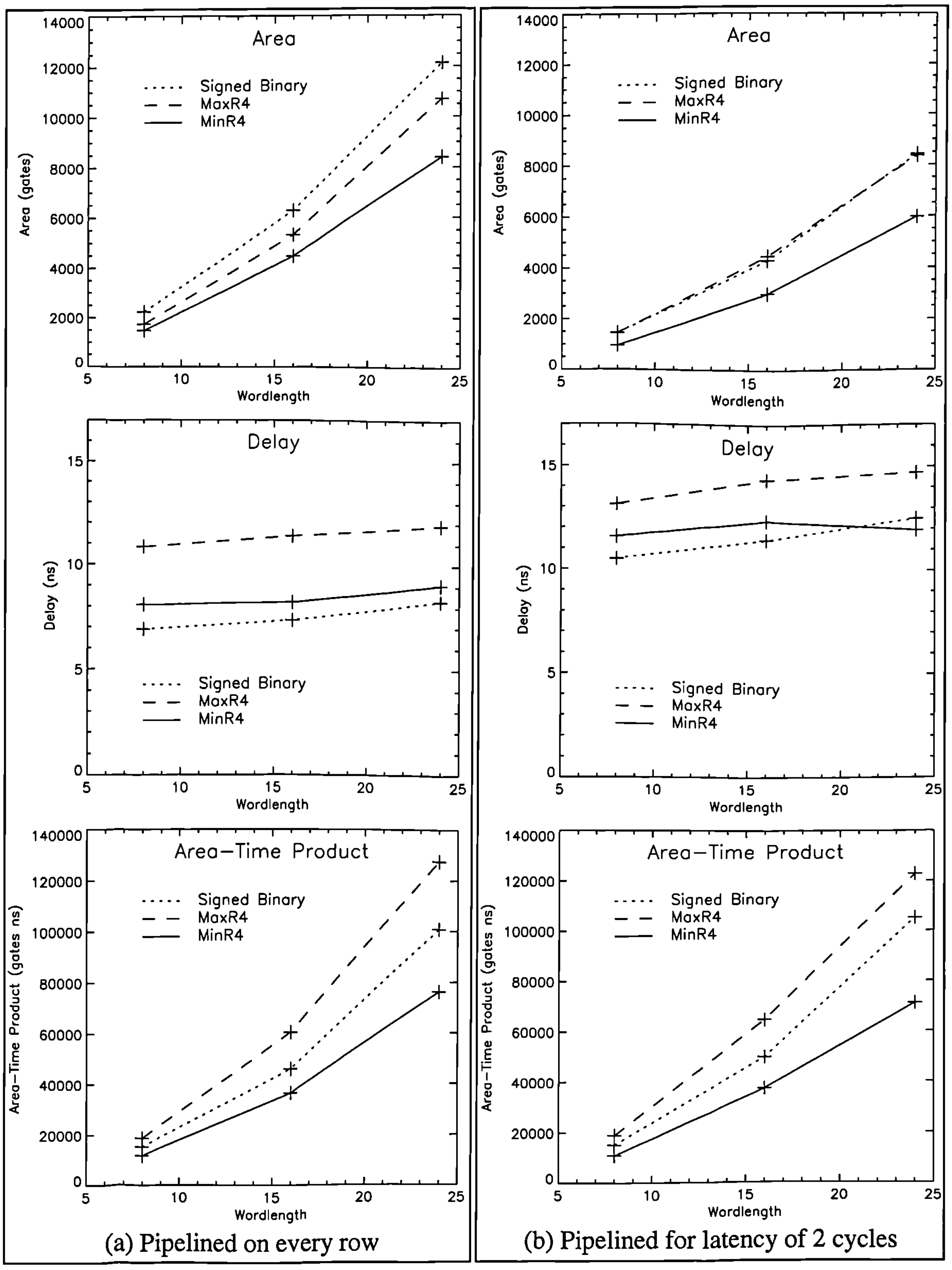


Figure 3.6 Delay and area comparison of msdf multipliers[1]

A low-latency msdf multiplier architecture based on the MinR4 digit-set was presented by Koppenhofer[Kopp93] for application in a decision-feedback equaliser (DFE). Essentially

1. The MaxR4 results were obtained using an architecture developed by McQuillan[McQu94b].

the algorithm employed is the same, although an analysis is not provided within the paper. There are some significant differences in the architecture worth noting. A carry-save representation is used within the body of the array rather than a MinR4 representation, which increases the number of pipelining latches required. A 4-bit representation is used for the MinR4 output digits which will also increase the cost of signal delays in the implementation of a filter. The additive input is not in an msdf format, so the multiplier cannot be cascaded, even though this is required by the 2nd-order DFE architectures presented in the paper.

Recently it has been shown that the pipelining can be applied to msdf-multipliers to obtain single-cycle latency[McGo95]. The delay of such an arrangement will be greater than that given for the 2-cycle latency case, and the relative benefits of each architecture should stay the same.

3.3 Tree-Based Multipliers

3.3.1 Background

Msdf arithmetic offers one approach for obtaining low-latency which has been applied very effectively in a simple recursive system to increase the sample-rate. In more complex systems the latency of the operations required to convert to and from the msdf representation can be significant and costly in terms of pipelining latches required to maintain correct timing elsewhere in the system. To avoid this, Montuschi[Mont93] suggested that the msdf array multiplier be used without msdf data skewing. In effect, this was achieved by removing all the pipelining latches from the body of the array and presenting the inputs in a parallel form. Unfortunately, due to the array structure there is still a relatively long carry chain. In conventional multiplication, the delay of an array multiplier is reduced by using adder-trees, which changes the dependence of the delay upon wordlength from a linear one to a logarithmic one. Although the delay of the tree is not independent of the wordlength, as is the case in msdf arithmetic, it can be relatively small, particularly for low and medium wordlengths, and so the approach has the potential to offer low-latency without the inconvenience of an msdf format.

One disadvantage of tree multipliers is that they are much less regular than array multipliers, and so are more difficult to layout. However, in recent years there has been a rapid improvement in design tools, and it is now possible to automatically synthesise, place and route irregular designs such as tree multipliers.

Wallace[Wall64] first proposed the use of a tree of full-adders to accumulate partial-products and reduced the number of adders in the critical path to produce the product. Later Dadda [Dadd65] saw this as a special case of trees composed of parallel (n, m) counters, which would count n input bits to give an m-bit binary output. This has been extended by Stenzel *et al.*[Sten77] to the case of generalised counters which take several weighted columns of bits and produce their weighted sum. Alternatively, *compressors* can be used to provide partial addition of bits. One very popular example is the 4:2 compressor, which generates 2 outputs (sum and a carry) from 4 input bits[Wein81]. It also accepts a carry-in and generates a carry-out which is independent of it. This compressor has been widely used in multiplier tree implementations.

In this chapter it is proposed that adder-trees be constructed from simple full-adders for the reasons given in section 2.3, i.e. the full-adder is a good circuit primitive when available as a standard-cell component. Indeed, Oklobzija *et al.* [Oklo96] have demonstrated, that by careful choice of full-adder interconnection, very high performance multipliers can be constructed from full-adders.

3.3.2 Tree Multipliers with Single Redundant Input

Briggs and Matula [Brig93] presented a redundant multiplier-adder based on a tree of signed-binary adders. This enabled the signed-binary output to be fed back as either the additive input or the multiplicand (as in the latter case the signed-binary partial-products formed between digits of the multiplier and the signed-binary multiplicand could be summed by the signed-binary adder-tree). The multiplier was in 2's complement form, and so ultimately the design performed the same function as the msdf multiplier-adder described in this chapter. It achieved single-cycle latency for a 17-bit multiplier and was used to implement multiply, divide, square-root and transcendental functions using a series of multiplications for a floating-point co-processor chip. The 17-bit, non-redundant multiplier input was recoded into the minimally-redundant, radix-8 digit-set $[\bar{4}\ldots4]_8$ to reduce the number of partial-products by a factor of 3 and the tree depth to only three signed-binary adders. The radix-8 digit-set required that partial-products be formed with the digits $\bar{3}$ and 3. This is obtained by adding multiples of 1 and 2 times the multiplicand and distributing the result to the partial-product generators. In this case, the multiplier is in a redundant form so the addition may be performed very quickly, without significant carry-propagation using a redundant adder.

A disadvantage of using signed-binary adders is that they are approximately twice the size of a conventional full-adder. They can be avoided by using a conventional tree of full-adders and feeding back the carry-save output and recoding it, so that it is applied as the multiplier input. A generalised adder can be used to perform the conversion from carry-save to a MinR4 representation. Such a scheme, at a block diagram level, was recently proposed by Lyu and Matula[Lyu95], and is, in effect, the tree version of the MinR4 multiplier presented earlier. A practical realisation of this architecture is now developed.

One critical aspect of the multiplier design not considered by Lyu and Matula was the need to avoid redundancy overflow and maintain a fixed wordlength for the feedback variable. This can be done, as in the msdf multipliers presented earlier, and will only incur a small delay. A signed-binary representation for the output is also advantageous, as this provides unbiased truncation error without the need for rounding, and as a consequence the lower half of the partial-product need not be calculated. As will be shown later, this leads to a significant saving in area over the conventional approach. It is also shown how a signed-binary output may be generated directly from a conventional adder-tree by using the generalised interpretation of the full-adder. The architecture of the proposed multiplier is presented in Figure 3.7.

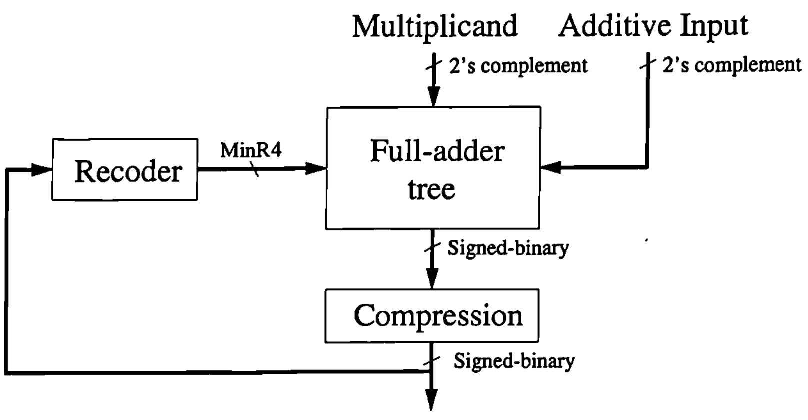


Figure 3.7 Tree-based redundant multiplier

Figure 3.8 shows an adder tree suitable for adding partial products. Each slice of the tree adds a number of bits of equal weight. The inputs enter the array from above and the result is obtained at the bottom. A view of the inputs from above is shown in the right-hand side of the figure as a rectangular grid. The partial-products are entered into the tree via the inputs contained within the parallelogram, and the other inputs are set to zero. Describing the tree with a rectangular array of inputs rather than a parallelogram is much simpler in VHDL, and does not lead to extra hardware, as the circuit optimisation tool will eliminate all adders with fixed inputs.

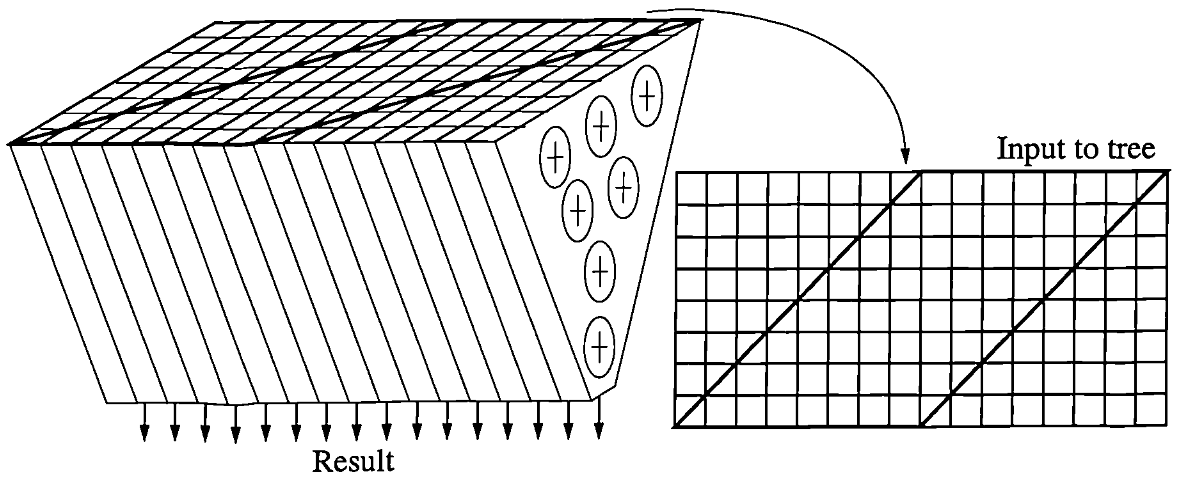


Figure 3.8 Simplified adder-tree

A conventional adder tree provides a carry-save output. However, a signed-binary output can be obtained using a generalised interpretation of the full-adders within the tree and by noting that the sign of the tree outputs reflects the signs associated with its inputs. If only one bit within each bit-slice (i.e. column) of the tree input is negatively-weighted then the sum bit of the output is also negatively weighted and the sum and carry bits form a signed-binary output. This principle is demonstrated for a variety of adders in Figure 3.9.

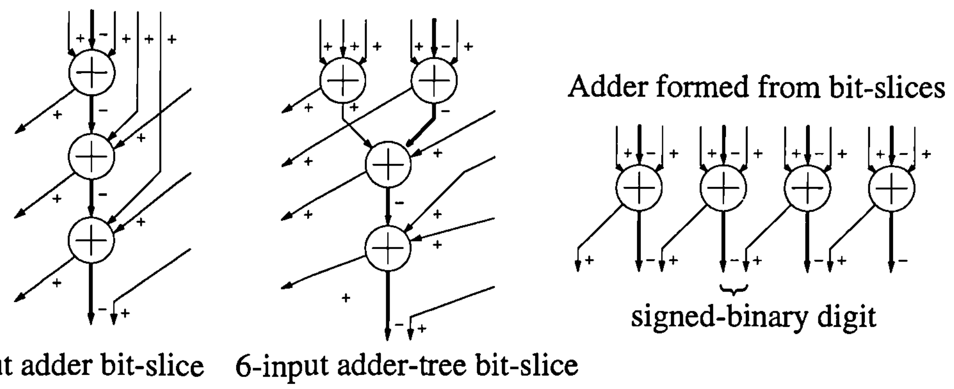


Figure 3.9 Obtaining signed-binary output from adders

Figure 3.10 shows an array of inputs to a tree which will generate a signed-binary output. This has been arranged by ensuring one bit in each column of the adder tree inputs is negatively-weighted. These negatively-weighted inputs are indicated using a '-' sign. Positively weighted inputs are indicated using a '+' sign, unless they are unused, in which case they are left blank. As discussed in Chapter 2, the input or output of an adder may be given a negative weight by inverting its bit-level coding (there is no need to change the logic of the adder-tree itself).

As in a conventional tree multiplier, the partial-products are entered in a 2's complement form in rows denoted (a) in Figure 3.10. The number of partial-products has been halved by recoding the multiplier. The negative weight of the msb of each 2's complement partial-product is accommodated by negatively weighting the associated tree input. An extra row of inputs with positive weight are used (row (b) in Figure 3.10) to enter the bits associated with the +1 operation required when a 2's complement partial product is negated. To ensure that each column has one negatively-weighted input, a final row of inputs has been included (and denoted (c) in Figure 3.10). These inputs are arithmetically zero, so must be set to logic 1 (see section 2.3.3). Therefore, in effect, a signed-binary output is obtained from the tree by simply adding a constant term. The generalised interpretation of full-adders has been a powerful tool in establishing this fact.

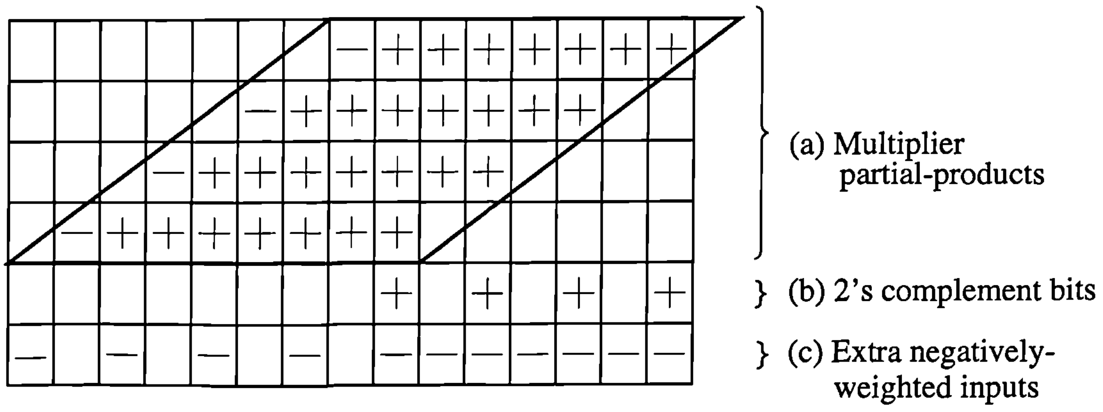
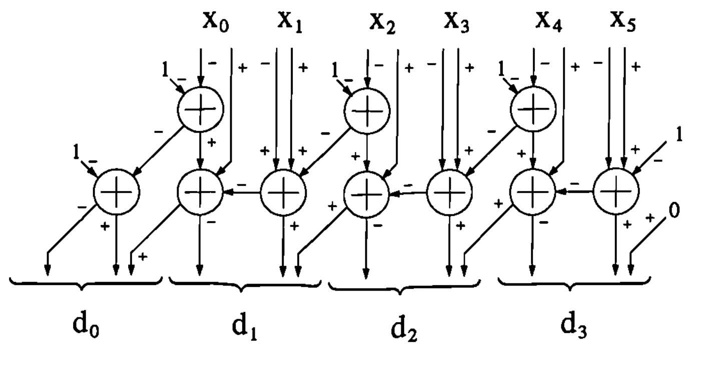

(a) Multiplier partial-products
(b) 2's complement bits
(c) Extra negatively-weighted inputs

Figure 3.10 Obtaining signed-binary from a conventional adder-tree

The signed-binary output from the tree, when fed back, forms the multiplier input. To reduce the number of partial-products, the input is recoded into a MinR4 representation. Recoding is conventionally performed from binary using modified Booth's recoding. However, it is possible to recode from redundant representations, although the number of bits which must be examined to generate each output digit is likely to be greater than 3. An effective and convenient way of designing the recoder is to construct it from generalised full-adders, as shown in Figure 3.11.



$$X = \sum_{i=0} 2^{-i} x_i, \quad x_i \in \{\bar{1}\ldots 1\}$$

$$D = \sum_{i=0} 4^{-i} d_i, \quad d_i \in \{\bar{2}\ldots 2\}$$

Figure 3.11 Recoding from signed-binary to MinR4

3.3.3 Comparison of Tree-Based and Msdf Multipliers

In this section a comparison is made between the tree and the MinR4 msdf multipliers. Figure 3.12 presents the estimates of the delay and area of the multipliers obtained by circuit synthesis. The properties of an equivalent modified Booth's encoded multiplier, consisting of a tree followed by a carry-propagate adder, are also shown.

The tree based multiplier offers lower area and delay than the msdf multiplier at the wordlengths considered. The latency of the signed-binary tree is significantly less than that of the other two, principally because it requires only one clock cycle, whereas the msdf and conventional tree multipliers require two. However, the latency of the msdf multiplier is almost independent of wordlength, whereas tree multiplier delay will increase with wordlength, and at some point will be greater than that of the msdf multiplier.

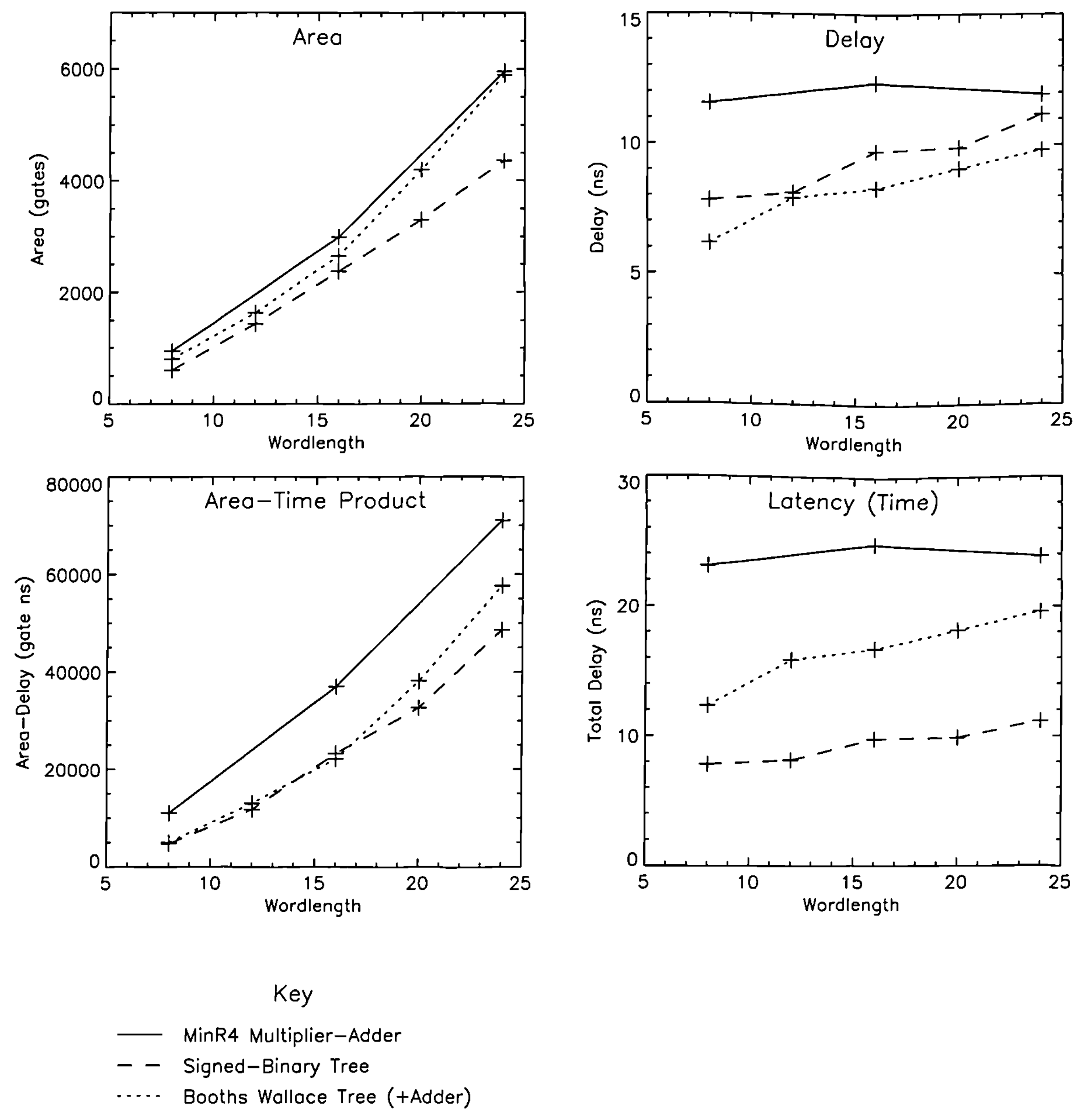


Figure 3.12 Speed and area comparison of tree-based, redundant multipliers

It should be noted that the functions of the multipliers are not identical. The tree multipliers considered here do not include an additive input, although incorporating it will only increase the delay and area by a small amount. More significantly, however, the msdf multiplier was designed for an IIR filtering application where the additive input is supplied in an msdf format, the magnitude of the coefficient may be as large as 2, and saturation is applied. These three factors significantly increase digit selection time over that of a multiplier where the magnitude of the coefficient is restricted to 1 and the additive input supplied in a parallel form. (The latter can be simply entered into the array via the first row of adders). Therefore, in other applications, the latency of the msdf maybe significantly reduced, although it will not be better than that of the signed-binary tree multiplier for the wordlengths considered here.

One important benefit of the msdf multiplier, is that saturation of the output may be implemented in an msdf manner. If employed on a tree multiplier it will incur a significant delay, as the whole result must be examined, and the latency will increase to that of the conventional tree multiplier.

Note that the signed-binary tree multiplier is smaller than the conventional one. This is because the signed-binary multiplier only needs to evaluate the most significant half of the partial-product sum. This more than compensates for the extra area of the recoding and compression circuits.

In conclusion, the choice of multiplier approach will depend upon the wordlength and whether saturation is required. If saturation is not required, the tree multiplier will provide the best solution for small and medium wordlengths (i.e. less than approximately 32-bits).

3.3.4 Tree Multipliers with Two Redundant Inputs

If both inputs to a multiplier are in a redundant form, it is necessary to design a multiplier where the multiplicand can take a redundant form. This will increase the size of the multiplier, as the tree will need to accommodate the extra bits associated with a redundant representation of the partial-products. In this section a multiplier architecture is presented where the multiplicand is recoded into a representation with reduced redundancy so that the number of partial-product bits and the area of the multiplier are reduced.

3.3.4.1 Algorithm

Figure 3.13 shows an example of a recoded redundant multiplier, where MinR4 recoding has been used on both the multiplier and multiplicand.

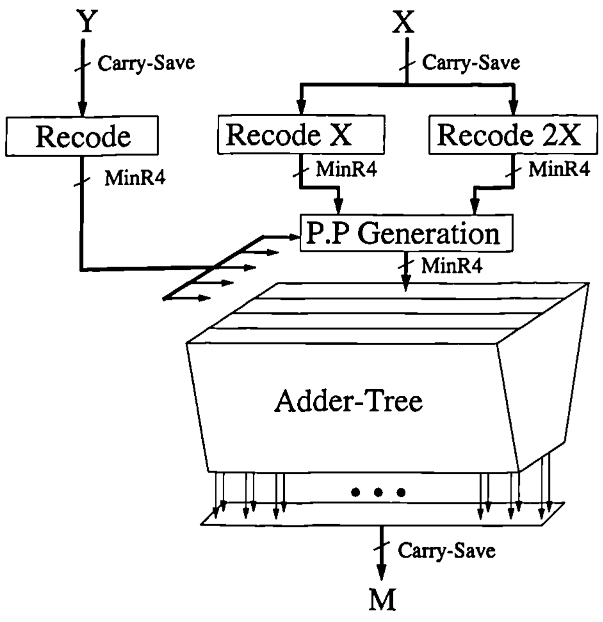


Figure 3.13 Block diagram of recoded, redundant multiplier

The partial-products are formed between the recoded multiplier, Y, and multiplicand, X, and are summed by a conventional tree of full-adders to give a carry-save result. Alternatively, the tree could be designed to generate a signed-binary result, using the approach presented in the previous section. The number of tree inputs is reduced by 25% using a MinR4 representation for the partial-products. The simplest way to achieve this is to precompute all the multiples of the multiplicand in a recoded form and distribute them to the partial-product generators, which select the appropriate multiple for the particular digit of Y. In fact, it is only necessary to distribute the positive multiples to the partial product generators, providing that the negative one can be easily obtained from them (as done in Figure 3.13). To achieve the recoding, generalised adders, similar to the one shown in Figure 3.11, can be used.

A MinR4 representation for the multiplier and multiplicand is one option, but there are other possibilities, and the next two sections consider more generally the issue of recoding the multiplicand and multiplier.

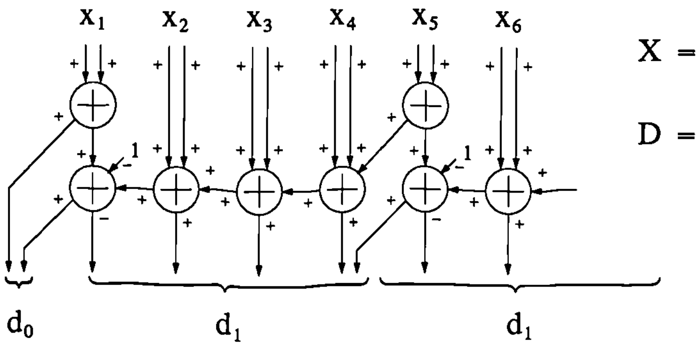
3.3.4.2 Recoding of Multiplicand X

The digit-set used to represent the multiplicand X should be chosen to satisfy the following criteria:

1. Products with negative multiplier digits should be easily obtained from the corresponding products with positive ones. In which case, only the positive products need be distributed, saving wiring and the logic required to form them.
2. The number of bits required to represent a redundant digit should be minimised.
3. The number of digits required to represent the multiplicand, X, should be minimised.
4. The recoding process should take no longer than the recoding of the multiplier, Y, so that it does not increase the delay of the multiplier.

The radix-r, minimally-redundant representation satisfies both (1) and (2). Point (3) can be addressed by maximising the radix, although this will increase the time to perform the recoding. Hence, the radix will be limited by the time available, as specified by (4).

For example, minimally redundant radix-16 (MinR16) can be obtained from carry-save using the generalised adder shown in Figure 3.14. In summary, as the radix is increased so the carry-chain becomes longer.



$$X = \sum_{i=1} 2^{-i}x_i, \quad x_i \in \{0\ldots2\}$$
$$D = \sum_{i=0} 16^{-i}d_i \quad d_i \in \{\bar{8}\ldots8\}$$

Figure 3.14 Recoding from carry-save to MinR16

Table 3.2 summarises the characteristics of a range of minimally redundant digit-sets. The bit-level coding shown is that obtained using similar recoding arrangements to those presented in Figure 3.14. The letter ‘p’ is used to represent a bit with positive weight and the letter ‘n’ to represent one with negative weight. Hence, for the case of MinR4 each radix-4 digit is represented by a three bits: two positive of equal significance, and one negative, which is positioned on their left and so has twice the significance of the positive ones (giving the digit range $[\bar{2}\ldots2]_4$). Note that the number of bits required to represent a number decreases as the radix is increased.

Table 3.2 Minimally redundant radix-r digit-sets

Name	Radix (r)	Digit-set	Proposed bit level coding (p: positive bit, n: negative bit)	Length of recoder carry-chain in terms of full-adders ($\log_2 r$)
MinR4	4	$[\bar{2}\ldots2]_4$	n p n p n p n p n p n p p p p p p p	2
MinR8	8	$[\bar{4}\ldots4]_8$	n p p n p p n p p n p p p p p p	3
MinR16	16	$[\bar{8}\ldots8]_{16}$	n p p p n p p p n p p p p p p	4

3.3.4.3 Recoding of Multiplier Y

One advantage of a redundant representation for the multiplicand is that it is possible to form multiples of the multiplicand, such as 3X, without significant carry-propagation using a redundant adder. Therefore, higher radices *can be used in the recoding of Y, as the time to* obtain the digit-multiples (y_iX) is shortened. The digit-set used for the multiplier Y should be chosen to:

1. Reduce the number of digits required to represent the multiplier, Y, and thereby reduce the number of partial-products which must be formed.
2. Simplify partial-product generation.
3. Reduce the number of product multiples of the multiplicand, X, which must be formed.

The MinR4 digit-set requires only two product multiples, X and $2X$ to be generated and distributed. If MinR8 is used the product multiples X, $2X$, $3X$ and $4X$ will be required. However, if MinR4 is used for the multiplicand coding, $4X$ can be obtained from the recoded X by a 2-bit shift to the left. Therefore, only X, $2X$ and $3X$ need be distributed.

3.3.4.4 Adder-Tree

The partial-products in their recoded form can be summed by a conventional tree by first converting them to 2's complement form. A minimally redundant number can be converted to 2's complement by first splitting the number into its positive and negative parts. The negative part is converted by first inverting the negatively-weighted digits (to remove the inverted logical coding) and applying the usual 2's complement procedure of inverting and adding 1 in the lsb position. For example, in the MinR4 digit-set, the negatively weighted bits form a word $0n0n0n0n0$ (where the subscripts of n, to indicated bit position, are not shown for clarity). Undoing the coding gives $0\bar{n}0\bar{n}0\bar{n}0\bar{n}0$ and taking the 2's complement provides $1n1n1n1n1+1$. This can be simplified to $0n0n0n0n0+101010110$, which is the original input plus a constant. The constant will be the same for all partial-products and their sum can be precalculated so that only one term need be entered in to the adder array. Ultimately, this term will also be eliminated by the circuit optimisation tool.

The result of the conversion can be merged with half of the positive bits, which in the MinR4 case yields

$$00p0p0p0p+0n0n0n0n0 = 0npnpnpnp \tag{3.7}$$

Therefore, two rows of tree inputs are required to add each partial-product. Table 3.3 shows the terms which must be added for other minimally redundant digit-sets. The third row is the constant term which must be accumulated for all partial-products and entered once into the array. Due to the recoding, the second row contains a proportion of logic 0s which increases with the size of the radix. The adders associated with these inputs will be eliminated by the optimisation tool, providing the saving in area which as been the goal of this technique.

Table 3.3 Coding of partial-product for adder-tree

Digit-set	Partial-product coded for adder-tree	Saving in tree area
MinR4	0 n p n p n p n p n p n p 0 0 p 0 p 0 p 0 p 0 p 0 p 1 0 1 0 1 0 1 0 1 0 1 1 0	25%
MinR8	0 n p p n p p n p p n p p 0 0 0 p 0 0 p 0 0 p 0 0 p 1 0 1 1 0 1 1 0 1 1 1 0 0	33%
MinR16	0 n p p p n p p p n p p p 0 0 0 0 p 0 0 0 p 0 0 0 p 1 0 1 1 1 0 1 1 1 1 0 0 0	38%

3.3.4.5 MinR4 Redundant Tree Multiplier

In this section the design of a redundant multiplier using a MinR4 representation for both the multiplier and multiplicand is considered.

To enable cascading of multipliers or results to be fed back, the multiplier should accept its inputs in the same representation as its output. The output of a conventional tree is not strictly carry-save as it has two negatively-weighted bits in the msd position. These arise from the addition of partial-products which may be negative due to the negative digits of the recoded multiplier. To perform the recoding from this representation into MinR4 the generalised adder shown in Figure 3.15 may be used. It has been assumed here that $0 \leq X, Y, M < 1$, and consequently the two negatively-weighted bits are exclusive. This fact has been used to simplify the circuit.

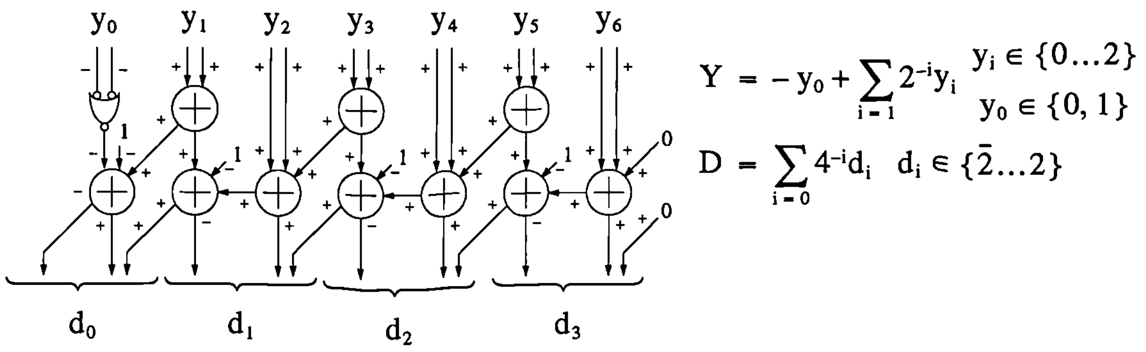

y_0
y_1
y_2
y_3
y_4
y_5
y_6
d_0
d_1
d_2
d_3
$Y = -y_0 + \sum_{i=1} 2^{-i} y_i$ $y_i \in \{0...2\}$ $y_0 \in \{0, 1\}$
$D = \sum_{i=0} 4^{-i} d_i$ $d_i \in \{\bar{2}...2\}$

Figure 3.15 Recoding from carry-save to MinR4

The recoder shown in Figure 3.15 is suitable for recoding the multiplier Y and the multiplicand multiple X. To recode the other multiple, i.e. 2X, the adder in Figure 3.16 can be used. Note that for an input less than one, no carry will be generated from the adder in the most significant bit position, so its carry output may be discarded.

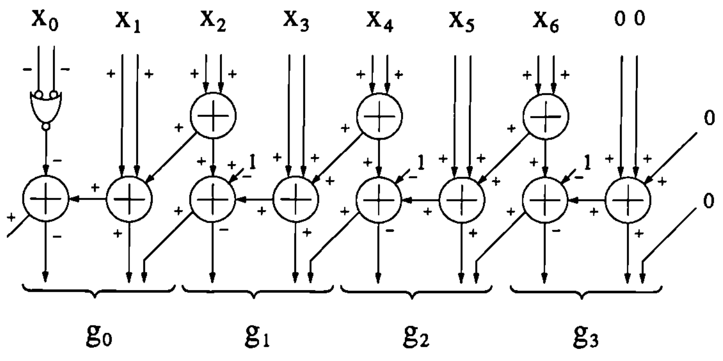
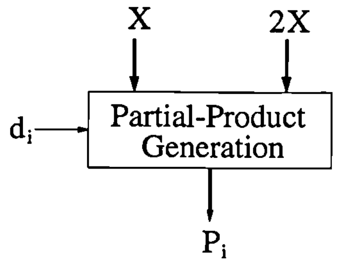


Figure 3.16 Recoding of 2X from carry-save to MinR4

The partial-products can be formed using the network shown in Figure 3.17.



$$P_i = \begin{cases} \mathrm{NOT}(2X_i) \text{ if } d_i = \bar{2} \\ \mathrm{NOT}(X_i) \text{ if } d_i = \bar{1} \\ 0 \text{ if } d_i = 0 \\ X_i \text{ if } d_i = 1 \\ 2X \text{ if } d_i = 2 \end{cases}$$

Figure 3.17 Partial-product generation

3.3.5 Comparison of Tree-Based Redundant Multipliers

Figure 3.18 presents estimates of the delay and area of a number of tree-based redundant multipliers. For comparison purposes the results for a conventional tree, preceded by carry-look-ahead adders (CLAs) to reduce the redundant carry-save input to non-redundant binary, have also been shown.

The results for three redundant adder-tree multipliers are presented in each graph. The first uses only a MinR4 representation in the multiplier. As expected this multiplier is about twice the size of the conventional non-redundant multiplier. It also takes twice as long, due to the more complex multiplier recoding and the larger number of product terms to be summed. The second redundant multiplier uses MinR4 on the multiplier and multiplicand, and achieves a saving of up to 25% in area and a small improvement in speed. The third redundant multiplier extends the recoding of the multiplier to MinR8 and further reduces the number of multiplier digits and associated partial-products. Unfortunately, this offers no further reduction in area over the previous multiplier, due to the relatively low multiplier wordlengths considered.

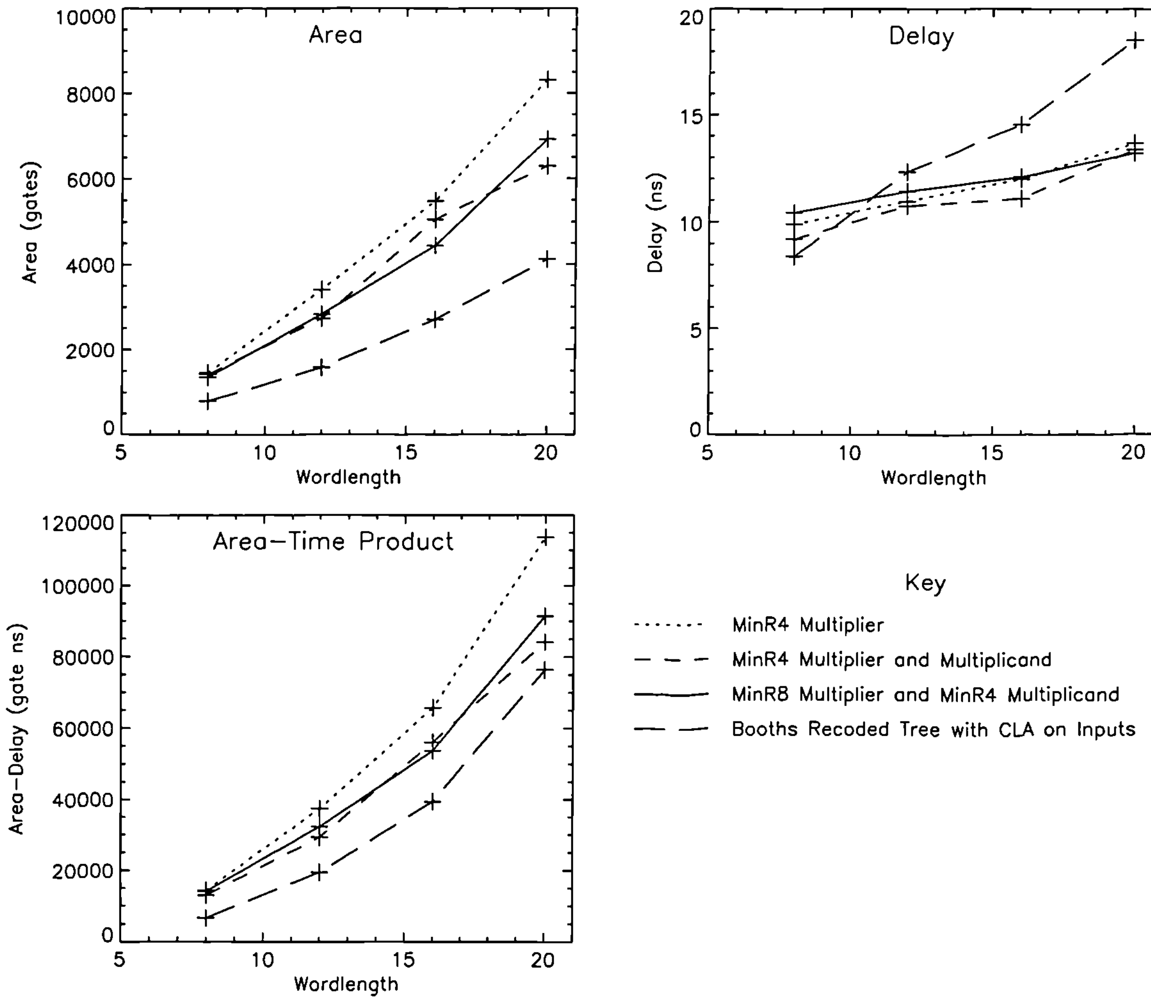


Figure 3.18 Speed and area comparison of tree-based redundant multipliers

In conclusion, this work has demonstrated that for the multiplication of two redundant numbers there is significant value in recoding the multiplicand. For larger wordlengths, it may also be beneficial to increase the radix of the recoding of the multiplier, and perhaps that of the multiplicand.

Flynn *et al.* in [Flyn95] describe a non-redundant multiplier which uses a redundant representation for the multiplicand so that higher-radix recoding of the multiplier may be used to reduce the number of partial-products. They employ a representation which is minimally redundant, but uses only positive digits. A reduction of 15%-20% in area for a non-redundant multiplier, without loss of performance.

3.4 Discussion of Multiplier Results

In this chapter new architectures for two forms of redundant multiplier have been presented. These use the minimally redundant digit-set in a number of ways to achieve significant reductions in area with little loss in speed. A conclusion, therefore, is that redundancy can be employed to obtain low-latency multiplication, and its cost can be significantly reduced by using a level of redundancy appropriate to the speed requirements of the circuit it which it is used.

Msdf multipliers offer highly regular architectures, and a latency which is almost constant with wordlength. For these reasons, the approach has been very attractive in the past for all but the smallest of multipliers. However, recent developments in circuit CAD tools allow irregular circuits such as the Wallace-tree to be realised just as easily as regular ones. Furthermore, the results presented in this chapter indicate that for medium wordlengths (i.e. 16- to 32-bits), where saturation of the output is not required, a redundant multiplier based upon a tree offers lower latency. Indeed, for these wordlengths the latency of a conventional tree multiplier using a carry-propagate adder is similar to that of the msdf multiplier. The conventional and redundant tree multipliers are smaller than the msdf multiplier and avoid the difficulties and overheads of a skewed data format. However, for larger wordlengths and situations where saturation is required, the msdf multiplier will still be of value.

It is clear that the cost of performing multiplication of two redundant numbers is high, requiring an increase in area of over 50% of that of non-redundant multiplication or multiplication by one redundant number. As power consumption is approximately proportional to area (see section 8.6.2) the redundant multiplication is expensive, and should only be used when the requirements for low-latency can justify it.

Chapter 4 High-Throughput, Low-Latency Dividers

4.1 Introduction

Recent VLSI implementations of dividers have considerably higher latency than those of multipliers, typically by a factor of 4 to 30. Research efforts have generally focused on producing fast multipliers, motivated by the view that division occurs relatively infrequently compared to multiplication. In particular, many common DSP operations do not require division, and those that do, use relatively few when compared with the total number of computations. Even so, the impact of a small number of division operations on the system can still be significant for both DSP and general purpose computing[Ober94].

For example, if the divide is present within a recursive loop, the sample-rate of a system will be limited by its high latency. Also, in a parallel array implementation of an algorithm, the timing of the divider employed within one processor may have implications in the synchronisation of data throughout the whole array. In a pipelined system, this may lead to the introduction of many extra latches to schedule the data.

It will be shown in Chapter 6 that the Givens rotation can be reformulated without the divide operation, but this leads to a doubling of the number of multiply operations and an increase in the dynamic range of variables, which must be accommodated by introducing scaling operations. Consequently, an algorithm employing division is used, and the objective of this chapter is to examine ways in which it may be performed efficiently, and with low latency, to minimise its impact on the parallel array implementations considered in Chapter 8.

This chapter proceeds with an overview of the available division methods. This is followed by a detailed examination of three more promising approaches, for which area and speed estimates are presented, and concludes with a comparison and discussion of them.

4.2 Overview of Division

Division approaches can be classified into the following three methods:

- Digit recurrence
- Multiplier-based approaches
- Variable latency algorithms

Digit recurrence methods: These are based on the conventional pencil-and-paper method, where multiples of the divisor are subtracted from the dividend to reduce it to zero. The intermediate quantity is known as the partial remainder, and the divisor multiples form the digits of the quotient. Only one digit is produced per iteration, so convergence to the solution is only linear, but the implementation is simple and the partial remainder may be used for exact rounding of the quotient (i.e. to produce the same value as if a quotient was calculated to infinite precision and then rounded).

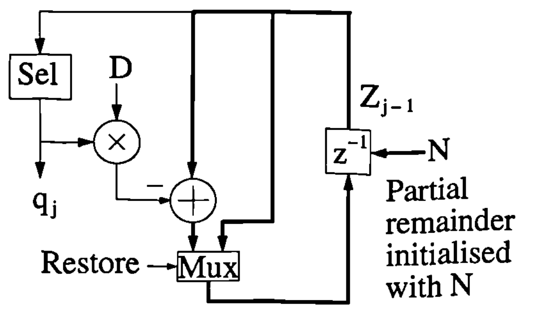
Multiplier-based methods: These methods start with an approximation of the quotient which is refined using a series of multiplications. *Newton-Raphson*, *series expansion* and *convergence division* are considered to be part of this class. Convergence to the result can be quadratic or more rapid[Ito95]. Alternatively, it may be made linear so that the multiplier wordlength can be fixed throughout the calculation[Brig93]. The principal disadvantage of the multiplier-based approach is that the remainder is not directly available, and rounding is not straightforward. (Either twice the number of quotient digits must be produced to enable correct rounding, or further operations must be performed to calculate the remainder.) The application of the divider in this thesis does not require exact rounding, only that the error is unbiased. Hence, the convergence method offers an opportunity to achieve low-latency division.

Variable latency methods: Considerable savings in latency may be achieved by allowing it to be variable. Self-timed logic can be used to give average timing figures much less than the worst-case values. Also, implementations can be designed to take advantage of short iterations, which occur in a digit recurrence algorithm when the quotient digit is zero[Mont91].

Variable latency can be accommodated in general purpose processor designs and asynchronous systems, but is of little use in a synchronous parallel processor, as adopted later in the thesis. Therefore, only the digit recurrence and convergence approaches are considered any further. For a more detailed overview of the approaches to division the reader is referred to a report by Oberman and Flynn[Ober95].

4.3 Digit Recurrence Methods

Division by the digit recurrence method is performed using three steps on each iteration as shown in Figure 4.1.



```
$Z_0 = N$
FOR $j = 1$ TO n DO
    $q_j = Sel(Z_{j-1})$
    $Z_j = rZ_{j-1} - q_jD$
    $Z_j = \begin{cases} Z_j & \text{if restore} = 0 \\ rZ_{j-1} & \text{if restore} = 1 \end{cases}$
    $Q_j = Q_{j-1} + r^{-j}q_j$
END FOR
```

Figure 4.1 Division using digit recurrence

The division starts with the partial remainder initialised to the dividend N. On each iteration, a digit of the quotient q_j is selected based upon the value of the partial remainder Z_{j-1}. This digit-multiple of the divisor, q_jD, is formed and subtracted from the partial remainder. If the partial remainder goes negative, the original *restoring algorithm* uses a fourth step to restore it to the previous partial remainder. This takes time and is avoided in the *non-restoring algorithm* by allowing the quotient digits to take on negative values by using the digits $\bar{1}$ and 1.

A redundant representation can be used for the partial remainder to avoid carry-propagation in the addition. To exploit this, Sweeney, Robertson and Tocher independently developed the SRT algorithm [Robe58] (a name obtained from the initials of the three researchers). A redundant signed-binary representation is used for the quotient, which enables the result digit selection to be based on only a few digits of the partial remainder. Subsequently, higher radix redundant representations have been employed with the aim of achieving higher throughput.

The digit recurrence is relatively simple for small digit sizes. Increasing the digit size reduces the number of iterations, but results in an exponential increase in the complexity of the digit selection circuitry, because greater numbers of remainder digits must be examined[Burg95]. Consequently, only radix-2 or radix-4 algorithms tend to be used in practice, and the number of iterations required is therefore high.

Three significant developments have occurred since the introduction of the SRT algorithm. These are prescaling of the divisor, overlapping of the steps within an iteration, and overlapping of the iterations. They are now considered in more detail in the following sections.

4.3.1 Prescaling of the Divisor

Prescaling has been exploited by Svoboda[Svob63] in a radix-10 divider and more recently by Ercegovac in radix-2[Erce89b] and radix-4[Erce90] designs to restrict the range of the divisor so that result digit selection logic is simplified. Prescaling is performed on both the divisor and dividend before division, and requires carry-propagate adders to produce non-redundant results so that simple adders can be used within the divider. The prescaling will increase the latency by one or more cycles, but enables quotient-digit selection time to be significantly reduced.

4.3.2 Overlapping of Division Steps and Iterations

Overlapping of the quotient-digit selection and remainder computation was achieved by Burgess[Burg91] for radix-2 arithmetic. McQuillan has generalised the solution to other radices using the *modified SRT* algorithm for division and square-root[McQu94a]. This solution also uses prescaling to achieve very simple digit-selection logic. This is considered to be an important algorithm, and the radix-2 implementation of the modified SRT algorithm is the first of the division algorithms examined later in this chapter.

Overlapping the iterations of the division enables groups of quotient digits to be determined more rapidly. Taylor proposed a radix-16 divider which overlapped the quotient digit selection of four radix-2 SRT stages to obtain a radix-16 digit of the quotient[Tayl85]. For example, overlapping of two stages is accomplished in the following way. The first stage forms all possible outcomes of the most-significant part of the partial remainder *while* the quotient digit is selected. The second stage determines the second quotient digit for each partial remainder. Once the quotient digit from the first stage is known it is used to select the correct quotient digit from the second stage.

Prabhu[Prab95] achieves a further speed increase in a radix-8 implementation of SRT by also overlapping the partial remainder computation for each radix-2 stage. This design was produced for the UltraSparc processor, and using a full custom layout and 0.5μm triple-metal CMOS technology, a single radix-8 stage was implemented producing radix-8 digits at 167MHz. This is a high rate for the technology, and may offer improvements over the modified SRT algorithm, so this forms the second division technique examined in this chapter.

Overlapping of the quotient-digit selection requires the parallel calculation of only the most significant part of the partial remainder, of sufficient accuracy to enable speculation of the digits in later stages. Once the quotient digits are known, it is necessary to calculate the rest of the partial remainder. This delay may be avoided by speculating on the whole partial remainder, but this is expensive. In Prabhu's implementation this is acceptable, as only a single radix-8 stage is implemented, which is re-used to produce the require quotient precision. This is possible in the UltraSparc general purpose processor as low-latency is the objective, and a throughput of 1 double-precision divide every 22 cycles is acceptable. High-throughput division is required in this thesis, needing all stages to be realised, and resulting in an excessive area. To overcome this, a modification of Prabhu's architecture has been proposed to significantly reduce the hardware, while maintaining the high speed of the approach. Firstly, the implementation of a divider using the radix-2 modified SRT algorithm is presented.

4.4 Radix-2 Modified SRT

4.4.1 Modified SRT Algorithm

The modified SRT algorithm was developed by McQuillan[McQu94a] in an effort to remove the sequential dependence between the quotient digit selection and the partial remainder updating operations. The modified SRT recurrence is:

$$\begin{aligned} T &= 2Z_{j-1} + \alpha D \\ &\text{where} \quad \alpha = \begin{cases} \bar{1} & \text{if } q \in \{1, 0\} \\ 1 & \text{if } q \in \{\bar{1}, 0\} \end{cases} \\ Z_j &= \begin{cases} T & \text{if } |q| = 1 \\ 2Z_{j-1} & \text{if } q = 0 \end{cases} \end{aligned} \tag{4.1}$$

A coarse selection is made to determine whether the quotient digit is in one of the digit-pairs $\{0, 1\}$ or $\{\bar{1}, 0\}$. The partial remainder is calculated for both digits in the selected pair. One of the digits is always a 0, so in either case only one new partial remainder needs to be calculated. Whilst this occurs, the quotient digit is determined more precisely. Once obtained, a multiplexer is used to select the correct partial remainder. In this last respect the algorithm is of the restoring kind.

To simplify the result digit selection, prescaling is performed to restrict the range of the divisor to $\frac{1}{2} \le D < \frac{3}{4}$. This is achieved by scaling the divisor by 0.75 when $D \ge \frac{3}{4}$, by adding $\frac{1}{2}D$ to $\frac{1}{4}D$. The same scaling is applied to the dividend to maintain their ratio. A carry-propagate adder is needed to ensure a non-redundant representation so that simple adders may be used for partial remainder calculation in the divider.

4.4.2 Modified SRT Divider Architecture

The architecture for the radix-2 modified SRT algorithm is shown in Figure 4.2. The selection cell is required to generate 3 signals. The coarse selection signal α is generated by inverting one bit of the partial remainder, and the restore signal is generated from only 2 digits. The third signal, compress, is required to detect overflow of the redundant residual and signal the type 2 adder cells to remove it.

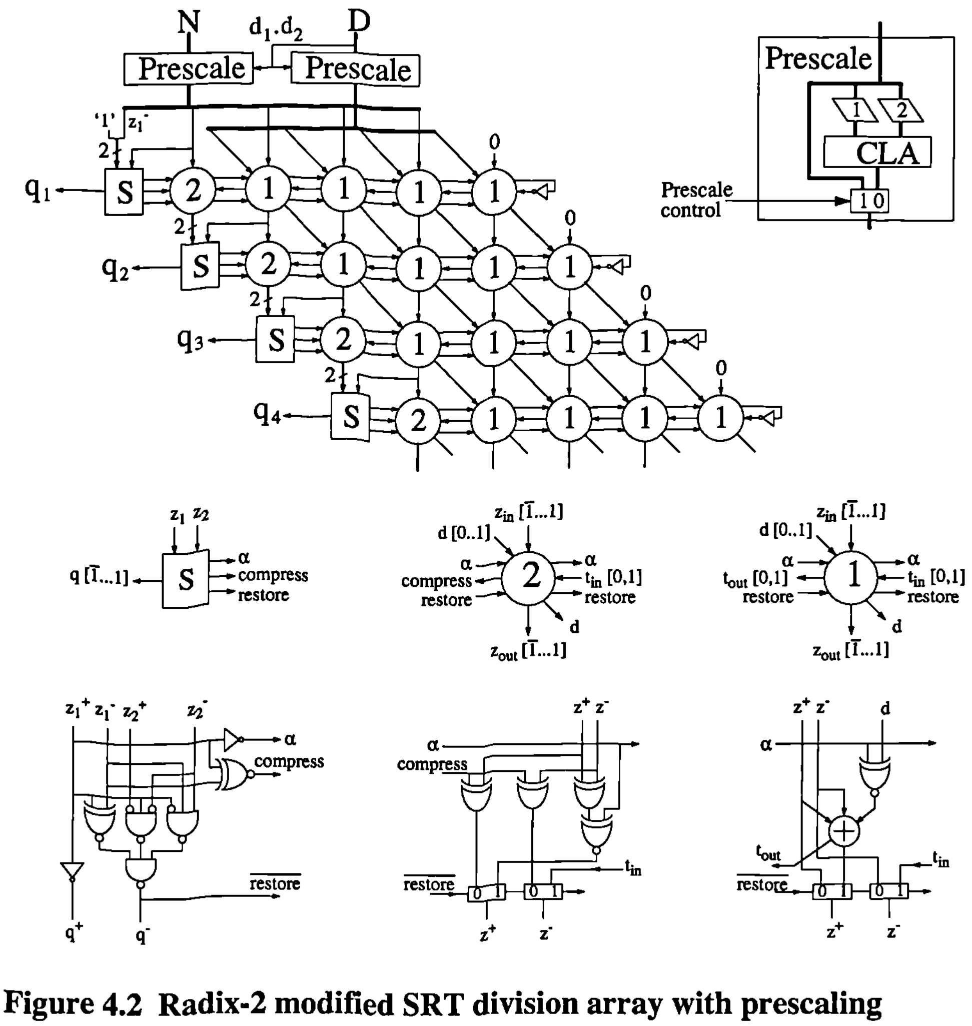


Figure 4.2 Radix-2 modified SRT division array with prescaling

Figure 4.3 shows the gate count and delay of the modified SRT divider, obtained by circuit synthesis, for a range wordlengths and levels of pipelining. The circuit includes prescaling and on-the-fly conversion circuitry (the latter by Knowles as described in Chapter 2), but neither contributes towards the delays shown, only the latency. As presented in the figure, the delay and area-time products are improved by increasing the level of pipelining. Values have not been obtained for the pipelining of every row due to the high synthesis times. It is anticipated that this level of pipelining would give a small delay reduction, but an increase in the area-time product.

As expected, the area of the divider has a square-law dependence upon wordlength. The delay should be independent of it, but increases slowly with wordlength due to the increased loading of gates driving signals along rows, and the increased complexity of the circuit optimisation task. Latency is increased by increased levels of pipelining. The latency in terms of clock cycles is given in brackets on the graph.

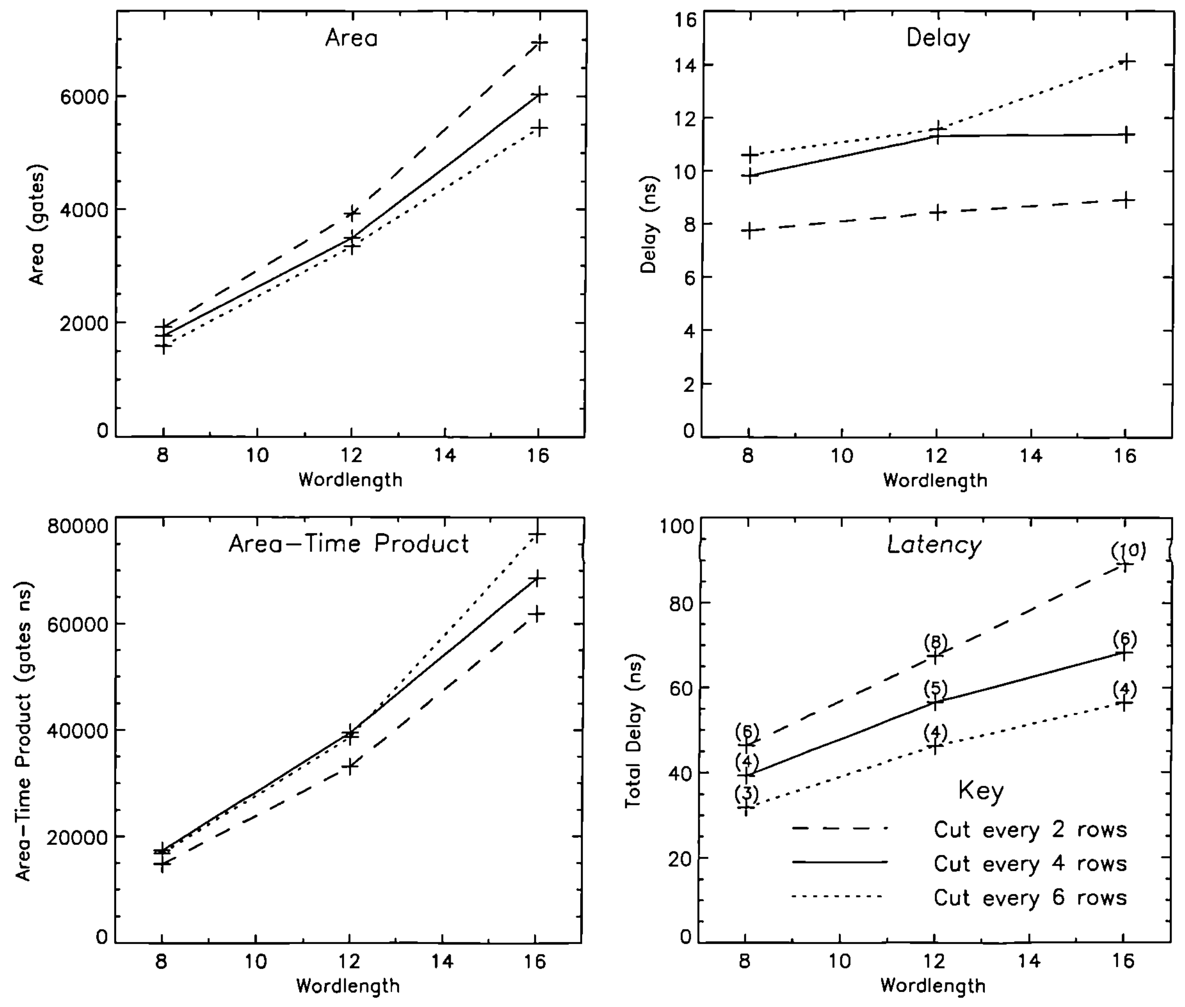


Figure 4.3 Area and speed of modified SRT divider

The improvement that the modified SRT algorithm offers over the basic SRT algorithm is shown in Figure 4.4. The modified SRT algorithm offers advantages in delay and latency, whilst maintaining similar area-time products. This is particularly the case for lower levels of pipelining (not shown on the graph), as the modified SRT algorithm allows better merging of rows.

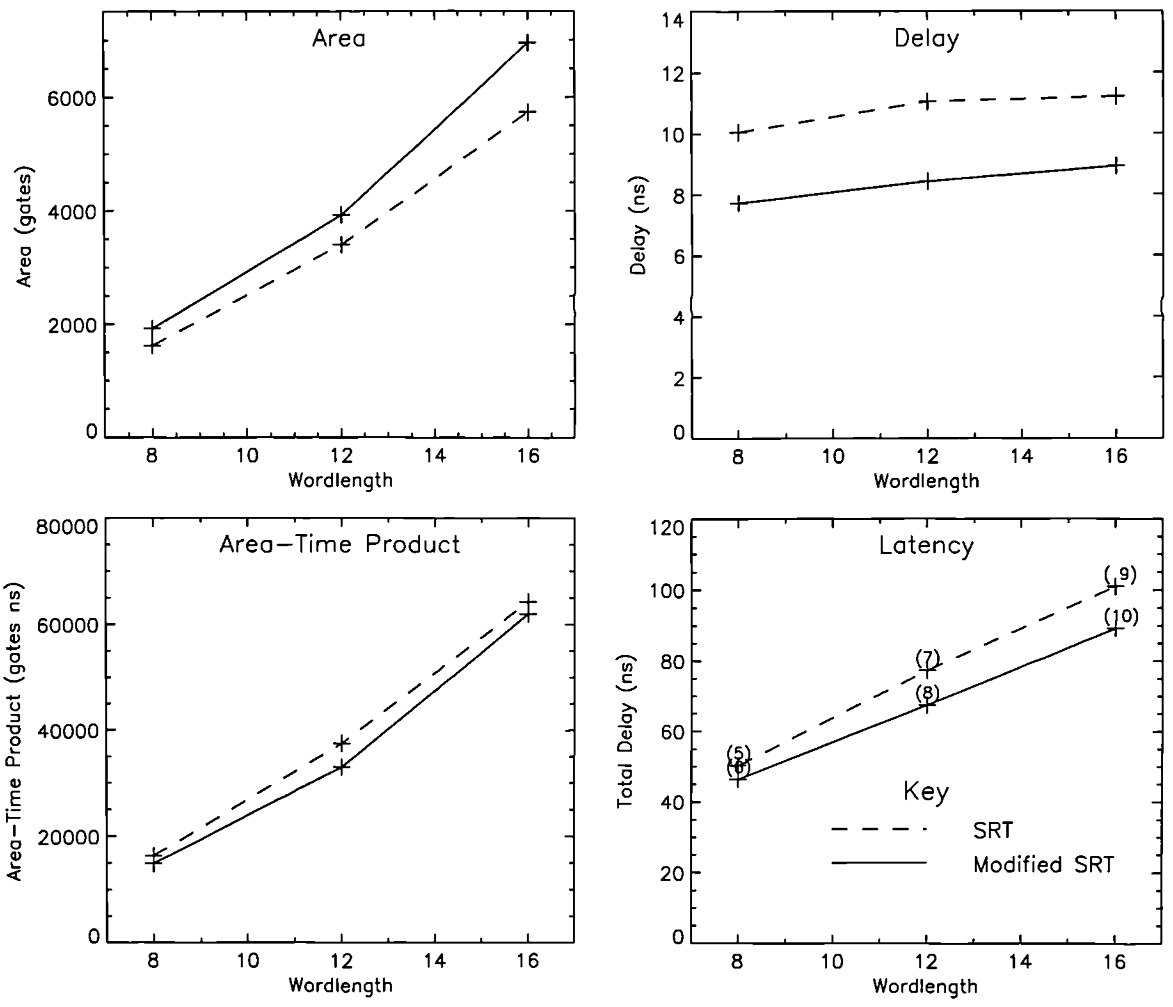


Figure 4.4 Comparison between SRT and modified SRT algorithms

4.5 Speculative SRT

Consider now the radix-8 overlapped SRT divider presented by Prabhu [Prab95]. This uses the radix-2 quotient digits from three rows of the radix-2 SRT algorithm to deliver one radix-8 quotient digit per iteration. Whilst the first of the radix-2 digits is being determined, the most significant bits of the partial remainders for all possible combinations of the first and second quotient digits are calculated and examined to speculate on the outcome of the second and third digits. This requires that seven partial remainders be calculated in parallel as shown in Figure 4.5.

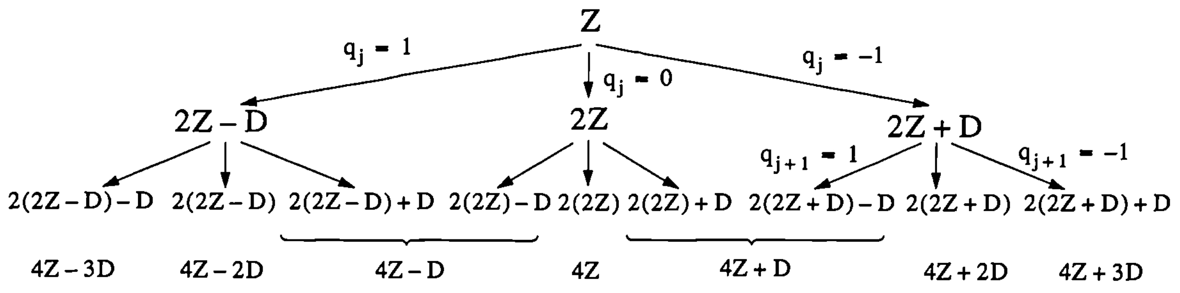


Figure 4.5 Speculative calculation of SRT partial remainders

In parallel with the speculation of the quotient digits, the complete partial remainder is calculated for each of the three values of the quotient digit. Although this approach is fast, it is expensive to implement. In this thesis it is proposed that the number of gates be reduced by speculating on only those digits of the partial remainder required by the quotient-digit selection logic of the next stage. The other digits of the partial remainder are calculated in the next stage, in a non-speculative manner, as the quotient digits are known at this point. Figure 4.6 shows one radix-8 row of the divider which does this.

The divider operates as follows. The most significant part of the partial remainder Z^h_{j-1} enters the top of the array in a carry-save representation. The 4 msbs are examined by the selection logic to obtain the first digit q_j. The selection logic consists of a carry-propagate adder, as shown in Figure 4.7, which implements the selection function specified in Table 4.1. Whilst this occurs, the partial remainders resulting from the three possible outcomes of the first quotient digit are calculated using the 14-bit carry-save adders denoted in the diagram as CSA_14. Only two adders are needed, as a quotient digit of zero requires no subtraction, only that the wordlength be compressed using the X-cell. The 4 msbs of the three partial remainders are fed to selection cells to determine the second quotient digit q_{j+1} for each case. In parallel with this, the partial remainders for the seven possible outcomes of both the first and second quotient digits are calculated. These are used to determine the third quotient digit for all outcomes for the previous two digits.

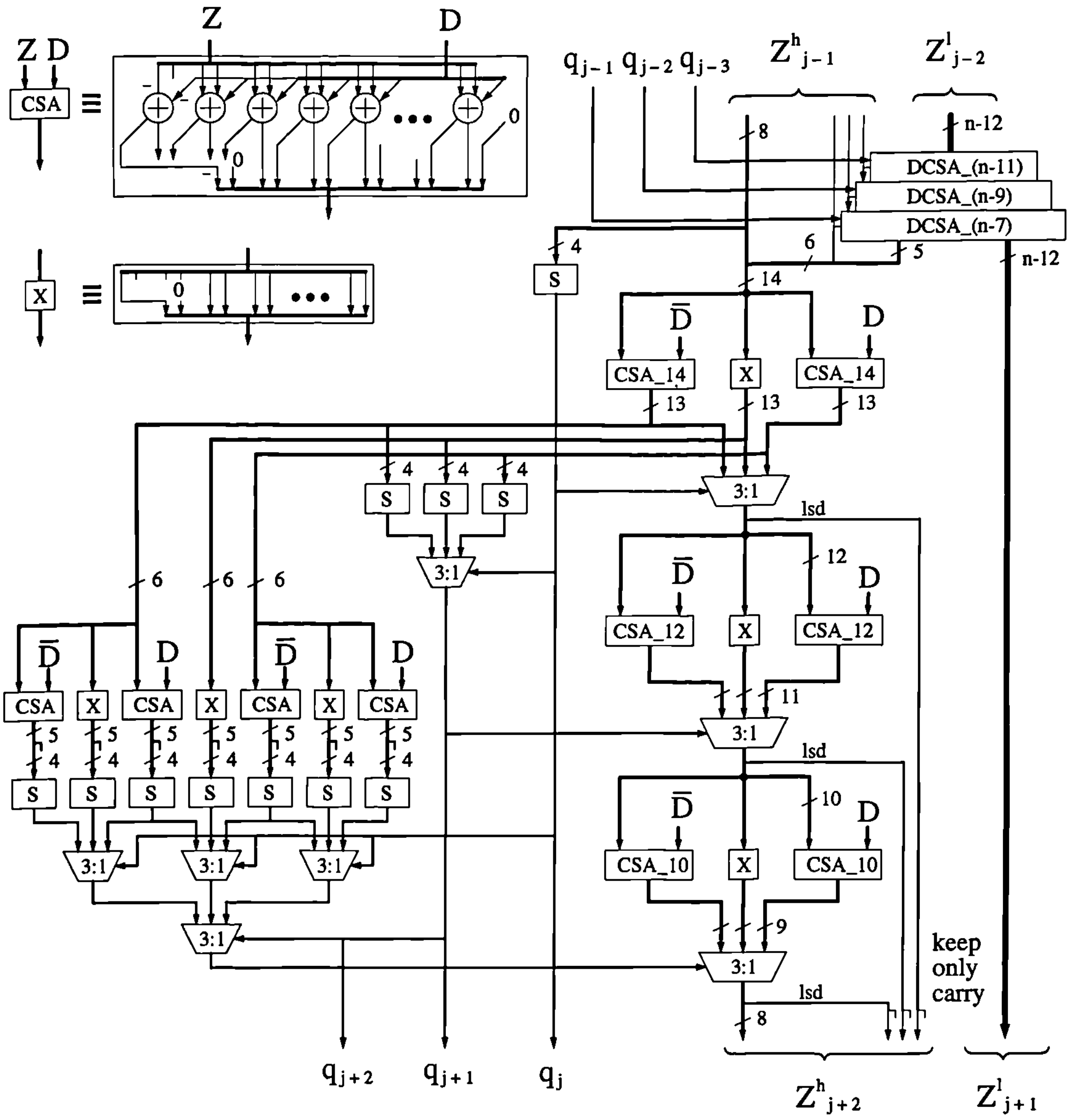


Figure 4.6 Block diagram of one row of the radix-8 speculative divider

Once the first quotient-digit becomes available, the correct second digit is selected using multiplexers. The first and second digits are used to select the third. The appropriate partial remainders can also be selected once the quotient digits become available. Speculation of the partial remainder is performed only for those bits required by the selection logic of the next stage. The rest of the partial remainder is calculated in the next stage using the known quotient digits. The components to do this are shown in the top right-hand corner of Figure 4.6.

Delaying the calculation of the lower part of the partial remainder, Z^l, substantially reduces the area of the divider, and should not significantly increase its critical path.

4.5.1 Selection Logic

The truth-table for the selection function is shown in Table 4.1.

Table 4.1 Selection function

Input		Output		
Value	$[x_1, x_2, x_3, x_4]$	Value	s	m
≥ 0	1xx.x	+1	0	1
$=-\frac{1}{2}$	011.1	0	1	0
≤ -1	0xx.x (excluding case above)	-1	1	1

A logical implementation of the selection function is shown in Figure 4.7.

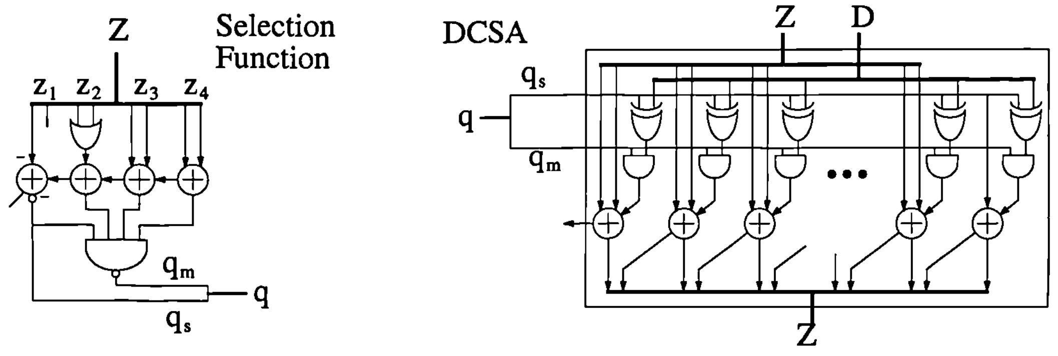


Figure 4.7 Logical implementation of selection function and DCSA

The critical path of the divider consists of a selection cell, two carry-save adders and three multiplexers. The lowest delay is obtained by ensuring that the partial remainder calculation is simple. Therefore, a carry-save representation has been used for the remainder, as compressing it to avoid overflow is particularly simple and requires no logic gates. A signed-binary representation for the partial remainder would reduce the number of digits examined for selection of the quotient-digit selection by one[Kuni87], but would require logic to compress the most significant digits into range after each addition[McQu94a].

4.5.2 Circuit Synthesis Results

Only the single block of three rows, as shown in Figure 4.6, has been coded and synthesised, as the synthesis times of the full divider would be too great. The results are shown in

Figure 4.8. Two area results are presented, one for a row, and the other for the full divider. The number of gates required by a row is high at low wordlengths due to high cost of the speculation circuitry, but grows relatively slowly with wordlength because speculation is not performed on all the additional bits of the partial remainder in this new implementation.

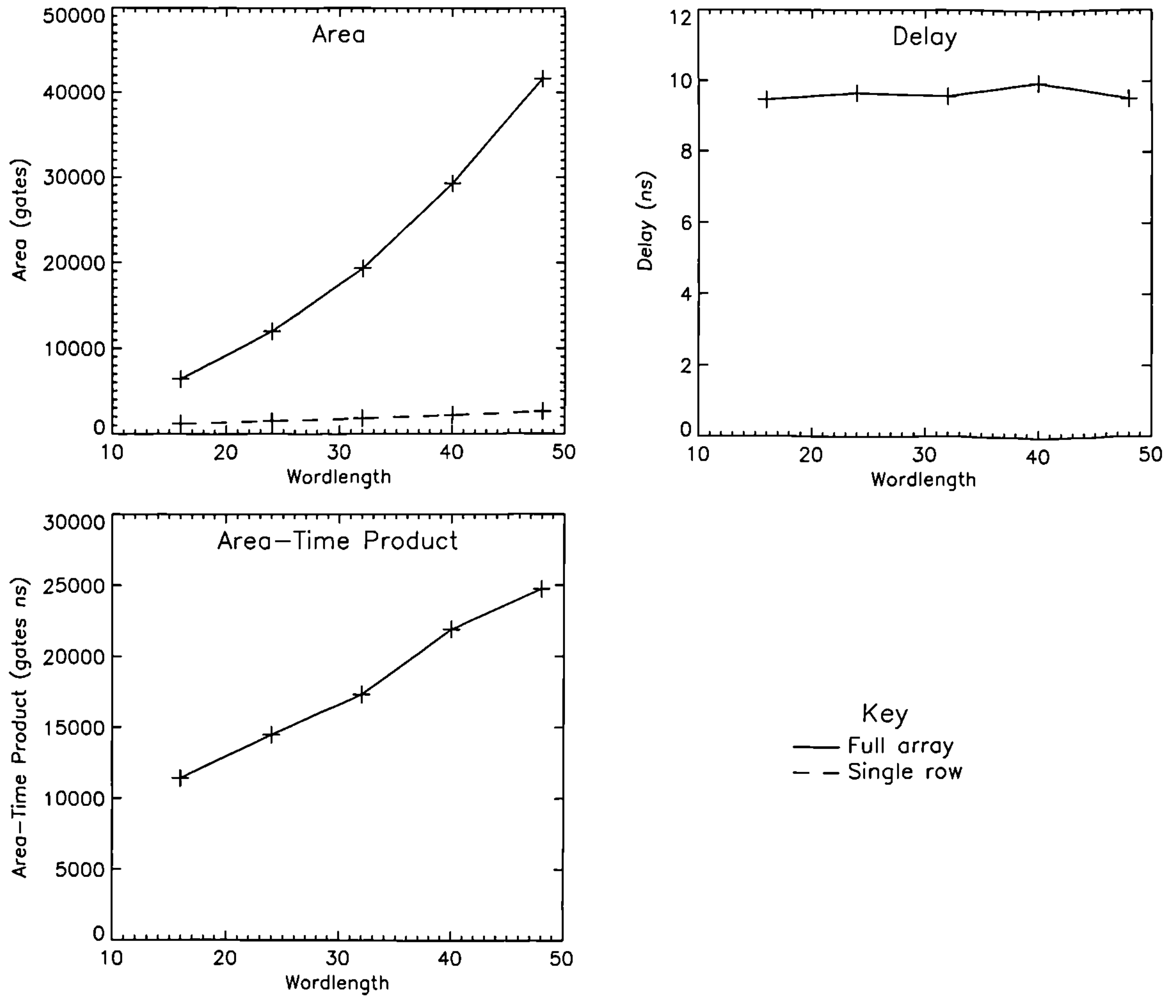


Figure 4.8 Circuit synthesis results for the speculative divider

The delay appears to be constant at 9.5ns, and its fluctuation can be attributed primarily to its estimation errors. The wordlength independence of the delay is expected for a digit-recurrence algorithm, but is more pronounced here than for the full modified-SRT divider, as the smaller circuit simplifies its optimisation. This delay figure is similar to that of the modified-SRT divider. To make an accurate comparison with the modified-SRT divider it should also be synthesised for a block of 3 rows. The modified SRT will require an additional clock cycle to perform pre-scaling, but offers a greater degree of choice in the level of pipelining used, which is an important systems issue.

4.6 Multiplier-Based Division Methods

The multiplier-based division methods use a series of multiplications to refine an initial approximation of the quotient. Consequently, the iterations are more complicated than for the digit recurrence method, but the number of quotient digits determined on each iteration is greater than one and grows quadratically or higher with the iteration. One disadvantage of this approach is that rounding is more difficult to implement as the remainder is not directly available to determine rounding direction. However, the divider required by the Givens rotation operation does not have to be correctly rounded, it is only important that any bias in the quotient error is kept small (how small is discussed later). Therefore, with the availability of low-latency multipliers, the multiplier-based method offers a way of achieving low-latency division or reciprocation.

There are three methods commonly found in the literature:

- Newton Raphson
- Series Expansion
- Convergence

Each method will now be discussed in more detail.

4.6.1 Newton Raphson

A division can be performed using a two-step process in which the reciprocal of the divisor is first obtained and then used to multiply the dividend to give the result. The reciprocal can be found using a popular technique known as Newton Raphson for finding the roots of an equation. The technique solves the equation $f(x) = 0$ where $f(x)$ may take a number of forms. A form which can be used to obtain the reciprocal is $f(x) = \frac{1}{x} - D$, where D is the divisor.

The procedure is iterative, and strives to improve the estimate of a root of $f(x)$ by finding where the tangent of the function at each estimate intercepts the x-axis i.e.

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)} \tag{4.2}$$

To find the reciprocal, the recurrence equation is

$$Q_{j+1} = Q_j(2 - Q_jD) \qquad j = 1, 2, 3, \ldots, n \qquad Q_0 = \text{initial estimate of } \frac{1}{D} \tag{4.3}$$

Where Q_j is the j^{th} estimate of the root and reciprocal. For each iteration, two sequential multiplications are required to double the accuracy of Q_j. For example, if Q_0 was obtained from a table with 7-bits accuracy, then Q_1 would be accurate to 14-bits and Q_2 to 28-bits. Any error made in the computations will also be attenuated quadratically by subsequent iterations, and so the wordlength of the multipliers can be tailored to each iteration, providing considerable savings in area and time.

4.6.2 Series Expansion

Alternatively, the reciprocal can be expressed as a series expansion by representing the divisor with $D = 1 + x$. If $0 < D < 2$ then the reciprocal is given by

$$f(x) = \frac{1}{D} = \frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + \ldots \tag{4.4}$$

the above can be factored to give

$$\frac{1}{D} = (1-x)(1+x^2)(1+x^4)\ldots \tag{4.5}$$

which is easily calculated, as each higher order term can be obtained from the product of the previous terms. For example, the next term in equation (4.5) is $(1 + x^8)$, which can be obtained by first calculating $(1 - x^8)$ using

$$1 - x^8 = (1 - x^4)(1 + x^4) \tag{4.6}$$

The term $(1 + x^8)$ is calculated using the relationship $1 + x^8 = 2 - (1 - x^8)$, which is simply the 2's complement operation.

If P_j is defined as the expansion of the reciprocal up to terms of order 2^j, then a recurrence equation can be written as

$$P_{j+1} = P_j(2 - P_j(1 + x)) \qquad j = 1, 2, 3, \ldots, n \qquad P_0 = \text{initial estimate of } \frac{1}{D} \tag{4.7}$$

As observed by Flynn [Flyn70] the recurrence equations obtained by series expansion and Newton-Raphson are just two variants of the same algorithm related by $D = 1 + x$.

4.6.3 Convergence Methods

The third multiplicative method is convergence division, which is performed using the following process:

$$Q = \frac{N}{D} = \frac{N \cdot R_0 R_1 \ldots R_{m-1}}{D \cdot R_0 R_1 \ldots R_{m-1}} \rightarrow \frac{Q}{1} \tag{4.8}$$

Both the divisor and the dividend are multiplied by m factors R_0, $R_1 \ldots R_{m-1}$ which are chosen so that the divisor converges to 1. The dividend, N, then converges to the quotient Q.

Convergence methods can be divided into those that use multiplication and those that simplify the multiplication to an addition. The multiplication based convergence methods offer quadratic or higher order convergence. Additive convergence algorithms offer linear convergence, generating a digit of the quotient for each iteration. In this respect, the additive technique is similar to the digit recurrence method, but iterates two terms instead of one, which relaxes the digit selection criteria (*cf.* CORDIC). Rodrigues *et al.* [Rodr81] exploits this fact to implement a range of functions in radix-4 arithmetic using the MinR4 representation to reduce the circuit complexity to that of radix-2.

The multiplicative convergence algorithm used by Anderson *et al.* [Ande67] can be derived by noting that the multiplier R_j can be obtained using the relationship:

$$(1 - x)(1 + x) = 1 - x^2 \tag{4.9}$$

Therefore, the divisor $D = (1 + x)$ can be driven closer to one by multiplying it by $R = (1 - x)$. The latter term can be obtained from the first by a 2's complement operation i.e. $R_j = 2-(1+x_j)$. The recurrence equations to perform a division become,

$$\begin{aligned} R_j &= 2-D_j \\ D_{j+1} &= D_jR_j \\ N_{j+1} &= N_jR_j \end{aligned} \tag{4.10}$$

This bears strong similarities with the Newton-Raphson method. If the substitution $D_j = Q_jD$ is made then

$$Q_{j+1}D = Q_jD(2-Q_jD) \tag{4.11}$$

and dividing through by D yields

$$Q_{j+1} = Q_j(2-Q_jD) \tag{4.12}$$

which is the Newton-Raphson recurrence. Unlike the Newton-Raphson method, a division is performed instead of a reciprocal operation. More significantly, the two multiplications required on each iteration are independent and can be performed in parallel. This allows reduced latency when the multipliers are implemented in parallel. However, their independence also means that the errors made in the calculation of N_j are not reduced by subsequent operations, which will require the use of extra wordlength in this datapath.

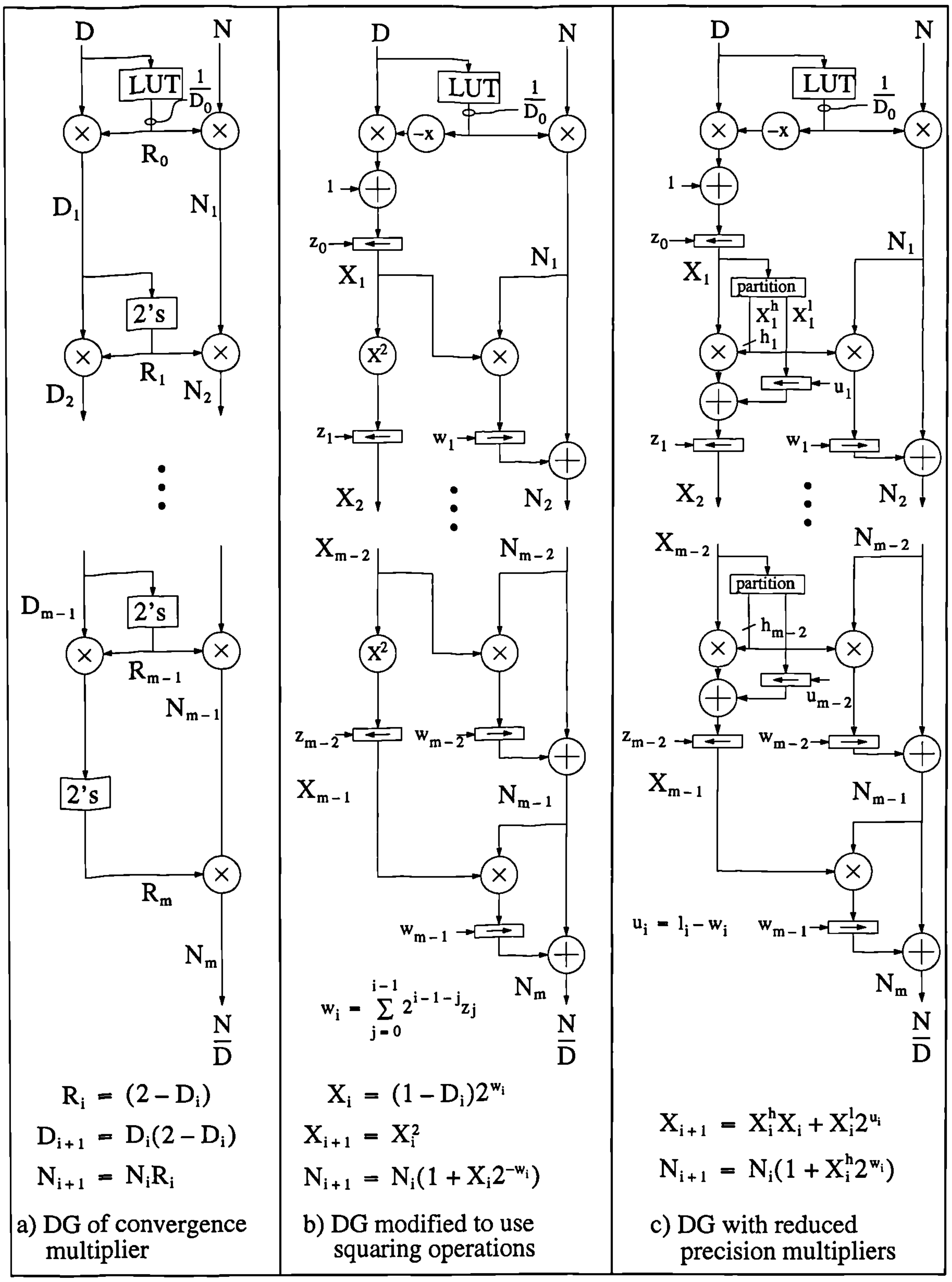
The multiplicative convergence algorithm offers the greatest potential for a low-latency implementation, so a divider design based on this approach has been investigated further. The results of this study are presented in the following section.

4.7 Low-Latency Convergence Divider

Conventionally, convergence dividers are implemented by re-using a single multiplier a number of times. This is an effective approach in microprocessor applications where wordlengths are high and latency is more of an issue than throughput. In this thesis, it is not suitable, as high-throughput is also required. A further disadvantage of re-using a single multiplier is that it is not always used optimally. Therefore, in this section an implementation using dedicated multipliers for each multiplication is proposed to obtain high-throughput and allow individual multiplier wordlengths to be minimised.

4.7.1 Reducing the Multiplier Wordlength in the Divider

Figure 4.9 (a) presents the dependence graph (DG) for a convergence divider.



$R_i = (2 - D_i)$

$D_{i+1} = D_i(2 - D_i)$

$N_{i+1} = N_i R_i$

a) DG of convergence multiplier

$w_i = \sum_{j=0}^{i-1} 2^{i-1-j} z_j$

$X_i = (1 - D_i)2^{w_i}$

$X_{i+1} = X_i^2$

$N_{i+1} = N_i(1 + X_i 2^{-w_i})$

b) DG modified to use squaring operations

$X_{i+1} = X_i^h X_i + X_i^l 2^{u_i}$

$N_{i+1} = N_i(1 + X_i^h 2^{w_i})$

c) DG with reduced precision multipliers

Figure 4.9 Dependence graphs for division by multiplicative convergence

The D_j are calculated in the left-hand column and the N_j in the right-hand column. The R_j factors are obtained by a 2's complement operation on the D_j, and applied to both the D_j and N_j terms using the multipliers in the respective columns.

A number of improvements can be made to the basic DG. The first can be obtained by noting that R_j converges to one. This fact has been exploited by Anderson *et al.*[Ande67] and others to reduce the size of the multiplications, by eliminating the leading 1s that arise in their binary representation. The D_j term also approaches 1, a fact which may now be exploited to further reduce the size of the dedicated multipliers.

The wordlength reductions that may be achieved become more obvious if the graph is reformulated to evaluate the division in terms of a new variable $X_j = 1 - D_j$. In this case $R_j = 1 + X_j$ as shown in Figure 4.9 (b). The first term, X_0, is small relative to one, so a shift-left operation can be performed by a fixed number of bits, z_0, to normalise the number and reduce its wordlength. The effect of this scaling and further scaling by z_j is accounted for with an exponent w_j, which is applied in any calculations involving X_j by a right-shift of the result by w_j bits. Both z_j and w_j are fixed for a particular wordlength of divider, so the shifters can be implemented using wire. Another advantage of this reformulation is that X_j is updated using dedicated squaring operators, which are almost half the size of a multiplier of the same wordlength.

The N_j can be updated using X_j and $N_{j+1} = N_j + N_jX_j$. Due to the quadratic reduction in the size of X_j, the value of N_jX_j rapidly becomes less significant than N_j, and so the wordlength of the multiplication can be reduced with each update step.

Another reduction in multiplier size can be obtained by noting that the accuracy of R_j may be reduced in early steps without affecting the final accuracy of the result, providing that the errors introduced are no bigger than those already present in X_j. Figure 4.9 (c) shows the DG with this modification. The partitioning circuit splits X_j into two parts, such that

$X_j = X_j^h + 2^{-h}X_j^l$. The multiplier R_j is obtained from the most significant part using $R_j = 1 - X_j^h$, which reduces the size of the multiplication required to update X_j and N_j. The term X_j is correctly updated by performing $X_j = X_jX_j^h + 2^hX_j^l$. This arises from the need to obtain

$$\begin{aligned} 1 - X_{j+1} &= (1 - X_j)R_j \\ &= (1 - X_j^h - 2^hX_j^l)(1 + X_j^h) \\ &= 1 - (X_jX_j^h + 2^hX_j^l) \end{aligned} \tag{4.13}$$

This involves an extra addition, but the multiplication will be significantly smaller. The multiplication can, in part, be implemented using a squarer to reduce the hardware.

This same principle may be applied to limit the multiplier wordlength, but will result in a convergence rate less than quadratic.

4.7.2 Initial Estimate of the Reciprocal

The number of stages required to calculate the reciprocal can be reduced by starting with a more accurate initial estimate of $\frac{1}{D}$. The most common approach is to use a look-up table implemented by a ROM or combinational logic. This offers speed, but requires an area which grows exponentially with the wordlength. The simplest approach is to use a piece-wise constant approximation to the reciprocal. Alternatively, interpolation using a piece-wise linear or polynomial approximation may be used to reduce the table size. These latter approaches use multipliers to calculate values within the intervals, but the wordlengths are small and so the extra complexity is not great. Interpolation may also be obtained using ***bipartite tables***. In this case, a multiplier is avoided by performing the interpolation using a second look-up table. The number of interpolation table entries is reduced by grouping intervals which may be approximated by the same function. In this way Das Sarma[DasS95] has achieved 2 to 4 times reduction in size for 9-bit tables, and 4 to 16 times reduction for 10 to 16-bit tables over the piece-wise table approach.

Another method is to modify the partial product arrays of a multiplier so that it can provide an approximation to a reciprocal. The cost of doing this in additional hardware is relatively small, but it assumes that a multiplier is available for this purpose, otherwise it is very costly.

In this thesis only the piecewise constant look-up table approach is considered, as this can be determined sufficiently quickly to allow it to be merged with the first multiplier. Alternative look-up approaches may provide sufficient accuracy to require one less iteration, but the increased complexity would mean an extra cycle being devoted to the look-up operation, and any latency benefit lost.

4.7.2.1 Reciprocal LUT

For a piecewise constant approximation, the largest look-up error is minimised by calculating the table entry at the mid-point of the interval and rounding the result[DasS94]. For a table input truncated to k fractional bits and the output rounded to g fractional bits, the entries are given by

$$\mathrm{table}(\hat{D}) = \mathrm{round}\left(\frac{1}{\hat{D} + 2^{-k-1}}, g\right) \qquad \hat{D} = \mathrm{trunc}(D, g) \tag{4.14}$$

The error in the approximation is a function of both truncation error (ε_D) of the table input and rounding error of the stored value (ε_T). i.e.

$$\frac{1}{D} = \frac{1}{\hat{D} + \varepsilon_D} + \varepsilon_T \cong \frac{1}{\hat{D}} - \frac{\varepsilon_D}{\hat{D}^2} + \varepsilon_T \tag{4.15}$$

For a binary input, the truncation error is $|\varepsilon_D| < 2^{-k}$, and for a binary table entry the rounding error is $|\varepsilon_T| \leq 2^{g-1}$. If the input is normalised within the range $0.5 \leq D < 1$, it will be of the form $0.1b_2b_3 \ldots b_k$ and the output of the form $1.b_1'b'_2b'_3b'_4 \ldots b'_g$. The table size will be proportional to $2^{k-1}g$ and the total error $|\varepsilon_L| < 2^{-(k-2)} + 2^{-(g+1)}$. It is clear from the latter expression that the error can be reduced by increasing k and g, and equal contributions are made by each component of the error when $g = k-3$. However, it is less costly to increase g, due to its linear relationship with the size of the table, and therefore a better choice is to make $g = k-1$.

Figure 4.10 shows the number of gates and delay obtained for a simple reciprocal look-up table with a range of input wordlengths k and output wordlength $g = k - 1$.

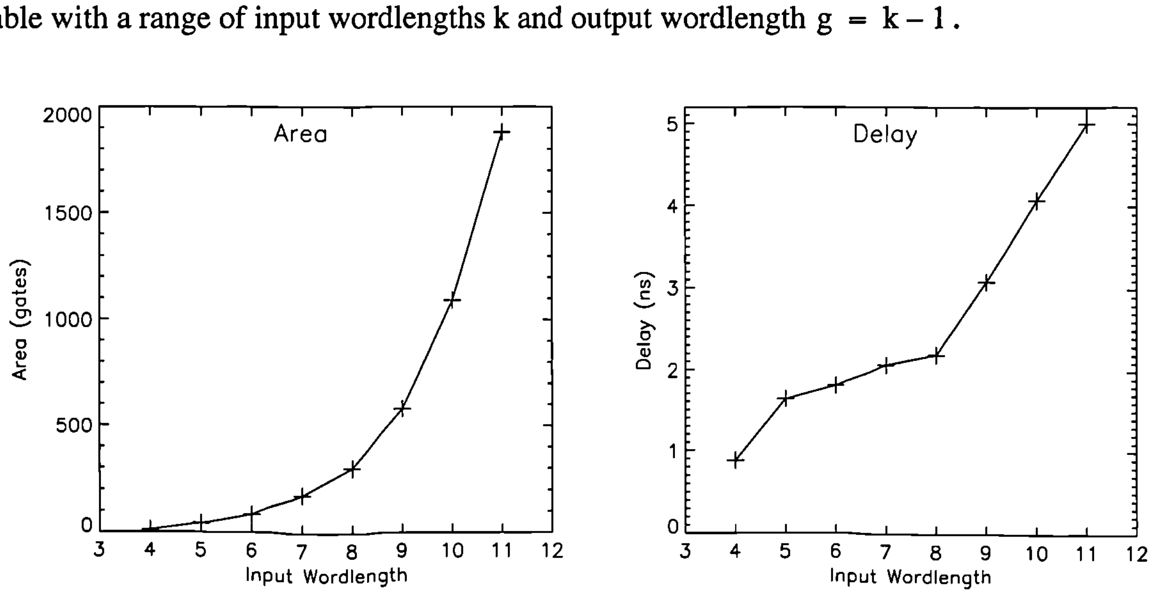


Figure 4.10 Circuit synthesis results for reciprocal LUT

The exponential dependence of the area on wordlength is very clear, and at 11-bits the area is similar to a 16-bit multiplier. Beyond this, the synthesis task becomes too great. All the area figures are lower than would be achieved with an SRT divider, and the delay is significantly lower. For wordlengths up to 8-bits the delay is less than 2.2ns. After this it rises linearly, which indicates the point at which the optimisation tool is starting to having difficulty. It would be prudent to limit the wordlength of the table to 8-bits to enable it to be combined with the first multiplier.

4.7.3 Rounding

The remainder of a division, performed by convergence, is not directly available, and so exact rounding is not possible. One solution is to calculate the quotient to twice the precision of the input, but this can be expensive in area (as parallel multipliers have an area proportional to the square of their wordlength). Alternatively, schemes which use an additional multiply step with only a small increase in wordlength can be used[Darl90][Brig93].

As mentioned previously, the Givens rotation does not require exact rounding, only that the bias in the error is kept sufficiently small. Error bias is important in this application, as numbers are accumulated and biased errors will grow faster than unbiased ones. The convergence division technique converges on the quotient from below, and so there is always a negative bias in the error. This can be made small relative to the error in the dividend and divisor by calculating the quotient to a greater precision. One option it to calculate the quotient to a wordlength where the bias introduced is similar to that which is naturally present in the dividend or divisor (due to the finite number of values they may take). If this is done, then the quotient precision needs to be approximately 50% greater than the dividend and divisor precision.

The convergence divider will be large, even with the optimisation of the multiplier sizes, particularly if extra quotient precision is required to reduce the bias in the output error. However, if used for generating the reciprocal of a number the multiplier sizes in the N_j path can be reduced substantially. A reciprocal circuit is of value in the Givens rotation as two or three divisions are performed using the same divisor. Therefore, the VLSI design of a reciprocal circuit is considered in the remainder of this section.

4.7.4 Reciprocal Circuit

Figure 4.11 shows the DG for a convergence divider, which has been modified to produce a reciprocal. This has been achieved by setting the dividend input to one. As a consequence, the first multiplier is no longer needed, and has been removed. The number of stages and the multiplier wordlengths chosen for a 16-bit input and 16-bit output (once rounded) are shown. These have been established using a computer simulation to calculate the error generated over all divisor inputs.

A lists of the parameters for a range of other input and output wordlengths is presented in Appendix A.

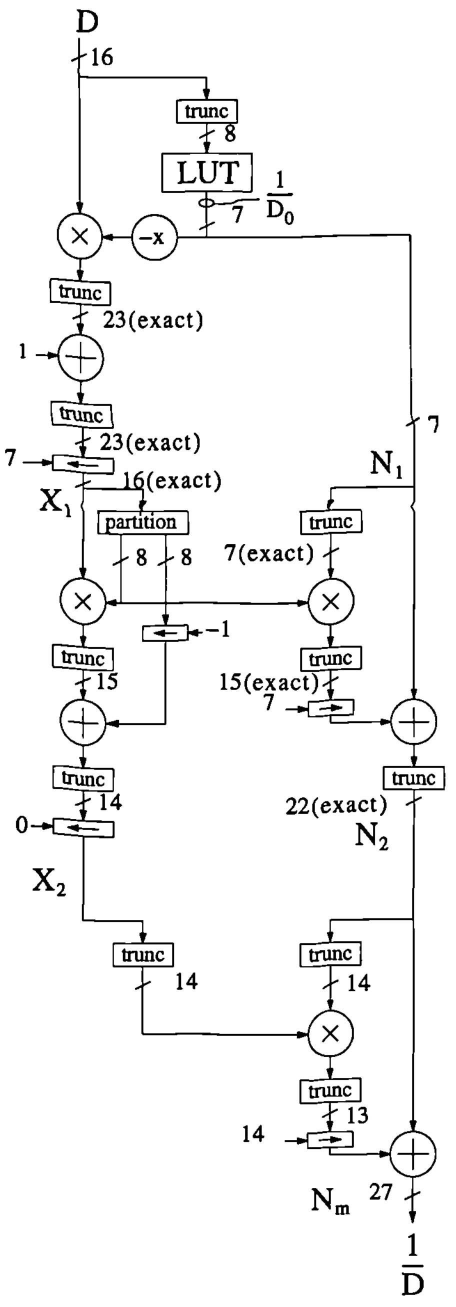


Figure 4.11 DG for 16-bit reciprocal

4.7.5 Circuit Implementation

The circuit is ideally suited to the application of redundant arithmetic to obtain high speed. The adders could be merged with the multipliers and implemented using the tree structures presented in Chapter 3. However, this will be costly in area, even if the techniques presented in this chapter and Chapter 3 are used. Therefore, a non-redundant approach has been adopted.

Figure 4.12 shows the number of gates and delays obtained for the range of non-redundant reciprocal circuits, as summarised in Table 4.2.

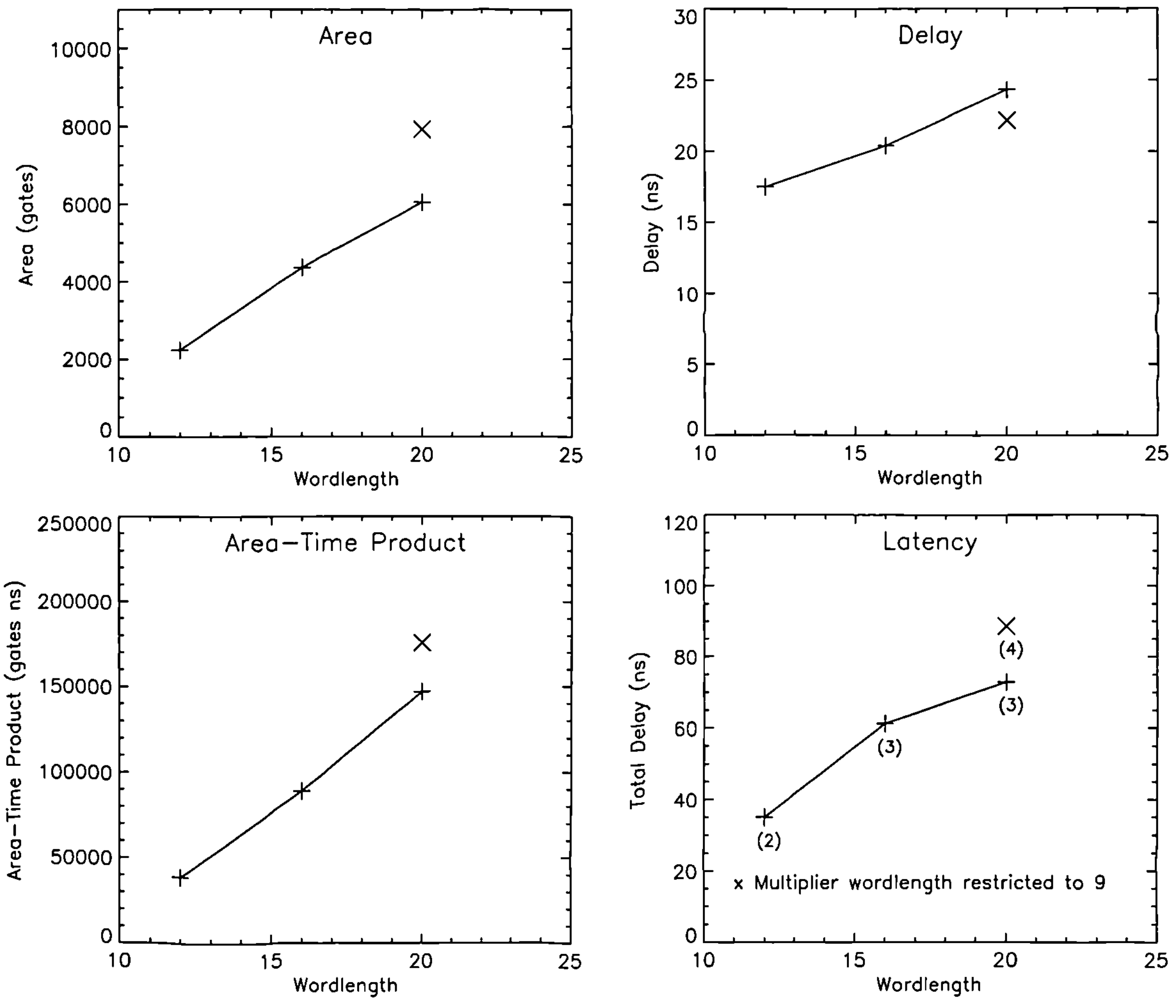


Figure 4.12 Circuit synthesis results of the convergence reciprocal circuits

The points of the graphs marked by an X are the results of the reciprocal circuit where the multiplier wordlength has been restricted to 9-bits. This technique can be used to reduce the circuit delay to meet a particular system clock speed. However, it increases the number of cycles of latency from 3 to 4 and increases the area and total delay.

In all cases the stage delay is high, which can be attributed to the use of non-redundant arithmetic and associated carry-propagate adders. Pipelining was applied only between stages. Extra cuts could be made within the stages between the adders and the multipliers, and may enable a 10ns stage delay to be obtained, but would also double the latency. The delay of the adders can be reduced by combining them with the multiplier.

Table 4.2 Circuit synthesis results for reciprocal circuit

Input Wordlength	Output Wordlength	Stages (Latency)	Gate Count	Speed (ns)	Latency (ns)	Modified SRT Latency
12	20	2	2226	16.09	32.18	67.4
16	26	3	4367	19.00	57.00	89.2
20	32	3	6055	22.90	68.70	112[a]
24[b]	32	4	7931	20.75	83.00	135[a]

a. Estimated from 12-bit divider results.
b. Multiplier size restricted to 9.

The convergence method, in the non-redundant form, allows the 12-bit and 16-bit reciprocals to be obtained significantly quicker than if a modified SRT divider was used and require approximately 33% and 18% less area respectively.

4.8 Summary of Divider Results

The divider research can be summarised as follows:

- The modified SRT algorithm offers the lowest area-time product of all the approaches considered. From a system design perspective it also offers great flexibility as the level of pipelining may be chosen to match the system clock frequency.

- The speculative divider offers low-latency and comparable area-time product. However, it is less flexible than the modified-SRT, as it is designed to be pipelining only every 3 rows.

- In its current form, the convergence divider will offer some advantage in latency over the other approaches. It is also well suited to the application of redundant arithmetic, which may allow considerably lower latency to be achieved, but with greater area. For this reason convergence division would appear the more attractive solution for dividers in future microprocessors (where large circuit area is becoming less critical than latency).

- The convergence reciprocal circuit, if implemented in a more highly pipelined form, may offer some area-time advantage over the digit-recurrence approaches. If used as a first step for division, it will only offer a benefit in area if more than two divisions by the same number are required. If implemented in a redundant form it will offer a significant reduction in latency but with approximately 50% increase in area over current levels.

Note that these conclusions have been drawn from results in the case of relatively low word-lengths (which is required for Givens rotations implementations considered later in the thesis). Other observations are:

- The circuit synthesis of the unpipelined block of 3 rows of the speculative divider provided a delay independent of wordlength. When the whole SRT divider was synthesised the results were less impressive, probably due to the complexity of the optimisation problem. Therefore, a two step synthesis approach is proposed for improving the circuit results and run times. The first step is to synthesise and optimise the logic between pipelining stages and assemble the divider from these circuit blocks. In a second step, this block can be used as a starting point from which to obtain an optimisation of the circuit as a whole.

- One of the most surprising results was the low delay and relatively low size of the look-up-tables generated for the initial reciprocal estimate. This approach was less effective at wordlengths greater than 8-bits due to the enormous optimisation problem that arises, but below this the results were very good. Future research could investigate how the table look-up function could be posed in a way which simplifies the synthesis and optimisation task. Alternatively, a more directed optimisation strategy could be used for this task, rather than a general purpose synthesis tool.

Chapter 5 Floating-Point Operators

5.1 Introduction

In the previous two chapters the design of high-throughput, fixed-point operators with low latency was considered. Floating-point arithmetic is important for the implementation of certain digital signal processing algorithms and there are now many DSP processors which support it. Indeed, floating-point arithmetic is a requirement of some of the Givens rotation algorithms examined later in the thesis, and in particular the most computationally efficient algorithm requires it. In this chapter the design of the floating-point operators for this algorithm is considered. The algorithms and architectures of these operators are presented, and the area and speed of their VLSI implementations has been determined for a range of wordlengths.

To achieve high-throughput floating-point operators, the bit-parallel architectures presented in the previous two chapters are used to perform the fixed-point mantissa operations. Rounding has been included to achieve an output with little or no bias of the truncation error.

For reasons which will become apparent in Chapter 7, very low latency is required from only the adder, so a non-redundant, parallel binary representation has been used for all other operator inputs and outputs so as to minimise their area. For the adder, two approaches have been adopted. One approach uses the same non-redundant representation as the other operators, and relies upon low wordlength and parallel hardware to obtain low latency. The other obtains lower latency by using a fixed-point representation for one of the inputs and the output. This also enables a redundant representation to be used to avoid carry-propagation, and so obtain very low-latency which is independent of wordlength.

The steps to implement floating-point operators with rounding are well known and relatively straightforward, but these steps are numerous and lengthy. Therefore, the challenge in designing operators with low latency is to determine ways of reducing the number of steps. Many of the techniques used in this chapter have been obtained from the literature.

5.2 Floating-Point Representation

The IEEE Standard for Binary Floating-Point Arithmetic[IEEE85] has become the standard for VLSI floating-point units. If adhered to in the implementation of application specific DSP circuits it is possible to use the FPU of an IEEE compliant workstation to perform exact numerical simulation of the circuits at much higher speeds than a software emulation could achieve. The full standard can be complex to implement and only supports two wordlengths, known as single and double precision formats. Therefore, the standard is used only as a starting point here. In particular, *rounding-to-nearest* has been implemented, as this is important in reducing the build-up of numerical errors.

The number format shown in Figure 5.1 has been adopted (where MBits and EBits represent the number of bits in the mantissa and exponent respectively).

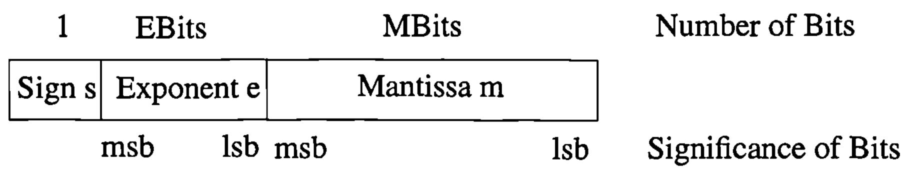


Figure 5.1 Floating-point number format

The exponent is stored in a two's complement format, with the most negative value (i.e. $100...00$) indicating that the floating-point number is zero. The mantissa, m, is positive and normalised such that $1.0 \le m < 2.0$. This means that the leading bit is always a logic 1, and so it is not stored. The sign bit is used to indicate when the mantissa is negative, in which case it takes the value logic 1. Otherwise, the number is positive or zero, and the sign bit takes the value logic 0.

When the number is zero (i.e. $e = 100...00$) the sign and mantissa are superfluous. In this case their values are left undefined as the logic to set these numbers to defined values will only delay the result.

Rounding-to-nearest is achieved by examining the fraction removed by truncation. If this is greater than one half of an lsb (i.e. $0.5\mathrm{lsb}$), the result is rounded up by adding a bit in the lsb position. If a binary representation is used, this condition can be determined by examining the msb of the truncation error. As described, the rounding procedure introduces a small bias in the error as there are more occasions when the number is rounded up rather than down. To avoid this the IEEE standard specifies that on occasions when the truncation is exactly half an lsb that rounding-up should only be implemented when it will generate an even result (i.e. the lsb of the result is zero, which occurs on average half the time). The truncation error is exactly $0.5\mathrm{lsb}$ when all its bits bar the msb are zero. A signal, referred to as a *sticky-bit,* is generated to indicate this condition, and the lsb of the result can be examined before rounding to determine if the result will be even. This aspect of the rounding is not implemented in the operators presented in this chapter as its effect on the error is small and the sticky-bit takes significant time to generate.

The design of each operator is now considered in detail in the next few sections.

5.3 Floating-Point Adder

The floating-point add operation is the most complicated of the floating-point operators to implement, and the area of the final adder is significantly greater than that of a fixed-point one. The complexity arises from the need to align the mantissa before addition and normalise the result afterwards. These operations significantly increase the adder's circuit area, particularly when they are required to be performed quickly.

5.3.1 Operation of Basic Floating-Point Adder

A dependence graph for a floating-point adder is shown in Figure 5.2. This simple solution offers minimum hardware, but results in very high latency. There are seven stages to the addition process, each of which could be one stage in a pipeline.

1. **Alignment of mantissas to a common exponent:** The mantissa of the number with the smallest exponent is right-shifted by a number of bits sufficient to make the exponents equal.
2. **Add:** The signed-magnitude numbers are converted to 2's complement and added.
3. **Absolute value:** The 2's complement sum is converted to sign-magnitude representation.
4. **Leading zero detection:** Leading-digit cancellation may have occurred, so the number of leading zeros in the absolute value of the sum is determined for normalisation. If both adder inputs are positive then overflow can occur, in which case a 1-bit right-shift operation is performed to normalise the sum, and the exponent is incremented.
5. **Normalisation:** A left-shift operation is performed to normalise the mantissa.
6. **Rounding:** Alignment or overflow may give a sum with more bits than the input mantissa. Rounding is performed to return the wordlength to that of the input. Rounding-to-nearest is implemented by examining the bit to the right of the lsb, and incrementing the lsb if its value is logic 1.
7. **Overflow normalisation:** When the mantissa bits are all logic 1 prior to rounding (i.e. $m = 1.11\ldots11$), and rounding is performed, the mantissa will overflow. This produces the result $m = 10.00\ldots00$ which must be normalised by shifting the mantissa 1-bit to the right. Alternatively, the operation can be simplified by noting that the normalised result differs from the unnormalised one by only the msb. Therefore, the normalised mantissa can be obtained directly from the unnormalised one, providing that the msb is obtained by ORing the msb of the unnormalised number and the overflow bit.

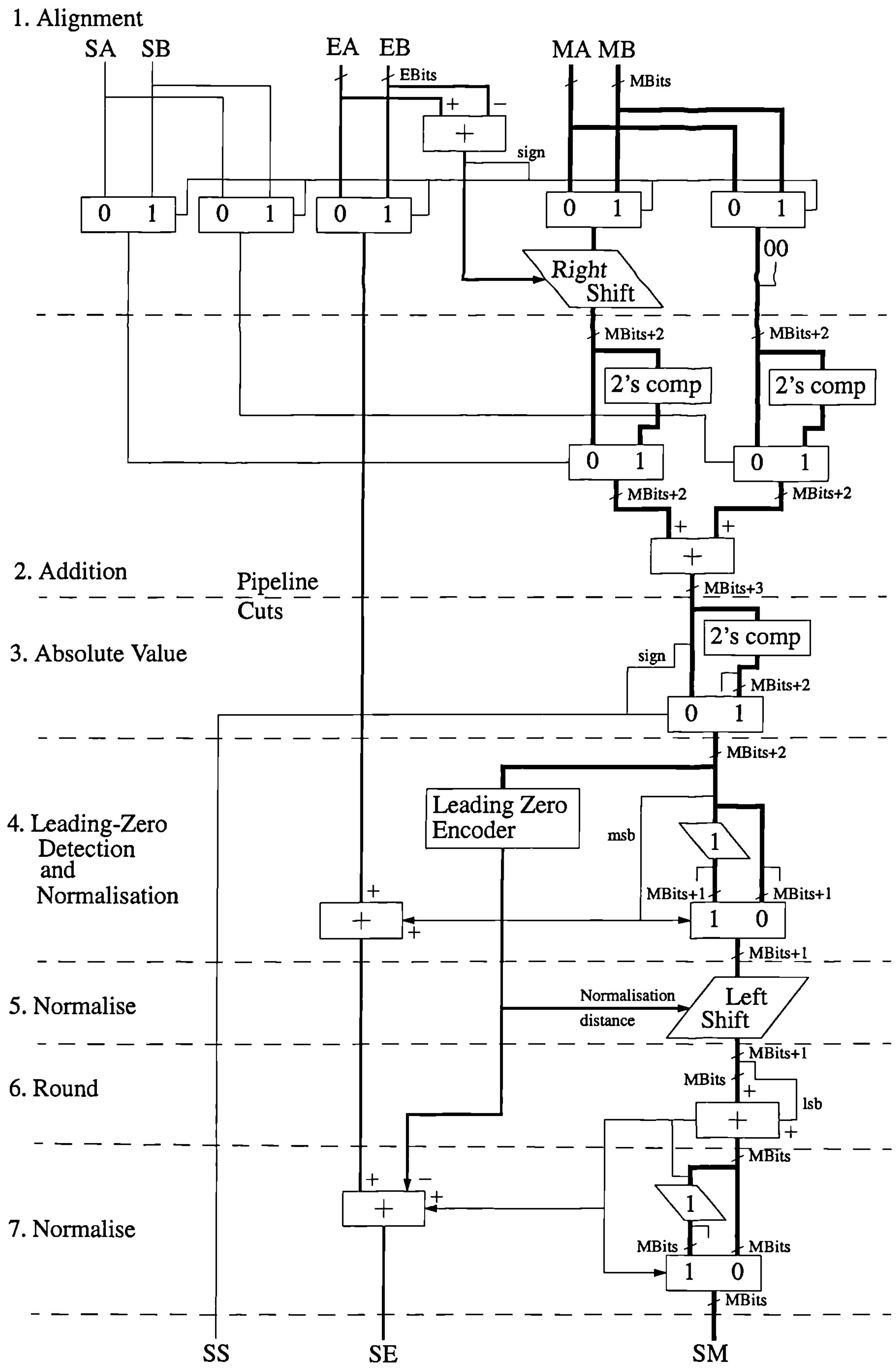


Figure 5.2 Block diagram of simple floating-point adder

5.3.2 Low-Latency Floating-Point Adder

The simple floating-point adder requires a high level of pipelining to achieve high throughput. This introduces a large latency and would be expensive to implement due to the large number of latches required. The time to perform the operation can be reduced by calculating all possible outcomes of certain steps using parallel hardware, so that the next step may start, albeit on several potential results, before the last has been completed. Figure 5.4 shows a detailed block diagram of the floating-point adder that was implemented. Its latency has been reduced from 7 to 3 by the following modifications:

1. **Alignment:** Here the mantissa of the number with the smallest exponent must be right-shifted by the difference of the exponents. Two outcomes of mantissa and exponent are possible, depending upon which exponent is the largest. Both are calculated so that the correct one can be selected by a multiplexer when the largest exponent is eventually determined.
2. **Addition and absolute value:** Determination of the sign and magnitude of the mantissa sum can be avoided[Know91]. This is because only four additions are possible, depending upon the sign of the mantissas. These are $A + B$, $-A - B$, $A - B$ and $B - A$. In the first two cases the sign of the output is known, but in the second two the result can be either positive or negative, so sign detection is necessary. In the latter case, the results have the same magnitude but opposite sign. Therefore, if both are performed in parallel the positive one can always be obtained. Two adders are required, and Table 5.1 summarises the operations of each adder in the four cases detailed above.

Table 5.1 Obtaining a positive output from adder

A Sign	B Sign	Adder 1	Adder 2	Adder 1 Sign	Sum Sign	Output
+	+	A+B	X	+	+	Adder 1
+	-	A-B	B-A	+	+	Adder 1
				-	-	Adder 2
-	+	A-B	B-A	+	-	Adder 2
				-	+	Adder 1
-	-	A+B	X	+	-	Adder 1

For example, if the sign of A is positive and the sign of B is negative then the operations $A - B$ and $B - A$ are performed by Adder 1 and 2 respectively. The positive output forms the magnitude component of the sum and the sign is obtained from Adder 1.

3. **Normalisation and Rounding:** The normalisation and rounding operations are never required at the same time, so they are performed in parallel rather than sequentially. In the normalisation path, the leading-digit detection and shift-left operations have been optimised for speed. Figure 5.3 shows a normalisation circuit developed for this purpose for a 16-bit mantissa. The number of leading zeros is encoded into binary, and may be added to the exponent while the mantissa is being shifted.

Rounding may require the exponent to be incremented. In anticipation of this, an adder is used to increment the exponent, whether or not this is actually required, and a multiplexer is used to select the correct exponent once the result of rounding is known.

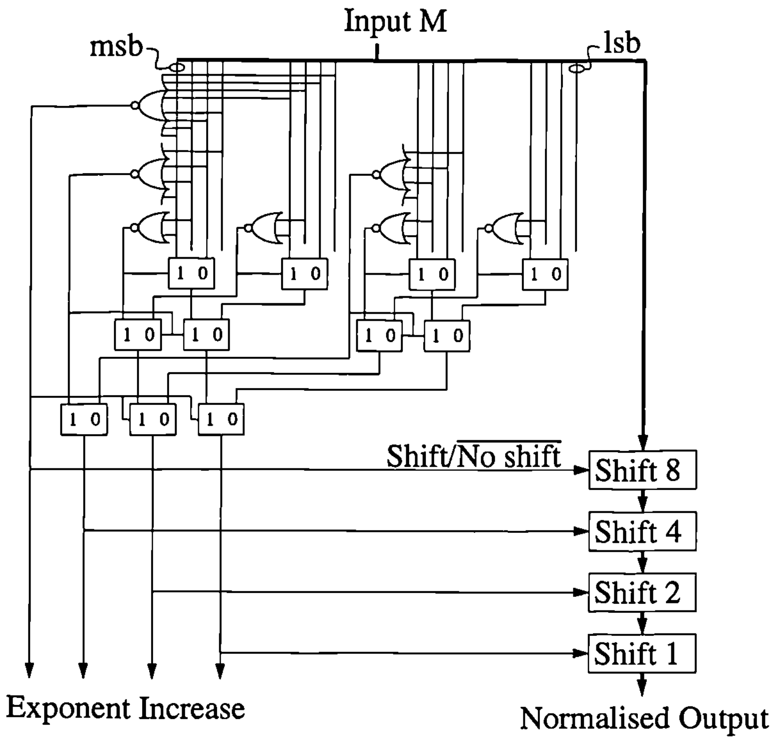


Figure 5.3 Normalisation using speculation of leading zeros

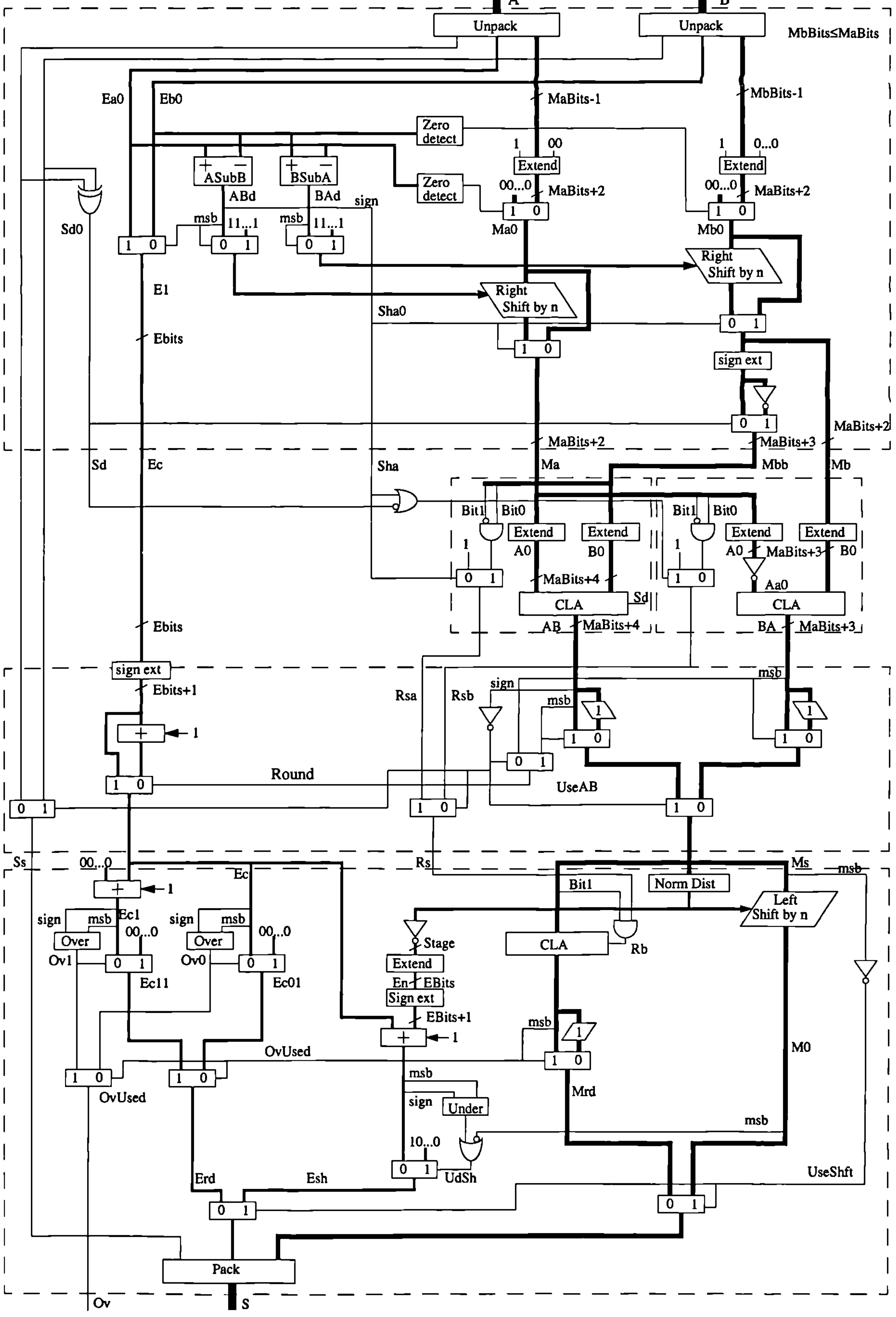


Figure 5.4 Detailed block diagram of floating-point adder

5.3.3 Circuit Synthesis Results

Figure 5.5 shows the delay and the number of gates obtained for the floating-point adder for a range of wordlengths and pipelining. For each level of pipelining two sets of results are given. One where both inputs have the same wordlength, and the other where one input is fixed at 16-bits and the wordlength of the other input and the output is extended. In the former case the horizontal axis indicates the wordlength of both inputs and the output, and in the latter case it indicates the wordlength of the extended input and output.

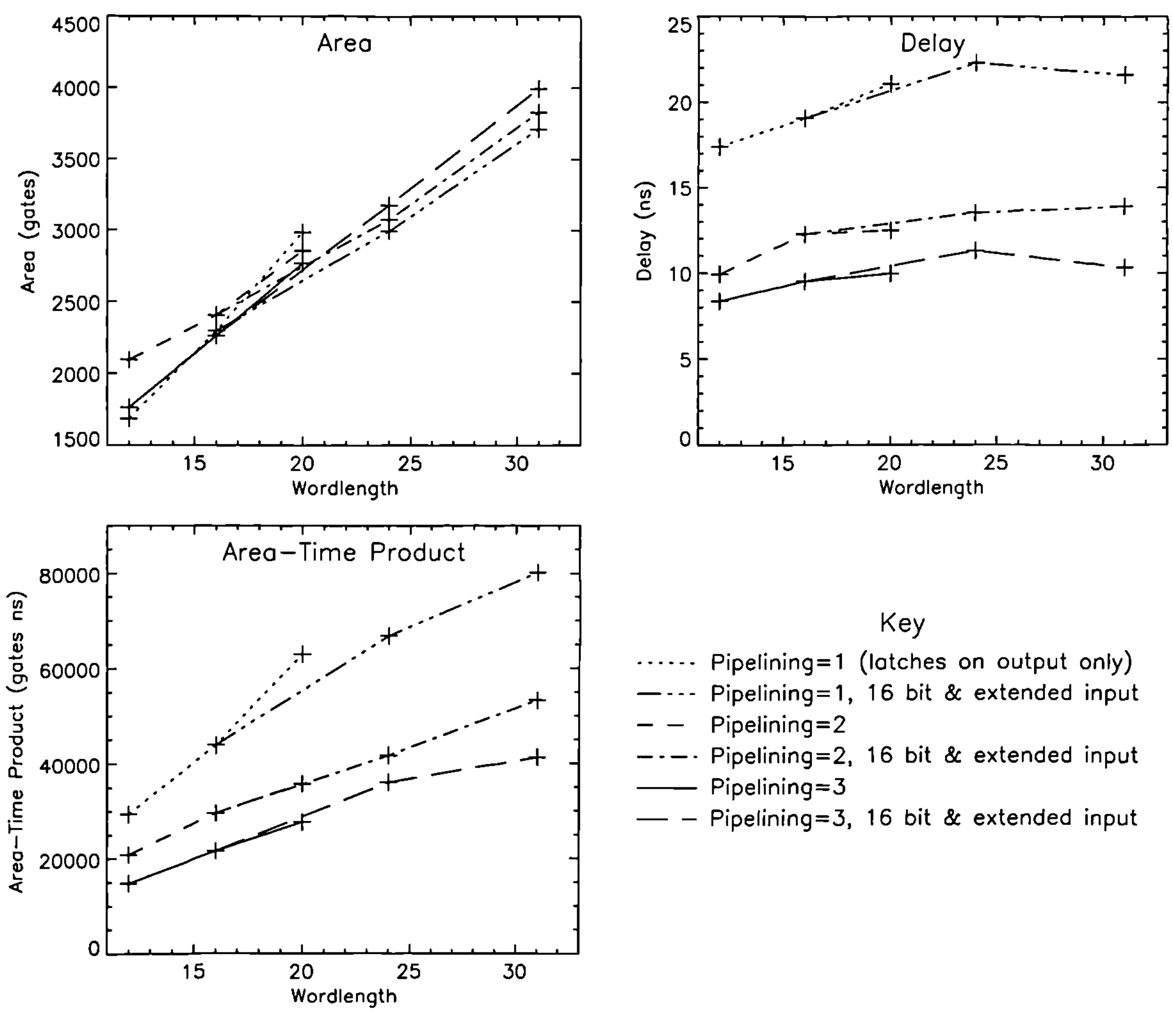


Figure 5.5 Circuit synthesis results for the floating-point adder

The lowest delay and area-time products are obtained when the adder is pipelined into three stages. Pipelining at any higher level would be difficult to achieve with the current architecture, and unlikely to yield a significant reduction in delay. The number of gates is almost linearly dependent upon wordlength, whereas the delay is less dependent upon wordlength.

The lowest latency is obtained when pipelining latches are used only on the output. However, this gives a very high area-time product and a delay which is likely to be too great to be compatible with the rest of the system (a target of 100MHz would be realistic, allowing a maximum delay of 10ns). Therefore, a level of pipelining of 3 is likely to be used in practice. Note that pipelining is implemented at no additional cost in area because the circuit between latches is constrained to a manageable size. Consequently, the optimisation results are much better.

For the floating-point adder, low-latency has been achieved using parallelism, but at a high cost in area. It may be possible to achieve a better area-time product by allowing the latency to increase to 4 or 5, in which case more stages could be used and less parallelism required. However, a reduction in the area-time product may not be as large as expected due to the additional pipelining latches that would be required between stages.

5.4 Fixed/Floating Point Adder

Floating-point operation of the adder increases latency principally due to the need to align the inputs and normalise the output. If it is acceptable for one input and the output to be represented in a fixed-point, then the latency for that input can be reduced to a single cycle. This adder is shown in Figure 5.6. It has also been modified to apply scaling by $(1-2^{-s})$ to the fixed-point input. This is required by the Givens rotation algorithm, and its incorporation here avoids additional latency elsewhere, but necessitates a three-input-adder.

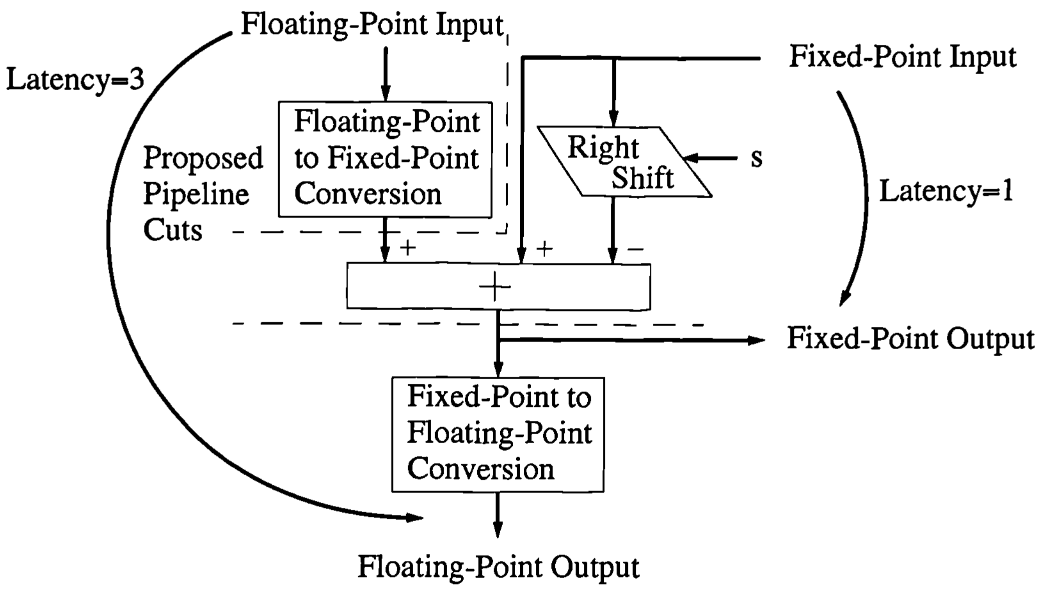


Figure 5.6 Fixed/floating-point adder

Using a signed-binary, redundant representation for the fixed-point input and output, and a redundant adder, it is possible to achieve very fast addition times which are independent of the wordlength. This allows the wordlength of the fixed-point input to be extended to increase its dynamic range, without increased delay. It also allows truncation to be used, instead of a more complex rounding operation.

Figure 5.7 shows the circuit synthesis results for the fixed/floating-point adder using CLA and signed-binary adders for a range of wordlengths. For each adder type two sets of results are given. One where both the fixed- and floating-point inputs have the same wordlength, and the other where the floating-point input is set to 16-bits and the wordlength of the fixed-point input and output is extended. In the former case the horizontal axis indicates the wordlength of the inputs and output, and in the latter case it indicates the wordlength of the extended fixed-point input and output. Fixed-to-floating-point conversion is implemented separately and hence not included in these figures.

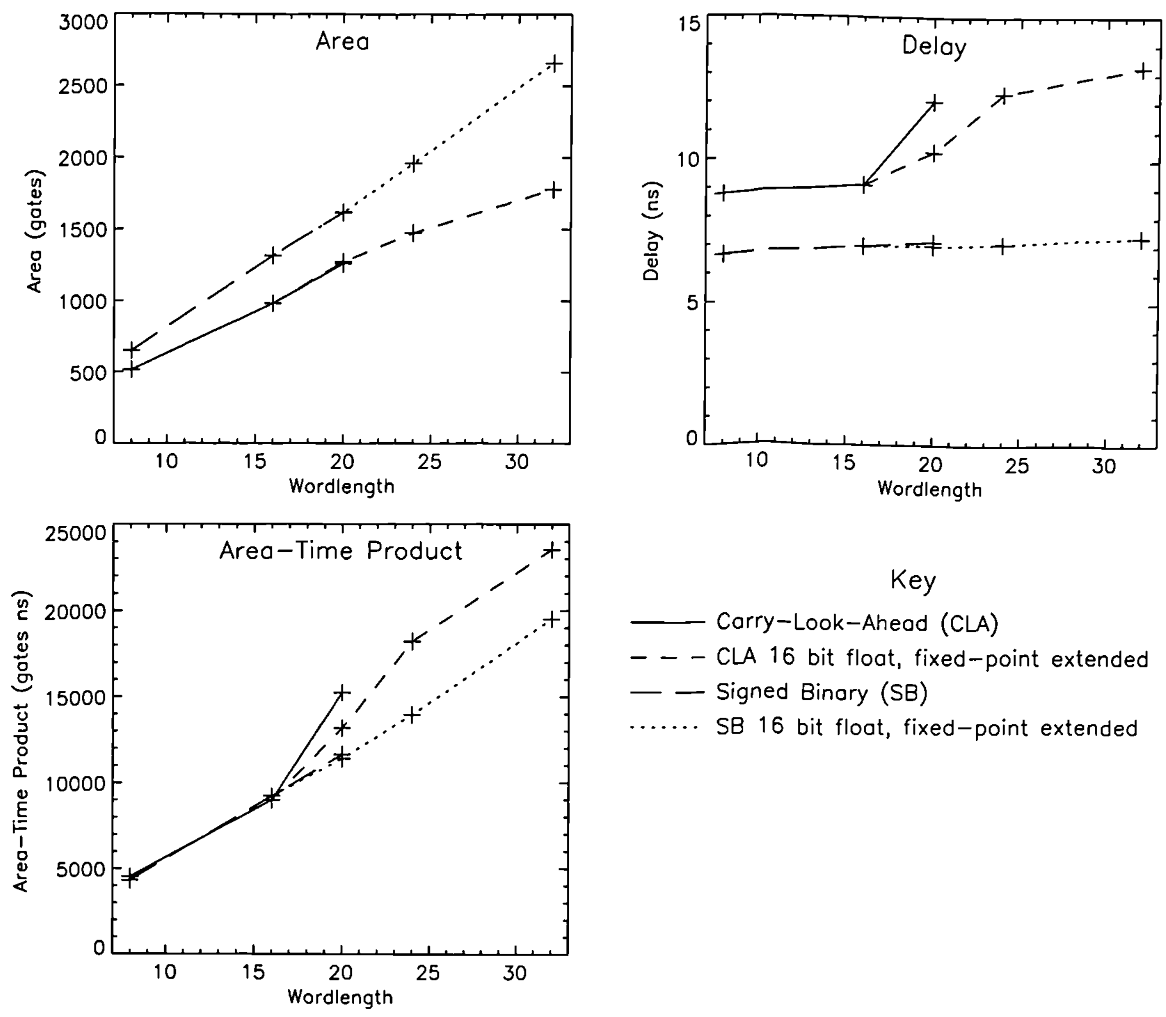


Figure 5.7 Circuit synthesis results for fixed/floating-point adder

Very low-latency is achieved using the signed-binary representation. Care must be taken to ensure that the floating-to-fixed-point and fixed-to-floating-point conversion do not dominate at high wordlengths, and pipelining can be used within the converters to avoid this. The CLA-based scheme takes twice as long, but is still relatively fast considering that scaling must also be performed.

5.5 Floating-Point Multiplication

5.5.1 Operation of Basic Floating-Point Multiplier

The additional complexity of implementing a floating-point multiplier over a fixed-point one is small. There is only a modest increase in the overall latency, due to the need to perform normalisation after multiplication of the mantissas. Achieving rounding with low-latency is the main challenge, and this will arise whether fixed- or floating-point operation is used. Figure 5.8 shows a block diagram of a basic floating-point multiplier. It consists of the following stages:

1. **Multiplication:** Calculation of the product of mantissas, in a carry-save format, the sum of the exponents and the sign.
2. **Addition:** Carry-propagate addition of the carry-save product to obtain a non-redundant mantissa. This product has twice the wordlength of the input mantissa, so a fast adder will be required (also true of fixed-point).
3. **Normalisation:** If both inputs are normalised i.e. $1 \le X, Y < 2$ the output will be in the range $1 \le P < 4$. So a 1-bit right-shift may be needed to normalise the result. If so, the exponent must also be decremented. The sticky bit, required for IEEE compliant rounding, is also calculated here.
4. **Rounding:** Rounding-to-nearest is performed by examining the msb of the truncated part of the product to determine if it is logic 1. If so, the mantissa is incremented.
5. **Normalisation:** As with addition, rounding can cause the mantissa to overflow if previous to rounding all bits of the mantissa are logic 1. If so, the mantissa must be normalised and the exponent incremented.

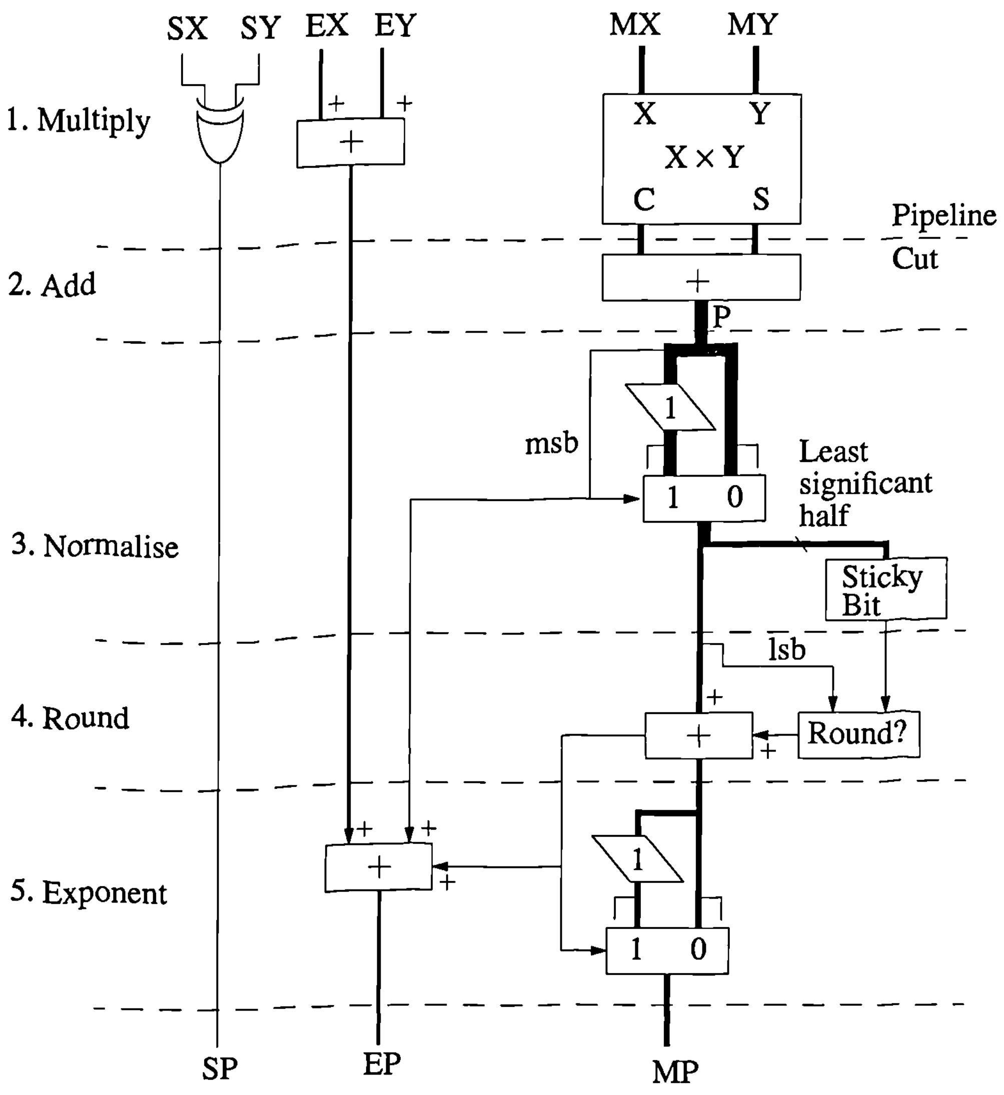


Figure 5.8 Block diagram of the basic floating-point multiplier

5.5.2 Improved Multiplier

The basic multiplier has a latency of 5 cycles assuming the pipeline cuts proposed in Figure 5.8. This has been reduced to 2 by performing the addition, rounding and normalisation in one stage as described below.

The carry-save output of the multiplier is partitioned into an upper and a lower part as shown in Figure 5.9. The carry-save adder on the upper-part may be ignored initially. The upper-part represents the truncated result without the lsb, and the lower-part is used for rounding and determining the lsb. The boundary between the two parts has been chosen so that there can be only one carry from the lower-part to the upper-part resulting from the addition of the carry-save bits and any rounding that may be required. As there is only one carry, there are only two possible results for the upper-part, one with a carry and one without. Both of these two possibilities are calculated in parallel to establish whether, or not, there is overflow.

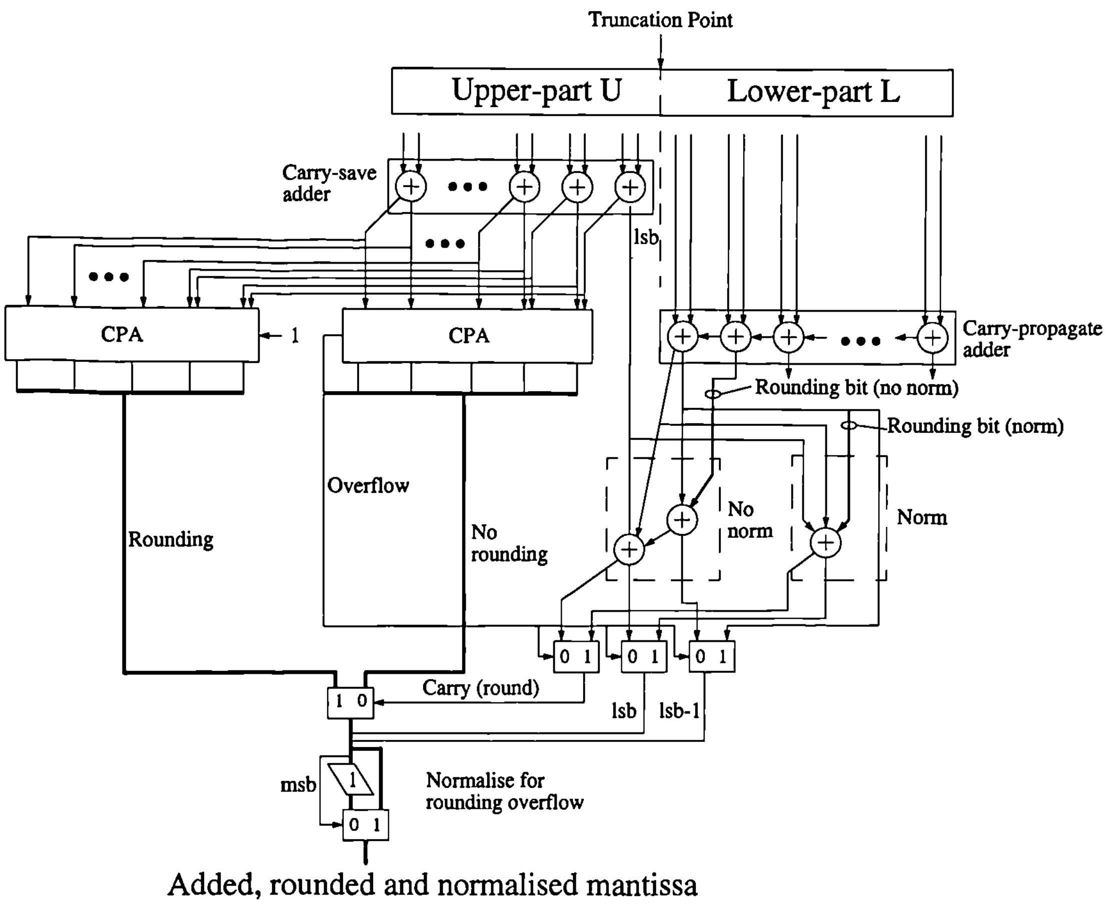


Figure 5.9 Performing the addition, rounding and normalisation together

Arranging for only one carry is achieved by ensuring that the terms remaining in the lower part cannot generate more than one carry when added. Rounding may require the addition of a bit in the position of the lsb or one bit to the right of the lsb (where the latter is required if overflow occurs and the result needs to be normalised by a 1-bit right-shift).

For there to be only one carry, the lower sum must bounded such that $0 \leq \mathrm{L} < 4\mathrm{lsb}$. The upper bound is obtained from the worst case sum $11.11\ldots111$, where the leading bit is the carry. This bound is not met by a simple partitioning of the carry-save number, and so it is first necessary to remove a bit in the lsb position using the carry-save adder as shown in Figure 5.9.

In the right of the figure, the remaining lsb and the lower partition are added and rounded for the cases of normalisation and no normalisation. The correct result is selected once the need to normalise has been established from the addition of the upper part. Also, a multiplexer is used to select the corresponding carry-out of the lower part, and this is used to determine which of the two upper-part adder results should be used. Note that overflow will not occur due to rounding (i.e. a product of 4 cannot be generated from rounding the product of the two largest mantissas).

Figure 5.10 provides a detailed block diagram of the floating-point multiplier implemented. The multiplication is performed using a modified Booths recoded, Wallace-tree multiplier, as this offers high-speed and low-area.

As mentioned previously, the rounding-to-nearest rule, as defined by the IEEE standard, was not used, and so the sticky-bit was not required. If IEEE compliant rounding were required then the approach proposed by Yu [Yu95] could be used for rapid generation of the sticky-bit.

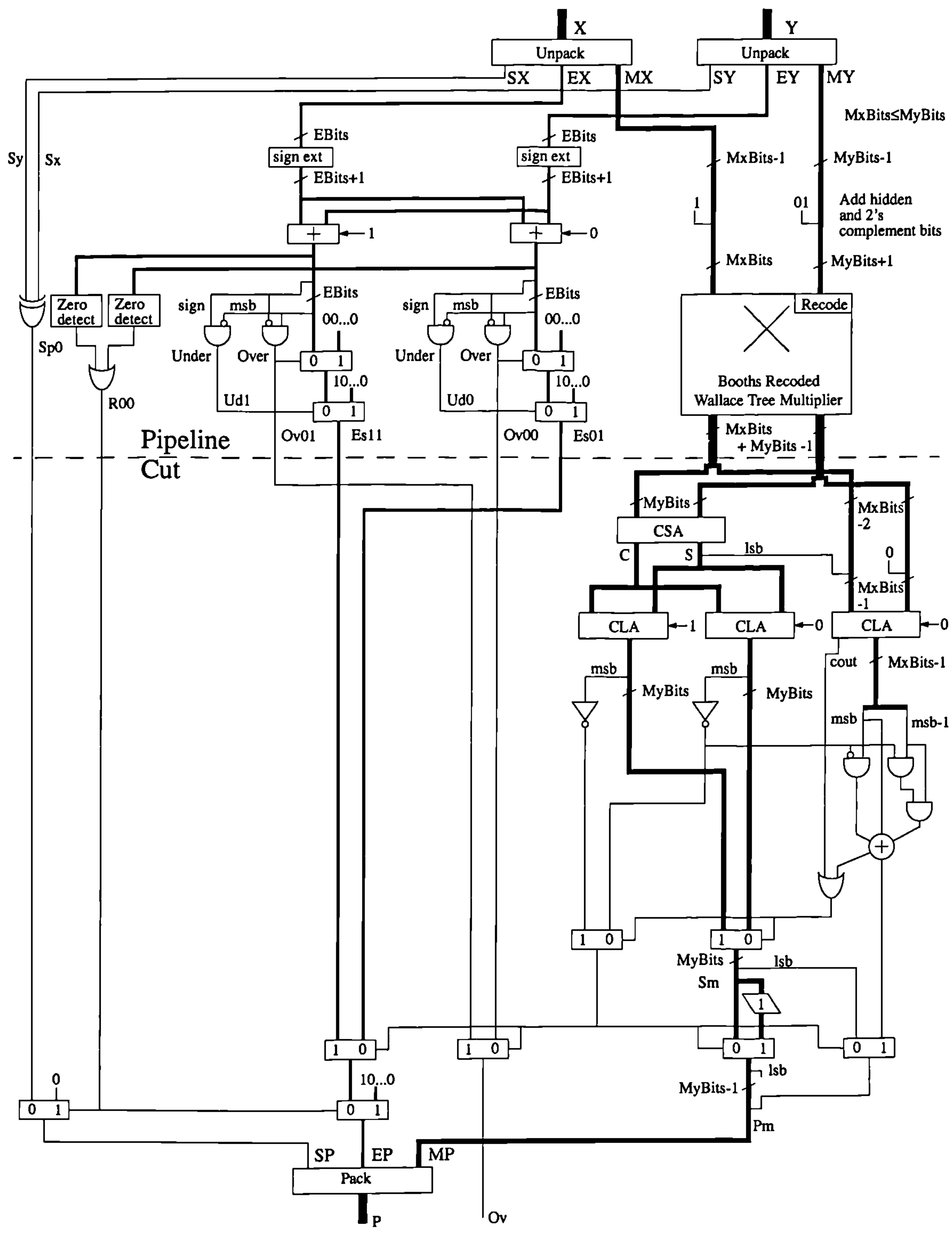


Figure 5.10 Detailed block diagram of floating-point multiplier

5.5.3 Circuit Synthesis Results

Figure 5.11 shows the circuit synthesis results for the floating-point multiplier for a range of wordlengths. Two cases of pipelining are presented, one where latches are only present on the output of the multiplier, and the other where latches are also placed after the Wallace-tree, as indicated in the figure.

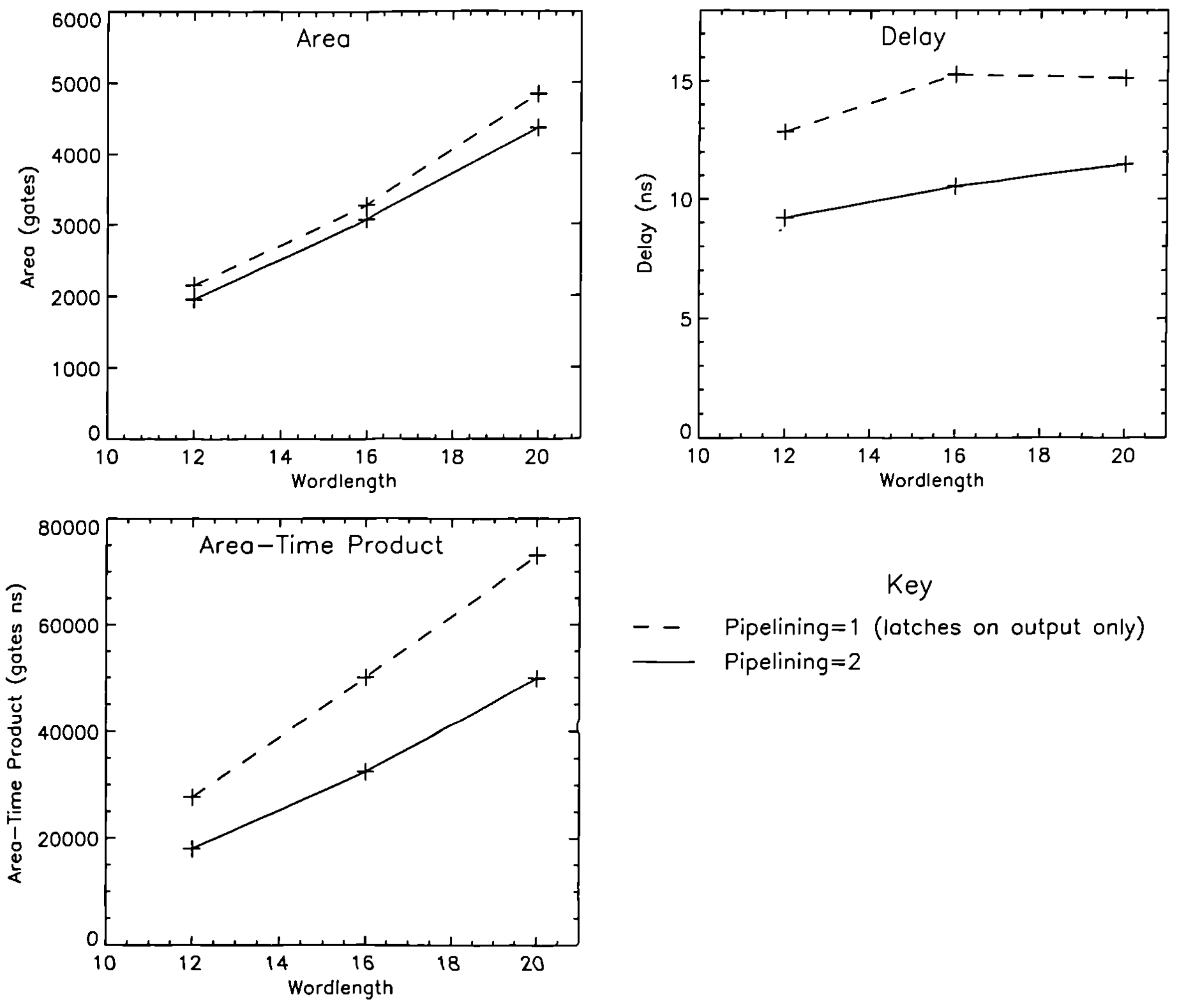


Figure 5.11 Circuit synthesis results for the floating-point multiplier

As with the adder, pipelining actually reduces the number of gates as it breaks down the circuit into smaller parts which can optimised more effectively. Pipelining also gives a large improvement in the stage delay. Increasing the level of pipelining above that shown, would require the Wallace-tree to be partitioned, and the improvement is likely to be small for the wordlengths considered here.

The area has a square-law dependence upon the wordlength, which is not obvious from the graph, but arises from the architecture. The delay will have a logarithmic dependence upon the wordlength due to the use of a Wallace-tree and carry-look-ahead adders.

5.6 Floating-Point Division

As with multiplication, the extra hardware required to implement a floating-point divider over that of a fixed-point one is relatively low. This is particularly true here as practical fixed-point division algorithms also require that the divisor be normalised prior to division. The challenge in implementing the floating-point divider lies in performing rounding and normalisation after division as quickly as possible.

5.6.1 Operation of Basic Floating-Point Divider

The block diagram of the basic floating-point divider is shown in Figure 5.12. It consists of the following stages:

1. **Division:** Here division of the normalised mantissas is performed, and the difference of the exponents and the sign of the quotient are evaluated. The division process is usually very lengthy and needs to be heavily pipelined to achieve high-throughput. The output of the divider-core is generated in two parts, a quotient Q and a remainder Z.
2. **Normalisation:** The mantissas are bounded, i.e. $1 \leq N, D < 2$, so the quotient will be in the range $0.5 < Q < 2$. Therefore, a shift-left operation may be required to normalise the result, and the exponent decremented accordingly.
3. **Rounding:** The sign of the remainder of the division (Z) must be examined to determine if rounding is required, and if so, the quotient must be incremented.
4. **Normalisation:** If overflow occurs after rounding, the mantissa must be normalised by a shift-right operation and the exponent incremented accordingly.

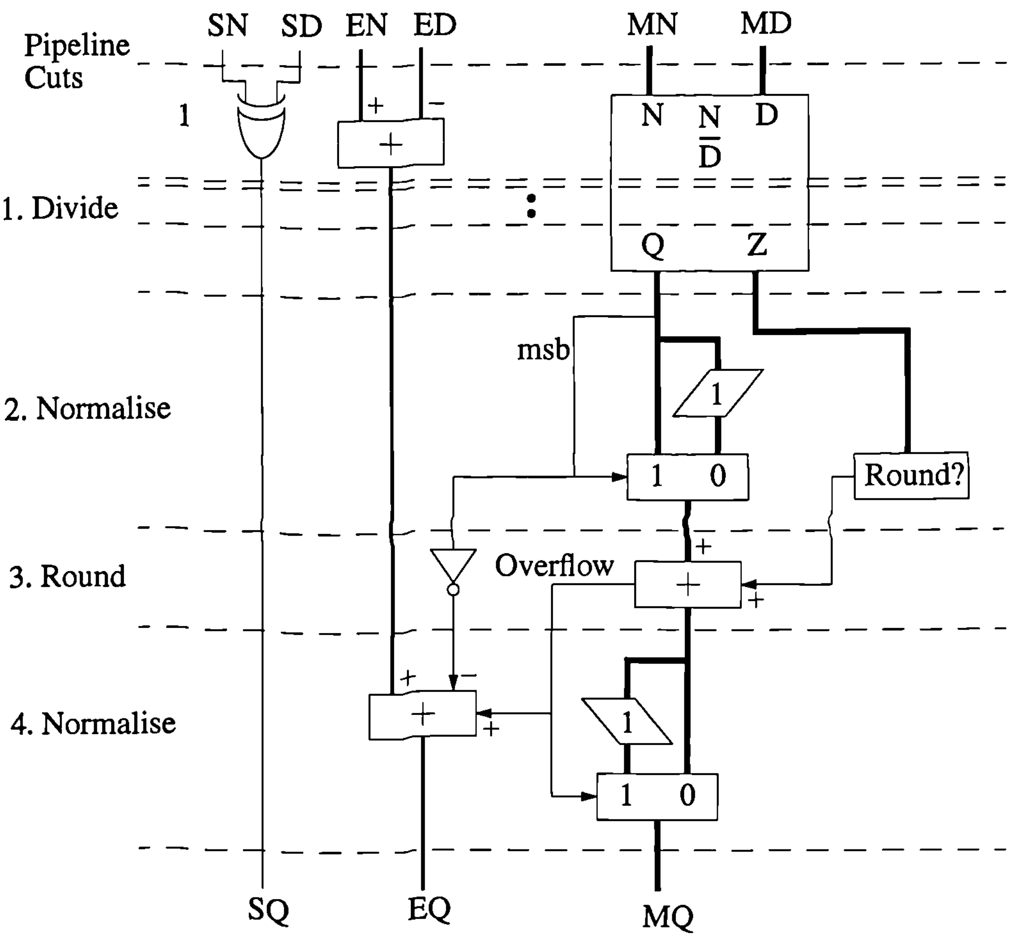


Figure 5.12 Block diagram of a basic floating-point divider

5.6.2 Improved Divider

Figure 5.13 shows a block diagram of the floating-point divider implemented. The modified-SRT algorithm has been used to realise the divider core, as it offers good area-time figures and provides a good degree of flexibility in the choice of pipelining-level to match the speed of other operators in a system. The architecture for the floating-point divider is also directly compatible with a range of recurrence division schemes. A number of optimisations have been made to reduce the time to perform normalisation and rounding, and these are now discussed.

The modified-SRT divider core generates a quotient and a remainder in a signed-binary representation. The quotient is converted to binary within the divider block using an on-the-fly conversion scheme. Two additional bits are generated, one for rounding and another in case normalisation is required. The only other item of information required to determine if rounding should take place is the sign of the remaining quotient bits, which can be established from the sign of the residual. Table 5.2 shows when rounding should be performed.

Table 5.2 Divider rounding

Q (including guard and rounding bits)	Normalisation	Rounding Bit(s)	Sign of Residual Z	Rounded Quotient
01xx...xxx0	Shift-left	0	x	1xx...xxx
01xx...xxx1	Shift-left	1	-	1xx...xxx
01xx...xxx1	Shift-left	1	0/+	1xx...xxx+1
1xxx...xx00		00	x	1xxx...xx
1xxx...xx01		01	x	1xxx..xx
1xxx...xx10		10	-	1xxx..xx
1xxx...xx10		10	0/+	1xxx..xx+1
1xxx...xx11		11	x	1xxx..xx+1

The first column of the table shows the quotient output of the modified-SRT divider with two additional bits. The x's are used to represent bits of Q which are not used to determine the rounding direction. If the msb of the quotient is 0 then normalisation is required. In which case, the first of the additional quotient bits forms the lsb and rounding is based on the second bit. If this bit is 1 and the residual is positive or zero then the combination (which forms the truncated part) is equal to or greater than 0.5lsb, so the quotient should be rounded up by incrementing it. If the residual is negative or the rounding bit is 0, then the truncated part is less than 0.5lsb, and no rounding is performed.

If normalisation is not required, then both additional bits and the sign of the remainder must be used to determine if rounding is required, and the principles presented above are used to determine rounding direction.

In all cases rounding has one of two outcomes. Either the normalised quotient, or an incremented version of it, provides the rounded result. To improve the performance of the rounding circuit an incremented quotient is produced, even if not required, in parallel with the remainder sign detection operation. A multiplexer is used to select the correct quotient when the rounding detection circuit completes.

Note that the incremented version of the result can be obtained from the on-the-fly conversion circuit[Erce89a]. Here a simpler approach was adopted as it does not significantly increase the hardware or critical path of the divider.

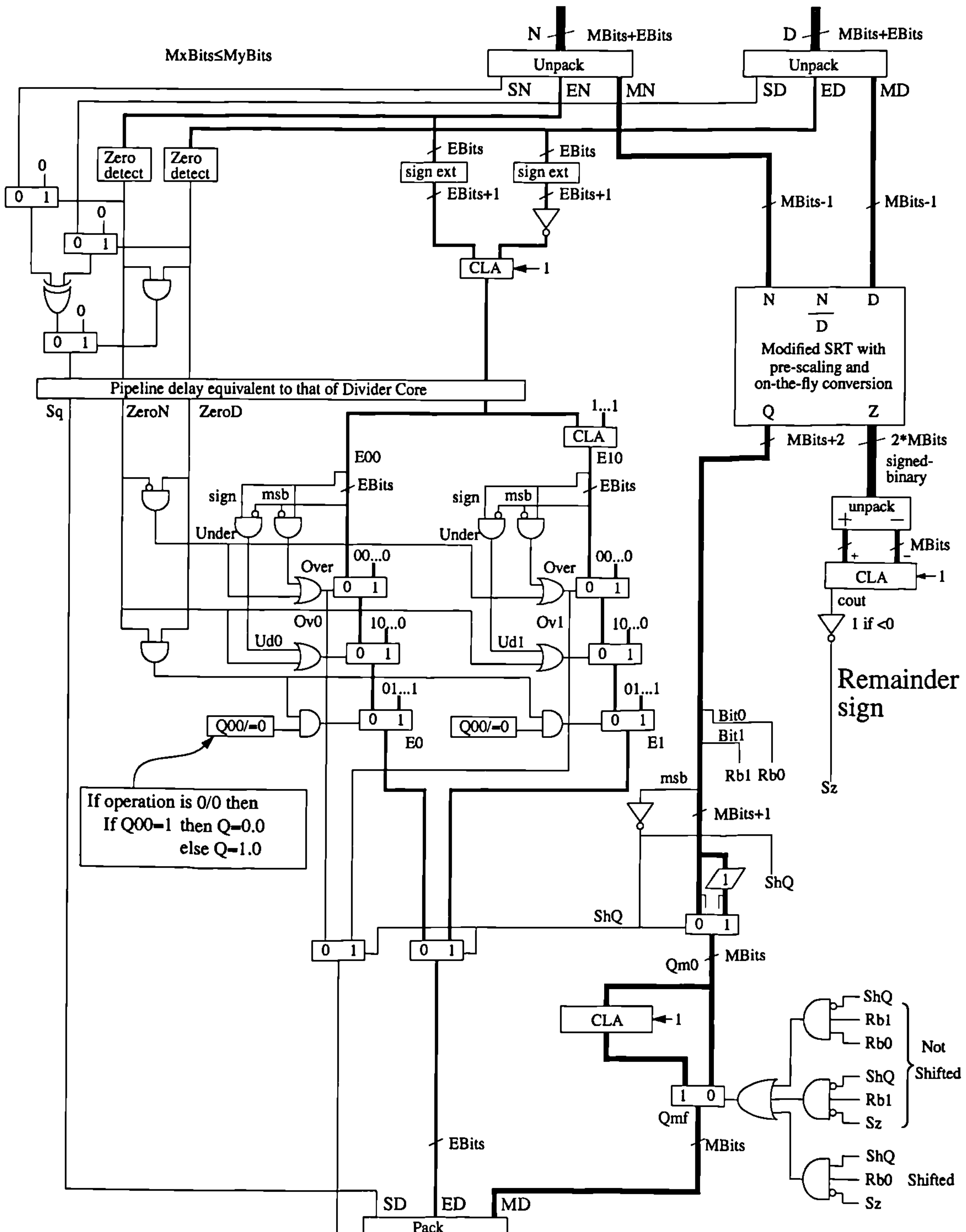


Figure 5.13 Detailed block diagram of floating-point divider

5.6.3 Circuit Synthesis Results

Figure 5.14 shows the circuit synthesis results for the floating-point divider.

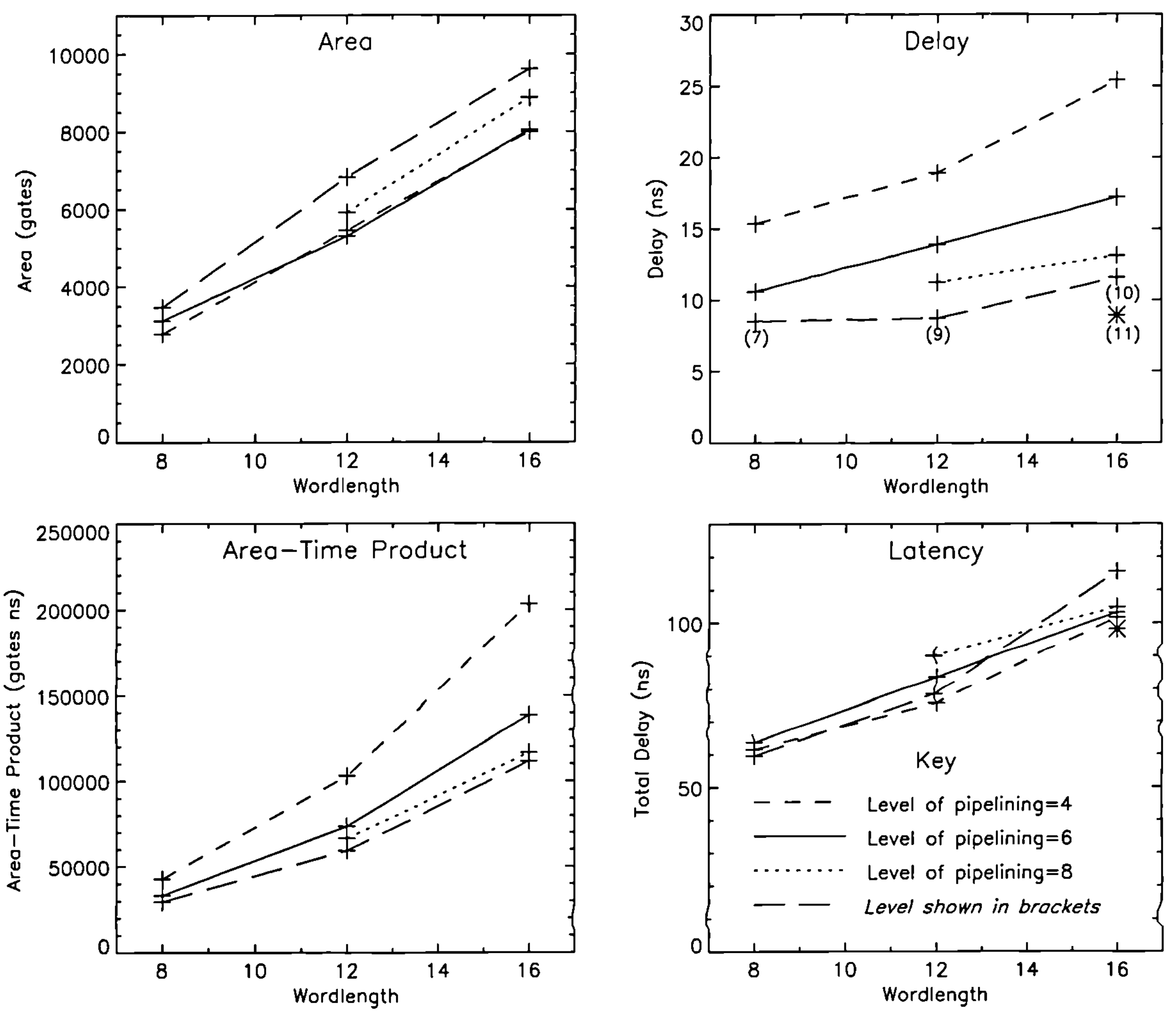


Figure 5.14 Circuit synthesis results for the floating-point divider

The area-time product is improved by increasing the level of pipelining. The maximum level of pipelining used represents a cut every two rows. In this case the delay for the 16-bit divider is not as low as the core presented in Chapter 4. This is because the critical path is in the prescaler. This can be avoided by pipelining the prescaler, which increases the latency to 11, but will reduce the delay to that marked on the figure by the star.

For all pipelining levels, the number of gates is significantly greater than that of the divider core presented in the previous chapter, as two extra quotient bits are generated for normalisation and rounding. Also a large number of latches are required to pipeline the exponent. Note that much of this would also be required by a fixed-point implementation of a divider.

5.7 Other Operators

Other operators which have been developed are:

1. **Rounder:** This reduces the precision of a number by rounding.
2. **Beta Multiplier:** Performs the multiplication $X\beta$ where $\beta = (1 - 2^s)$ and $s > 1$. This requires that X be right-shifted by s-bits, complemented and added to X. Only rounding is performed on the output as normalisation is not required.
3. **Delay Block:** This delays a floating-point number by a fixed number of clock cycles D. The number of gates is $5(\mathrm{EBits} + \mathrm{MBits})\mathrm{D}$ and the delay is approximately 3ns.
4. **Fixed-to-Floating-Point Converter.**

Figure 5.15 shows the circuit synthesis results for the other floating-point operators.

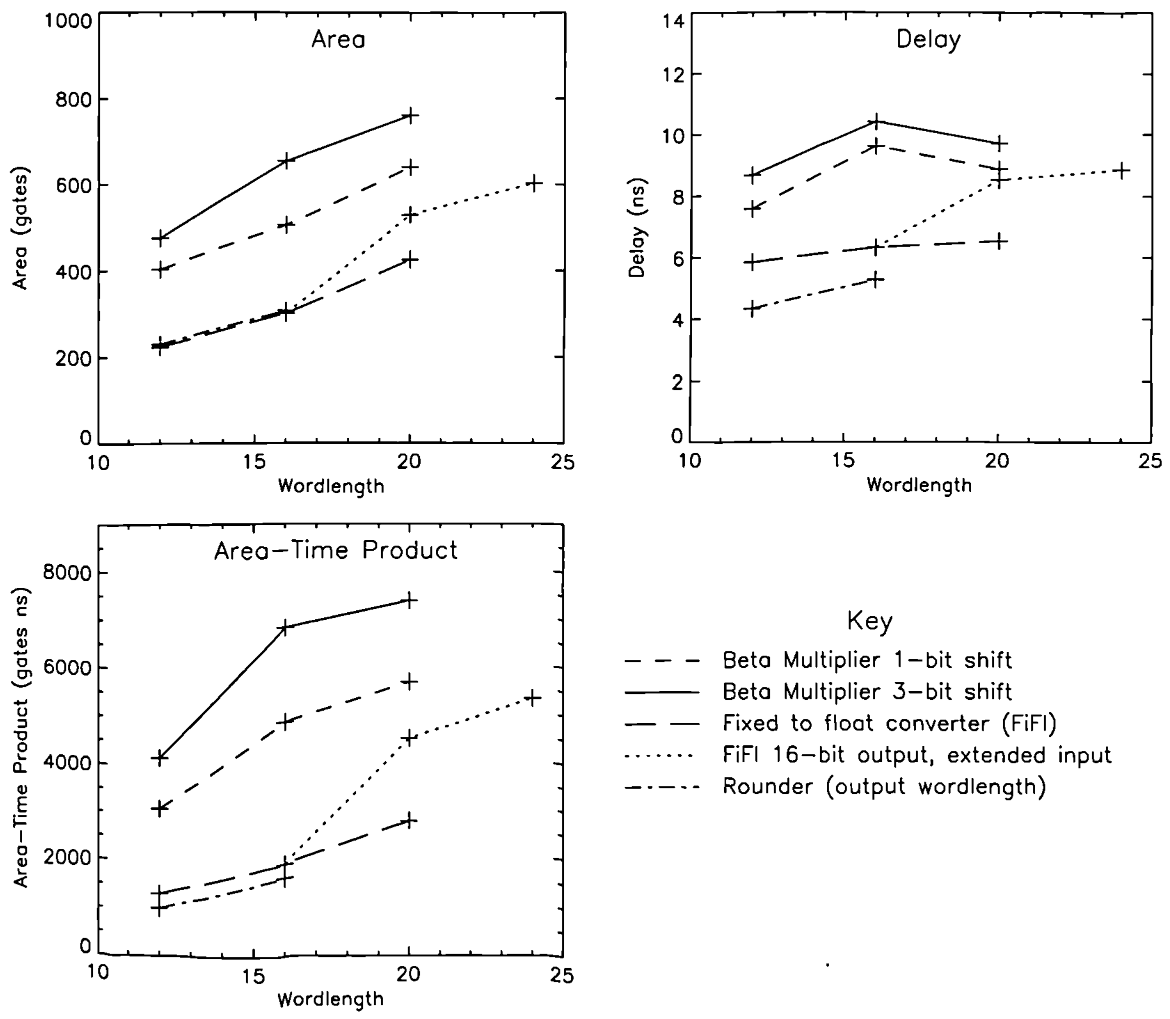


Figure 5.15 Circuit synthesis results for other floating-point operators

5.8 Comments on Floating-Point Operator Designs

A 95MHz throughput rate could be achieved using the floating-point designs presented in this chapter and the fabrication process described in Chapter 1. A rate of 100MHz should be possible by increasing the optimisation effort, but this will require longer computing times. The operator properties are summarised in Table 5.3. In particular, the fixed/floating-point adder provides single-cycle latency on the fixed-point input for all practical wordlengths.

Table 5.3 Summary of the performance of 16-bit floating-point operators

Operator 16-bit mantissa	Second Input Wordlength	100MHz Operation		Max. Clock Frequency
		Latency	Gate Count	
Adder	16	3	2266	105
	32		3996	96.8
Fixed/Floating-Point Adder	16	1	1318	143
	32		2661	136
Multiplier		2	3075	94.9
Divider		11	9600	108

For the operators add, multiply and divide the cost of obtaining floating-point operation is only significant for the adder. In this case, the area has increased by approximately a factor of 10 and the latency by a factor of 3 over the fixed-point design. Therefore, further work to develop a more area-time efficient floating-point adder would be worthwhile.

For multiplication and division, the additional hardware to achieve floating-point operation is small and its effect on latency negligible. The major effort has been to implement rounding, and this would affect fixed-point operators in a similar way. If rounding were not required and truncation used, it would be unnecessary to calculate the lower-half of the product in the multiplier and the remainder in the divider, and considerable savings in area could be made. Therefore, achieving an unbiased error using truncation would be worth investigating. One approach would be to arrange for a signed-binary representation at the output of the divider core, or multiplier tree (as done in Chapter 3). Applying normalisation and truncating at the right point could probably be done using similar techniques to those already applied.

Chapter 6 Givens Rotation Algorithm Variants

In the previous chapters, the design of low-latency fixed- and floating-point operators was considered. The purpose of this work was to identify and develop operators for the implementation of a Givens rotations processor suitable for a high sample-rate implementation of the QR-algorithm for adaptive filters. In this and the next two chapters, the design of a Givens rotation processor and its use in the construction of an adaptive filter are considered.

The Givens rotation is a relatively simple operation, yet there are many forms of it for performing the QR-algorithm. Each of these has properties that significantly influence the performance of any VLSI implementation. In this chapter, the most promising variants of the algorithm are identified and discussed. This results in a wide variety of algorithms; however in the next chapter the properties of the algorithms are examined in more detail and a single variant is identified as offering significant advantages over the others for VLSI implementation. This is followed by a chapter which examines how the Givens rotation processor could be used to construct an adaptive filter for a range of problem sizes.

6.1 Overview of Givens Rotation Algorithm Variants

The use of unitary rotations to triangularise a matrix was proposed by Givens[Give58] in 1958. (Note that Householder[Hous58] was also publishing his method at the time). Since then a number of variants of the algorithm have been developed for various reasons. The conventional Givens rotation (introduced in Chapter 1) requires both square-root and division operations in the boundary cell and four multiplications are needed to implement the rotation within the internal cells of the array. The square-root and divide operations are widely regarded as much more complex than multiply. Gentleman[Gent73] provided the first variant that did not require a square-root operation; the number of multiplications required to implement the rotation in the internal cell was also reduced to three in one variant, and at Golub's suggestion to two in another. The two-multiplier variant was derived using the output of the cell to update the stored parameters r and u (and is referred to as the *square-root-free, x-feedback* (SQF-XFB) algorithm). This algorithm halved the number of multiply operations within the internal cell and so reduced the number of computations required to implement the QR-algorithm by almost one third. Hammarling [Hamm74] observed, however, that significant numerical errors could arise in the two-multiply variant and it was not possible to guarantee stability when this form is used. He proposed five two-multiplier solutions as alternatives. More recently Döhler[Döhl91] proposed another, which is a simplified variant of one of these solutions, to give what he called the *Squared Givens Rotation (SGR)*. It will be shown in Chapter 7 that this variant offers particular benefits in achieving an efficient application specific VLSI implementation of the Givens rotation.

The variants of the Givens rotation which are square-root-free and also use only two multiplications to perform the rotations in the internal cell are known as *fast Givens rotations*. Although these offer reduced numbers of computations, there is still some concern over their numerical properties[Anda94]. One reason for this concern is that numbers may overflow or underflow, and the effort required to monitor these conditions is high. Hammarling[Hamm74] proposed that underflow be overcome by storing an exponent separately, by normalising occasionally, or performing row interchanges. The latter option makes the design of a good VLSI architecture difficult - one of the reasons why an approach based on Givens rotations was adopted. Barlow and Ipsen[Barl87] proposed a scaling approach based on powers of 2 so that it could be implemented using shifters. Unfortunately, their algorithm uses 4 multipliers in the internal cell.

Golub and Van Loan[Golu89] use a combination of two fast algorithms which can be selected depending upon the rotation to be performed so that the scaling is limited to 2 on each iteration. Anda *et al.* [Anda94] have taken this one step further by using a combination of two algorithms which act either to increase or reduce the stored quantity in the boundary cells. By selecting the appropriate rotation it is possible to ensure that the scaling of each row is maintained close to unity (in fact the scaling factor, s, is bounded such that $\frac{1}{\sqrt{2}} \le s \le \sqrt{2}$).

In the variants mentioned so far, division operations are still required to calculate the rotation parameters in the boundary cell. This operation is of similar VLSI circuit complexity as the square-root operation, so to avoid division in both the boundary and internal cells Götze and Schwiegelshohn[Götz91] developed the *divide-and-square-root-free* variant of the Givens rotation. Only multiplications and additions are required by the boundary and internal cells, but four multiplications are required in the internal cells to perform the rotation. Another problem associated with removing the division is that the dynamic range requirements of the arithmetic increase dramatically, although this can be overcome relatively easily by scaling the variables by powers of two[Götz91][Fran94].

The QR-algorithm is usually implemented using floating-point arithmetic due to dynamic range requirements of variables. Fixed-point implementations are simple, and therefore attractive, and possible by applying scaling to the input of the conventional Givens rotation algorithm. However, scaling must be precalculated, and the numerical errors introduced can be relatively high. To address this issue McWhirter, Walke and Kadlec[McWh95] recently presented the normalised Givens rotation, in which scaling is applied dynamically based upon the energy of the signal in each cell. This ensures fixed-point number ranges. However, the cost of normalising at each cell can be high. Therefore, two new algorithms, which are presented later in this chapter, have been developed which apply normalisation to either whole columns or the whole array. In summary, the algorithms can be grouped into the following classes:

- Conventional algorithm
- Square-root-free algorithms
  - Scaled-Givens rotations
  - Hammarling's two-multiply rotation
  - Squared-Givens rotations
- Divide-and-square-root-free algorithm
- Normalised algorithms

All but the normalised algorithms can be described using a generalised form of the Givens rotation equations. This is presented next and then used to derive some of the more relevant algorithm variants.

6.1.1 Generalised Givens Rotation Algorithm

The following generalised Givens rotation equations were first proposed by Gentleman[Gent73], and similar approaches have been used in the literature subsequently[Hamm74][Hsie93]. As indicated in Chapter 1, the Givens rotation is a 2-dimensional unitary rotation which eliminates an element of a matrix. In the QR-array shown in Figure 1.4, the rotation is used to eliminate the input to the boundary cell, x_B, i.e.

$$\begin{bmatrix} c & s^* \\ -s & c \end{bmatrix} \begin{bmatrix} \beta r_B & \beta r_I \\ x_B & x_I \end{bmatrix} = \begin{bmatrix} r_B' & r_I' \\ 0 & x_I' \end{bmatrix} \tag{6.1}$$

where r_B represents the stored parameter in the boundary cell, and x_I and r_I are, respectively, the input and stored parameter of one of the internal cells in the same row as the boundary cell. In practice the rotation is applied to the parameters associated with all the internal cells in a row, but only one set is shown in equation (6.1) for clarity. The results of the rotation are the internal cell output x_I' and the boundary and internal cell parameters r_B' and r_I' updated to time t_n (i.e. elements of $\mathbf{R}(n)$).

Note that later in the chapter, when dealing with an array of cells, the cell to which a variable relates is indicated by subscripts. For example, the input to the cell on the ith row and the jth column is denoted by $x_{i,j}(n)$.

For the purposes of generality, the Givens rotation has been described for the case when x, and consequently r, are complex quantities, as this is required for adaptive beamforming. In this case, s is also complex quantity and s* represents its complex conjugate. The rotation equations for the case of real x are a degenerate case of the equations for complex x.

The Givens rotation can be generalised by introducing two scaling terms using the following substitution

$$\begin{bmatrix} r_B & r_I \\ x_B & x_I \end{bmatrix} = \begin{bmatrix} d^{1/2} & 0 \\ 0 & \delta^{1/2} \end{bmatrix} \begin{bmatrix} \bar{r}_B & \bar{r}_I \\ \bar{x}_B & \bar{x}_I \end{bmatrix} \tag{6.2}$$

Making this substitution in equation (6.1) yields

$$\begin{bmatrix} c & s^* \\ -s & c \end{bmatrix} \begin{bmatrix} \beta d^{1/2}\bar{r}_B & \beta d^{1/2}\bar{r}_I \\ \delta^{1/2}\bar{x}_B & \delta^{1/2}\bar{x}_I \end{bmatrix} = \begin{bmatrix} d'^{1/2}\bar{r}_B' & d'^{1/2}\bar{r}_I' \\ 0 & \delta'^{1/2}\bar{x}_I' \end{bmatrix} \tag{6.3}$$

where the two rotation parameters are given by

$$s = \frac{\delta^{1/2}\bar{x}_B}{d'^{1/2}\bar{r}_B'} \qquad c = \frac{\beta d^{1/2}\bar{r}_B}{d'^{1/2}\bar{r}_B'} \tag{6.4}$$

Equation (6.3) can be rearranged to express the rotation in terms of a new transformation matrix acting on the new variables i.e.

$$\begin{bmatrix} \frac{\beta c d^{1/2}}{d'^{1/2}} & \frac{s^*\delta^{1/2}}{d'^{1/2}} \\ -\beta\frac{s d^{1/2}}{\delta'^{1/2}} & \frac{c\delta^{1/2}}{\delta'^{1/2}} \end{bmatrix} \begin{bmatrix} \bar{r}_B & \bar{r}_I \\ \bar{x}_B & \bar{x}_I \end{bmatrix} = \begin{bmatrix} \bar{r}_B' & \bar{r}_I' \\ 0 & \bar{x}_I' \end{bmatrix} \tag{6.5}$$

Hence, the updated values are given by

$$\bar{r}_B'^2 = \frac{\beta^2 d\bar{r}_B^2 + \delta|\bar{x}_B|^2}{d'} \tag{6.6}$$

$$\bar{r}_I' = \left(\frac{\beta c d^{1/2}}{d'^{1/2}}\right)\bar{r}_I + \left(\frac{s^*\delta^{1/2}}{d'^{1/2}}\right)\bar{x}_I \tag{6.7}$$

$$\bar{x}_I' = \left(\frac{c\delta^{1/2}}{\delta'^{1/2}}\right)\bar{x}_I - \left(\frac{\beta s d^{1/2}}{\delta'^{1/2}}\right)\bar{r}_I \tag{6.8}$$

where equation (6.6) has been obtained by making the substitution for s and c as defined in equation (6.4).

For the standard Givens rotation, the residual is given by $\gamma(n)\alpha(n)$, where $\gamma(n)$ is the product of the rotation cosines down the array diagonal and $\alpha(n)$ is the output from the bottom cell of the right-hand column i.e. $\alpha(n) = x_{p,p}(n)$ (noting that p is the number of array inputs). In the generalised case, this output is $\bar{\alpha}(n)$, where $\alpha(n) = \delta_p^{1/2}(n)\bar{\alpha}(n)$. So the residual is

$$e(n) = \gamma(n)\delta_p^{1/2}(n)\bar{\alpha}(n) \tag{6.9}$$

This completes the definition of the generalised Givens rotation. The new equations contain the freedom to specify some constraints on the variables. In the following sections specific variants of the Givens rotation are obtained by choosing appropriate constraints.

6.2 Square-Root-Free Algorithm

The conventional Givens rotation formula contains a square-root operation in the boundary cell, however, a range of square-root-free variants have been developed.

6.2.1 Gentleman's Solution

If substitutions for s and c are made in equation (6.8), the generalised internal cell equation for the cell output takes the form

$$\bar{x}_I' = \beta\sqrt{\frac{d\delta}{d'\delta'}}\left(\frac{\bar{r}_B\bar{x}_I - \bar{x}_B\bar{r}_I}{\bar{r}_B'}\right) \tag{6.10}$$

If δ is updated using the formula

$$\delta' = \frac{\beta^2 d}{d'}\delta \tag{6.11}$$

then the square-root in equation (6.10) is avoided. There is also sufficient freedom to impose the condition $\bar{r}_B = \bar{r}_B' = 1$, avoiding the need for a square-root operation to compute $\bar{r}_B'$ in the boundary cell. The boundary cell equations become

$$d' = \beta^2 d + \delta|\bar{x}_B|^2 \tag{6.12}$$

$$c = \beta^2\frac{d}{d'} \qquad s = \delta\frac{\bar{x}_B}{d'} \tag{6.13}$$

$$\delta' = c\delta \tag{6.14}$$

and the internal cell equations

$$\bar{x}_I' = \bar{x}_I - \bar{x}_B \bar{r}_I \tag{6.15}$$

$$\bar{r}_I' = s^* \bar{x}_I + \bar{x}_B \bar{r}_I \tag{6.16}$$

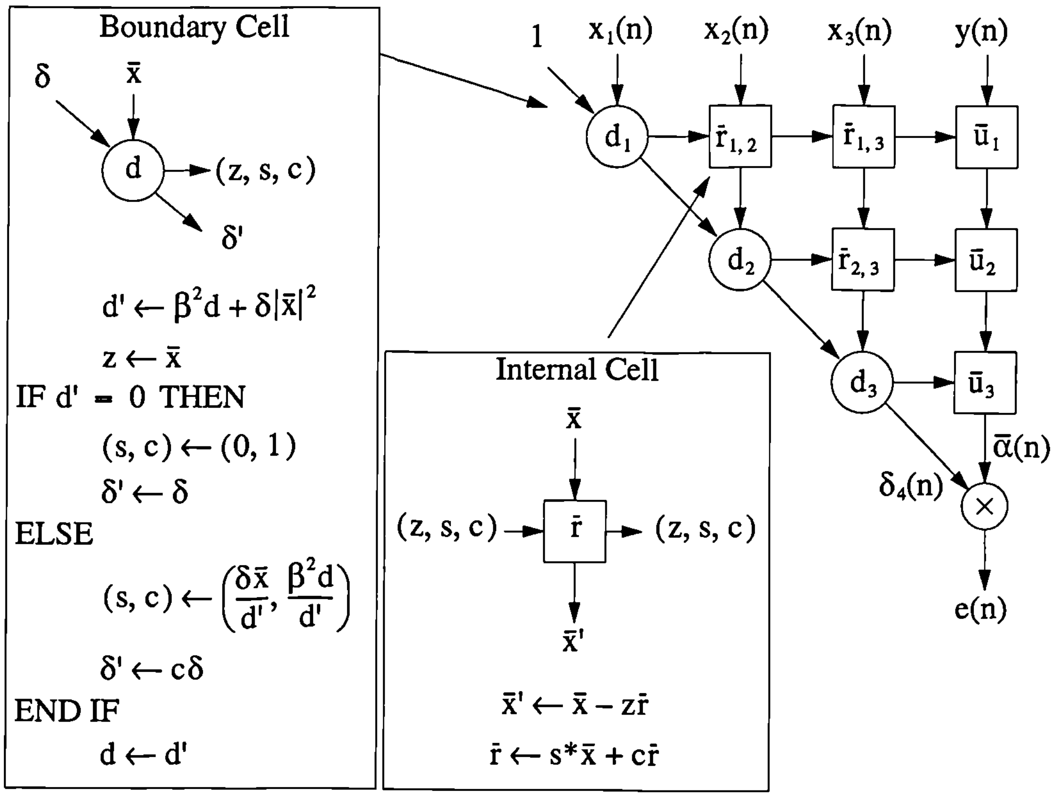
The resulting algorithm was proposed by Gentleman[Gent73] and is summarised in Figure 6.1.

The least squares residual is given by $e(n) = \delta_p^{1/2}\bar{\alpha}(n)\gamma(n)$, where $\bar{\alpha}(n)$ is the output from the bottom cell in the right hand column at time t_n. If $c_i(n)$ is defined to be the cosine generated by the boundary cell on the ith row of the array at time t_n, then

$$\begin{aligned}\gamma(n) &= \left(\prod_{i=1}^{p} c_i(n)\right) = \prod_{i=1}^{p-1} \frac{\delta_{i+1}^{1/2}(n)}{\delta_i^{1/2}(n)} \\ &= \delta_1^{1/2}(n) \cdot \frac{\delta_2^{1/2}(n)}{\delta_1^{1/2}(n)} \cdot \ldots \cdot \frac{\delta_p^{1/2}(n)}{\delta_{p-1}^{1/2}(n)} = \delta_p^{1/2}(n)\end{aligned} \tag{6.17}$$

and hence

$$e(n) = \delta_p(n)\bar{\alpha}(n) \tag{6.18}$$



```
$d' \leftarrow \beta^2 d + \delta|\bar{x}|^2$
$z \leftarrow \bar{x}$
IF $d' = 0$ THEN
    $(s, c) \leftarrow (0, 1)$
    $\delta' \leftarrow \delta$
ELSE
    $(s, c) \leftarrow \left(\frac{\delta\bar{x}}{d'}, \frac{\beta^2 d}{d'}\right)$
    $\delta' \leftarrow c\delta$
END IF
    $d \leftarrow d'$
```

```
$\bar{x}' \leftarrow \bar{x} - z\bar{r}$
$\bar{r} \leftarrow s^*\bar{x} + c\bar{r}$
```

Figure 6.1 Summary of square-root-free algorithm

6.2.2 Golub's Solution

The square-root-free algorithm presented above requires three multiplications in the internal cell and three parameters to describe the rotation. Golub suggested a more efficient form which requires only two multiplications and two additions. It is obtained from Gentleman's solution by using the cell output $\bar{x}_I'$, rather than its input, to update $\bar{r}_I$. Substituting equation (6.15) for $\bar{x}_I$ into equation (6.16) gives

$$\begin{aligned} \bar{r}_I' &= s^*(\bar{x}_I' + z\bar{r}_I) + c\bar{r}_I' \\ &= s^*\bar{x}_I' + (s^*z + c)\bar{r}_I \end{aligned} \qquad (6.19)$$

But from equation (6.12) and (6.13) it follows that $s^*z + c = 1$ and so

$$\bar{r}_I' = \bar{s}^*\bar{x}_I' + \bar{r}_I \qquad (6.20)$$

This solution requires only two multiplies in the internal cell. Furthermore, only two quantities (s and z) are required to represent the rotation, and this reduces the communication costs of an implementation.

Hammarling [Hamm74] observed that it is no longer possible to guarantee stability when using Golub's solution with finite precision arithmetic. This can be attributed to the use of $1 - s^*z$ instead of c in the internal cells. When c is small, this substitution is poor and numerical errors can be relatively high. This is particularly significant during the initialisation phase leading to the extremely slow convergence observed in simulations. However, once convergence has been attained and $c \rightarrow \beta^2$, the substitution is more accurate and a significant reduction in the numerical errors has been observed.

6.2.3 Hammarling's Two-Multiply Rotations

As alternatives to Golub's solution, Hammarling [Hamm74] proposed another five two-multiply rotations. These were obtained by choosing relationships to update the scale factors so that two of the multipliers applied in the internal variables reduce to unity. These cases can be identified by considering the generalised equations applicable in the internal cell as given by equations (6.7) and (6.8). Four possible cases can be directly obtained; the first by choosing $d'^{1/2} = \beta c d^{1/2}$ and $\delta'^{1/2} = c\delta^{1/2}$. This gives the following two-multiply formulae

$$\bar{r}_I' = \bar{r}_I' + a\bar{x}_I' \tag{6.21}$$

$$\bar{x}_I' = \bar{x}_I - b\bar{r}_I \tag{6.22}$$

where

$$a = \frac{s^*\delta^{1/2}}{d'^{1/2}} = \frac{\delta\bar{x}_B^*}{d'\bar{r}_B'} \qquad b = \frac{\beta s d^{1/2}}{\delta'^{1/2}} = \beta\frac{\delta^{1/2}\bar{x}_B^* d^{1/2}}{d'^{1/2}\bar{r}_B'\delta'^{1/2}} = \frac{d^{1/2}\delta^{1/2}}{d'^{1/2}c\delta^{1/2}}\frac{\bar{x}_B^*}{\bar{r}_B'} = \frac{\bar{x}_B^*}{\bar{r}_B'} \tag{6.23}$$

In the boundary cell, the scaling-factor update equations become

$$d' = \frac{\beta^2 d}{\kappa} \qquad \delta' = \frac{\delta}{\kappa} \tag{6.24}$$

where

$$\kappa = \frac{1}{c^2} = \beta^2 + \frac{\delta|\bar{x}_B|^2}{d\bar{r}_B^2} \tag{6.25}$$

Also, using the first substitution (i.e. $d'^{1/2} = \beta c d^{1/2}$) and the expression for c given in equation (6.4), the expression to update $\bar{r}_B$, given by equation (6.6), becomes

$$\bar{r}_B' = \bar{r}_B + \frac{\delta|\bar{x}_B|^2}{\beta^2 d\bar{r}_B} = \frac{\kappa r_B}{\beta^2} \tag{6.26}$$

The least squares residual is computed using $e(n) = \delta_p^{1/2}\bar{\alpha}(n)\gamma(n)$, where $\bar{\alpha}(n)$ is the output from the bottom cell in the right hand column, and

$$\begin{aligned}\gamma(n) &= \left(\prod_{i=1}^{p} c_i(n)\right) = \prod_{i=1}^{p-1}\frac{\delta_{i+1}^{1/2}(n)}{\delta_i^{1/2}(n)} \\ &= \delta_1^{1/2}(n)\cdot\frac{\delta_2^{1/2}(n)}{\delta_1^{1/2}(n)}\cdot\ldots\cdot\frac{\delta_p^{1/2}(n)}{\delta_{p-1}^{1/2}(n)} = \delta_p^{1/2}(n)\end{aligned} \tag{6.27}$$

Therefore $e(n) = \delta_p(n)\bar{\alpha}(n)$.

A simplification of Hammarling's solutions can be obtained by defining two new variables

$$k = \frac{1}{d} \qquad q = \frac{1}{\delta} \tag{6.28}$$

The boundary cell equations become

$$a = \frac{k\overline{x}_B}{\beta q\overline{r}_B} \qquad b = \frac{\overline{x}_B}{\overline{r}_B} \tag{6.29}$$

$$k' = k\kappa \qquad q' = q\kappa \qquad \bar{r}_B = \frac{\bar{r}_B\kappa}{\beta^2} \tag{6.30}$$

where

$$\kappa = \beta^2 + \frac{k|\overline{x}_B|^2}{q\overline{r}_B^{\,2}} \tag{6.31}$$

Consequently, κ is greater than β^2 and will increase the diagonal matrix scaling terms (d and δ) on each iteration. Potentially, the growth can be quite high, but Golub and Van Loan[Golu89] avoid this by ensuring that $\kappa \le 2$ by selectively using a second two-multiply algorithm which meets this condition when the first does not.

The substitutions which enabled the other three of Hammarling's direct two-multiply solutions to be obtained are listed in Table 6.1. Further solutions are possible by using one of the updated internal cell variables to calculate the other. In this case the rotation matrix can be written as the product of two matrices, where each uses only one multiplication to update one of the variables. The combined effect, as shown below, is to perform three multiplications.

$$\begin{bmatrix} 1 & 0 \\ v & 1 \end{bmatrix}\begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & \alpha \\ v & 1+\alpha v \end{bmatrix} \tag{6.32}$$

For example, if $\bar{r}_I$ is updated first and $d'^{1/2} = \beta c d^{1/2}$ then

$$\bar{r}_I' = \bar{r}_I + a\bar{x}_I \tag{6.33}$$

where

$$a = \frac{s^*\delta^{1/2}}{d'^{1/2}} \tag{6.34}$$

Now equation (6.33) can be rewritten as

$$\bar{r}_I = \bar{r}_I' - a\bar{x}_I \tag{6.35}$$

and substituted into equation (6.8) to give

$$\bar{x}_I' = \frac{c\delta^{1/2}}{\delta'^{1/2}}\bar{x}_I - \frac{\beta s d^{1/2}}{\delta'^{1/2}}\bar{r}_I' + \frac{\delta^{1/2}s^*s}{\delta'^{1/2}}\left(\frac{\beta d^{1/2}}{d'^{1/2}}\right) \tag{6.36}$$

the term in brackets can be replaced by $\frac{1}{c}$ and the equation rearranged to give

$$\bar{x}_I' = \frac{\delta^{1/2}\bar{x}_I(c^2 + ss^*) - csd^{1/2}\bar{r}_I'}{c\delta'^{1/2}} \tag{6.37}$$

But $c^2 + ss^* = 1$, so

$$\bar{x}_I' = \frac{\delta^{1/2}}{c\delta'^{1/2}}\bar{x}_I - \frac{sd^{1/2}}{\delta'^{1/2}}\bar{r}_I' \tag{6.38}$$

Now there are two further solutions which reduce this second equation to just one multiply. These are $\delta'^{1/2} = \frac{\delta^{1/2}}{c}$ or $\delta'^{1/2} = sd^{1/2}$.

Two solutions based on this approach were suggested by Hammarling and are provided in Table 6.1. Further analysis suggests that there are another 14 solutions. All 16 solutions are listed in Table 6.2. The implications of each solution on the signal flow graph of the internal cell is considered in the next chapter.

6.2.4 Squared Givens Rotation

Döhler[Döhl91] developed a simplified version of one of Hammarling's two-multiply solutions by making the substitutions $d = \frac{1}{\bar{r}_B}$, $d' = \frac{1}{\bar{r}_B'}$ and $\delta' = c^2\delta$ into the generalised equations. The internal cell formulae are

$$\bar{r}_I' = \beta^2\bar{r}_I + a\bar{x}_I \tag{6.39}$$

$$\bar{x}_I' = \bar{x}_I - b\bar{r}_I \tag{6.40}$$

$$a = \delta\bar{x}_B^* \qquad b = \frac{\bar{x}_B}{\bar{r}_B} \tag{6.41}$$

The boundary cell formula are simplified to

$$\overline{r_B}' = \beta^2\overline{r_B} + \delta|\overline{x_B}|^2 \qquad \delta' = \beta^2\frac{\overline{r_B}}{\overline{r_B}'}\delta \tag{6.42}$$

This algorithm is known as the Squared-Givens Rotation (SGR) and is summarised in Figure 6.2.

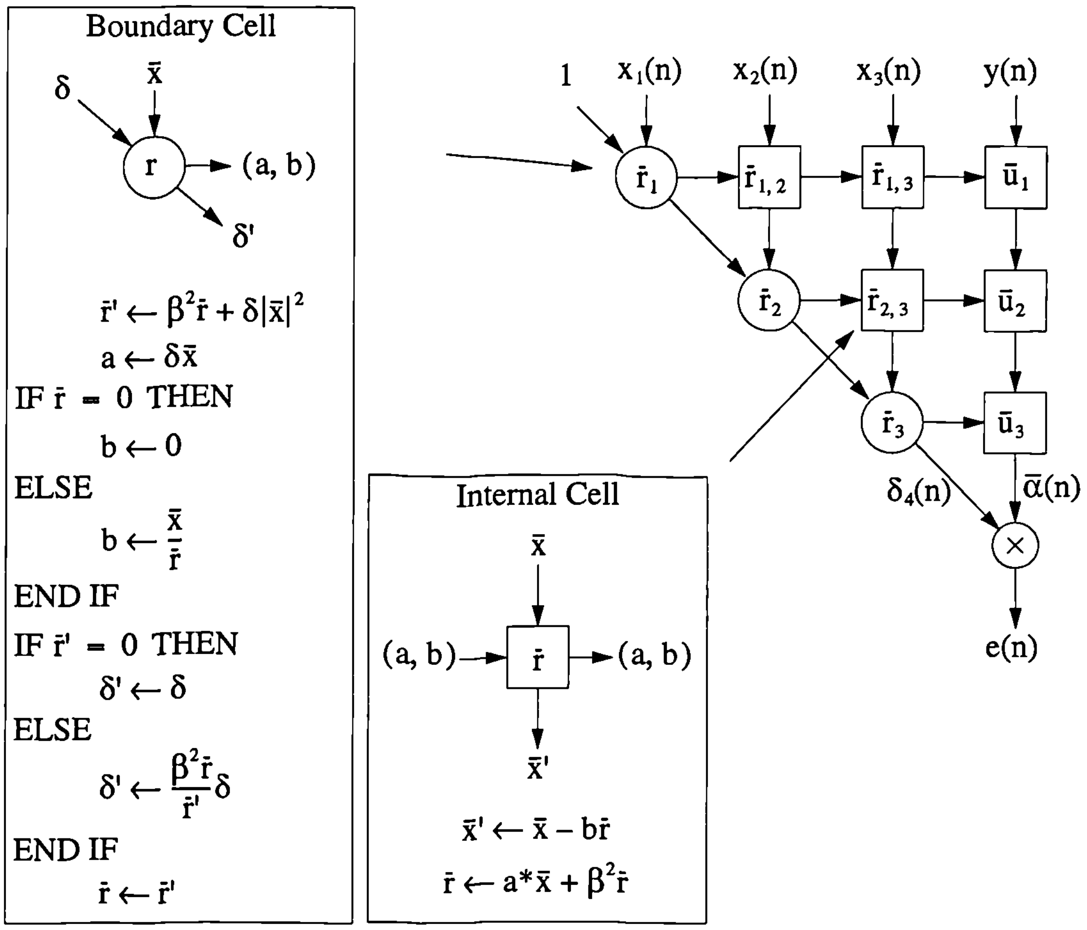


Figure 6.2 Summary of SGR algorithm

6.3 Divide-and-Square-Root-Free Algorithm

A divide-and-square-root-free (DSF) algorithm can be obtained by choosing the updates

$$d'\bar{r}_B' = d\delta \qquad \delta' = \frac{1}{\bar{r}_B'} \tag{6.43}$$

and defining new variables $k = \frac{1}{d}$ and $q = \frac{1}{\delta}$. The boundary cell equations then become

$$\bar{r}_B' = \beta^2 q \bar{r}_B^{\,2} + k|\bar{x}_B|^2 \tag{6.44}$$

$$k' = kq\bar{r}_B' \qquad q' = \bar{r}_B' \tag{6.45}$$

The internal cell formula become

$$\bar{r}_I' = \left(\frac{\beta^2 d\bar{r}_B}{d'\bar{r}_B'}\right)\bar{r}_I + \left(\frac{\delta\bar{x}_B^*}{d'\bar{r}_B'}\right)\bar{x}_I \qquad (6.46)$$

$$= \beta^2\bar{r}_B'\bar{r}_B\bar{r}_I + k\bar{x}_B^*\bar{x}_I$$

and

$$\bar{x}_I' = \beta(\bar{r}_B\bar{x}_I - \bar{x}_B\bar{r}_I) \qquad (6.47)$$

The DSF algorithm is summarised in Figure 6.3.

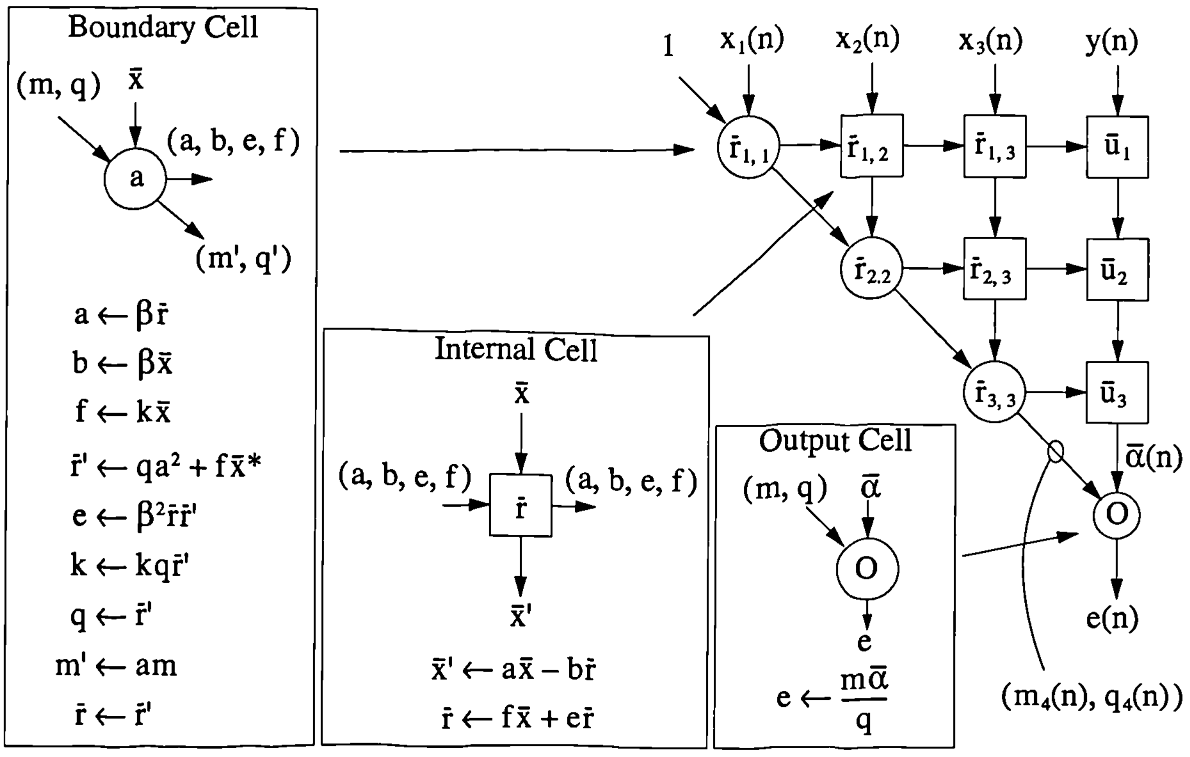


Figure 6.3 Summary of the DSF algorithm

The least-squares residual is computed using $e(n) = \gamma(n)\bar{\alpha}(n) = \gamma(n)\delta^{1/2}\alpha(n)$, where

$$\gamma(n) = \left(\prod_{i=1}^{p-1} c_i(n)\right) = \prod_{i=1}^{p-1}\frac{\beta q_i^{1/2}\bar{r}_{i,i}}{q_{i+1}}$$

$$= \frac{\beta\bar{r}_{1,1}}{q_1^{1/2}}\cdot\frac{\beta q_1^{1/2}\bar{r}_{2,2}}{q_2^{1/2}}\cdot\ldots\cdot\frac{\beta q_{p-1}^{1/2}\bar{r}_{p-1,p-1}}{q_p^{1/2}} = \frac{\prod_{i=1}^{p-1}\beta\bar{r}_{i,i}}{q_p^{1/2}} = \frac{m_{p+1}}{q_p^{1/2}} \qquad (6.48)$$

Therefore $e(n) = \frac{\bar{\alpha}(n)}{q_p}$. This is the only division required by the algorithm.

6.4 Summary of Givens Rotation Algorithms

Table 6.1 summarises the choices required to obtain some of the more important Givens rotation algorithm variants.

Table 6.1 Summary of Givens rotation algorithms

Givens Rotation	Substitution
Conventional, Givens[Give58]	$d = d' = 1 \qquad \delta = \delta' = 1$
Square-Root-Free (SQF-XFB), Gentleman[Gent73]	$\bar{r}_B = \bar{r}_B' = 1 \qquad \delta' = c\delta$
Squared-Givens Rotation (SGR), Döhler[Döhl91]	$d' = \frac{1}{\bar{r}_B'} \qquad d = \frac{1}{\bar{r}_B} \qquad \delta' = c\delta$
Divide-and-Square-Root-Free (DSQF), Götze and Schwiegelshohn[Götz91]	$d'\bar{r}_B' = d\delta \qquad \delta' = \frac{1}{\bar{r}_B'}$
Scaled Givens Rotations, Barlow and Ipsen[Barl87]	$d' = \frac{1}{2^{\alpha}\bar{r}_B'} \qquad \delta' = \delta d d' \frac{2^{2\alpha}}{2^{2\beta}}$
Two-Multiplication, Feed-Forward Solutions, Hammarling[Hamm74]	$d' = cd \qquad \delta' = c\delta$
	$d' = cd \qquad \delta' = -sd$
	$d' = s\delta \qquad \delta' = sd$
	$d' = s\delta \qquad \delta' = c\delta$
Feedback solutions, Hammarling[Hamm74]	$d' = s\delta \qquad \delta' = c\delta$
	$d' = s\delta \qquad \delta' = -\frac{s^2\delta}{c}$

Table 6.2 summarises the 16 possible two multiply solutions which can be obtained using feedback.

Table 6.2 All two-multiply solutions obtained by feedback

$d'^{1/2} = cd^{1/2}$ $\delta'^{1/2} = \frac{\delta^{1/2}}{c}$	$\delta'^{1/2} = c\delta^{1/2}$ $d'^{1/2} = \frac{d^{1/2}}{c}$
$d'^{1/2} = cd^{1/2}$ $\delta'^{1/2} = sd^{1/2}$	$\delta'^{1/2} = c\delta^{1/2}$ $d'^{1/2} = s\delta^{1/2}$
$d'^{1/2} = cd^{1/2}$ $\delta'^{1/2} = \frac{d^{1/2}}{s}$	$\delta'^{1/2} = c\delta^{1/2}$ $d'^{1/2} = \frac{\delta^{1/2}}{s}$
$d'^{1/2} = cd^{1/2}$ $\delta'^{1/2} = \frac{c^2 d^{1/2}}{s}$	$\delta'^{1/2} = c\delta^{1/2}$ $d'^{1/2} = \frac{c^2\delta^{1/2}}{s}$
$d'^{1/2} = s\delta^{1/2}$ $\delta'^{1/2} = c\delta^{1/2}$	$\delta'^{1/2} = sd^{1/2}$ $d'^{1/2} = \frac{\delta^{1/2}}{s}$
$d'^{1/2} = s\delta^{1/2}$ $\delta'^{1/2} = \frac{d^{1/2}}{s}$	$\delta'^{1/2} = sd^{1/2}$ $d'^{1/2} = cd^{1/2}$
$d'^{1/2} = s\delta^{1/2}$ $\delta'^{1/2} = \frac{\delta^{1/2}}{c}$	$\delta'^{1/2} = sd^{1/2}$ $d'^{1/2} = \frac{d^{1/2}}{c}$
$d'^{1/2} = s\delta^{1/2}$ $\delta'^{1/2} = \frac{s^2\delta^{1/2}}{c}$	$\delta'^{1/2} = sd^{1/2}$ $d'^{1/2} = \frac{d^{1/2}s^2}{c}$

6.5 Normalisation of Conventional Givens Rotation Algorithm

6.5.1 Fixed-point Operation

Fixed-point operation is obtained by defining the QR-algorithm in terms of parameters whose magnitude is guaranteed to be less than one. Also, for good numerical performance it is necessary to do this in a way which makes good use of the available wordlength. In this section a normalised version of the conventional Givens rotation algorithm is presented which can be implemented using fixed-point arithmetic. Unfortunately, this algorithm is costly to implement, so other schemes with relaxed levels of normalisation are derived. Normalisation of the other Givens rotation variants is not considered due to its cost. In particular, two multiply solutions are not possible when normalisation is applied.

6.5.2 Normalised Algorithm

The normalised algorithm is achieved in the following manner. The input of the cell on the i^{th} row and the j^{th} column is normalised using

$$\bar{x}_{i,j}(n) = \frac{x_{i,j}(n)}{g_{i,j}^{1/2}(n)} \tag{6.49}$$

where

$$g_{i,j}(n) = \sum_{k=1}^{n} \beta^{2(n-k)} x_{i,j}(k)^2 = \beta^2 g_{i,j}(n-1) + x_{i,j}(n)^2 \tag{6.50}$$

Consequently $|\bar{x}_{i,j}(n)| \le 1$, as required. The internal cell (i, j) stores and updates the parameter $r_{i,j}(n)$ as follows:

$$r_{i,j}(n) = \frac{\beta^2 r_{i,i}(n-1) r_{i,j}(n-1)}{r_{i,i}(n)} + \frac{x_{i,i}(n) x_{i,j}(n)}{r_{i,i}(n)} \tag{6.51}$$

and so

$$\begin{aligned} r_{i,j}(n) r_{i,i}(n) &= \beta^2 r_{i,i}(n-1) r_{i,j}(n-1) + x_{i,i}(n) x_{i,j}(n) \\ &= \sum_{k=1}^{n} \beta^{2(n-k)} x_{i,i}(k) x_{i,j}(k) \end{aligned} \tag{6.52}$$

Similarly, the parameter stored in the ith boundary cell may be written in the form

$$r_{i,i}(n) = \left(\sum_{k=1}^{n} \beta^{2(n-k)} x_{i,i}(k)^2 \right)^{\frac{1}{2}} \equiv g_{i,i}^{1/2}(n) \tag{6.53}$$

If, for every cell in the array, a normalised parameter is defined as

$$\bar{r}_{i,j}(n) = \frac{r_{i,j}(n)}{g_{i,j}^{1/2}(n)} = \frac{\sum_{k=1}^{n} \beta^{2(n-k)} x_{i,i}(k) x_{i,j}(k)}{\left(\sum_{k=1}^{n} \beta^{2(n-k)} x_{i,i}(k)^2 \right)^{\frac{1}{2}} \left(\sum_{k=1}^{n} \beta^{2(n-k)} x_{i,j}(k)^2 \right)^{\frac{1}{2}}} \tag{6.54}$$

then it follows from the Cauchy-Schwarz inequality that $|\bar{r}_{i,j}(n)| \le 1$, and so this quantity is also normalised.

The function of the ith boundary cell may now be expressed in terms of normalised parameters by noting that $\bar{r}_{i,i}(n) = 1$ and

$$\begin{aligned} s_i(n) &= \frac{x_{i,i}(n)}{r_{i,i}(n)} = \frac{x_{i,i}(n)}{g_{i,i}^{1/2}(n)} = \bar{x}_{i,i}(n) \\ c_i(n) &= \sqrt{1-s_i^2(n)} \end{aligned} \tag{6.55}$$

The function of the internal cell in the i^th row and j^th column may also be written in terms of normalised quantities. From equation (6.1) it follows that

$$\begin{aligned} \bar{r}_{i,j}(n) &= \frac{r_{i,j}(n)}{g_{i,j}^{1/2}(n)} \\ &= \beta c_i(n)\frac{r_{i,j}(n-1)}{g_{i,j}^{1/2}(n)} + s_i(n)\frac{x_{i,j}(n)}{g_{i,j}^{1/2}(n)} \end{aligned} \tag{6.56}$$

but $r_{i,j}(n-1) = \bar{r}_{i,j}(n-1)g_{i,j}^{1/2}(n-1)$ so

$$\bar{r}_{i,j}(n) = \beta\frac{g_{i,j}^{1/2}(n-1)}{g_{i,j}^{1/2}(n)}c_i(n)\bar{r}_{i,j}(n-1) + s_i(n)\bar{x}_{i,j}(n) \tag{6.57}$$

The term $\beta\frac{g_{i,j}^{1/2}(n-1)}{g_{i,j}^{1/2}(n)}$ represents the change in normalisation of $\bar{x}_{i,j}(n)$ from time t_{n-1} to time t_n, and is the generalisation of the $c_i(n)$ parameter (which represents the change in normalisation for the i^th boundary cell). The term can be determined from its definition, i.e.

$$g_{i,j}(n) = \beta^2 g_{i,j}(n-1) + x^2_{i,j}(n) \tag{6.58}$$

dividing through by $g_{i,j}(n)$, re-arranging and taking the square-root yields

$$\frac{\beta g_{i,j}^{1/2}(n-1)}{g_{i,j}^{1/2}(n)} = \sqrt{1-\bar{x}^2_{i,j}(n)} \tag{6.59}$$

Hence, equation (6.57) becomes

$$\bar{r}_{i,j}(n) = \sqrt{1-\bar{x}^2_{i,j}(n)}\,c_i(n)\bar{r}_{i,j}(n-1) + s_i(n)\bar{x}_{i,j}(n) \tag{6.60}$$

The output of a cell can also be expressed in terms of normalised parameters

$$\bar{x}_{i+1,j}(n) = \frac{x_{i+1,j}(n)}{g_{i+1,j}^{1/2}(n)} = \frac{c_i(n)x_{i,j}(n) - s_i(n)\beta r_{i,j}(n-1)}{g_{i+1,j}^{1/2}(n)} \tag{6.61}$$

Substituting for $r_{i,j}(n-1)$ and multiplying top and bottom by $g_{i,j}^{1/2}(n)$ gives

$$\begin{aligned}\bar{x}_{i+1,j}(n) &= \left(\frac{c_i(n)x_{i,j}(n) - s_i(n)\beta\bar{r}_{i,j}(n-1)g_{i,j}^{1/2}(n-1)}{g_{i,j}^{1/2}(n)}\right)\frac{g_{i,j}^{1/2}(n)}{g_{i+1,j}^{1/2}(n)} \\ &= (c_i(n)\bar{x}_{i,j}(n) - s_i(n)\bar{r}_{i,j}(n-1)\sqrt{1-\bar{x}^2_{i,j}(n)})\frac{g_{i,j}^{1/2}(n)}{g_{i+1,j}^{1/2}(n)}\end{aligned} \tag{6.62}$$

The new term $\frac{g_{i,j}^{1/2}(n)}{g_{i+1,j}^{1/2}(n)}$ represents the change in normalisation from cell to cell down a column of the array. An expression for it can be obtained by using the fact that each cell performs an orthogonal transformation. Clearly

$$x^2_{i+1,j}(n) + r^2_{i,j}(n) = x^2_{i,j}(n) + \beta^2 r^2_{i,j}(n-1) \tag{6.63}$$

and hence

$$\begin{aligned}\sum_{k=1}^{n}\beta^{2(n-k)}x^2_{i,j}(k) - \sum_{k=1}^{n}x^2_{i+1,j}(k) &= \sum_{k=1}^{n}\beta^{2(n-k)}(r^2_{i,j}(k)-\beta^2 r^2_{i,j}(k-1)) \\ &= r^2_{i,j}(n)\end{aligned} \tag{6.64}$$

that is $g_{i+1,j}(n) = g_{i,j}(n) - r^2_{i,j}(n)$

Hence, by rearranging and expressing in terms of $\bar{r}_{i,j}(n)$

$$\frac{g_{i,j}^{1/2}(n)}{g_{i+1,j}^{1/2}(n)} = \frac{1}{\sqrt{1-\bar{r}^2_{i,j}(n)}} \tag{6.65}$$

Substituting this into the expression for the normalised cell output (i.e. equation (6.62)) gives

$$\bar{x}_{i+1,j}(n) = \frac{c_i(n)\bar{x}_{i,j}(n) - s_i(n)\bar{r}_{i,j}(n-1)\sqrt{1-\bar{x}^2_{i,j}(n)}}{\sqrt{1-\bar{r}^2_{i,j}(n)}} \tag{6.66}$$

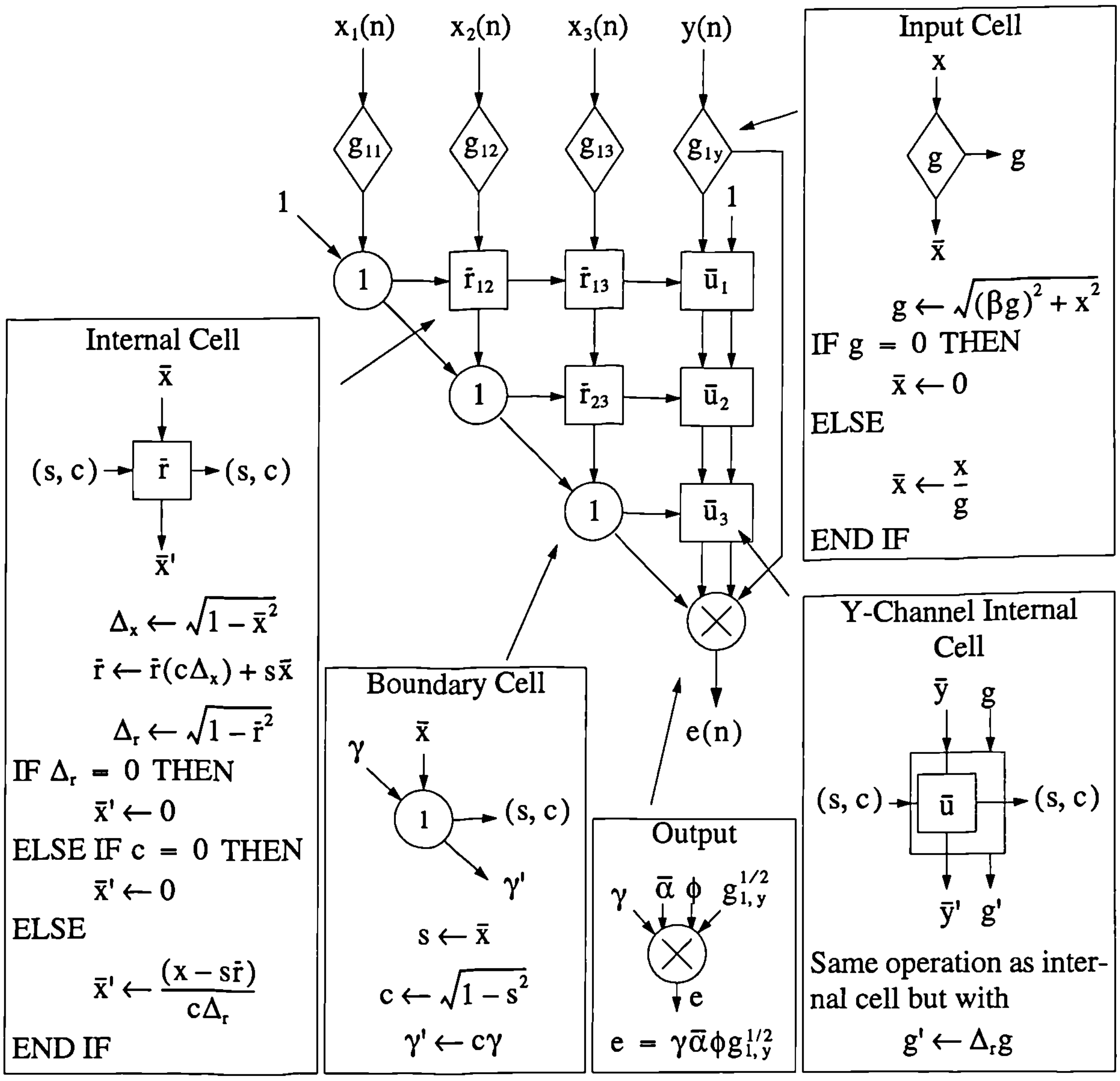
Alternatively, the cell output may be expressed in terms of the updated value of $\bar{r}$, i.e. $\bar{r}_{i,j}(n)$, by using equation (6.60) to make the following substitution

$$\bar{r}_{i,j}(n-1)\sqrt{1-\bar{x}^2_{i,j}(n)} = \frac{\bar{r}_{i,j}(n) - s_i(n)\bar{x}_{i,j}(n)}{c_i(n)} \tag{6.67}$$

This leads directly to the simpler, alternative formula for the output

$$\bar{x}_{i+1,j}(n) = \frac{\bar{x}_{i,j}(n) - s_i(n)\bar{r}_{i,j}(n)}{c_i(n)\sqrt{1-\bar{r}^2_{i,j}(n)}} \tag{6.68}$$

The normalised algorithm is summarised in Figure 6.4.



$x_1(n)$ $x_2(n)$ $x_3(n)$ $y(n)$
g_{11} g_{12} g_{13} g_{1y}
1
1
$\bar{r}_{12}$ $\bar{r}_{13}$ $\bar{u}_1$
$\bar{r}_{23}$ $\bar{u}_2$
$\bar{u}_3$
$e(n)$
Input Cell
x g g $\bar{x}$
$g \leftarrow \sqrt{(\beta g)^2 + x^2}$
IF $g = 0$ THEN
$\bar{x} \leftarrow 0$
ELSE
$\bar{x} \leftarrow \frac{x}{g}$
END IF
Internal Cell
$\bar{x}$ (s, c) $\bar{r}$ (s, c) $\bar{x}'$
$\Delta_x \leftarrow \sqrt{1-\bar{x}^2}$
$\bar{r} \leftarrow \bar{r}(c\Delta_x) + s\bar{x}$
$\Delta_r \leftarrow \sqrt{1-\bar{r}^2}$
IF $\Delta_r = 0$ THEN
$\bar{x}' \leftarrow 0$
ELSE IF $c = 0$ THEN
$\bar{x}' \leftarrow 0$
ELSE
$\bar{x}' \leftarrow \frac{(x - s\bar{r})}{c\Delta_r}$
END IF
Boundary Cell
γ $\bar{x}$ 1 (s, c) γ'
$s \leftarrow \bar{x}$
$c \leftarrow \sqrt{1-s^2}$
$\gamma' \leftarrow c\gamma$
Output
γ $\bar{\alpha}$ ϕ $g_{1,y}^{1/2}$ e
$e = \gamma\bar{\alpha}\phi g_{1,y}^{1/2}$
Y-Channel Internal Cell
$\bar{y}$ g (s, c) $\bar{u}$ (s, c) $\bar{y}'$ g'
Same operation as internal cell but with
$g' \leftarrow \Delta_r g$

Figure 6.4 Summary of the normalised Givens rotation algorithm

The boundary and internal cells perform the functions derived previously, and can be implemented using fixed-point arithmetic. The diamond shaped input cells normalise the input data and should be realised using floating-point arithmetic, or fixed-point arithmetic with extended wordlength. The output cell at the bottom of the array calculates the least squares residual using the following relationship

$$e(n) = \gamma(n)\alpha(n) \equiv \gamma(n)\bar{\alpha}(n)g_{p,p}^{1/2}(n) \tag{6.69}$$

and makes use of the expression

$$g_{p,p}^{1/2}(n) = g_{1,p}^{1/2}(n)\prod_{i=1}^{p}\sqrt{1-\bar{r}_{i,p+1}^{2}(n)} \equiv g_{1,p}^{1/2}(n)\phi(n) \tag{6.70}$$

which is readily deduced from equation (6.65). The normalised scalar $\phi(n)$ is computed in a recursive manner by extending the function of the cells in the right-hand column. The unnormalised parameter $g_{1,p}^{1/2}(n)$ ($\equiv g_{1,y}^{1/2}(n)$) is computed in the right-hand input cell and passed directly to the output cell. The output cell forms the product of its four inputs and must be implemented using floating-point arithmetic, or fixed-point arithmetic with sufficient wordlength.

The normalised algorithm uses a level of scaling specific to each cell to make good use of the wordlength. However, the extra arithmetic operations required to do this are significant. Therefore, a number of alternative schemes have been proposed based on the simplification of the normalised algorithm.

6.5.3 Column Normalisation

The column normalised algorithm only normalises the array inputs once at the top of each column. As the input normalisation changes from one time instance to the next, the $\bar{r}$ and $\bar{u}$ parameters stored within the cells are renormalised. This is achieved by transmitting the change in input normalisation (i.e. Δ_x) down each column and within each cell applying it to $\bar{r}$ or $\bar{u}$ using a multiplier. The effect of normalisation is removed from the residual by multiplying the right-hand column output by the normalisation applied at the column input (i.e. $g_{1,y}^{1/2}$). The algorithm is summarised in Figure 6.5.

6.5.4 Array Normalisation

The normalised algorithm can be further simplified by using only a single normalisation quantity applied to all inputs of the array. If the greatest value of column normalisation (i.e. $\max(g_{1,j}(n))$ is used, then the magnitude of all parameters will be less than one. Calculating this quantity will introduce latency in the input cells, as the normalisation terms for all inputs must be examined to determine the quantity before it may be applied to all inputs. The $\bar{r}$ and $\bar{u}$ parameters must also be renormalised, as done in the column normalised algorithm. These requirements can be avoided by precalculating a fixed worst-case value of the input normalisation, v, and applying it to all the array inputs. Where,

$$\begin{aligned} v &= \max\left(\frac{1}{g_{1,j}^{1/2}(k)}\right) \\ &\leq \frac{\sqrt{1-\beta^2}}{\max(x_{1,j}(k))} \end{aligned} \tag{6.71}$$

This is the simplest of the normalised algorithms allowing fixed-point operation.

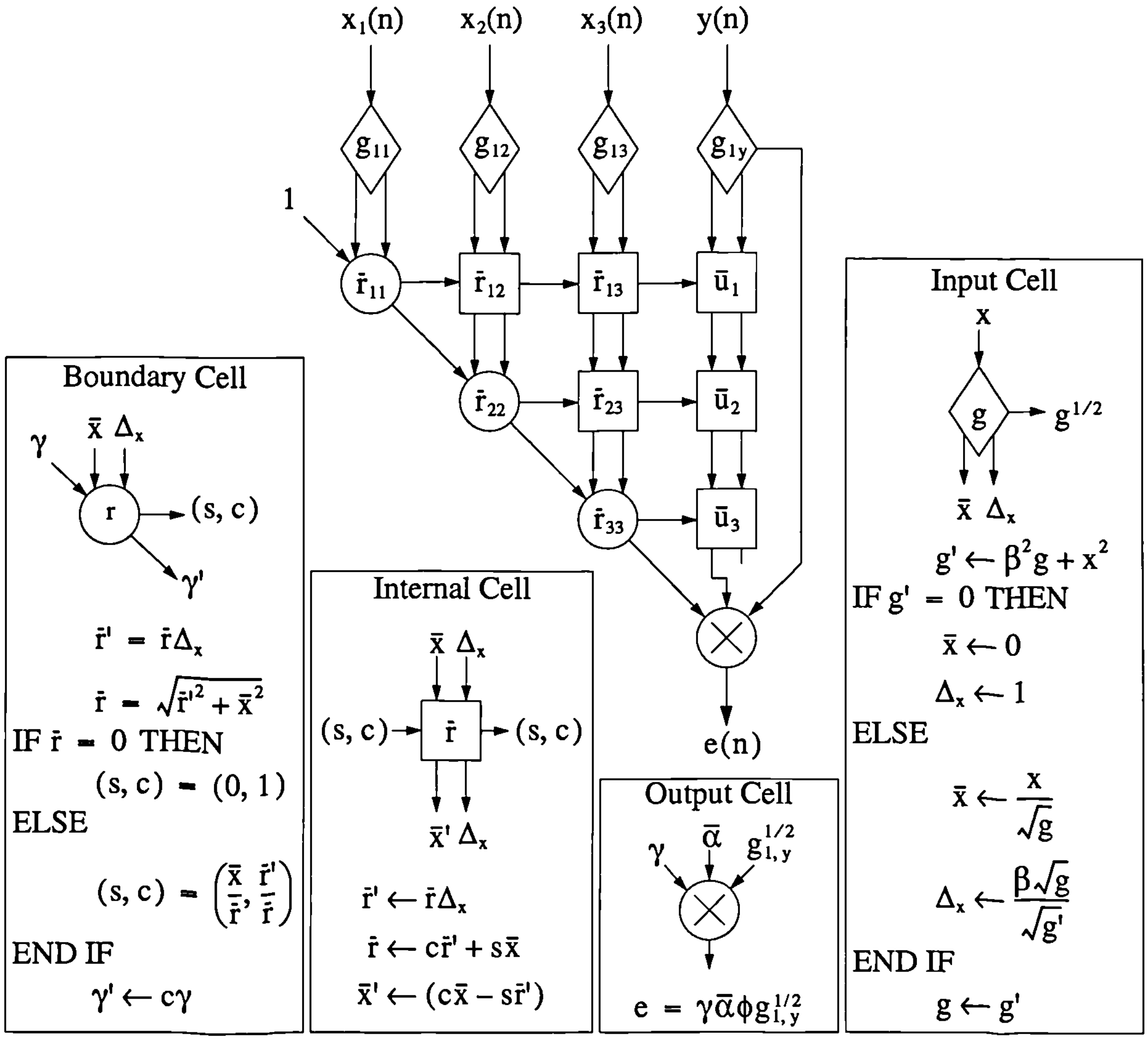


Figure 6.5 Summary of column normalised Givens rotation algorithm

This concludes the discussion on the variants of the Givens rotation algorithm. In the next chapter the implications of each algorithm on a VLSI implementation are considered.

Chapter 7 Comparison of Givens Rotation Algorithms

7.1 Introduction

7.1.1 VLSI Signal Processing

It is essential to consider the constraints of the technology, when implementing an algorithm in VLSI, if optimum results are to be achieved. In this chapter the Givens rotation algorithms are examined in detail to determine their suitability for implementation using a VLSI application specific circuit approach. Consideration is given to issues such as the number of operations, their type and order, whether fixed- or floating-point arithmetic is required and the wordlength necessary to meet a particular signal-to-noise ratio (SNR).

Number, Type and Dependence of Operations: Minimising the number of operations reduces either the area or execution time. In a parallel implementation of an algorithm, where all operations are implemented to achieve high throughput, a reduction in the number of operations leads directly to a reduction in the circuit area. In a sequential implementation, where one or more operators are reused to implement the algorithm, a reduction in the number of operations will reduce the time to generate a result.

The type of operation is significant as the area and latency requirements vary considerably. Division and square-root have high latency, which can complicate timing and increase the number of latches required throughout a system to maintain the synchronisation of data. The location of these operations is also important, for if such operations are within a recursive loop their high latency can severely limit the sample-rate of the system. The order of the operators can influence the numerical properties and can restrict the effectiveness of a redundant representation to reduce latency.

Wordlength Requirements: In an application specific circuit, the wordlength of arithmetic is often under the designer's control. By minimising the wordlength it is possible to reduce circuit area and increase speed. Wordlength is particularly critical in the implementation of parallel multipliers, dividers and square-root operations, as their area is proportional to the square of the wordlength.

Fixed or Floating-Point Arithmetic: Fixed-point arithmetic offers simplicity and speed, but limited dynamic range. A floating-point representation uses an exponent to increase the dynamic range but requires additional circuitry to handle normalisation and alignment operations, which increases latency and the complexity of its implementation. The simplicity of fixed-point arithmetic must be weighed against the cost of increases in wordlength necessary to meet an algorithm's dynamic range requirements.

Algorithm: The algorithm determines the type, number and order of operations, and the optimisations which can be applied to minimise wordlength and simplify the arithmetic. A range of Givens rotation variants was presented in the previous chapter, and in this chapter their suitability for a high sample-rate application specific VLSI implementation is examined.

7.1.2 Overview of the Chapter

The first algorithm examined in section 7.2 is a floating-point implementation of the conventional Givens rotation. Floating-point operation results in high latency within a critical feedback loop. To avoid this, fixed-point operation using the normalised algorithms is investigated in section 7.3. The normalised algorithms are found to offer high sample-rate, but at the expense of a large number of operations and high wordlength. Therefore, floating-point arithmetic is reconsidered in section 7.4, but using the more efficient fast-Givens rotation algorithms. A variant is developed which contains only an adder within the critical feedback path. This enables the wordlength of the feedback path to be extended to improved numerical performance and give significant savings in wordlength requirements elsewhere. Latency of the adder is still a problem, so fixed-point redundant arithmetic is used within the feedback path to give a sample-rate which is extremely high and independent of wordlength.

This is referred to as the *enhanced-SGR algorithm.*

In section 7.5 the properties of the DSF algorithm are briefly examined. The chapter concludes with a direct comparison of the estimated area and sample-rate of various implementations of the QR-array. This clearly demonstrates the benefits, in terms of high sample-rate and low area, of the enhanced-SGR algorithm.

One aspect of this study has been to consider the numerical performance of the variants under the conditions of limited precision arithmetic, because wordlength requirements are so critical to the area and speed of a VLSI implementation. This has been done using a computer simulation of the algorithms in a channel equaliser application, similar to that presented in section 1.3.2. This provides a simple and useful application with which to test the numerical performance of each algorithm variant.

Channel equalisation can be performed more efficiently using a lattice filter, but the triangular array is required by adaptive beamforming (the primary application of this research) and so is of more interest here. The findings from the simpler channel equaliser simulations are directly relevant to the adaptive beamforming application.

Further details of the channel equaliser application are given in Appendix E. Here it is sufficient to say that the QR-array is used to form an FIR filter to undo certain effects of an imperfect channel. This approximate 'inverse' channel filter is estimated over a window of input data determined by the forget-factor β, where $\beta < 1$. The length of this window is approximately given by $\frac{1}{1-\beta}$. As $\beta \rightarrow 1$ the window becomes larger and the estimate of the filter parameters in the presence of noise is improved, but the slower is the response of the filter to changes in the channel. Typical values for β range from 0.9 to 0.9999.

For the purposes of obtaining estimates of area, latency and speed of the VLSI implementations of the algorithms, the synthesis results presented in earlier chapters of the thesis are used. A system clock frequency of 100MHz is assumed (as it is a convenient figure and a realistic clock frequency for the operators described in the previous chapters). This choice does not constrain the conclusions to a single technology, as the relative merits of each approach will be the same irrespective of the technology. It is convenient, however, for putting realistic figures to the sample-rates achievable.

7.2 Conventional Givens Rotations

7.2.1 Numerical Performance of Conventional Givens Rotation Algorithm

The conventional Givens rotation algorithm requires floating-point arithmetic. Figure 7.1 shows the signal-to-noise ratio (SNR) obtained for a range of arithmetic wordlengths and forget factors. Notice, that as $\beta \rightarrow 1$ the SNR falls. This is because the length of window over which the numerical error is accumulated is increasing.

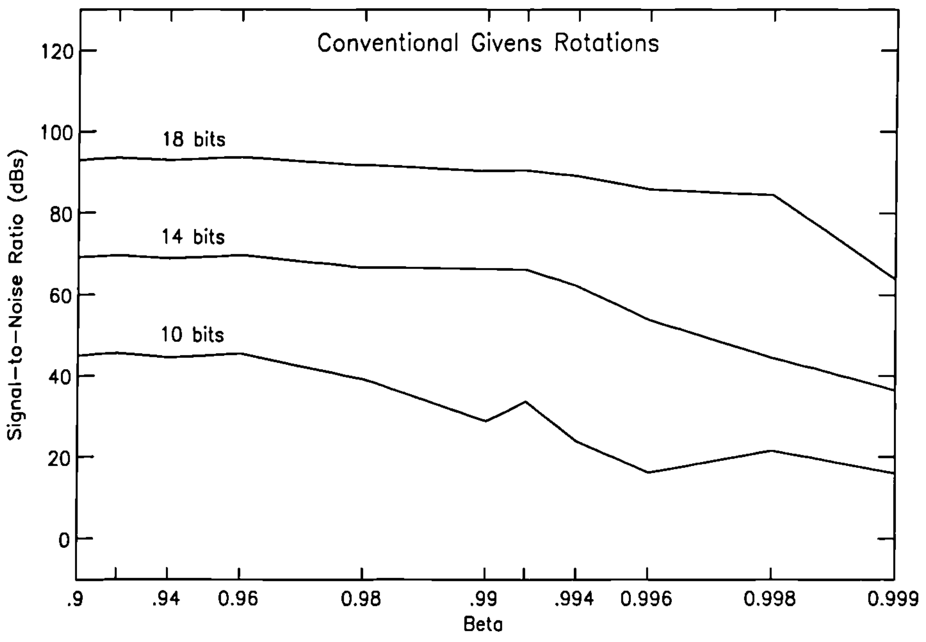


Figure 7.1 Numerical performance of conventional algorithm

7.2.2 Signal Flow Graph of Conventional Algorithm

The SFG for the boundary and internal cells of the conventional algorithm are shown in Figure 7.2.

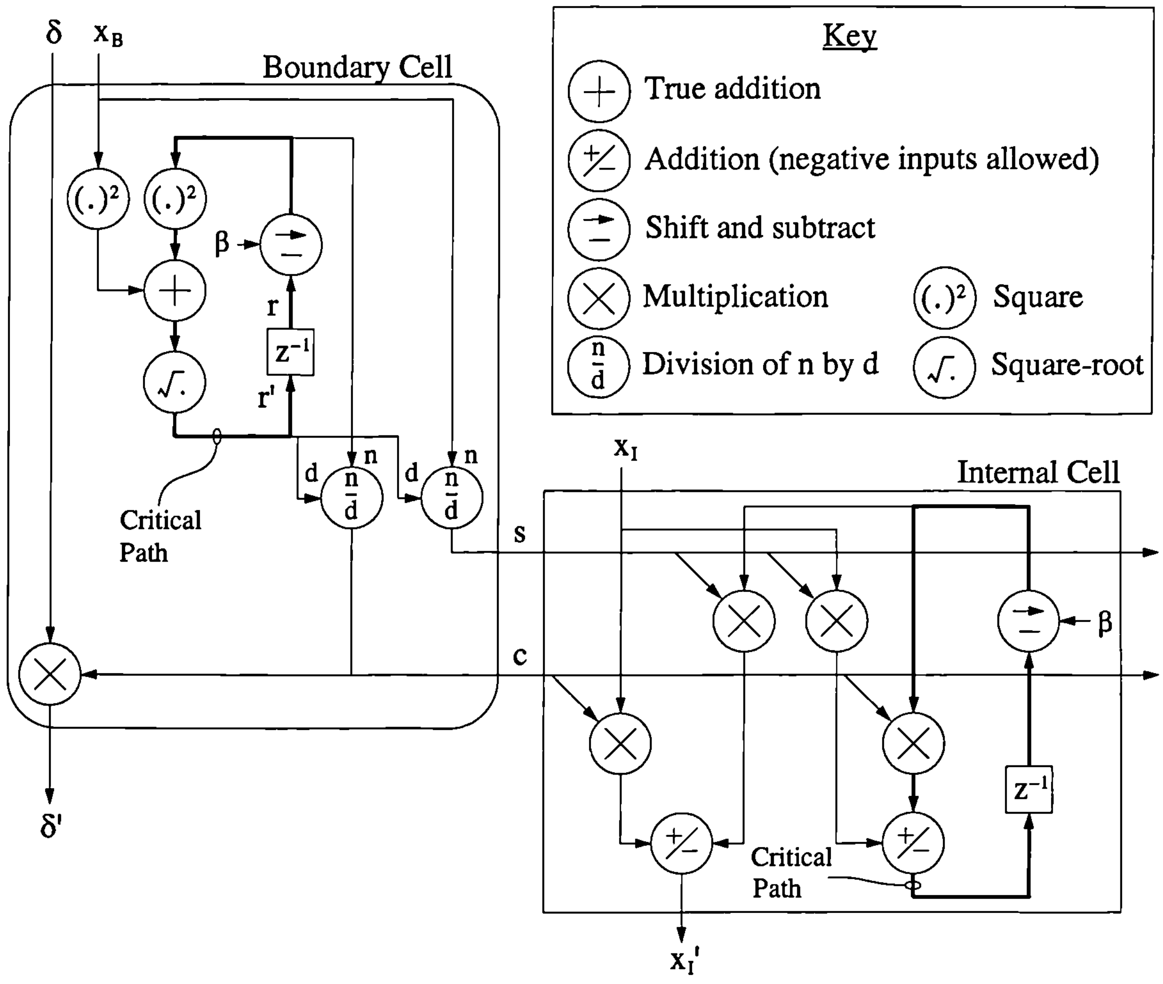


Figure 7.2 SFG of boundary and internal cell for the conventional algorithm

Both the boundary and internal cells contain loops, which are emphasised in the figure using a thick line to represent the critical signal paths. The multiplier applying the forget-factor has been simplified to a shift-and-subtract operation by restricting β to values which can be represented by $(1-2^{-s})$, where s is an integer. This operation will subsequently be referred to as the *beta multiplier*. This limits the number of values which β may take, but as shown by Table 7.1, not to an unreasonable extent.

Table 7.1 Permissible values of beta

s	3	4	5	6	7	8	9	10
$\beta = (1-2^{-s})$	0.875	0.9375	0.9688	0.9844	0.9922	0.9961	0.9980	0.9990

The SFG shows that the loop in the boundary cell involves a square-root operation which has considerably greater latency than the multiply and add operations in the internal cell. It is possible to remove the square-root operation from the loop by maintaining r in a squared form. This gives the modified signal flow graph for the boundary cell shown in Figure 7.3.

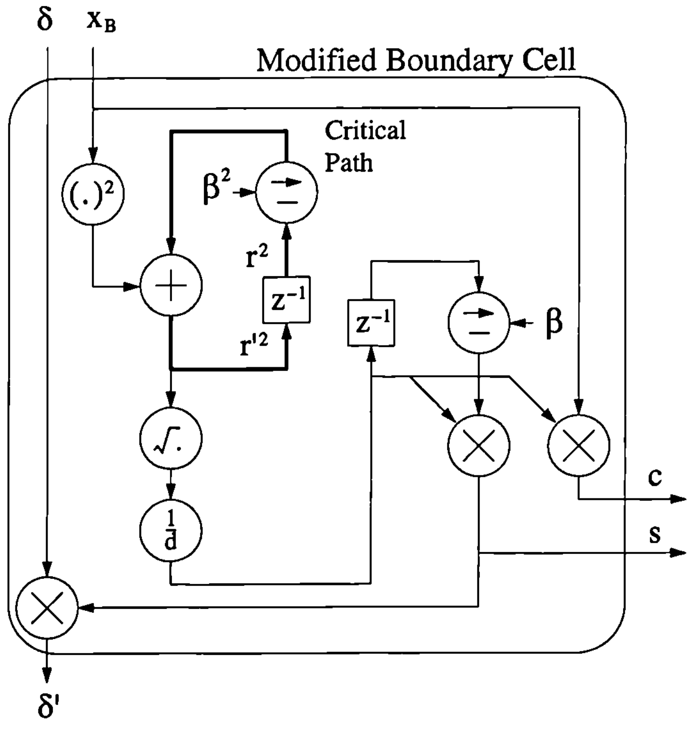


Figure 7.3 Modified SFG for the conventional algorithm

The loop now consists of only an adder and a beta multiplier. Another modification made in the figure is the replacement of the two dividers by a single reciprocal circuit and two multipliers. Dividers are significantly larger than multipliers (see Chapter 5), so this option may reduce area. Furthermore, the reciprocal can be generated with low-latency using a convergence technique. This is likely to be particularly worthwhile in a complex arithmetic implementation of the boundary cell, where three divisions by a common divisor are required.

7.2.3 Floating-Point Design of the Conventional Algorithm

A conventional floating-point implementation of both the modified and unmodified SFGs is relatively straight-forward once the operators have been designed. For the modified algorithm the sample-rate would be limited by the latency of internal cell loop, which would be 6 (multiplier: 2, adder: 3, beta-multiplier: 1), allowing a sample-rate of 16.7 MHz to be achieved. It may be possible to reduce the latency by one by merging the beta-multiplier with the recoding circuits in the multiplier.

A floating-point, msdf format would offer little benefit to a implementation of the algorithm due to the latency of the normalisation operation within the adder/subtractor. A parallel redundant format would also suffer from normalisation and the alignment operations within the adder.

In conclusion, a minimum latency of 5 is probable with a floating-point implementation of the modified conventional Givens rotation. Much of this latency is due to the use of floating-point arithmetic, so algorithms which can exploit fixed-point arithmetic are investigated next.

7.3 Normalised Algorithms

The latency in the floating-point implementation of the conventional algorithm arises primarily from the alignment and normalisation operations required by floating-point addition/ subtraction. Fixed-point avoids these operations, and enables redundant arithmetic to be considered, which will allow the merging of the multiply and add operations.

In the previous chapter, a normalised algorithm was presented which allowed fixed-point arithmetic to be used. Two simpler versions, referred to as the column and array normalised algorithms were also proposed. The numerical performance of all three is considered below.

7.3.1 Numerical Performance of Normalised Algorithms

The SNR of the normalised algorithms is shown in Figure 7.4.

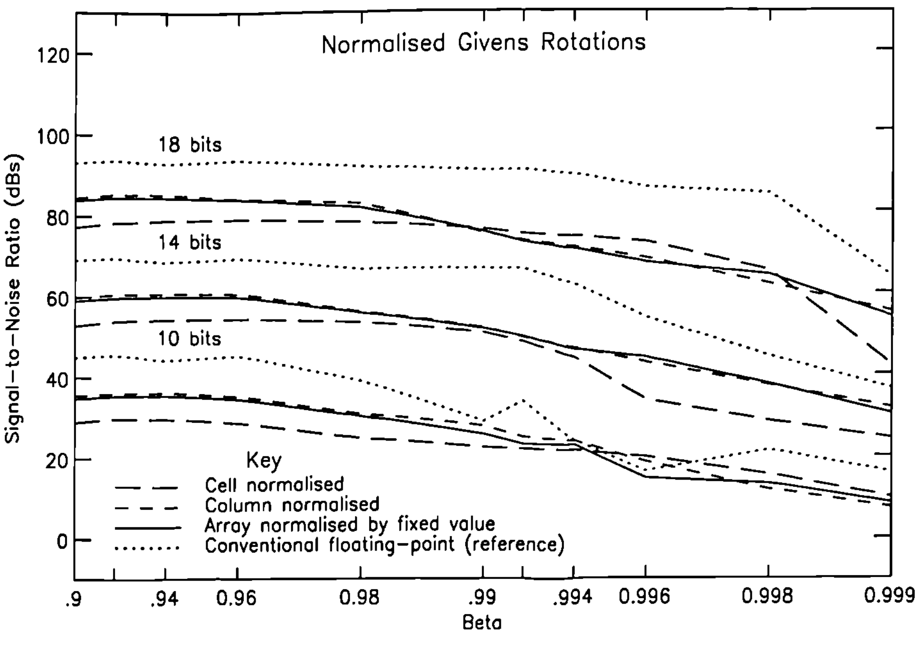


Figure 7.4 Numerical performance of the normalised algorithms

The array-normalised algorithm presented here, uses a fixed, precalculated quantity. It offers similar SNR to the column-normalised algorithm in this application, as the column inputs are just delayed versions of the one input signal, and so have the same normalisation requirements.

A surprising result is that the cell normalised algorithm generates numerical errors far in excess of the other two algorithms. In the process of tracking down the cause of this, it was noted that the same errors were produced when floating-point arithmetic was used, demonstrating that it was not due to the arithmetic, but the algorithm. Further investigation showed that it was the result of a numerical effect, introduced by one of the normalisation operations. Its source is considered further in Appendix B.

Even with this numerical effect, the SFG of the normalised algorithm is of interest and is considered next, as the algorithm achieves dynamic normalisation of each term, much like floating-point does, but without the need for an exponent and the associated alignment and normalisation requirements of the addition/subtraction operation.

7.3.2 Signal Flow Graph of Normalised Algorithm

The SFGs for the input, boundary and internal cells of the cell normalised algorithm are shown in Figure 7.5. Loops are present in the input and internal cells. Within the internal cell the loop is relatively simple and contains only a multiply-add operation. Within the input cell the loop appears to be more complicated, but as in the case of the conventional algorithm, it can be simplified by removing the square-root operation from the loop.

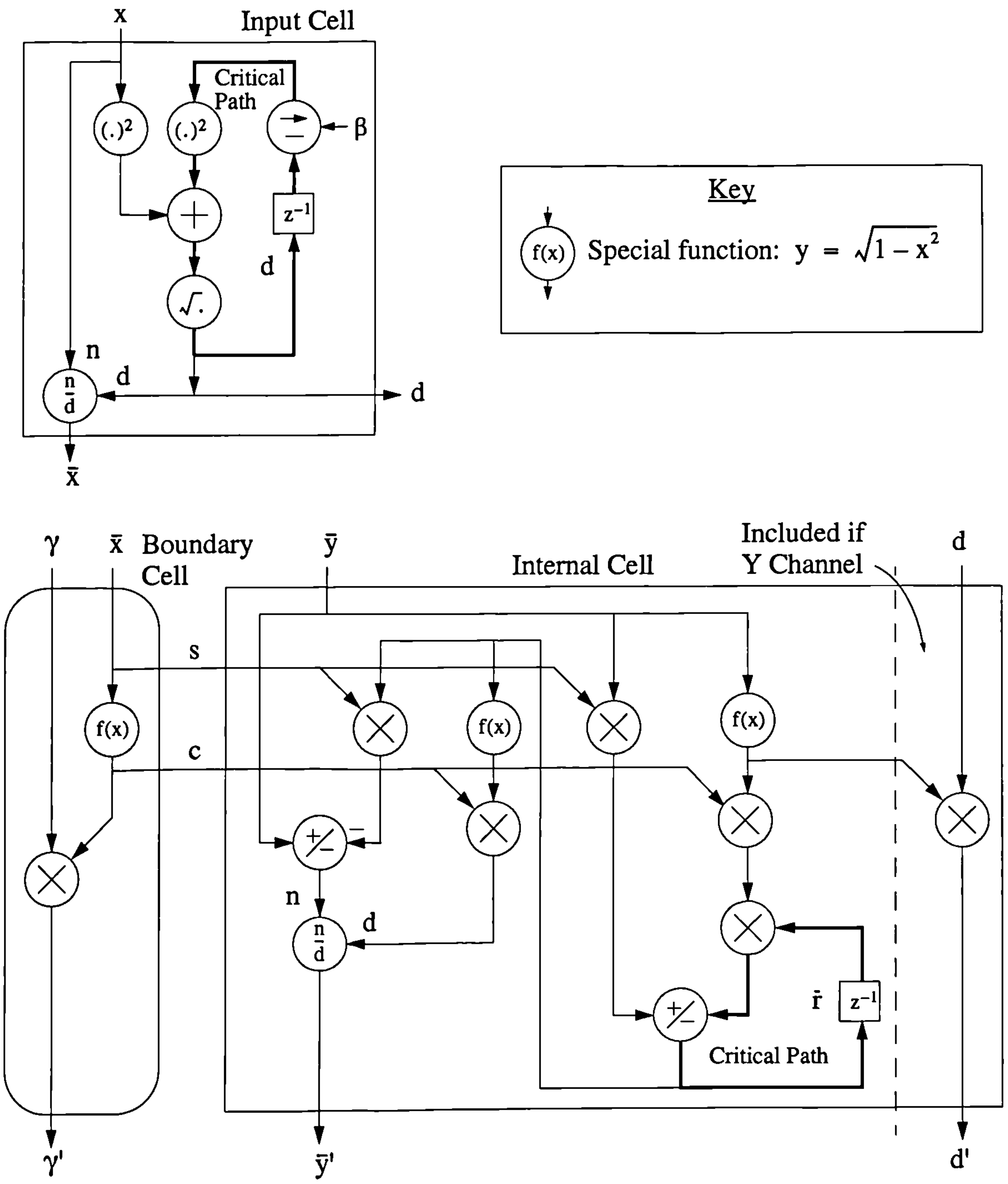


Figure 7.5 SFG for the normalised algorithm

7.3.3 Fixed-Point Design of Normalised Algorithm

The input cells convert the input of the array into a normalised fixed-point format. If the array input is floating-point then the input cells must be implemented using floating-point arithmetic. However if the input is fixed-point, which may be the case if its source is an analogue-to-digital converter, then a fixed-point implementation of the input cell may be employed, provided that sufficient wordlength is used for the loop variable.

The internal cell requires a divide operation and two special functions to be evaluated. The special functions are unary, and could be implemented quite efficiently at low word lengths (i.e. 12-bits) using a look-up table and interpolation. For higher wordlengths some functional iteration could be used to improve accuracy. The divide operation is costly in terms of both area and latency.

In contrast to the complexity of the internal cell, the boundary cell is relatively simple. Unfortunately, internal cells dominate the array, so this change is not beneficial.

A particularly interesting aspect of the normalised algorithm is how the loop variable within the internal cell is dynamically normalised using only a multiplier within the loop. Scaling-down of the variable is achieved by multiplication by a number less than one within the loop. Whereas scaling-up of the variable, relative to the loop input, is achieved by *scaling down the loop input*. Hence, the operations of alignment (scaling-down of the loop variable) and normalisation (scaling-up of the loop variable) are achieved without the need for explicit shift operations. Also, the level of normalisation applied is not determined in the loop by examining the size of the variable, but is obtained from the energy in the signal, which is evaluated outside of the loop and so out of the critical path. However, the extra computation is high, and due to the numerical anomaly the algorithm's performance is unreliable. Therefore, the implementation of the array normalised algorithm shall be addressed instead, as this offers better numerical performance (in this application), with fewer operations and the same simple loop.

7.3.4 Implementation of the Array Normalised Algorithm with Fixed Normalisation

The array normalised algorithm, as considered here with a fixed level of normalisation, is simply a fixed-point implementation of the modified conventional Givens rotation algorithm. Fixed-point operation of the algorithm has been achieved as described in section 6.5.4 by scaling the input of the array by the quantity

$$v = \frac{\sqrt{1-\beta^2}}{\max(x)} \tag{7.1}$$

Note that r^2 rather than r, is stored in the modified boundary cell, which has greater dynamic range, and so the wordlength used to represent it should be increased in a fixed-point implementation. This will also increase the size of any storage latches.

The longest loop is present in the internal cell, but the latency of this could be reduced by combining the addition and multiplication and employing redundant arithmetic. Either parallel redundant arithmetic and MSDF arithmetic could be used within the loop to achieve high sample-rate operation.

7.3.4.1 Approach Based on Parallel Redundant Arithmetic

If a parallel redundant representation was used for the loop variable and a non-redundant representation was used for the s and c inputs to the internal cell, then single-cycle latency could be achieved using the adder-tree presented in section 3.2.2 (for wordlengths of approximately 16-bits). This multiplier is no larger than a conventional tree-multiplier, and the signed-binary output representation would provide an unbiased error using truncation; thus avoiding an extra clock-cycle to round. Saturation would not be required as $\bar{r}_{i,j} \le 1$, so a sample-rate of 100 MHz would be feasible. There is little benefit in using a parallel redundant format outside of the loop, so it should be avoided as it is more costly to implement.

7.3.4.2 Approach Based on Msdf Redundant Arithmetic

An msdf approach is also worth considering, as it offers a latency that is independent of wordlength, and wordlength has to be increased to maintain numerical performance in fixed-point implementations.

Figure 7.6 shows how an msdf representation may be used within the loop of the internal cell to obtain high sample-rate operation. One of the msdf multiplier-adders presented in section 3.2 could be used within the internal cell loop. In particular, the MinR4 multiplier is only slightly larger than a conventional tree-multiplier, but offers a 2-cycle latency which is independent of wordlength. The delay of the msdf multipliers required here is likely to be less than those presented in Chapter 3, as the additive input is not in an msdf form and the magnitude of the non-redundant coefficient input would be 1 rather than 2 (as the magnitude of both s and c are less than 1). Therefore, a 50 MHz sample-rate should be achievable.

An msdf format is not required for the boundary cell loop variable, as only addition is required in the loop, and this can be achieved with single-cycle latency using a parallel redundant representation.

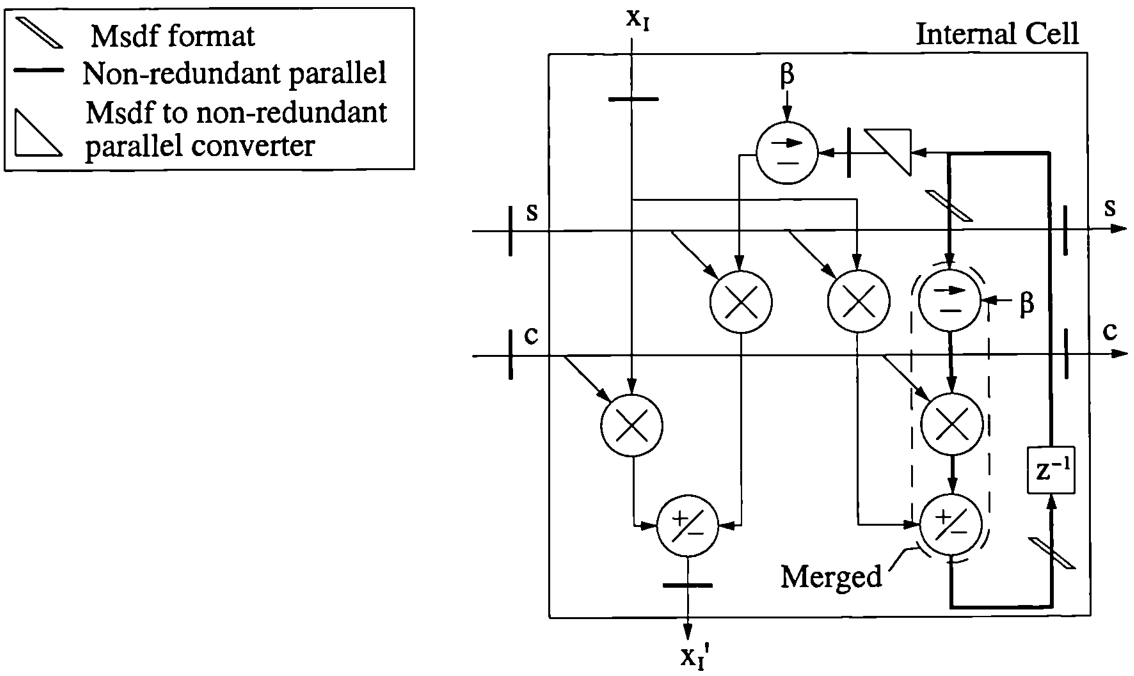


Figure 7.6 Fixed-point Msdf implementation of the conventional algorithm

Another potential application of msdf arithmetic is to reduce the latency of the square-root and divide operations within the boundary cells. Although the latency of these operators does not limit the sample-rate is does introduce similar delays into the internal cells which can be costly in terms of additional latches. This is demonstrated by Figure 7.7 which shows that any pipelining cut through a boundary cell must also continue through all the internal cells in a row of the array. However, any more extensive use of the msdf representation will require multiplier-adders in the internal cell that could accept two redundant inputs, which would make them 50% larger and outweigh any latch reduction obtained by the reduced boundary cell latency. Therefore, more extensive use of the msdf representation is considered no further.

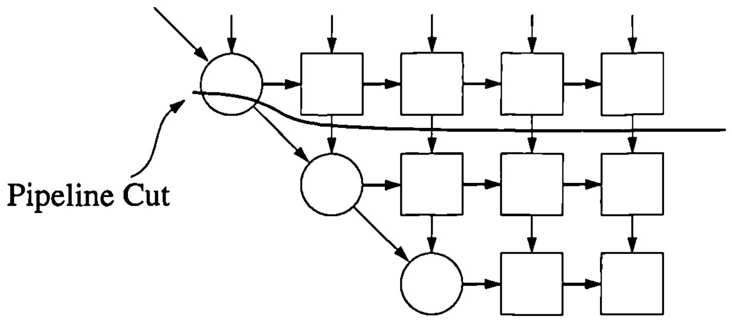


Figure 7.7 Section of QR-array showing effect of pipeline cut through boundary cell

7.3.4.3 Summary of Redundant Arithmetic Implementation Approaches

In summary, an adder-tree redundant multiplier could be used to reduced the latency of the loop to 1-cycle, and enable a sample-rate of 100 MHz to be achieved for medium wordlengths (i.e. 16-bits). An msdf approach would give a latency of 2-cycles and a sample-rate of 50MHz. This would be independent of wordlength, and it may also be possible to achieve higher clock rates (alternatively, the single-cycle pipelining of the msdf multiplier could be investigated to reduce latency). The two approaches would require similar sized multipliers.

7.3.5 Possible Improvements to Normalised Algorithm

The array normalised algorithm uses a fixed level of scaling, which would be acceptable in applications where the inputs were fixed-point. However, in cases where the input has a higher dynamic range then its performance may not be so good. To overcome this and avoid the need to choose an appropriate level of scaling in the first place, an approximate form of column scaling could be used. Here it is proposed that the input cells determine the normalisation requirements of each input as in the column normalised case, but apply it in an approximate manner down a column by scaling by powers of two, using shifters. This is also compatible with a redundant arithmetic implementation. In the case of msdf arithmetic, scaling down operations can be performed without latency, although scaling-up introduces a latency dependent upon the amount of scaling. However, only scaling by 1 or 2 is ever required, as the input energy can only decay slowly with β. So, this approach could be introduced with only one additional cycle of latency.

The idea could also be extended to an approximate form of cell normalisation (without the numerical problem), by using the boundary cell to specify an approximate normalisation term for each row. This can be combined with the column normalisation and applied in a single shift operation within each cell.

7.3.6 Summary of Normalised Algorithms

In summary, normalised algorithms offer high sample-rate operation, but are costly to implement, due to their increased wordlength requirements and extra operations. This is particularly true of the cell normalised algorithm, which due to a numerical effect, has generated errors far in excess of the other simpler normalisation schemes. Normalisation is applied in a similar way in lattices. It would be worthwhile investigating whether this numerical error could also occur there.

7.4 Square-Root-Free Algorithms

7.4.1 Introduction

The wordlength necessary to meet a particular SNR is very important in a bit-parallel arithmetic implementation of the Givens rotation cells, as their area is dominated by multipliers and dividers, which have an area dependent upon the square of their wordlength. Therefore, floating-point arithmetic shall be reconsidered, since for a particular SNR, lower wordlengths can be obtained using the conventional algorithm rather than the normalised ones. Another advantage of floating-point arithmetic is that fast-Givens rotation algorithms can be considered. As shown in the previous chapter, these avoid the square-root and reduce the number of multiplication operations in the internal cells from 4 to 2. They also offer much greater variety of SFG, which helps to address another issue, considered next.

In the simulation results of algorithms presented in previous sections, the SNR falls as the window length increases. This was found to be due to accumulations of numerical error in the loop variable. As shown later, additional wordlength for representing the loop variable leads to significant improvements in SNR. One potential benefit of this is to enable reductions in the wordlength of arithmetic used elsewhere in the algorithm, and so make savings in area. Unfortunately, increasing the wordlength of the loop variable in the algorithms considered so far is not straightforward, as they all contain a multiplier in the loop of the internal cell. Consequently, increased wordlength results in increased multiplier area and latency, but more significantly, requires an increase in the wordlength of the parameters c and s and the area of the operators which calculate them.

Motivated by this issue, a fast Givens rotation algorithm was developed which simplifies the loop in the internal cell to an add operation. This has allowed the wordlength of the loop variable to be increased to give superior SNR and single-cycle latency. Before considering the algorithm in more detail, it is worth elaborating on the importance of additional wordlength in the representation of the loop variable.

7.4.2 Error Accumulation in the Loop Variable

It is clear from the simulation results and rudimentary error analysis that the output error is dominated by the round-off errors which accumulate in the recursive update of the loop variable, r. Build-up of errors in accumulators is a well known problem[Wilk63]. Here it can be explained by considering the accumulation process which updates r:

$$r' = cr + y \tag{7.2}$$

where $y = ax$.

Initially consider the case where $c = 1$, in which a simple accumulation operation is being performed. In this circumstance, two errors affect the value of r: the error in the input y and the error due to rounding the result of the addition. If the accumulator is large enough, then the error in r' is dominated by the error in the input. If this input error is due to rounding in previous calculations or thermal noise in the input signal, then it is likely to be random, so the sum of these errors will grow as $\sqrt{n}$, where n is the number of iterations. The purpose of equation (7.2) is to accumulate correlated quantities, which will grow with n. Therefore, the accumulation process is able to produce a result with greater relative accuracy than the input.

If the wordlength of the accumulator is not large enough it is possible for the error in r to become dominated by the accumulator round-off error. For example, consider the calculation of r when the correlation is high and the sum grows as n. In this case r will approach 1 to indicate high correlation. Figure 7.8 (a) shows graphically the input and the accumulator contents for an accumulator which has the same wordlength as the input. The sum in the accumulator is a factor of n times greater than the input. Therefore, the input is shown right-shifted to align it with the contents of the accumulator. The erroneous bits of the input, which are shaded in the figure, make no contribution to the error in r, which is dominated instead by the accumulator round-off error.

Figure 7.8 (b) shows the accumulation of error for an accumulator with twice the wordlength of the input. The accumulator error is now dominated by the error in the input. Depending upon the size of the input error ε_y the reduction in the error in r (denoted ε_r in the figure) can be as much as a factor of $\frac{1}{n}$.

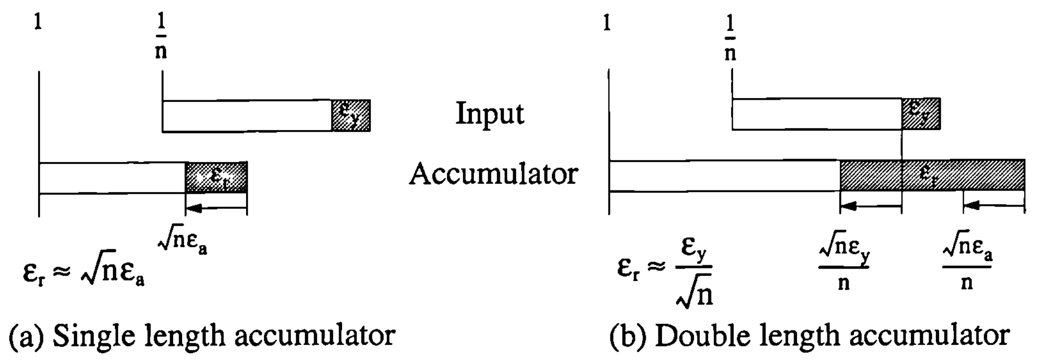

1
$\frac{1}{n}$
ε_y
ε_r
$\sqrt{n}\varepsilon_a$
$\varepsilon_r \approx \sqrt{n}\varepsilon_a$
(a) Single length accumulator
Input
Accumulator
$\varepsilon_r \approx \frac{\varepsilon_y}{\sqrt{n}}$
$\frac{\sqrt{n}\varepsilon_y}{n}$
$\frac{\sqrt{n}\varepsilon_a}{n}$
(b) Double length accumulator

Figure 7.8 Effect of accumulator length on errors

In the case of the conventional algorithm $c \rightarrow \beta^2$. This has the effect of limiting n even though the sum is repeated infinitely (i.e. an effective window length can be defined where $n_{effective} = \frac{1}{\sqrt{1-\beta^2}}$). As β approaches 1 the effective window length over which r is accumulated increases, and the effect of a short accumulator becomes more pronounced.

This simple analysis suggests that the numerical performance of the QR-algorithm implementation can be improved by increasing the wordlength of the accumulator above that used by the rest of the algorithm. This can be done in the conventional algorithm, but as discussed previously, it increases the size of the multiplier in the loop and the wordlengths of operators elsewhere. These side-effects can be avoided by transforming the algorithm to remove the multiplication from within the loop. A fast-Givens rotation algorithm was derived which achieved this, although further investigation of the literature showed that the algorithm was the Squared Givens Rotation (SGR), recently proposed by Döhler[Döhl91]. As published, it offers little advantage over other fast-algorithms and there has been only limited interest in it[Deit93a][Deit93b]. However, later in the chapter it is shown how the algorithm can be enhanced to offer significant benefits over other variants when an application specific circuit solution is adopted.

7.4.3 Signal Flow Graphs for Fast-Givens Rotation Algorithms

It is interesting to note that the fast-Givens rotation algorithms offer a whole range of SFG possibilities for the internal cell computation. Figure 7.9 shows the SFGs of the internal cells for the 4 feed-forward variants and Figure 7.10 shows the 8 SFGs resulting from the 16 feed-back variants.

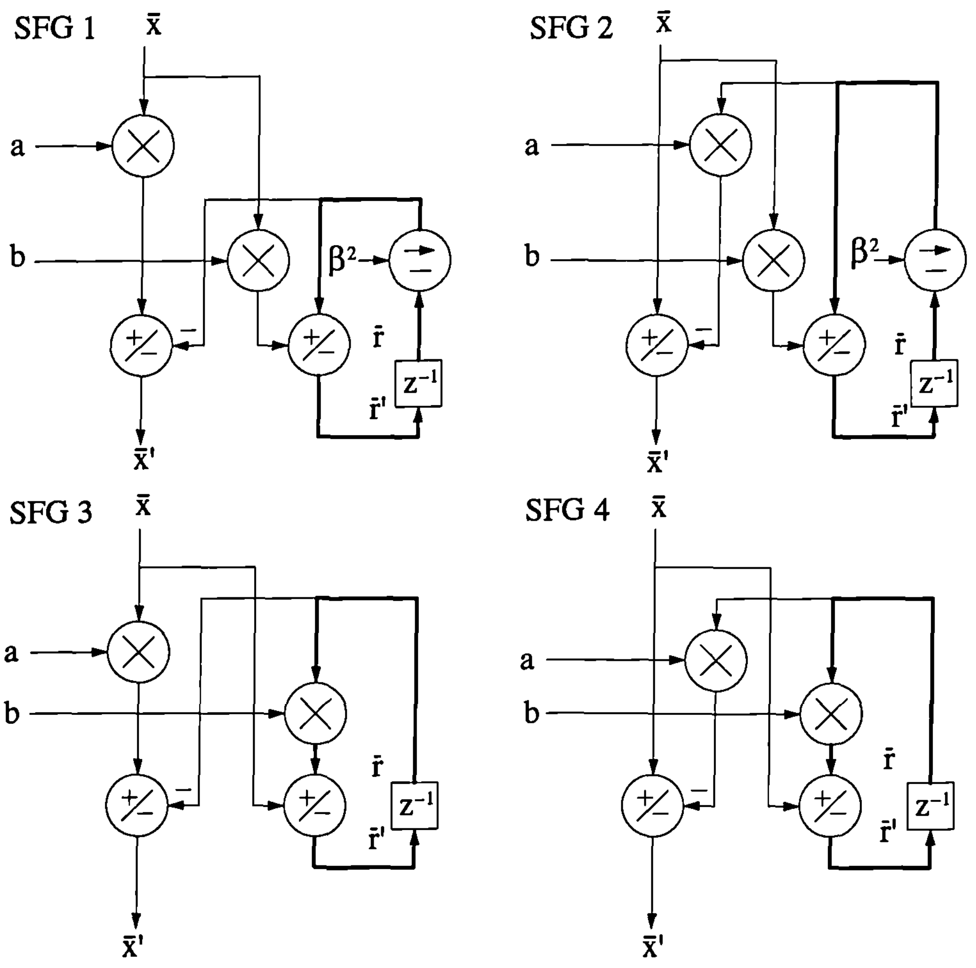


Figure 7.9 SFGs of internal cells for two-multiply feed-forward algorithms

The popular SQF-XFB algorithm, is presented as SFG 11 in Figure 7.11. This has 4 operators in the loop and numerical problems[Hamm74], and so is of no further interest. Half of the SFGs offer simple loops in the internal cells. The SFG of the internal cell of the SGR algorithm is the same as SFG 2. However, the SGR algorithm also offers a simple boundary cell without multipliers in the loop. The other SFGs have not been considered, as they offer no obvious benefits over that of the SGR algorithm.

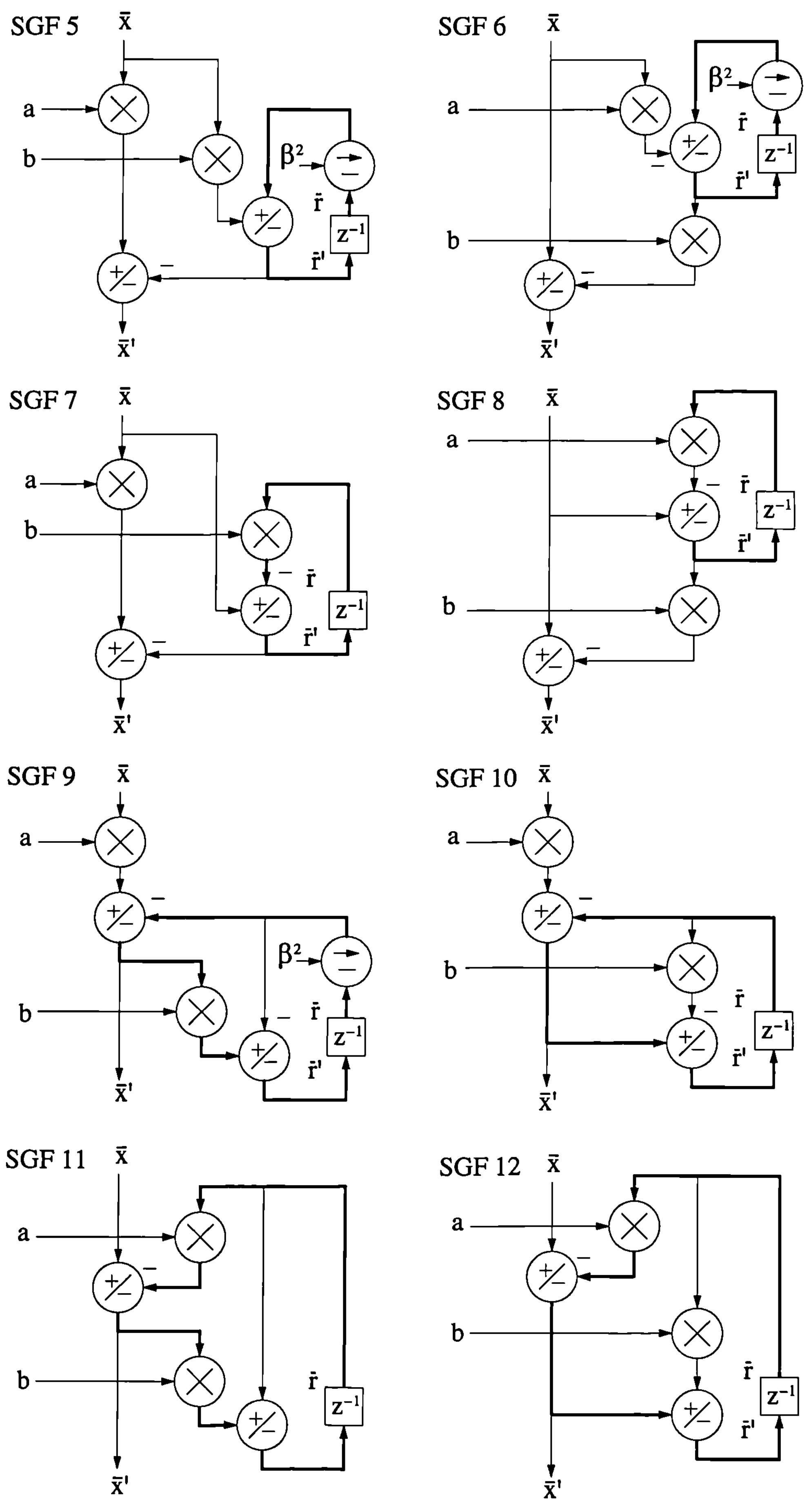


Figure 7.10 SFG for two-multiply feedback algorithms

The SFG for both the boundary and internal cells of the SGR algorithm are shown in Figure 7.11.

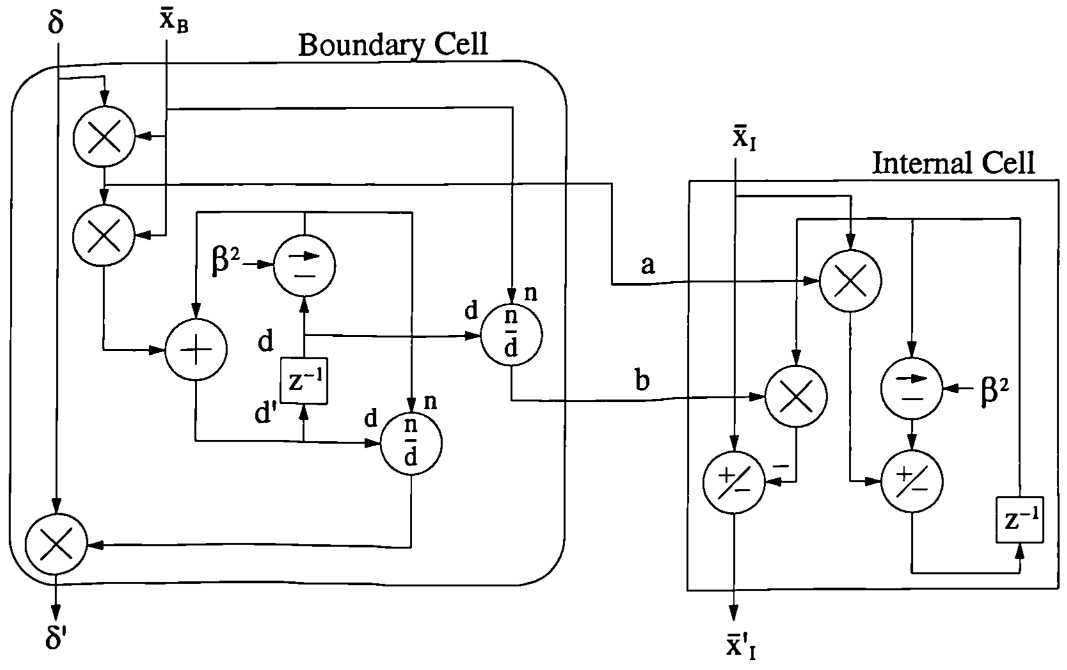


Figure 7.11 SFG of SGR Algorithm

The boundary cell has a very similar loop to the internal cell, and only differs in its use of a true-add rather than an add/subtract operation. The wordlength of both loop variables can be increased at little cost in hardware. The numerical performance of the SGR algorithm, when it is enhanced with increased wordlength in the loop variable, is considered next.

7.4.4 Numerical Performance of SGR Algorithm

Figure 7.12 shows the SNR of the SGR algorithm for a range of wordlength and beta factor. For each, the SNR is shown for three wordlengths of the loop variable mantissa. For the original algorithm (denoted SGR-FL0 in the figure i.e. no additional wordlength in the loop), there is a marked decrease in the SNR as beta approaches one. Increasing the wordlength in the loop by 4-bits (denoted SGR-FL4) provides a marked improvement in the SNR, which becomes very significant for large window lengths. The SNR now *increases* with the window length. Increasing the loop wordlength by more than 4-bits only provides a small additional improvement, when compared with SGR-FL40 where a 40-bit mantissa is used in the loop.

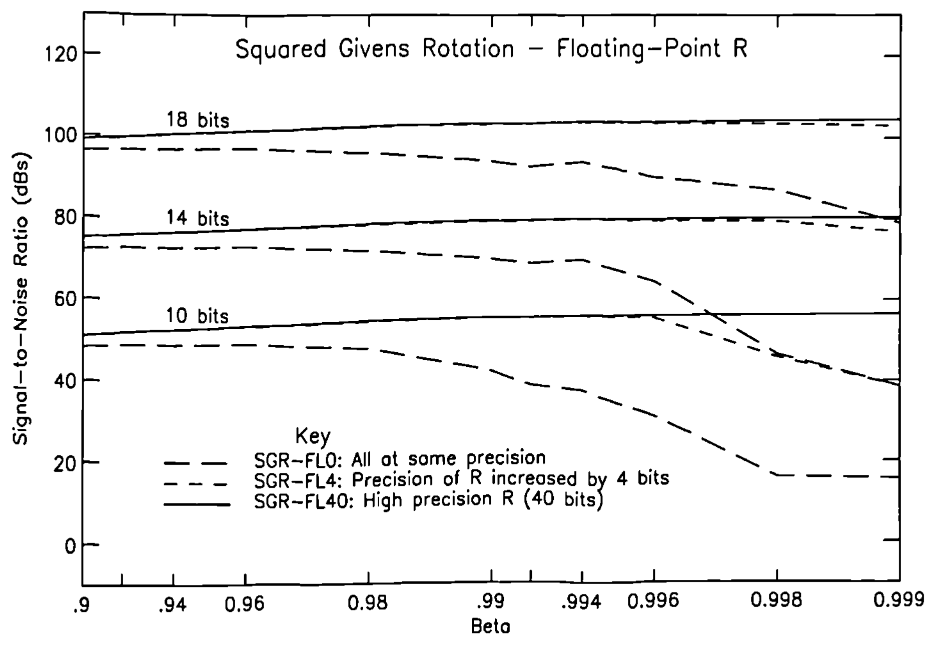


Figure 7.12 Performance of SGR algorithm with floating-point loop variable

Although, the enhanced SGR algorithm improves the SNR and enables reduced area implementations, the latency is still quite high. The loop in the internal cell consist of a floating-point adder and beta-multiplier which have a combined latency of 4, resulting in a maximum sample-rate of 25MHz. By merging the beta-multiplier with the adder it should be possible to achieve a latency of 3 and a sample-rate of 33MHz. However, as shown in the next section, the latency can be reduced to single-cycle by using fixed-point arithmetic within the loop and a single 3-input adder to implement both operations.

7.4.5 Fixed-Point Loop in the SGR Algorithm

A fixed-point number range can be obtained in the loop, as done for the conventional algorithm, by scaling the array input to ensure that the loop variables, i.e. the $\bar{r}_{i,j}$ terms, are less than one. In the SGR algorithm the loop variable is given by

$$\bar{r}_{ij} = \sum_{k=1}^{n} \beta^{2(k-n)} x_{ii} x_{ij} \tag{7.3}$$

which can be normalised by noting that

$$\frac{\sum_{k=1}^{n} \beta^{2(k-n)} x_{ii} x_{ij}}{\sqrt{\sum_{k=1}^{n} \beta^{2(k-n)} x_{ii}} \sqrt{\sum_{k=1}^{n} \beta^{2(k-n)} x_{ij}}} \le 1 \tag{7.4}$$

As in the array-normalised algorithm, a single fixed worst-case value of normalisation can be applied to the inputs. In this case, multiplication of the inputs by v^2 is sufficient (where v is defined in equation (7.1)). In practice, a slightly greater level of scaling should be applied to provide headroom for correct representation of the errors. If this is not done, the errors can cause numbers close to 1 to overflow. Note that if saturation is used to avoid the large errors associated with overflow, it will introduce a bias in the errors which will increase the overall level of error in r considerably.

The simulation results obtained using fixed-point arithmetic in the loop are shown in Figure 7.13.

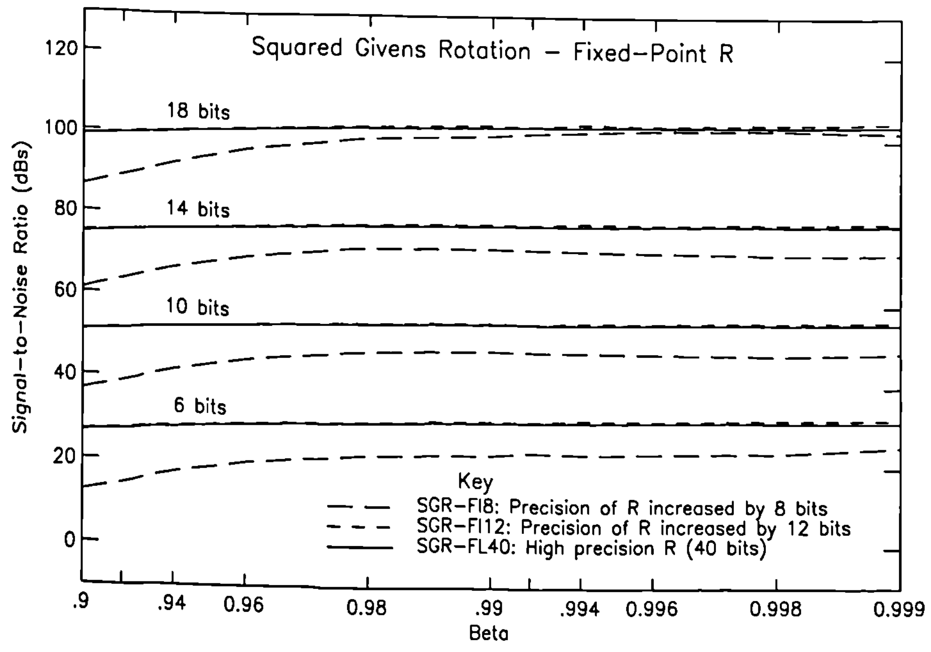


Figure 7.13 Performance of SGR algorithm with fixed-point loop variable

It is clear from the simulation results that an extension of the wordlength by 12-bits is sufficient to obtain optimum performance (represented here by SGR-FL40). This can be obtained at little cost in area due to the simplicity of the loop. In Chapter 5 a delay of 13.24ns was obtained using a carry-look-ahead adder with a 16-bit floating-point input and a 32-bit fixed-point input for the loop variable. This is not fast enough for single-cycle latency at 100MHz. However, redundant arithmetic may be used within the loop to reduce the time required to perform the addition and make it independent of the wordlength. In this case, a delay of 7.33ns can be achieved, which would allow single-cycle latency and a sample-rate of up to 136 MHz to be achieved for any realistic wordlength. (Although, to achieve a sample-rate above 100 MHz it would be necessary to apply higher levels of pipelining in the other operators outside of the loop.)

7.4.6 Further Simplification of the Loop of the SGR Algorithm

A further simplification of the loop, which would allow even higher sample-rates to be achieved, is to replace the beta-multiplier by a simple single-bit shifter. This can be achieved in the following way.

The beta-multiplier is used to implement an exponential decay of the loop variable, which can also be achieved by scaling-up the input, rather than scaling-down the loop variable. In this way, the ratio of input to loop variable is maintained, but the multiplication is performed outside of the loop.

The scaling applied outside the loop is β^{-n}, and so grows exponentially with the iteration number n (because $\beta^{-1} > 1$). Its magnitude can be limited by periodically scaling-down both it and the loop variable by 0.5. If this is done each time the input scaling exceeds 2, the scaling of the loop variable can be maintained between 1 and 2.

Figure 7.14 (a) shows the SFG for the original loop, and Figure 7.14 (b) the SFG for the case with beta removed from the loop. In the latter figure, the second loop is present to keep track of the level of scaling which must be applied to the input. In any practical implementation it would be replaced by a circuit to generate the necessary sequence at high sample-rates (i.e. a pre-loaded shift register). The same shift operation would be applied globally to all the loop variables within the array, and only a simple 1-bit shifter would be required in each cell to implement it. The scaling would be removed from the output by a division.

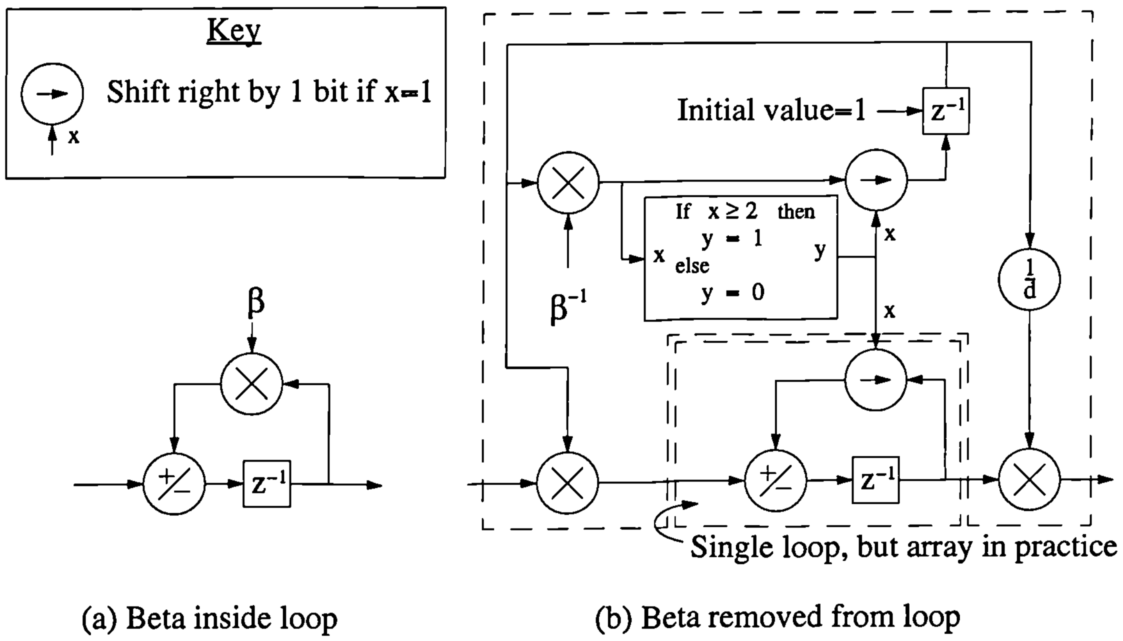

Key
Shift right by 1 bit if x=1
x
β
z⁻¹
Initial value=1
z⁻¹
If x ≥ 2 then
y = 1
else
y = 0
x
y
x
x
β⁻¹
1/d
z⁻¹
Single loop, but array in practice
(a) Beta inside loop
(b) Beta removed from loop

Figure 7.14 Removing beta from the loop

This technique does not restrict the values of beta to those which can be implemented using a shift-and-subtract operation, and it also enables more complicated windowing functions to be implemented very easily.

7.5 Divide-and-Square-Root-Free Algorithms

The fast square-root-free algorithms require a divide operation within the boundary cells. As discussed in Chapter 4, the divide operation is lengthy and needs to be heavily pipelined to achieve high-throughput. This introduces a large latency into the output, and although this does not affect the sample-rate it will require that similar levels of pipelining be applied in other signal paths, which do not benefit from it.

To avoid the divide operation, the Givens rotation algorithm has been reformulated as the DSF algorithm (as discussed in section 6.3)[Götz91]. The SFG for this algorithm is shown in Figure 7.15.

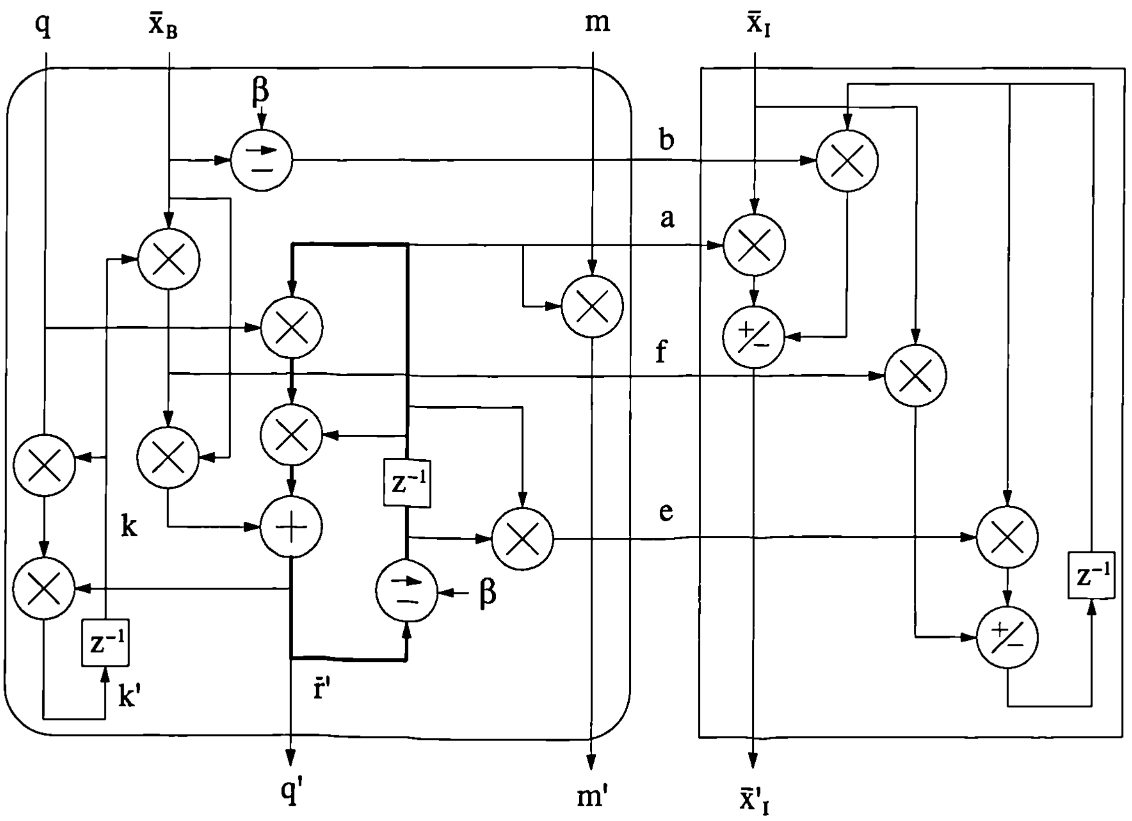


Figure 7.15 SFG of DSF algorithm

It is clear from the SFG that the DSF algorithm has been achieved at the cost of a more complex internal cell and greater communication between the cells. Also the recursive loop of the boundary cell is quite complicated. For these reasons it is not discussed further, other than making the following comments:-

A potential benefit of the DSF algorithm is that both the boundary and internal cells can be realised from a multiplier-adder building block. A disadvantage of this algorithm is a lack of stability, as terms are not bounded[Fran94]. This is to be expected, as the division operation served to provide normalisation of terms. A solution has been to introduce scaling by powers of 2 to bring the variables into a more acceptable range. The scaling is applied on a row by row basis, and in effect implements the exponent of the missing division.

7.6 Comparison Between Algorithms

7.6.1 Gate-Count Requirements

So far, a range of algorithms has been examined where the primary objective has been to achieve high sample-rate solutions. Another important property for VLSI implementation is the area-time product of each variant as this will be a key factor in determining the silicon area required to achieve a particular sample-rate for a given problem size. Therefore, in this section the time-area products are estimated for the algorithms so that they may be compared.

The time element of the area-time metric can be fixed by implementing all operators with the same throughput, and a figure of 100 MHz is considered realistic for the operators developed in the earlier chapters of this thesis.

Determining the area is a little more complicated as it depends upon a range of factors. It can be established for a particular array size by determining the number of cells and the area required by each. The cell area is a function of the number, type and wordlength of the operators used. The latter can be established for a particular beta-factor and SNR using the simulation results presented in this chapter.

The simulation results were obtained for an array with 11-primary and 1-auxiliary inputs. For this array size the number of each cell type is shown in Table 7.2.

Table 7.2 Numbers of cells required by QR-array

Cell Type	Formula	Number of cells operations when $p = 11, q = 1$
Input Cell	$p+q$	12
Boundary Cell	p	11
Internal Cell	$\frac{(p-1)p}{2}+qp$	66
Output Cell	q	1

The number and type of arithmetic operations required by each cell is shown for each variant of the Givens rotation algorithm in Table 7.3.

Table 7.3 Operation count of algorithm variants

Algorithm	Cell	Add	Mult	Square	Beta Mult	Div	Square -Root	Special Function
Square-root	BC	1	1	2	1	2	1	
	IC	2	4		1			
	OP		1					
Cell Normalised	IP	1		1	1	1	1	
	BC		1					1
	IC	2	5			1	2	
	YIC	2	6			1		2
	OP		3					
Column Normalised	IP	1		1	1	2	1	
	BC	1	2			2	1	
	IC	2	5					
	OP		2					
SGR	BC	1	3		1	2		
	IC	2	2		1			
	OP		1					
DSF	BC	1	8		2			
	IC	2	4					
	OP					1		

The number of gates required by each operator as a function of wordlength has been extracted from the circuit synthesis results presented in earlier chapters. These are summarised in Table 7.4, where w is the basic wordlength used and z is any extra wordlength used in the representation of the loop variable. (Note that the relationships marked by an asterisk have been estimated based on the results of the floating-point operators.)

Table 7.4 Operator area as a function of wordlength

Operator		Fixed-Point		Floating-Point	
		Gate Count	Latency	Gate Count	Latency
Adder	CLA	$14.7w$	1	$260 + 125w + 115z$	3
	SBNR			$81.6w+35.3z$	1
Multiplier (Wallace Tree)		$9.43w^2$ *	2	$613 + 9.43w^2$	2
Squarer (Wallace Tree)		$9.43\frac{(w+1)w}{2}$ *	2	$613 + 9.43\frac{(w+1)w}{2}$	2
Beta Multiplier (CLA)		$36.4w$ *	1	$46.9 + 36.4(w + z)$	1
Divider (Mod. SRT)		$1534 + 32.9w^2$ *	$\frac{w}{2} + 3$	$1534 + 32.9w^2$	$\frac{w}{2} + 3$

Table 7.4 Operator area as a function of wordlength

Operator	Fixed-Point		Floating-Point	
	Gate Count	Latency	Gate Count	Latency
Square-Root (Mod. SRT)	$1534 + 32.9\frac{(w+1)w}{2}*$	$\frac{w}{2}+3$	$1534 + 32.9\frac{(w+1)w}{2}*$	$\frac{w}{2}+3$
Special Function	$11w^2*$	$\frac{w}{8}$	N/A	

The number of gates required to implement the array with a sample-rate of 1-MHz and an SNR of 60dBs are shown in Figure 7.16 as a function of the forget-factor, β.

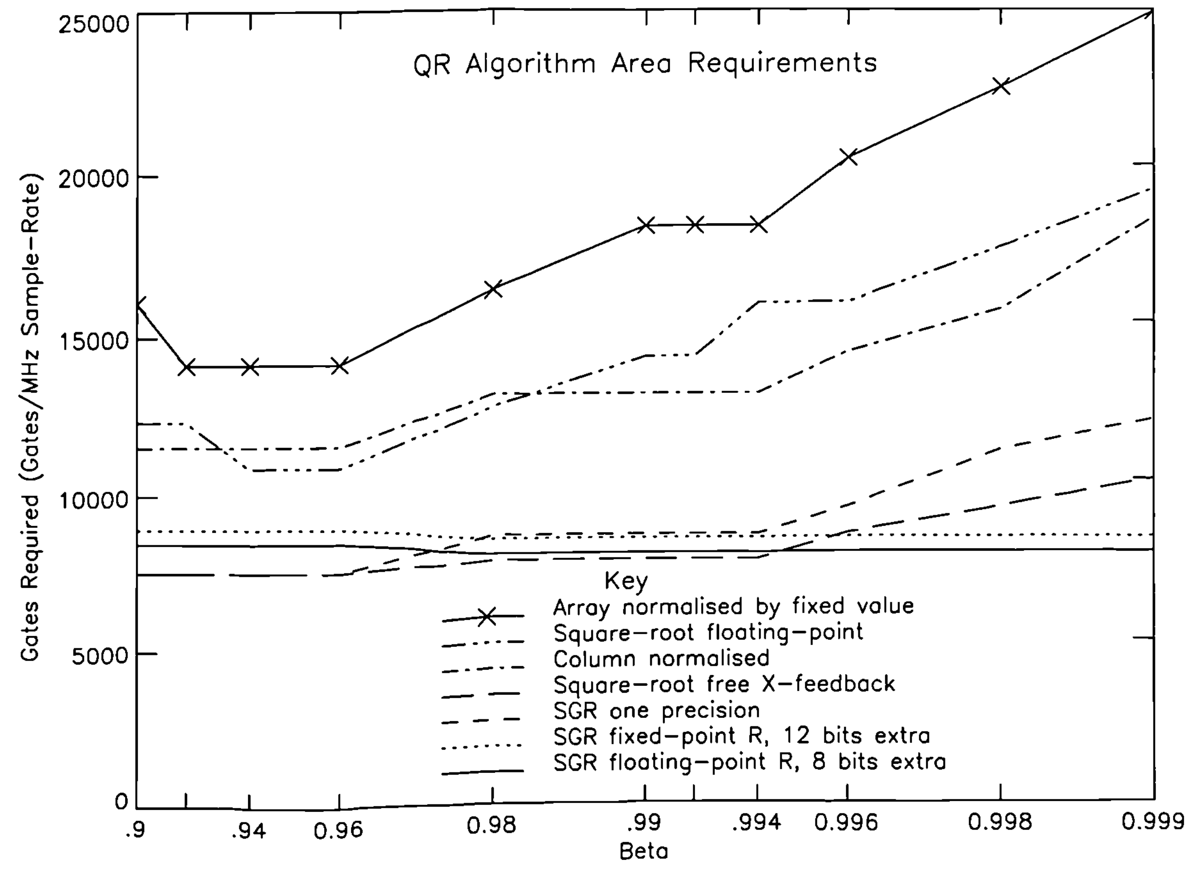


Figure 7.16 Number of gates required to meet an SNR of -60dBs

The cell normalised algorithm is not shown, as it is very expensive due to the numerical anomaly, and would be off the scale.

The numbers of gates required to implement the conventional algorithm using fixed- and floating-point arithmetic are similar. The improvement offered by the enhanced-SGR algorithm is significant for all values of β, but becomes very significant as $\beta \rightarrow 1$ and the window length increases. Also shown is the number of gates required for the SQF-XFB algorithm. This algorithm has been quite popular, but can exhibit numerical problems which result in slow convergence. Once convergence has been achieved, however, it provides similar levels of SNR to the enhanced-SGR algorithm and so similar numbers of gates are required, as indicated in the figure. However, as $\beta \rightarrow 1$ the SNR and wordlength requirements of the SQF-XFB algorithm increase. For very large window lengths the benefits of the enhanced-SGR algorithm over this and all the other variants becomes very significant.

7.6.2 Sample-Rate

A summary of the sample-rates which may be achieved using each algorithm variant is presented in Table 7.5.

Table 7.5 Latency of operations in loop and QR-array sample-rate

Algorithm Variant		Number Representation	Latency	Sample rate, (MHz) f_{ck} = 100MHz
Conventional (with modified BC)	Floating-point	Non-redundant	6[a](5)[b]	16.7(20)[b]
Normalised (fixed-point input)	Fixed-point	Parallel redundant	1[a]	100
		msdf	2	50
SGR	Floating-point r	Non-redundant	4[a](3)[b]	25(33.3)[b]
	Fixed-point r	Parallel redundant	1	100(136)[c]
DSF		Non-redundant	5[a]	20

a. Wordlength dependent, given for 16-bit arithmetic.
b. If beta-multiplier simplified to a 1-bit shifter or merged with another operation.
c. Maximum sample-rate if operators outside of loop are more highly pipelined.

The enhanced-SGR algorithm, using a fixed-point redundant representation for the loop variable r, offers the highest sample-rate, and is the only algorithm to offer latency which is single-cycle and independent of wordlength.

7.6.3 Conclusions

The enhanced-SGR algorithm offers minimum area and the highest sample-rate. For very large window lengths, it will offer very significant savings in area over other algorithms.

Chapter 8 Architecture of Adaptive Filter

8.1 Introduction

In the previous chapter, algorithms and architectures were examined for the VLSI implementation of the boundary and internal cell processes of the QR-algorithm. In this chapter a range of architectures are developed to use these cells to implement high-throughput parallel processor arrays to perform the QR-algorithm for a range of problem sizes. In particular, a new linear array has been developed which offers local interconnection and wide flexibility in number of processors used.

VLSI layout has been obtained for the boundary and internal cells for one array architecture using the enhanced SGR algorithm. This has provided an estimate of the level of performance achievable using an application specific approach, and enables a comparison to be made with a CORDIC based solution. The results are also compared with a solution implemented on an array of general purpose (GP) DSP chips. This second comparison provides an indication of the enormous performance improvements possible using an application specific VLSI approach.

8.2 Parallel Array Processing

To achieve high throughput for applications such as radar adaptive beamforming it is necessary to adopt a highly parallel computing approach. One attraction of the QR-algorithm is that it may be implemented very efficiently using the QR-array SFG. Maximum throughput is obtained by allocating a processor to each cell in the array, in which case the solution is referred to as *full-sized*[Bu90].

However, due to current technological constraints, such as chip size and package pin-count, it is not possible to realise the whole array on one chip. Also, the throughput of such an array is likely to be greater than required. Therefore, an array processor architecture is required which enables reduced area to be obtained at the expense of reduced speed. This is referred to as the *partitioning problem* and is a key issue which must be addressed in the development of a VLSI array processor.

Figure 8.1 shows the stages in the design of an array processor for the particular case of the QR-algorithm.

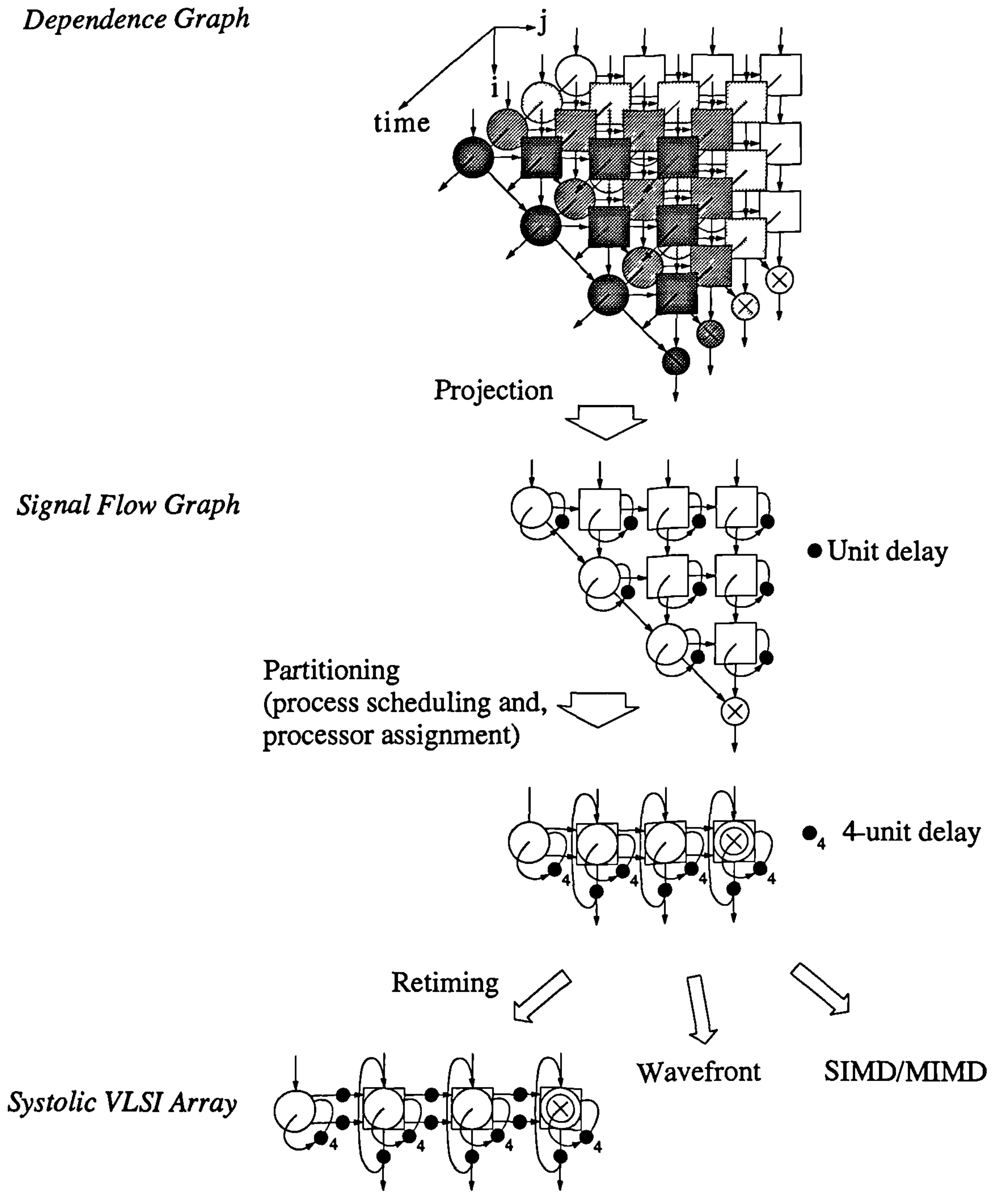


Figure 8.1 Stages in the design of a VLSI array processor

The dependence graph (DG) describes the algorithm in terms of the processes which must be performed and their data dependence. The QR-algorithm is recursive, so the DG has a time dimension, which shows the operations of the QR-array being performed at each time instant on the input vector.

The QR-array SFG is obtained by *projecting* the DG down the time axis onto a set of processing nodes. The data dependencies between the operations at one time instance and the next are maintained by placing latches on the data paths between them (in this case on all the r and u variables updated by the cells). The SFG can be viewed as a simplified DG which is a step closer to the hardware architecture used to implement it. It provides a good starting point, free from the semi-infinite time dimension of the DG, from which to perform the necessary architecture manipulations to obtain practical solutions.

One manipulation often required is to reduce the number of processors by partitioning. This may be performed by a further projection. The linear array shown in Figure 8.1 has been obtained by a projection down the i-axis. The new SFG contains additional latches on the outputs of a processor to store the output from one set of operations so that they may be applied as the inputs of the operations performed on the next time instance. Also the number of latches in the loops has been increased to 4 to store the parameters of each column of the original array which are updated by each of the processors in the linear array.

In the current form of the SGF, it is assumed that each processor performs its function instantaneously. Therefore, a final step is required to *retime* the graph to account for the actual delays of the physical VLSI processor. A *systolic array* is a special result, where processes and data movement are synchronised by a global clock. The clock is the only global signal, as all other connections between processors are local. This latter property avoids the delay and power associated with broadcasting signals over large distances.

In the next three sections, three array processor solutions are considered:

- **Full-sized array**
- **Locally sequential globally parallel (LSGP) array:** The array is partitioned into a number of blocks. The operations within a block are performed in a sequential fashion by one processor. An array of these processors is used to process the blocks in parallel.
- **Locally parallel globally sequential (LPGS) array:** The array is partitioned into blocks. The blocks are processed in a sequential manner by an array of processors which perform the operations within a block in parallel.

8.3 Full-Sized QR-Array Solution

The full-sized array solution offers maximum throughput, 100% utilisation of processor cells and an output in minimum time. When the enhanced-SGR algorithm is used, this solution also offers the maximum sample-rate of one input every clock cycle.

A rectangular shaped array is more convenient for VLSI implementation, and can be obtained by *folding* the array. Figure 8.2 (b) shows the folding proposed by Rader[Rade92]. As it stands, the array requires global interconnect to connect the two parts of the array. This can be avoided by a second folding, shown in Figure 8.2 (c), which interleaves the processors from each half of the array to give that shown in Figure 8.2 (d).

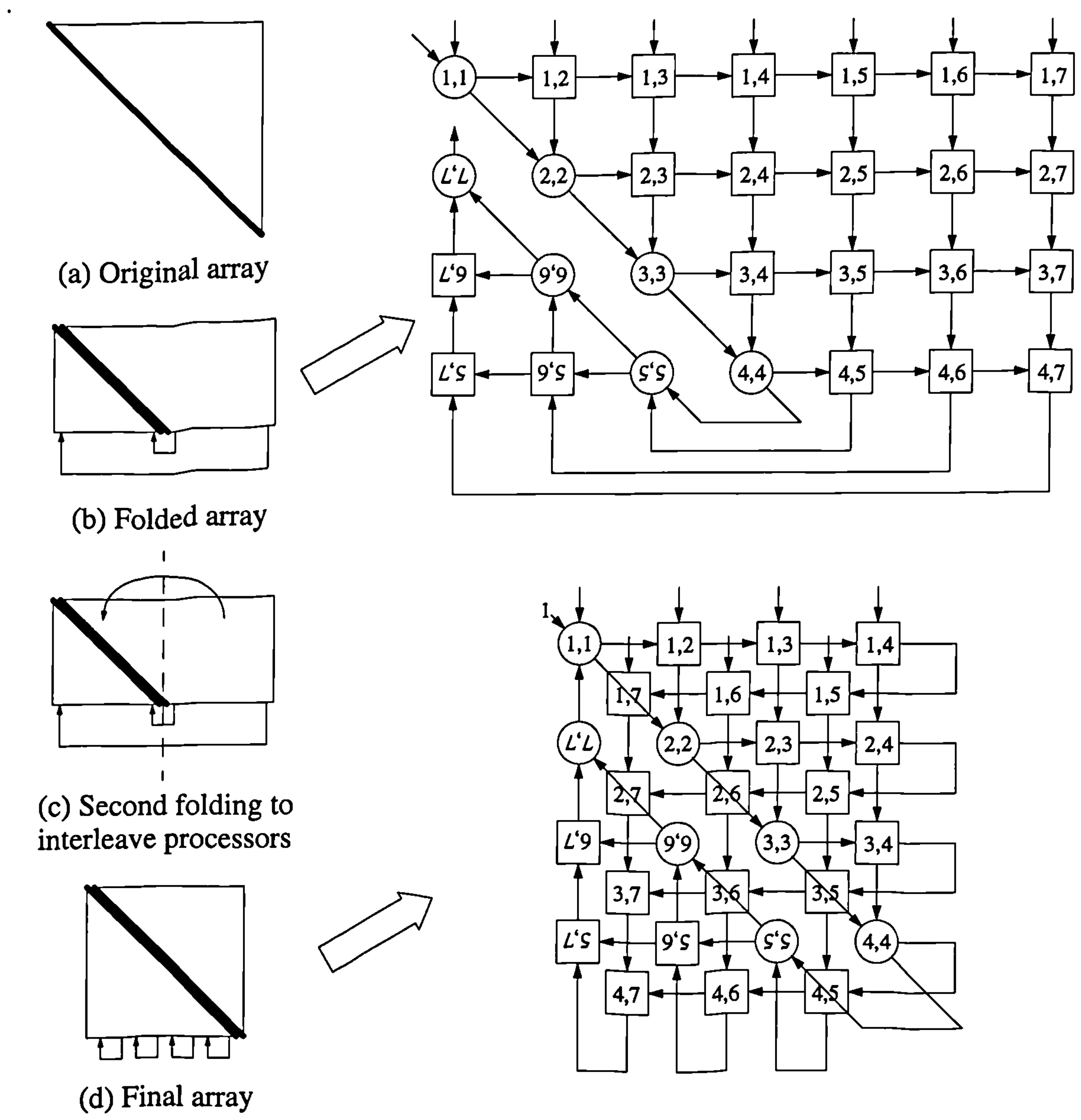
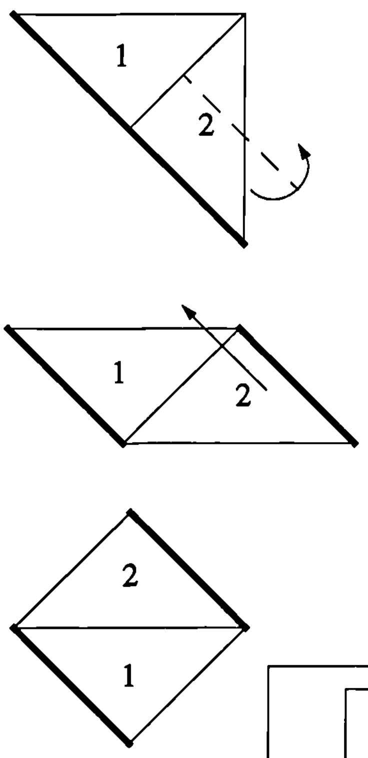
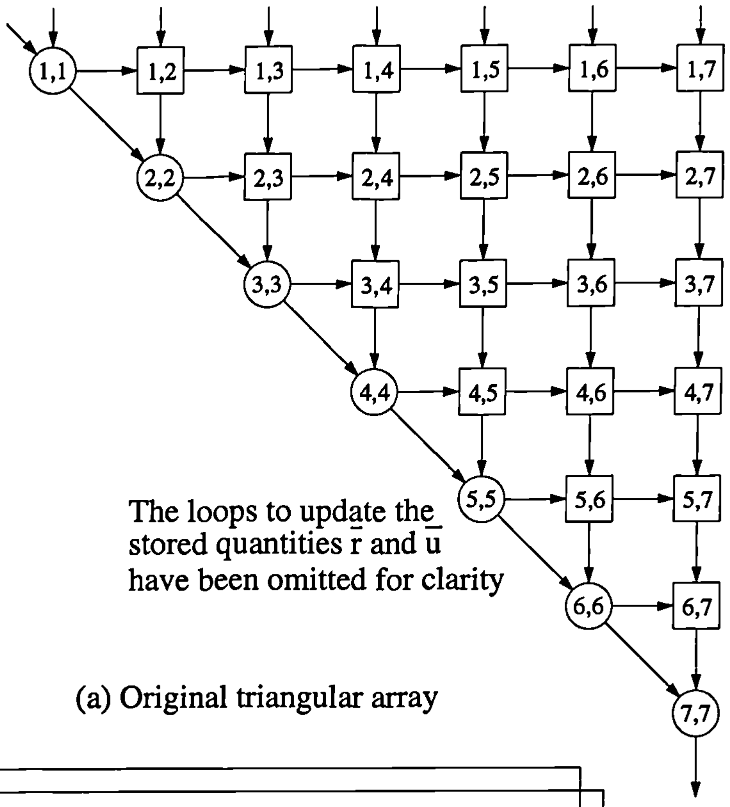


Figure 8.2 Rader's folding of the QR-array

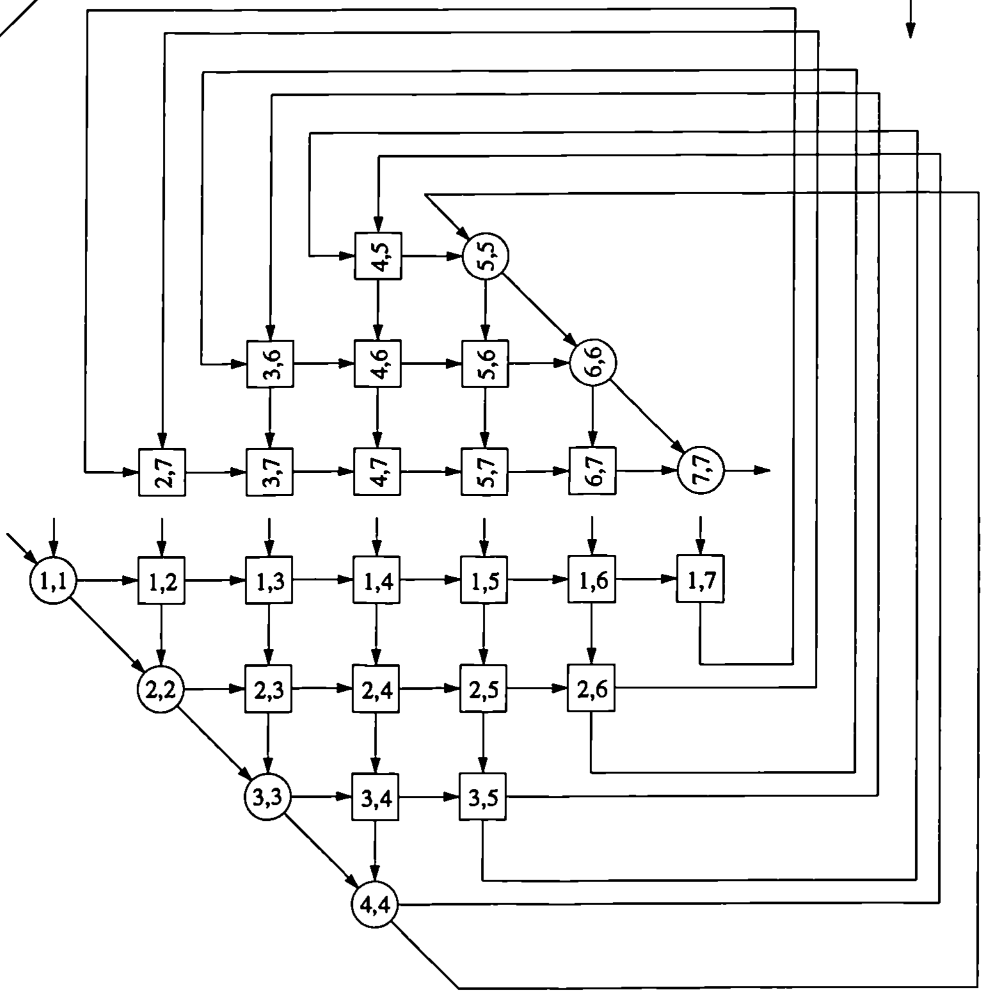
8.4 LPGS Linear Array Solution

The number of processors necessary to implement the QR-algorithm can be reduced by projecting the operations of the QR-array onto a smaller number of processors. If the projection is performed so that processors either perform the function of a boundary or an internal cell, it is possible to avoid the inefficiency of using a processor designed to implement both functions. This is not possible for the arrays shown in Figure 8.2 (without producing inefficient implementations), so a new folding has been proposed, which is shown in Figure 8.3.




The loops to update the stored quantities $\bar{r}$ and $\bar{u}$ have been omitted for clarity

(a) Original triangular array



(b) Folded array

Figure 8.3 An alternative folding of the QR-array

The bottom-right corner of the array has been reflected about the diagonal and moved to the top to obtain a rectangular shape (set at 45° in Figure 8.3). As will be shown shortly, the global interconnect will be removed by the projection on to a linear array, however, the connections are also transposed, and this must be removed first. This can be achieved by folding the array to interleave the processors, as shown in Figure 8.4.

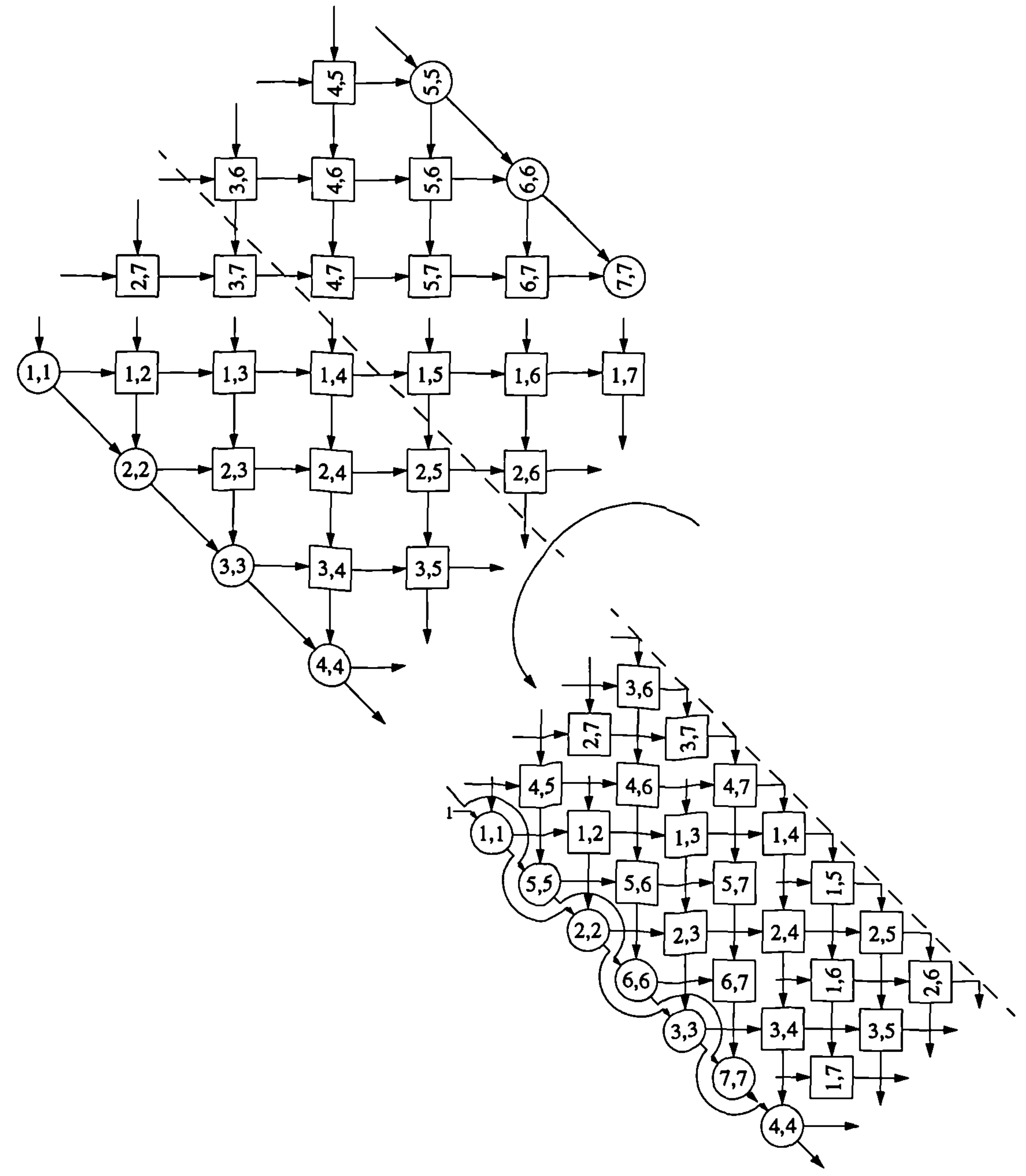


Figure 8.4 Array with diagonal fold marked

This fold also places all the boundary cell operations back on one diagonal. By projecting down the diagonal it is possible to assign all the boundary cell operations to one boundary cell processor and all the internal cell operations to a row of internal cell processors. In each diagonal there are the same number of operations, so the loading of each processor, in terms of cell computations per second, will be the same.

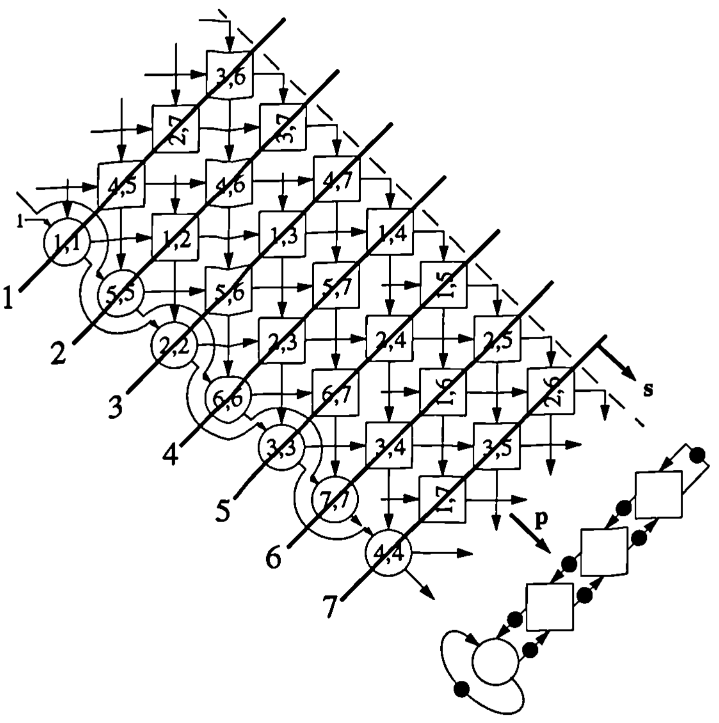
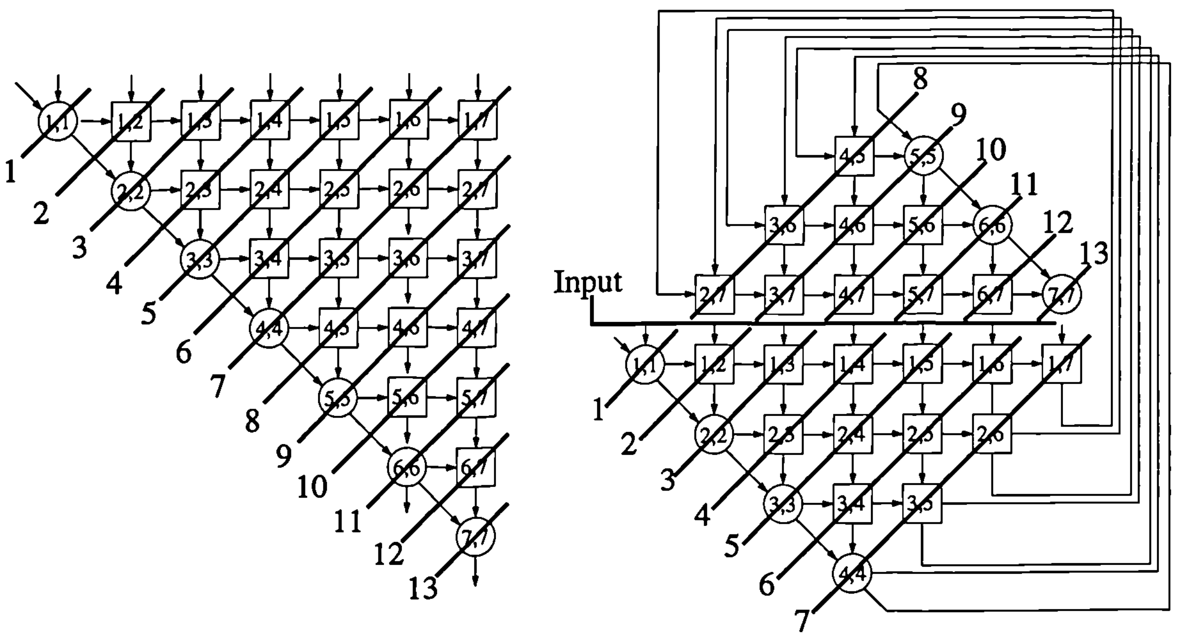


Figure 8.5 LPGS projection of the folded QR-array

The order in which the operations are performed on the linear array is identified by the *schedule*. This is shown in Figure 8.5 as a number of thick parallel lines (referred to as *hyperplanes* in cases where there are more than two dimensions to the SFG). These cut across operations in the array which should be performed at the same time. The schedule is also denoted more compactly in Figure 8.5 by a schedule vector s normal to the hyperplanes.

The schedule indicates that the linear array should perform, at each instance in time, the operations of one diagonal row of the array in parallel. The whole array is processed by performing the operations in each diagonal in a sequential manner, cycling from the top of the array to the bottom. For these reasons the solution is referred to as *locally parallel globally sequential* (i.e. LPGS).

A valid schedule is obtained by ensuring that the data required by each set of scheduled operations is available at the time of execution. This implies that data must flow across the schedule lines in the direction of the schedule vector. This is true in Figure 8.5 for all data connections, but those that pass from the bottom of the array to the top (not shown in the figure) due to the first fold. These dependencies can be removed by delaying the dependent operations until the data is available. This results in a delay of one cycle of the array, which can be easily justified by considering the scheduling of the unfolded array, as shown in Figure 8.6 (a). Figure 8.6 (b) shows the array and schedule with the first fold. The schedule for those operations that came from the bottom of the original array can be overlapped with that of the rest of the array, as at each time instance there are enough processors in a linear array to implement the operations on a diagonal from both parts of the array. However, the operations associated with the top part of the array will process the data output from the bottom part from the previous cycle of the array.



(a) Scheduled original triangular array (b) Scheduled folded array

Figure 8.6 Scheduling of the QR-array

Figure 8.7 shows a more detailed description of the SFG for the LPGS linear array.

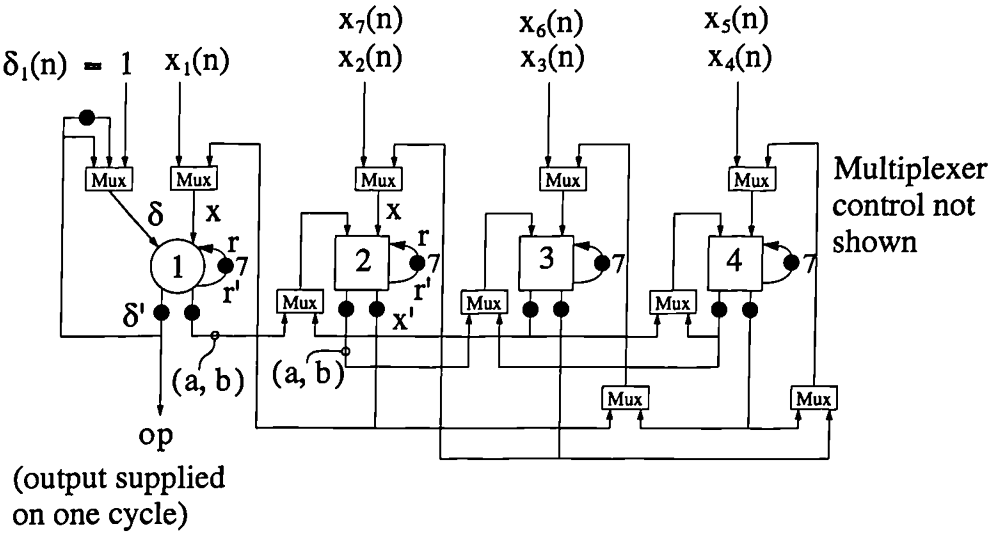


Figure 8.7 Linear systolic array for the LPGS schedule

The latches are present on all processor outputs to maintain the data between operations performed on one diagonal and the next. The latches for maintaining the parameters within the cells are also shown. In this case, a total of 7 latches are required; one for each diagonal of the 2D-array.

Multiplexers are present at the top of the array so that the inputs to the QR-array can be supplied to the cells at the right instances in time. The multiplexers at the bottom of the figure are present to cater for the different direction of data flow that occurs between rows of the original array (occurring due to the second fold).

8.5 LSGP Solution

Greater reductions in the number of physical processors required to implement the QR-array can be achieved by adopting a LSGP solution. Figure 8.8 shows a doubly-folded QR-array with 9 inputs. A schedule has been chosen to reduce the number of processors operating at any one time to three. Consequently, the array can be projected onto a 3-processor linear array, as shown in Figure 8.8.

Notice that a boundary cell operation is only required on alternate instances in time. Therefore, the boundary cell processor will be utilised only 50% of the time. This inefficiency can be avoided by designing the boundary cell processor to operate with a lower throughput (giving an associated reduction in silicon area).

Greater reductions in the number of processors can be obtained by further rotations of the schedule vector.

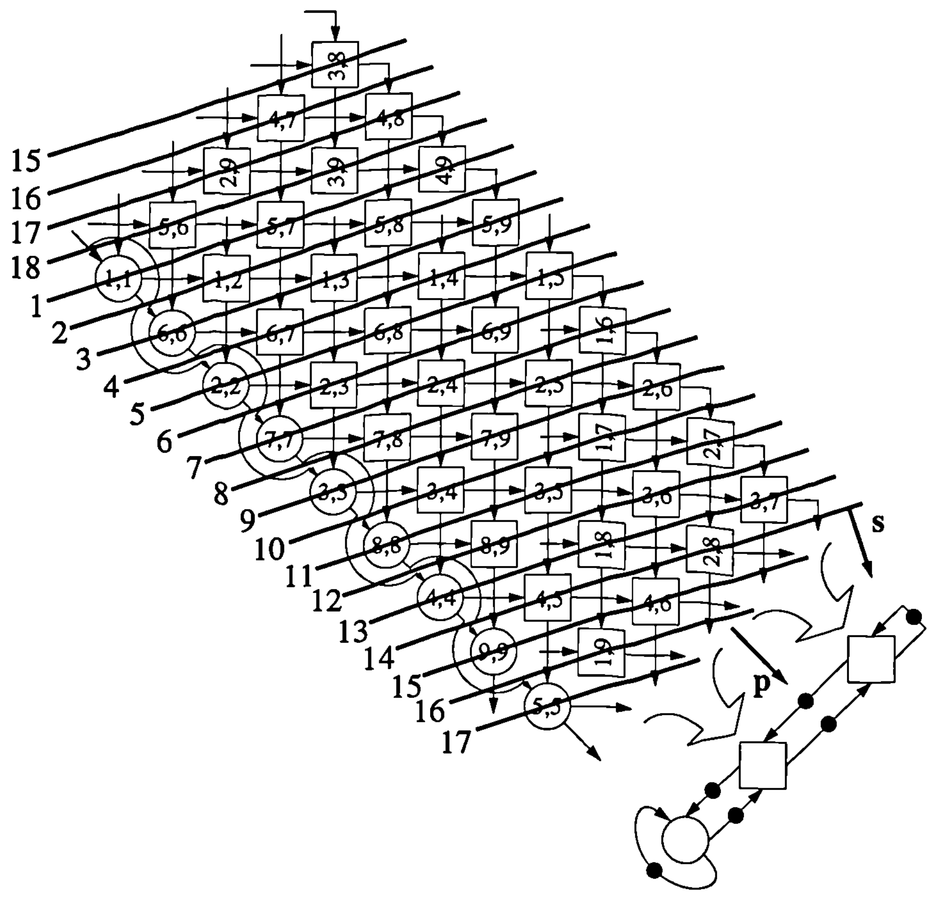


Figure 8.8 Reducing array size using an LSGP schedule

The LSGP schedule enables large arrays to be implemented using only a small array of processors and provides a good degree of flexibility over the number of processors required.

Further flexibility over the number of processors can be obtained by implementing the array using more than one linear array. This solution can be obtained by first dividing the 2D-array into a number of parts containing an equal number of diagonal rows of cells. Each part can then be implemented using a LSGP linear array. The global interconnect from the bottom linear array to the top can be avoided by a further fold. Alternatively, any delay associated with the interconnect can be accommodated by pipelining (i.e. allocating a clock cycle to the transmission of the signal).

8.6 VLSI Implementation of Full-Sized Array

Consideration will now be given to the physical implementation of the architectural solutions presented in this chapter. Two implementation approaches are considered. The first examines the use of an application specific VLSI approach and specifically considers the implementation of a small full-sized array. The layout has been produced for the boundary and internal cells using the enhanced-SGR algorithm, and area and speed figures have been obtained and compared with a CORDIC implementation of the cells. The second approach is based on the use of an array of programmable general purpose DSP chips, and a scalable implementation of the LSGP solution is presented.

8.6.1 VLSI Design of Full-Sized QR-Array

As presented in Figure 8.1 the final step in the design flow to obtain a VLSI architecture is to retime the SFG to accommodate the particular latencies of the physical processors used. Figure 8.9 shows a small full-sized QR-array which has been retimed for the enhanced-SGR algorithm. The latencies of the individual arithmetic operators are shown in the inset. The retiming process has been performed using a well established set of rules (see [Megs92] pages 49 to 54). In this particular case, the CAD tool *IRIS* was used to perform the process using a computer. IRIS has been developed by the Queen's University of Belfast[Train95], and is unique in that it can retime circuits with unconventional data formats such as msdf arithmetic. The tool generates a correctly pipelined description of the architecture in a VHDL format expressed hierarchically in terms of the operators used.

In this particular example, a floating-point representation has been used for the loop variable. A fixed-point loop could have been used to obtain high sample-rate operation just as easily but the fixed/floating-point adder operator was not available at the time. However, both adders require the same area, so the overall layout of both will be very similar.

A 0.6μm two-layer metal CMOS process was targeted, and a system clock frequency of 50 MHz was achieved for the operators (cf. to 100 MHz for 0.35μm). The area and latency obtained for each operator are summarised in Table 8.1.

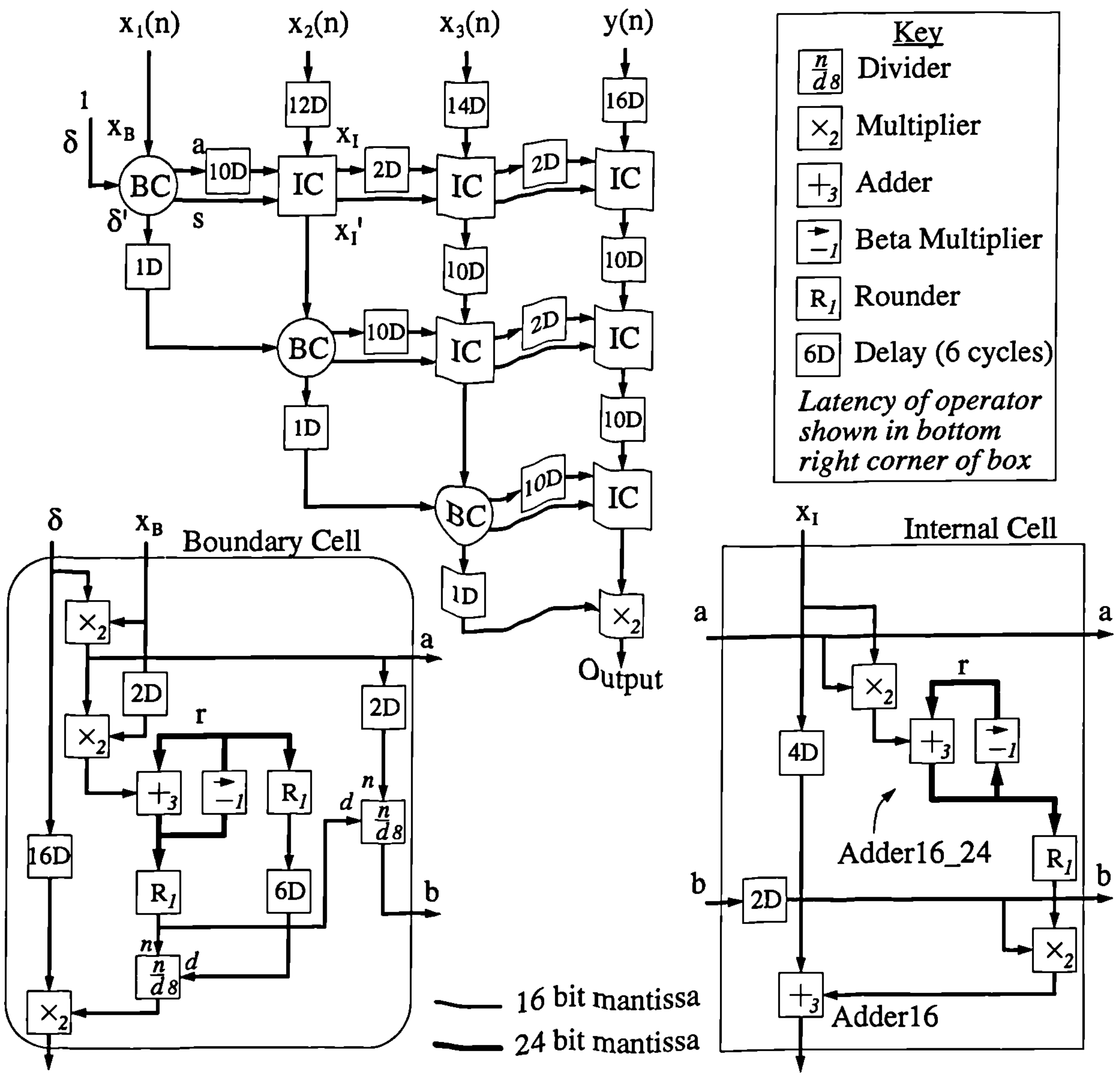


Figure 8.9 Retimed QR-array

Table 8.1 Properties of floating-point operators

Operator	Latency	Area (mm^2)[a]	Comments
Adder16	3	1.00	16+16=16-bits
Adder16_24	3	1.28	16+24=24-bits
Multiplier	2	1.50	
Divider	8	4.00	Latches every three rows
Rounder	1	0.095	Round 24-bits to 16-bits
Beta Adder	1	0.445	24-bit shift-and-subtract

a. Using 0.6μm two-layer metal CMOS process and for 50MHz operator throughput.

The layout for the boundary and internal cells is shown in Figure 8.10. The performance and area of the cells are summarised in Table 8.2.

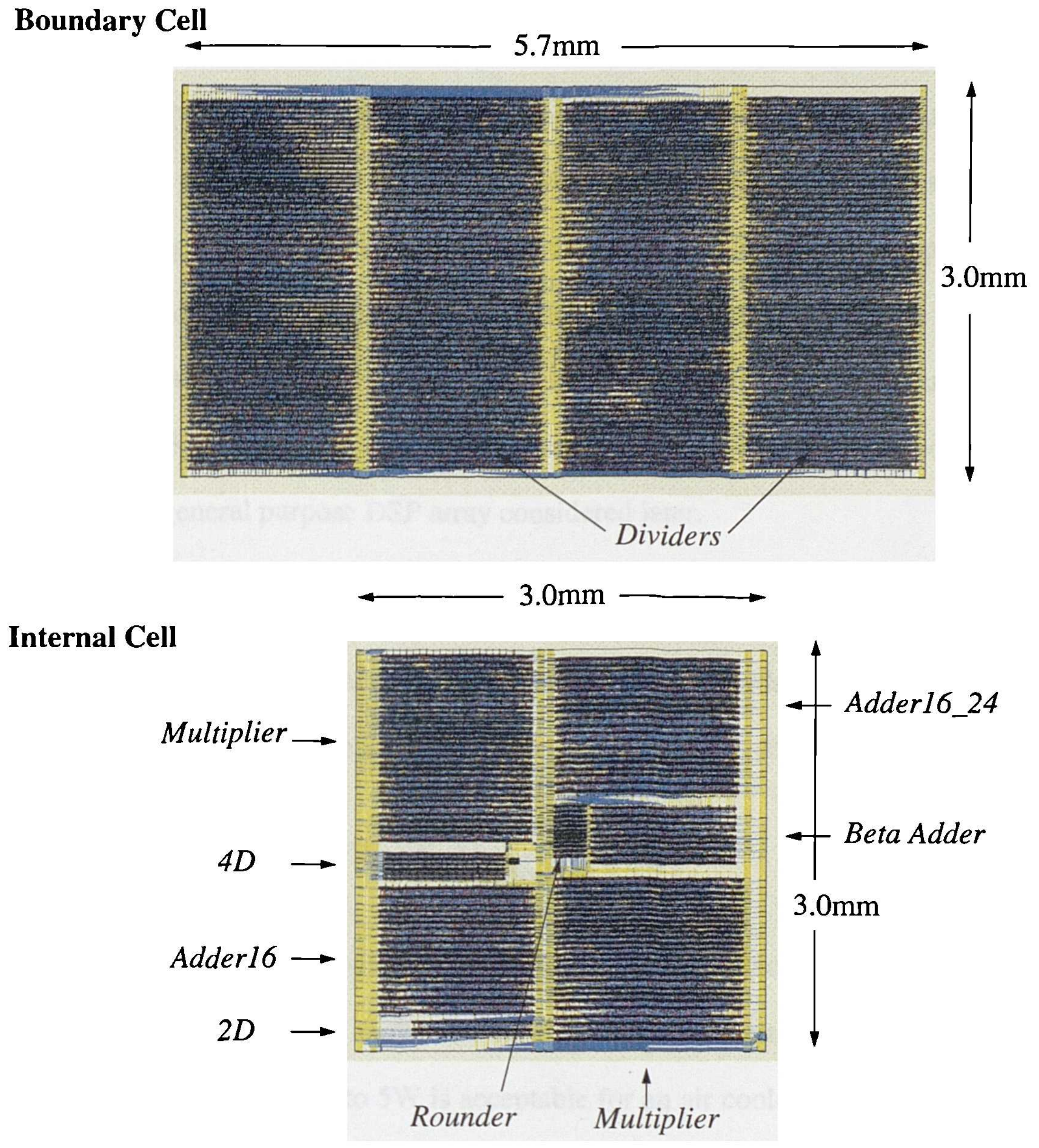


Figure 8.10 VLSI layout of the enhanced SGR boundary and internal cells

Table 8.2 Area and computation rate of the enhanced-SGR cells

Cell	Area[a] (mm^2)	Number of floating-point operations per cell	FLOPS at 50MHz clock (MFLOPS)	Latency (clock cycles)
Boundary	5.7x3.0=17.1	7	350	4
Internal	3.0x3.0=9.0	5	250	4

a. Implemented with a two-layer metal 0.6μm CMOS process

8.6.2 LPGS Architecture

Using the cell designs presented in the last section, it is possible to perform a realistic paper design of the LPGS linear array architecture for a medium sized array. Consider the case of an array with 16 auxiliary inputs and 1 primary input. This could be implemented using a linear array consisting of 1 boundary cell and 8 internal cells. A simple calculation suggests that an area of $17.2 + 8x8.15 = 82.4mm^2$ would be required for the processors. Allowing area for routing (which would be local) and the additional $17 - 4 = 13$ pipeline delays in the recursive loop, a single die of size 10x10mm should be sufficient. A sample-rate of 3.125MHz could be achieved, which relates to a sustained computation rate of 2,350 MFLOPS[1]. This is an impressive figure, particularly if it is compared to results obtained from a general purpose DSP array considered later.

The power consumption of such a device is difficult to estimate accurately as it depends on many factors. However, in practice it is found to be relatively independent of circuit function, and related to area and frequency in the following manner[West93], p 235:

$$P = K_p A \left(\frac{f_{ck}}{100 x 10^6}\right)\left(\frac{V}{3.3}\right)^2 \tag{8.1}$$

For 0.5µm technology K_p=45mW/mm^2 and for 0.35µm technology K_p=60mW/mm^2. Therefore, the chip described above will have a power consumption of approximately 4.6W. A power consumption of up to 5W is acceptable for an air cooled low cost package. Up to 15W can be sustained if fan cooling is used. Above this, a more sophisticated cooling scheme is required, e.g. using a thermo-syphon.

A very important issue in the past has been power dynamics. Currently, this is being solved using on-chip capacitors and extra layers of metal to provide power planes. Therefore, constructing large arrays of synchronous processors on chip is a practical proposition.

1. Note that these floating-point operations are for 16- and 24-bit arithmetic.

8.6.3 Comparison with CORDIC

As mentioned in Chapter 1, CORDIC is a popular method for implementing the Givens rotations. This option was considered by Hamill with the objective of comparing the CORDIC approach with that adopted in this thesis. Hamill has investigated two options. The first was based upon an msdf implementation of the CORDIC algorithm developed by Hamill and Walke[Hami95b]. This achieved a wordlength-independent latency of 10 cycles, but was considered to be very area intensive. The second approach used a more conventional implementation of CORDIC and consequently a much higher latency of 25 was obtained for a 16-bit mantissa. The latter has been synthesised and a direct comparison with the SGR algorithm can be made as presented in Table 8.3.

Table 8.3 Comparison between SGR and CORDIC approaches

Property		CORDIC		SGR	
		Boundary	Internal	Boundary	Internal
Latency (minimum)		25		1[a]	
Cell throughput (million cell computations per second)		52.6		50	
Maximum input sampling-rate		2.1		12.5/50[a]	
Cell area (mm^2)		20.9	20.0	17.1	8.15
Area required to implement array[b] (mm^2)	Per cell type	6.37	60.8	2.60	22.2
	Total	67.2		24.8	

a. If the enhanced-SGR algorithm is used with a fixed-point loop.

b. In this case the QR-algorithm has 16 primary inputs, 1 auxiliary input and a sample-rate of 1MHz. The SGR processor requires $136x10^6$ internal cell operations and $16x10^6$ boundary cell operations per second. The CORDIC processor requires an additional $16x10^6$ internal cell operations per second to implement the product of cosines for direct residual extraction.

It is clear from the table that the SGR algorithm offers almost a 3:1 advantage in area and a 25:1 advantage in sample-rate.

A reason why the SGR algorithm is so much better in terms of latency and area is that an approach based on standard arithmetic operators provides the flexibility to optimise the SFG. In this way it has been possible to simplify the loop and make significant savings in the size of the cells, particularly the most common one. The CORDIC algorithm implements the whole rotation in one combined operator; which leaves little scope for optimisation.

One important advantage of a CORDIC approach is that the boundary and internal cells can be implemented using a single processor design because the solutions are so similar. This considerably simplifies the VLSI implementation of the QR-algorithm with a programmable array size. Also the basic CORDIC operation can be extended to other functions, allowing the design of a single, very flexible chip.

The CORDIC approach can also be beneficial in applications, such as singular value decomposition, where rotations by one of a small number of angles are required, but where the new coordinates need to be calculated accurately[Götz95] (i.e. the rotations are orthogonal). In this case, only a small number of CORDIC iterations are required and the latency and area are very much reduced.

8.7 QR-Array Implementation using General Purpose DSP Processors

8.7.1 A Parallel Processor Array Implementation

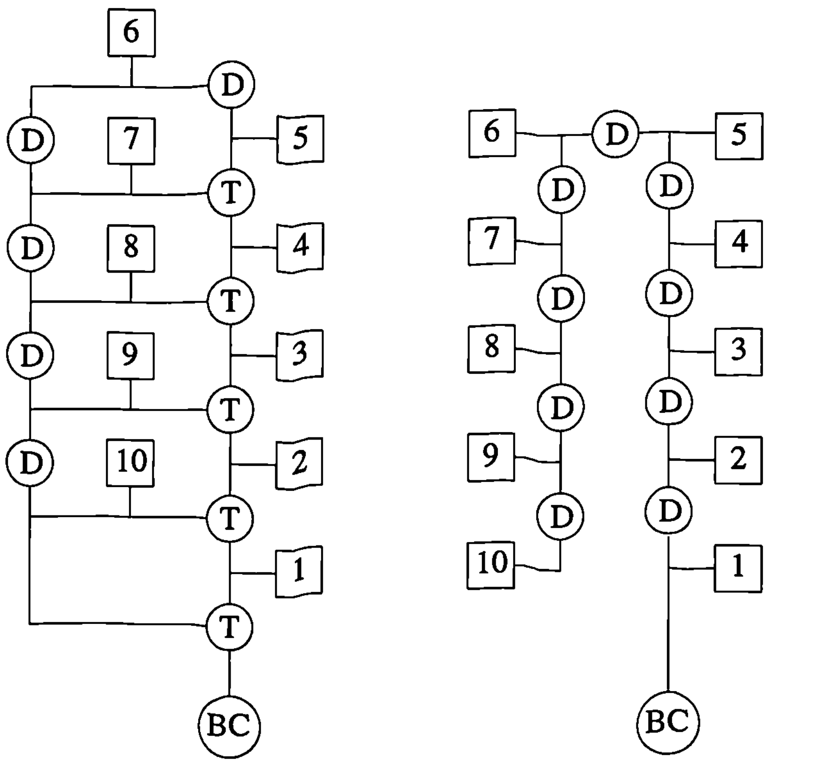
A more flexible alternative to an ASIC approach is to use programmable general purpose DSP processors to implement the QR-algorithm. The low throughput of these devices can be overcome by using many of them in parallel. Coffey *et al.* [Coff96] have developed a processor architecture specifically for implementing the QR-algorithm. The architecture is shown in Figure 8.11(a) and consists of a linear array of DSP32C 32-bit general purpose DSP processors interconnected using dual-port and tri-port RAMs. In the figure the architecture is shown using eleven processors. Ten processors have been used to implement internal cell operations and one processor to implement the boundary cell operations. Data is passed between processors via the RAMs using a dead-letterbox approach. The location from which an input is obtained and an output placed, is contained in a table supplied as part of each processor's program. The program is downloaded to each processor from a host PC computer.

A system has been constructed to implement an array of 45 DSP32C processors. Four

DSP32C processors are held on a card with the multi-port memories, and the size of the processor array is determined by the number of boards used.

8.7.2 Improved Architecture

This linear array architecture of Figure 8.11 (a) was not obtained using any formal design method. It has been obtained using a similar fold to that presented in Figure 8.3. However, the second fold, used to avoid the transposition of data as described in Figure 8.4, has not been performed. Consequently there is a requirement for tri-port RAMs to implement this data path. If the second fold were performed, then the architecture shown in Figure 8.11 (b), requiring only dual-port RAMs, could be used. This represents a significant simplification of the implementation.



(a) Scalable array of programmable DSP chips (b) Simplified array

Figure 8.11 Array processor of DSP chips

8.7.3 Comparison with ASIC Approach

In this section, the implementation of the QR-array using DSP32C processors is compared with an ASIC approach. The DSP array by Coffey *et al.* implements a QR-array with 81 auxiliary inputs using 45 processors. An array of this size requires the storage of 3340 r-variables. This storage would be expensive to implement on an ASIC. If the storage were off chip, then only a small number of processors could be implemented due to the I/O requirements of transferring the stored data on and off chip (i.e. the performance of an ASIC implementation is I/O limited rather than silicon area limited). Even so, such an implementation would nevertheless offer higher performance than the DSP array.

For example, Table 8.4 compares the properties estimated for an ASIC solution and for the DSP array. The ASIC estimates are based on 16-bit arithmetic, which is less than the 24-bit arithmetic provided by the DSP array, but sufficient for the beamforming applications of interest. The ASIC solution uses one boundary and two internal cells, giving a total of 6x32=192 pins to get 3 r-variables on and off the chip on each clock cycle. For an 81-input array the ratio of internal to boundary cell operations is 40:1, so in the proposed implementation the boundary cell in this ASIC implementation would be utilised only 5% of the time. Even with this inefficiency and low number of processors, the ASIC solution still offers significantly higher performance than the DSP array. It also requires significantly less volume and area, and the one-off construction costs are similar. One further benefit of an ASIC approach is that, in many ways, it is simpler to design. This is because the CAD tools are available to model the system and the fabrication process has been well characterised, whereas this is not the case for a board-level system.

Table 8.4 Comparison between programmable DSP and ASIC approaches

Property	45 Processor DSP Array	1 ASIC using SGR
Floating-point operations per second (FLOPS)	109	518
Volume (cm^3)	27,000	45
Power (W)	200	2
Estimated one-off construction cost (£ 1,000s)	65	100

8.8 Adaptive Beamforming Application

Adaptive beamforming for radar systems is an important application of the QR-array. It is possible to estimate the performance of an ASIC solution using the VLSI layout results presented in this chapter. The adaptive beamformer requires a complex arithmetic implementation of the cells which is about 2.5 times the size of the real ones considered up to this point. Typical applications require arrays of approximately 16 to 24 inputs. For this size, it is possible to store all the r-variables on chip. In which case, the number of processors which can be implemented on a single chip is determined by the silicon area available rather than the package pin-count. Figure 8.12 shows the expected sample-rate of a single chip ASIC implementation of an adaptive beamformer with 16 auxiliary inputs and 1 primary input (where the predictions of future technology have been obtained from [Nat94]).

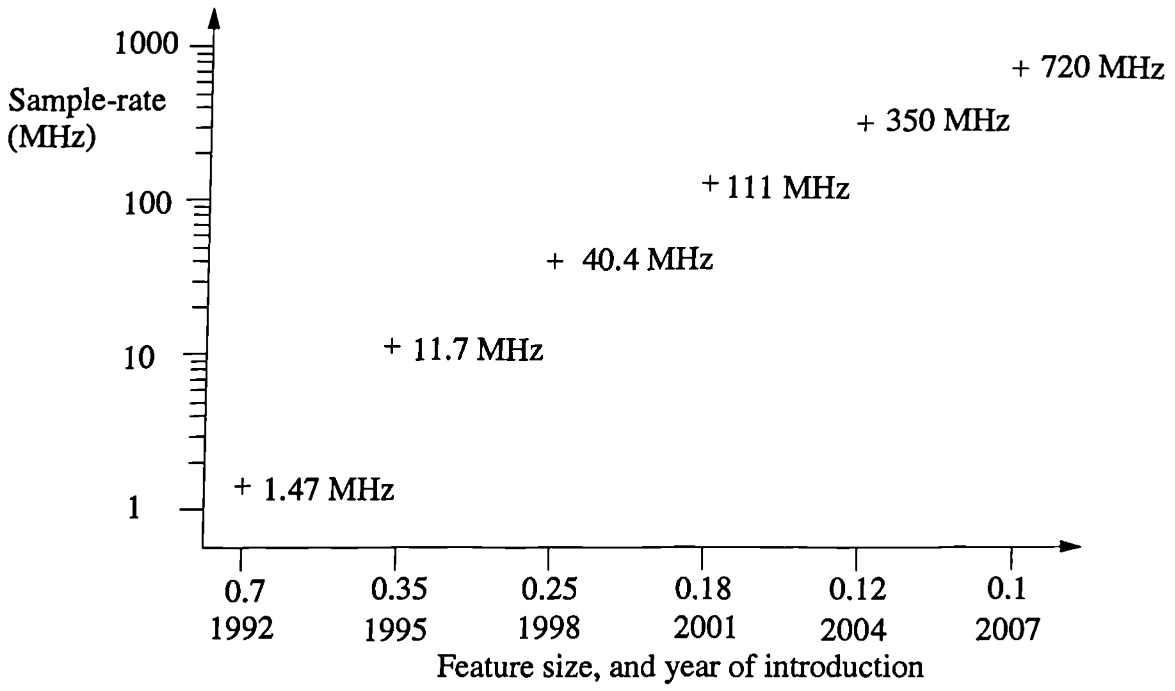


Figure 8.12 Estimated performance of an adaptive beamforming ASIC

Using current 0.35μm technology, it would be possible to implement the beamformer at a sample-rate of 11.7MHz. The power consumption would be about 8W, which is acceptable if fan cooling were used.

8.9 Discussion

It has been demonstrated in this chapter how methodical techniques can be used to obtain a broad range of VLSI architectures for implementing a fixed-size QR-array with a reduced number of processors. In particular, a new systolic array has been developed. This enables efficient implementation of the QR-algorithm using optimised processors designed to implement either boundary or internal cells.

There is currently a need for commercial tools to automate, or at least assist with, the array processor architecture design process.

The automation of the VLSI design route has improved enormously over the last few years, and the cell designs produced in this thesis were achieved in 1 week (once the VHDL operators had been coded in VHDL). This suggests that very rapid realisation of VLSI DSP designs is now possible, although it should be noted that other issues such as testability and verification of the design have not been addressed.

A CORDIC approach for implementing Givens rotations was found to require more than 3 times the area of the SGR algorithm implemented using standard arithmetic operators and incurred a latency between 11 and 25 times greater. This can be attributed to the greater scope for optimisation available when the rotation is constructed from a number of simple operators.

One slightly surprising result, was how inefficient the solution based upon an array of programmable DSP chips was when compared to an ASIC approach. For an 81 input array the ASIC solution was pin-count limited, yet it achieved more than a factor of 4 improvement over an array of 45 DSP chips. If a 16-input array were considered, where the ASIC solution is silicon area limited, then the figure would be 35. That is, the ASIC is a factor of 1575 faster than each GP DSP chip.

Why is this figure so large? One reason for this is that the DSP chips spend a high proportion of time communicating data in a multiprocessor configuration. This is particularly true of the DSP32C, although more recent processors have better support for this[SHAR95]. Secondly, the DSP chip has only one arithmetic unit, whereas 74 could be integrated on a single ASIC. Thirdly, the arithmetic unit of the DSP chip is of fixed size and may be considerably larger than required, whereas the wordlength of the operations in an ASIC may be tailored to save area. Multi-processor programmable DSP chips are possible, but the only one currently available commercially is difficult to program, yet has only 5 processors (4 of which are fixed-point).

The development time of a DSP array solution is high, and comparable to that of an ASIC. The one-off implementation costs are also similar. Therefore, in circumstances where the algorithm is application specific and well defined, there is a strong argument for the adoption of an ASIC solution.

Chapter 9 Conclusions

9.1 Overview

The VLSI design of an array processor for recursive least squares based on QR-decomposition using Givens rotations has been investigated. The emphasis has been on achieving high sample-rate operation, so particular interest has been paid to the complexity of the recursive loops in the algorithm and the delay of the operators used. In the search of the optimum design, the aspects of arithmetic, algorithm and architecture have been investigated in a way which accounts for the limitations and benefits of VLSI technology.

The main contributions of the work were:

- It has been shown that an ASIC design of the QR-algorithm based on floating-point arithmetic requires less silicon area than those based on fixed-point arithmetic.
- A novel algorithm has been presented, based on a combination of fixed- and floating-point arithmetic, which offers both higher sample-rate and reduced area requirements over previous Givens rotation algorithms when used in an ASIC implementation of the QR-algorithm.
- A novel linear array architecture has been proposed for the QR-array, which offers local interconnect and thereby greatly simplifies both ASIC and DSP parallel processor implementations.
- Novel multiplier-adder structures have been presented which uses minimally redundant radix-4 arithmetic to achieve low-latency multiplication with reduced area over previous techniques.

These are discussed in more detail below.

9.2 Low-Latency Arithmetic

Architectures for msdf multiplier-adders have been presented in the thesis as a method for obtaining low-latency for applications such as the QR-algorithm and IIR filtering. An architecture is presented which offers a 25% reduction in area over previous designs by a judicious use of redundancy whilst maintaining low-latency. The msdf approach is attractive as it uses an array multiplier with a very regular architecture, yet replaces the linear dependence of delay on wordlength with one which is fixed and relatively low. Indeed, the very regular architecture offered by msdf multipliers has enabled the automatic compilation of their layouts for any wordlength using a simple bit-slice approach.

Over the course of this research there have been rapid developments in the capabilities of integrated circuit design tools, and this means that it is now also possible to implement irregular circuits in an efficient and rapid manner. Therefore, low-latency multipliers based on adder trees have also been considered in the thesis, as they offer a delay which is only logarithmically dependent upon wordlength. It has been shown how these may be used to obtain a recursive multiply-add operation, and for medium wordlengths (i.e. 16 to 32-bits) a Wallace-tree multiplier will offer significant reductions in latency and area over an msdf approach without the inconvenience of a skewed data format.

In a study of high-throughput, low-latency dividers it has been shown that the modified SRT divider offers significantly higher levels of throughput than the standard SRT algorithm. It also compares well on speed with a version of a speculative divider which has been presented in the thesis in a modified form to provide acceptable area requirements. Although, it has higher latency, the modified SRT algorithm offers advantages in system design, as there is greater flexibility in the level of pipelining which may be applied and so greater control over throughput.

Fully parallel architectures for a convergence divider and reciprocal operator have been presented, which aim to offer low-latency with high-throughput. In this parallel form, it is possible to optimise multiplier wordlengths in order to obtain realistic area requirements. A design based on non-redundant arithmetic was produced and resulted in a similar level of latency to the modified SRT approach. A redundant implementation should significantly reduce the latency, and the techniques presented in this thesis can be used to obtain this with only a 50% increase in area rather than 100%.

9.3 Givens Rotation Algorithm

An important achievement has been the development of an enhanced version of the Squared Givens Rotation (SGR) algorithm as this offers extremely high sample-rate operation which is considerably better than any other algorithm examined, yet with reduced area and good numerical performance. Using a 0.35μm CMOS standard cell process, a sample-rate of 136 MHz is considered achievable.

The Givens rotation presented an interesting example of a recursive algorithm which was more complex than IIR filtering. In particular, there were a wide range of algorithm variants which exhibited quite distinct characteristics, and there was a requirement for more than just multiply-accumulate operations. In particular, fixed- and floating-point variants were possible, and although the simplicity of fixed-point arithmetic is generally sought in DSP implementations, it was found that a floating-point algorithm offered significantly reduced area and high sample-rate. The main reason for this was that the additional dynamic range provided by the floating-point format has enabled a transformation of the algorithm to be used which avoids the square-root operation, halves the number of multiplies and simplifies the recursive loop to just an add operation. The latter enables the wordlength of the loop variable to be increased at little cost in hardware, with significant reductions in the accumulated numerical error. Consequently, significant reductions can be made in wordlengths used elsewhere - particularly when large window lengths are required when error accumulation is at its worst. This provides large savings in the area occupied by the multipliers and dividers.

The extra cost of floating-point operators over fixed-point ones is only significant in the implementation of an adder, where the area increases by approximately a factor of 10 and the latency by a factor of 3. However, in the case of the Givens rotation, this increase in area is more than compensated for by reductions in wordlength and the number of operations used. The latency of the floating-point addition is avoided in the feedback loop by using a fixed-point representation for the loop variable. Also a signed-binary redundant representation has been used to reduce the addition time and make it independent of wordlength. Consequently, very high sample-rate operation is feasible which is independent of wordlength.

It is clear from this work that low-latency operators are expensive to implement in terms of their area-time product. For highly parallel implementations, maximum computation rates are obtained from a finite area of silicon by using operators offering the lowest area-delay product. If power is an issue, as for portable equipment and highly integrated systems, the area-speed-power product should be minimised. As power is approximately a function of area, then low-latency operators become even more expensive. Therefore, they should only be used for critical operations where their expense can be justified. It is interesting to note that in the QR-algorithm, a very high sample-rate design was achieved through the combination of an algorithm transformation and the use of only a redundant adder. This emphasises the need to consider the design process as a whole in order to obtain optimum results.

9.4 Application Specific Array Processor

The enhanced SGR algorithm provides exciting opportunities for realising very-high sample rate adaptive filters based on the QR-algorithm. It should be possible to use the SGR algorithm in a QR-based lattice filter. The current levels of integration and the efficiency of the algorithm are sufficiently high that the full lattice operating at maximum sample-rate could be implemented on a single ASIC (providing the length of the lattice were not too great). It should also be possible to implement realistic adaptive beamforming solutions using a complex arithmetic implementation of the QR-array and a single ASIC. For example, it has been estimated that an adaptive beamformer with 16 auxiliary and 1 primary input could be implemented with a sample-rate of 3.125 MHz using a single integrated circuit and a 0.35μm standard cell process. This would represent a sustained performance of 12,500 MFLOPS.

The enhanced-SGR algorithm has been compared with a CORDIC approach and shown to benefit by a factor of 3 in area and over 11 in sample-rate. Also when compared with a recent implementation of the QR-algorithm on a parallel array of GP DSP chips, it was estimated that a single application specific chip could offer as much as 1,500 times the level of computation as that obtained from a single GP DSP chip.

The automated design route used to obtain layout for SGR processor design was very rapid, once the arithmetic operators had been coded in VHDL. The layout was also very compact for the size of operators used, which suggests that obtaining an ASIC is no longer as difficult as it used to be. It is clear from the performance figures presented above that an application specific approach can offer levels of computation *several orders* of magnitude greater than can be achieved using programmable DSP chips, even when a standard cell approach is adopted. Therefore, there is a very strong argument for pursuing ASIC solutions for more complex DSP algorithms, and for developing the tools to enable the design process from algorithm to architecture to be performed as rapidly and as efficiently as possible.

9.5 Future Work

The research has identified a number of areas in which further work may be productive. These are summarised below:

The comparison of the msdf and tree-based multipliers suggested that multipliers offering minimum latency will depend upon the application - in particular, whether saturation is required and what wordlength is to be used. Further characterisation of the multipliers would be worthwhile to allow a more direct comparison to be made for particular applications. Specifically, it would be worthwhile obtaining performance figures for the msdf multiplier-adders pipelined for single-cycle latency.

Recent work by Oklobziga *et al.* [Oklo96] has shown that the delay of an adder-tree can be improved by careful construction and description of the tree in VHDL. This has been achieved by performing a high level optimisation on the tree using detailed knowledge of the full-adder building block. The synthesis and optimisation results obtained from Synopsys for the optimised tree should be compared with those obtained from a simple tree description to see if the approach provides any advantage over the current synthesis approach.

The convergence divider was implemented using non-redundant multipliers, which gave an area-efficient solution. The implementation of the divider and reciprocal circuit using redundant arithmetic is possible, and should be investigated. Redundant tree multipliers suitable for this purpose have been presented in this thesis and found to reduce the area required by 25%. Even so, the area will be substantially greater than the modified SRT divider (by approx. 50-100%), but the latency could be as much as halved.

The reciprocal LUTs synthesised using combinational logic in Chapter 4 were very fast and surprisingly small. The use of table based approaches (i.e. interpolation) should be further investigated for implementing low precision reciprocals and other functions.

The floating-point adder was designed to minimise latency in the recursive loop of the SGR algorithm. The design of a more area-time efficient floating-point adder may be possible by allowing a latency of 4 or 5. In many DSP implementations, the resulting sample-rate will be sufficient.

Much of the research on computer arithmetic is performed with microprocessor arithmetic units in mind, in which case the design objective is invariably high wordlength at low latency. This is not necessarily the requirement of DSP systems. Therefore, research on arithmetic techniques targeted at implementing operators with low area-time products for low and medium wordlengths is worthwhile.

The high sample-rate variant of the SGR algorithm currently achieves fixed-point number ranges for the loop variable by assuming that the input is in a fixed-point number range and applying scaling to the input appropriate to the value of forget-factor used. Simple normalisation schemes based on shifters have been suggested in the thesis which would allow floating-point inputs to be used, and would not be costly in time and area to implement. They should be pursued further.

The enhanced SGR algorithm may also be applied in the implementation of a QR based lattice filter[Prou91]. In the single channel case, the lattice generally requires fewer boundary and internal cells than the array, and is more likely to be applied in high sample-rate applications. Architectures for implementation of a lattice will be simpler, as it consists of only a linear array of boundary and internal cells. Consequently, a more flexible VLSI implementation could be achieved and would be worth investigating. The ratio of boundary to internal cells is 1 to 1, so there may be scope for developing a single optimised cell. Numerical simulations would need to be performed to confirm that the same numerical benefits can be obtained from the SGR algorithm.

Only cell designs for the SGR algorithm based on real arithmetic have been considered in this thesis. Complex arithmetic versions are required in adaptive beamforming applications. They could be achieved by modifying the SGR algorithm to accept complex quantities for the x and r terms. It is anticipated that the complex implementation would benefit from the improved numerical performance of the enhanced SGR algorithm, but this would need to be confirmed by simulation. One requirement of complex cells would be a complex multiplier. The latency of this device is not critical, as in the SGR cells it is not required within the feedback loop, but area-time efficient designs would be worth investigating.

Simple estimates, presented in the previous chapter, indicate that it is possible to solve a certain radar beamforming problem using a single application specific chip. The implementation of this chip would be an exciting objective. Obtaining measurements of real power consumption would be very useful, as it may be quite high. High power consumption is less likely to be a problem in military applications, but in many commercial ones it will be much less acceptable. Therefore, ways of reducing power consumption should be investigated. To achieve this objective, ways must be sought for minimising the number of logic transitions. Signed representations can be useful as they do not require large numbers of bit transitions when making small changes in value. Whereas, with a 2's complement representation it is necessary for all the bits to change when going from negative to positive number, even when the number is small.

Many of the basic arithmetic operators now exist in a parameterised VHDL form. This library could be exploited for the rapid implementation of range of application specific DSP problems. A wider study of relevant DSP algorithms and their architectures would be very worthwhile to see how many can currently be satisfactorily implemented in this way. It is likely that such a study would help define the requirements of software tools for rapid implementation of high performance DSP ASICS.

Chapter 10 References

[Anda94] A. A. Anda and H. Park, "Fast Plane Rotations with Dynamic Scaling", *SIAM J. on Matrix Analysis and Applications*, Vol. 15, No. 1, pp. 162-174, 1994.

[Ande67] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit", *IBM J.*, Jan. 1967.

[Aviz61] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic", *IRE Trans. on Compt.*, Vol. EC-10, pp. 389-400, 1961.

[Barl87] J. L. Barlow and I. F. Ipsen, "Scaled Givens rotations for the solution of linear least squares problems on systolic arrays", *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 5, pp. 716-733, Sept. 1987.

[Bedr62] O. J. Bedrij, "Carry-Select Adders", *IRE Trans. Electronic Computers*, Vol. EC-11, No. 3, pp. 340-346, 1962.

[Boot51] A. D. Booth, "A Signed Binary Multiplication Technique", *Qt. J. Mech. Appl. Math.*, Vol. 4, Part 2, 1951.

[Brac89a] R. H. Brackert Jr., M. D. Ercegovac and A. N. Willson Jr., "Design of an On-Line Multiply-Add Module for Recursive Digital Filters", *Proc. of the 9th Symp. on Compt. Arith.*, pp. 34-41, 1989.

[Brac89b] R. H. Brackert Jr., A. N. Willson, and M. D. Ercegovac, "A High-Speed Digital Filter Uisng On-Line Arithmetic", *Proc. of IEEE Int. Symp. on Circuits and Systems*, pp. 1552-1555, 1989.

[Brig93] W. S. Briggs and D. W. Matula, "A 17x69 Bit Multiply and Add Unit with Redundant Binary Feedback and Single Cycle Latency", *Proc. of the 11th Symp.*

on Compt. Arith., pp. 163-170, 1993.

[Bu90] J. Bu, E. Deprettere, and P. Dewilde, "A Design Methodology for Fixed-Size Systolic Arrays", *Proc. of the Int. Conf. of Application Specific Array Processors*, pp. 591-602, 1990.

[Burg91] N. Burgess, "A Fast Division Algorithm for VLSI", *IEEE Intl. Conf. on Compt. Design*, pp. 560-563, 1991.

[Burg95] N. Burgess and T. Williams, "Choices of Operand Truncation in the SRT Algorithm", *IEEE Trans. on Compt.*, Vol. 44, No. 7, pp. 933-938, 1995.

[Cart90] T. M. Carter and J. E. Robertson, "The Set Theory of Arithmetic Decomposition", *IEEE Trans. on Compt.*, Vol. 39, No. 8, pp. 993-1005, 1990.

[Coff96] A. S. Coffey, M. Johnson, and R. Jones, "Nonlinear Dynamical Systems Analyser", *Proc. SPIE 2296*, pp. 687-699, 1996.

[Dadd65] L. Dadda, "Some Schemes for Parallel Multipliers", *Alta Frequenza*, Vol. 34, pp. 349-356, March 1965.

[Darl90] M. Darley et al., "The TMS390C602A Floating-Point Coprocessor for Sparc Systems", *IEEE Micro*, Vol. 40, No. 3, pp. 36-47, 1990.

[DasS94] D. DasSarma and D. W. Matula, "Measuring the Accuracy of ROM Reciprocal Tables", *IEEE Trans. on Compt.*, Vol. 43, No. 8, pp. 932-940, 1994.

[DasS95] D. Das Sarma and D. W. Matula, "Faithful Bipartite ROM Reciprocal Tables", *Proc. of the IEEE 12th Symp. on Compt. Arith.*, pp. 17-28, 1995.

[Deit93a] C. R. Deitrich, "Computationally Efficient Cholesky Factorization of a Covariance-Matrix with Block Toepliz Structure", J. Statistical Computation and

Simulation, Vol. 45, No. 3-4, pp. 203-218, 1993.

[Deit93b] C. R. Deitrich, "Computationally Efficient Generation of Gaussian Conditional Simulations over Regular Sample Grids", *Mathematical Geology*, Vol. 25, No. 4, pp. 439-451, 1993.

[Deny85] P. Denyer and R. Renshaw, *VLSI Signal Processing: A Bit Serial Approach*, Addison-Wesley, IBSN 0-201-14404-2, 1985.

[Dini95] P. S. R. Diniz and M. G. Siqueira, "Fixed-Point Error Analysis of the QR-Recursive Least Squares Algorithm", *IEEE Trans. on Circuits and Systems - II: Analog and Digital Signal Processing*, Vol. 42, No. 5, 1995.

[Döhl91] R. Döhler, "Squared Givens Rotations", *IMA J. of Numerical Analysis*, Vol. 11, pp. 1-5, 1991.

[Erce77] M. D. Ercegovac, "A General Hardware-Oreiented Method for Evaluation of Functions and Computations in a Digital Computer", *IEEE Trans. on Compt.*, Vol. C-26, No. 7, pp. 667-680, 1977.

[Erce87] M. D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations", *IEEE Trans. on Compt.*, Vol. C-36, No. 7, pp. 895-897, 1987.

[Erce88] M. D. Ercegovac, "On-Line Scheme for Computing Rotation Factors", *J. of Parallel and Distributed Computing*, 5, pp. 209-227, 1988.

[Erce89a] M. D. Ercegovac and T. Lang, "On-the-Fly Rounding for Division and Square Root", *Proc. of the 9th Symp. on Compt. Arith.*, pp. 169-175, 1989.

[Erce89b] M. D. Ercegovac and T. Lang, "Fast Radix-2 Division With Quotient-Digit Prediction", *J. of VLSI Signal Processing*, Vol.1, No. 3, pp.169-180, 1989.

[Erce90] M. D. Ercegovac and T. Lang, "Simple Radix-4 Division with Operands Scaling", *IEEE Trans. on Compt.*, Vol. C-39, No. 9, pp1204-1207, 1990.

[Flyn70] M. J. Flynn, "On Division by Functional Iteration", *IEEE Trans. on Compt., Vol. C-19*, No. 8, pp. 702-706, 1970.

[Flyn95] M. J. Flynn, K. Nowka, G. Bewick, E. Schwarz and N. Quach, "The SNAP Project: Towards Sub-Nanosecond Arithmetic", *Proc. of the IEEE 12th Symp. on Compt. Arith.*, pp.75-82, 1995.

[Fran94] E. N. Frantzeskakis and K. J. R. Liu, "A Class of Square Root and Division Free Algorithms and Architectures for QRD-Based Adaptive Signal Processing", *IEEE Trans. on Signal Processing*, Vol. 42, No. 9, 1994.

[Gent73] W. M. Gentleman, "Least-Squares Computations by Givens transformations without square roots", *J. Inst. Math. Its Applics.*, Vol. 12, pp. 329-336, 1973.

[Gent81] W. M. Gentleman and H. T. Kung, "Matrix triangularization by systolic arrays", *Proc. SPIE 298, Real-Time Signal Processing IV*, pp. 19-26, 1981.

[Golu89] G. H. Golub and C. F. Van Loan, "Matrix Computations", 2nd Ed., John Hopkins, ISBN-08018-3739-1, 1989.

[Götz95] J. Götze and G. J. Hekstra, "An algorithm and architecture based on orthonormal m-rotations for computing the symmetric EVD", *Integration, the VLSI J.*, 20, pp. 21-39, 1995.

[Give58] W.Givens, "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form", *J. Soc. Indust. Appl. Math.*, Vol. 6, No. 1, pp. 26-50, March 1958.

[Götz91] J. Götze and U. Schwiegelshohn, "A square root and division free Givens rotation for solving least squares problems on systolic arrays", *SIAM J. Sci. Stat. Compt.*, Vol. 12, No. 4, pp. 800-807, 1991.

[Hami95a] R. Hamill, "VLSI Algorithms and Architectures for DSP Arithmetic Computations", *PhD Thesis*, The Queen's University of Belfast, 1995.

[Hami95b] R. Hamill, R. L. Walke and J. V. McCanny, "Constant Scale Factor, On-Line CORDIC Algorithm in the Circular Coordinate System", *VLSI Signal Processing, VIII*, ISBN 0-7803-2612-1, pp. 562-571, 1995.

[Hamm74] S. Hammarling, "A note on modifications to the Givens plane rotation", *J. Inst. Maths Applics.*, Vol. 13, pp. 215-218, 1974.

[Hayk91] S. Haykin, *Adaptive Filter Theory*, Second Edition, Prentice Hall, ISBN 0-13-013236-5, 1991

[Hous58] A.S. Householder, "Unitary triangularization of a nonsymmetric matrix", *J. ACM*, Vol. 5, pp. 339-342, 1958.

[Hsie93] S. F. Hsieh, K. J. R. Liu, and K. Yao, "A Unified Approach for QRD-Based Recursive Least-Squares Estimation Without Square Roots", *IEEE Trans. on Signal Processing*, Vol. 41, No. 3, pp. 1405-1409, March 1993.

[Hwan79] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley and Sons, 1979.

[IEEE85] "IEEE Standard for Binary Floating-Point Arithmetic", *IEEE Std. 754-1985*, IEEE, New York.

[Irwi87] M. J. Irwin, R. M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial", *IEEE Compt.*, pp. 61-73, 1987.

[Ito95] M. Ito, N. Takagi, and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root", *Proc. of the 12th Symp. on Compt. Arith.*, pp. 2-9, 1995.

[Kopp93] B. Koppenhofer, "A Novel Architecture for a Decision-Feedback Equalizer Using Extended Signed-Digit Feedback", *Proc. of the Int. Conf. of Application Specific Array Processors*, pp. 490-501, 1993.

[Know89a] S. C. Knowles, J. G. McWhirter, R. F. Woods, and J. V. McCanny, "Bit Level Systolic Architectures for High Performance IIR Filtering", *J. of VLSI Signal Processing*, Vol. 1, No. 1, pp. 9-24, 1989.

[Know89b] S. Knowles, J. G. McWhirter, "An Improved Bit-Level Systolic Architecture for IIR Filtering", *Systolic Array Processors*, Eds. J. V. McCanny, J. G. McWhirter and E. Swartzlander, Prentice Hall, pp. 205-214, 1989.

[Know91] S. C. Knowles, "Arithmetic Processor Design for the T9000 Transputer", *SPIE Vol. 1566 Advanced Signal Processing Algorithms, Architectures and Implementations II*, pp. 230-243, 1991.

[Kung78] S. Y. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)", In *Sparse Matrix Symp.*, pp. 256-282, SIAM, 1978.

[Kung87] S. Y. Kung, "VLSI Array Processors", in *Systolic Arrays*, Eds. W. Moore, A. McCabe and R. Urquhart, Adam Hilger, IBSN 0-85274-826-4, 1987.

[Kung88] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, IBSN 0-13-942749-X, 1988.

[Kuni87] S. Kuninobu, T. Nishyama, T. Tanguchi, and N. Takagi, "Design of High Speed MOS Multiplier and Divider using Redundant Binary Representation", *Proc. of the IEEE 8th Symp. on Compt. Arith.*, pp. 80-86, 1987.

[Lapo90] M. Lapointe, P. Fortier and H. T. Huynh, "A New Faster and Simpler Systolic Structure for IIR Filters", *Proc. of the IEEE Int. Conf. on Circuits and Systems*, pp. 1227-1230, 1990.

[Leis83] C.E. Leiserson and F. Saxe, "Optimizing Synchronous Systems", *J. of VLSI and Compt. Systems*, Vol. 1, No. 1, pp. 41-67, 1983.

[Lore95] F. Lorenzelli and K. Yao, "A Linear Systolic Array for Recursive Least Squares", *IEEE Trans. on Signal Processing*, Vol. 43, No. 12, 1995.

[Lyu95] C. N. Lyu and D. W. Matula, "Redundant Binary Booth Recoding", *Proc. of the 12th Symp. on Compt. Arith.*, pp. 50-57, 1995.

[Maje85] S. Majerski, "Square Rooting Algorithms for High Speed Digital Circuits", *IEEE Trans. in Compt.*, Vol. C-34, pp. 724-733, 1985.

[McGo95] B. P. McGovern, R. F. Woods, and C. McAllister, "Optimised multiply/accumulate architecture for very high throughput rate digital filters", *Electronics Letters*, Vol. 31, No. 14, pp. 1135-1136, 1995.

[McQu92] S. E. McQuillan and J. V. McCanny, "Algorithms and Architectures for High Performance Recursive Filtering", *Proc. of the Int. Conf. on Application Specific Array Processors*, pp. 230-244, 1992.

[McQu92] S. E. McQuillan, "Algorithms and Architectures for High Performance Processors", *Ph.D. Thesis*, The Queen's University of Belfast, 1992.

[McQu94a] S. E. McQuillan, "Fast VLSI Algorithms for Division and Square Root", *J. of VLSI Signal Processing*, Vol. 8, pp. 151-168, 1994.

[McQu94b] S. E. McQuillan, Y. Hu, "Algorithms and Architectures for Most Significant Digit First Arithmetic", Integrated Silicon Systems Ltd, Belfast, 1995

[McQu95] S. McQuillan and J. V. McCanny, "A Systematic Methodology for the Design of High Performance Recursive Digital Filters", *IEEE Trans. on Compt.*, Vol. 44, No. 8, pp. 971-982, Aug. 1995.

[McWh83] J . G. McWhirter, "Recursive least-squares minimization using a systolic array", *Proc. SPIE 431, Real-Time Signal Processing VI*, pp. 105-112, 1983.

[McWh95] J. G. McWhirter, R. L. Walke and J. Kadlec, "Normalised Givens Rotations for Recursive Least Squares Processing", *VLSI Signal Processing, VIII*, ISBN 0-7803-2612-1, pp. 323-332, 1995.

[Metr63] N. Metropolis and R. L. Ashenhurst, "Basic Operations in an Unnormalised Arithmetic System, *IEEE Trans. on Electronic Compt.*, Vol. EC-12, pp. 896-904, 1963.

[Megs92] G. M. Megson, *An Introduction to Systolic Algorithm Design*, Clarendon Press, Oxford, IBSN 0-19-853813-8, 1992.

[Mold86] D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays", *IEEE Trans. on Compt.*, Vol. C-35, No. 1, 1986.

[Mont91] P. Montuschi and L. Ciminiera, "Simple Radix 2 Division and Square Root with Skipping of Some Division Steps", *Proc. of the IEEE 10th Symp. on Compt. Arith.*, pp. 202-209, 1991.

[Mont93] P. Montuschi and L. Ciminiera, "$n \times n$ Carry-Save Multipliers without Final Addition", *Proc. of the IEEE 11th Symp. on Compt. Arith.*, pp. 54-61, 1993.

[Monz80] R. A. Monzingo and T. W. Miller, *Introduction to Adaptive Arrays*, John Wiley and Sons, ISBN 0-471-05744-4, 1980.

[Nat94] "The National Technology Roadmap for Semiconductors", Semiconductor Industry Association, 1994.

[Ober94] S. F. Oberman and M. J. Flynn, "Design Issues in Floating-Point Division", *Technical Report*, CSL-TR-94-647, Stanford University, 1994.

[Ober95] S. F. Oberman and M. J. Flynn, "An Analysis of Division Algorithms and Implementations", *Technical Report*, CSL-TR-95-675, Stanford University, 1995.

[Owen83] R. M. Owens, "Techniques to Reduce the Inherent Limitations of Fully Digit On-Line Arithmetic", *IEEE Trans. on Compt.*, Vol. C-32, No. 4, 1983.

[Oklo96] V. G. Oklobzija, D. Villeger, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach", *IEEE Trans. on Compt.*, Vol. 45, No. 3, pp. 294-306, 1996.

[Parh90] B. Parhami, "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations", *IEEE Trans. on Compt.*, Vol. 39, No. 1, pp. 89-98, 1990.

[Peng87] V. Peng, S. Sanudrala, M. Gavrielov, "On the Implementation of Shifters, Multipliers and Dividers in VLSI Floating Point Units", *Proc. of the IEEE 8th Symp. on Compt. Arith.*, pp. 95-102, May 1987.

[Prab95] J. A. Prabhu and G. B. Zyner, "167MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages", *Proc. of the 12th Symp. on Compt. Arith.*, pp.

155-162, 1995.

[Priv90] G. Privat, "A Novel Class of Serial-Parallel Redundant Signed Digit Multipliers", *Int. Symp. on Circuits and Systems*, pp. 2116-2119, 1990.

[Prou91] I. K. Proudler, J. G. McWhirter and T. J. Shepherd, "Computationally Efficient, QR Decomposition Approach to Least Squares Adaptive Filtering", IEE Proceedings, Vol. 138, Pt. F, No. 4, pp. 341-353, 1991.

[Rade92] C. M. Rader, "MUSE: A Systolic Array for Adaptive Nulling with 64 Degrees of Freedom using Givens Transformations and Wafer Scale Integration", *Proc. of the Int. Conf. of Application Specific Array Processors*, pp. 277-291, 1992.

[Rade96] C. M. Rader, "VLSI Systolic Arrays for Adaptive Nulling", *IEEE Signal Processing Magazine*, Vol. 13, No. 4, pp. 29-49, 1996.

[Reed74] I. S. Reed, J. D. Mallett, and L. E. Brennan, "Rapid Convergence Rate in Adaptive Arrays", *IEEE Trans. on Aerospace Electronic Systems*, Vol. AES-10, pp. 853-863, 1974.

[Robe58] J. E. Robertson, "A New Class of Division Methods", *IRE Trans. on Electronic Compt.*, Vol. EC-7, pp. 218-222, 1958.

[Rodr81] M. R. D. Rodrigues, J. H. P. Zurawski, and J. B. Gosling, "Hardware Evaluation of Mathematical Functions", *IEE Proc.*, Vol. 128, Pt. E, No. 4, pp. 155-164, 1981.

[Rubi75] L. P. Rubinfield, "A Proof of the Modified Booth's Algorithm for Multiplication", *IEEE Trans. on Compt.*, pp. 1014-1015, Oct. 1975.

[SHAR95] ADSP-2106x SHARC User's Manual, Analog Devices, Inc., First Edition, 1995.

[Shep93] T. J. Shepherd and J. G. McWhirter, “Systolic Adaptive Beamforming” in Chapter 5: *Array Processing*, Eds. S. Haykin, J. Litva and T. J. Shepherd, Springer-Verlag, ISBN 3-540-55224-3, pp. 153-243, 1993.

[Sten77] W. J. Stenzel, W. J. Kubitz, and G. H. Garcia, “A Compact High-Speed Parallel Multiplication Scheme”, *IEEE Trans. on Compt.*, Vol. C-26, pp. 948-957, 1977.

[Svob63] A. Svoboda, “An Algorithm for Division”, *Information Processing Machines*, No. 9, pp. 25-34, 1963.

[Tayl85] G. S. Taylor, “Radix 16 SRT Dividers with Overlapped Quotient Selection Stages”, *Proc. of the 7th Symp. on Compt. Arith.*, pp. 95-101, 1985.

[TMS95] TMS320C8x System Level Synopsys, Texas Instruments, Document SPRU113B, September 1995.

[Train95] D. W. Trainor, R. F. Woods and J. V. McCanny, “Architectural Synthesis on an Image Processing Algorithm Using IRIS”, , *VLSI Signal Processing, VIII*, ISBN 0-7803-2612-1, pp. 167-176, 1995.

[Triv77] K. S. Trivedi and M. D. Ercegovac, “On-Line Algorithms for Division and Multiplication”, *IEEE Trans. on Compt.* Vol. C-26, No. 7, pp. 681-687, 1977.

[Vold59] J. Volder, “The CORDIC Trigonometric Computing Technique”, *IRE Trans. Electron. Comput.*, Vol. EC-8, pp. 330-334, 1959.

[Walk93] R. L. Walke and R. A. Evans, “A Minimally Redundant Radix-4 Systolic Array for High Performance IIR Filtering”, *VLSI Signal Processing, VI*, ISBN 0-7803-0996-0, pp. 168-178, 1993.

[Wall64] C. S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Trans. on Electronic Compt.*, Vol. EC-13, pp. 14-17, 1964.

[Walt71] J. S. Walther, "A unified algorithm for elementary functions", *Proc. Spring Joint Compt. Conf.*, pp. 379-385, 1971.

[Ward86a] C. R. Ward, P. J. Hargrave, and J. G. McWhirter, "A Novel Algorithm and Architecture for Adaptive Digital Beamforming", *IEEE Trans. on Antennas and Propagation*, Vol. AP-34, No. 3, pp. 338-346, 1986.

[Ward86b] C. R. Ward and E. B. Davie, "The Application and Development of Wavefront Array Processors for Advanced Front-end Signal Processing Systems", in *Systolic Arrays*, W. Moore, A. P. H. McCabe, and R. B. Urqhart (eds.), Adam Hilger, Bristol, U.K., pp. 295-302, 1986.

[Wata81] O. Watanuki, M. D. Ercegovac, "Floating-Point On-Line Arithmetic: Algorithms", *Proc. of the 5th Symp. on Compt. Arith.*, pp. 81-86, 1981.

[Wein56] A. Weinberger and J. L. Smith, "A On-Microsecond Adder Using One-Megacycle Circuitry", *IRE Trans. Electronic Computers*, Vol. EC-5, pp. 65-73, 1956.

[Wein81] A. Weinberger, "4:2 Carry-Save Adder Module", *IBM Technical Disclosure Bull.*, Vol. 23, Jan, 1981.

[West93] N. Weste and K. Eshraghian, "Principles of CMOS VLSI Design", 2nd Ed. Addison-Wesley, IBSN 0-201-53376-6, 1993.

[Widr60] B. Widrow and M. E. Hoff, Jr, "Adaptive Switching Circuits", *IRE WESCON Conv. Rec.*, Pt. 4, pp. 96-104, 1960.

[Wilk63] J. H. Wilkinson, "Rounding Errors in Algebraic Processes", *Notes on Applied*

Science No. 32, HMSO, 1963.

[Wood88] R. F. Woods, S. C. Knowles, J. V. McCanny, and J. G. McWhirter, "Systolic IIR Filters with Bit-Level Pipelining", *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 2072-2075, 1988.

[Wood95] R. F. Woods, G. Floyd, K. Wood, R. Evans, and J. V. McCanny, "Programmable high-performance IIR filter chip", *IEE Proc. of Circuits, Devices and Systems*, Vol. 142, No. 3, pp. 179-185, June 1995.

[Wood91] R. F. Woods, O. C. McNally, and S. E. McQuillan, "Saturation Circuitry for Redundant Number System Based IIR Filters", *Electronics Letters*, Vol. 27, No. 21, pp. 1961-1963, Oct. 1991.

[Yu95] R. K. Yu, "167MHz Radix-4 Floating-Point Multiplier", *Proc. of the IEEE 12th Symp. on Compt. Arith.*, pp. 149-154, 1995.

[Zehe92] E. Zehendner, "Efficient Implementation of Regular Parallel Adders for Binary Signed Digit Number Representations", *Microprocessing and Microprogramming*, Vol. 35, pp. 319-326, 1992.

Appendix A Analysis of Msdf Multiplier-Adders

To provide an algorithmic basis from which to examine alternatives to the radix-2 number system, the analysis of msdf multiplier-adders by McQuillan[McQu95] has been extended from radix-2 to the general radix-r case, and is presented below:

The aim of this analysis is to derive an iterative algorithm to compute the multiply-and-add operation:

$$M = XY + A \tag{A.1}$$

The result M is computed, most significant digit first, with Y and A supplied in a digit-by-digit, msd-first manner. X is known at the start of the calculation and is presented in 2's complement parallel form.

A.1 General Radix Analysis

For the general radix case, the partial multiplier Y_j and partial addend A_j at the j^th iteration are given by

$$\begin{aligned} Y_j &= Y_{j-1} + y_j r^{-j} = \sum_{i=1}^{j} y_i r^{-i} \qquad y_i \in \{\kappa_{min}, \ldots, \kappa_{max}\} \\ A_j &= A_{j-1} + a_j r^{-j} = \sum_{i=1}^{j} a_i r^{-i} \qquad a_i \in \{\kappa_{min}, \ldots, \kappa_{max}\} \end{aligned} \tag{A.2}$$

Where κ_{min} and κ_{max} represent the two extreme values of the digit range. For a symmetrical digit set $-\kappa_{min} = \kappa_{max}$.

On each iteration, a digit is added to the partial terms Y_j and A_j, starting with the most significant digit and finishing with the least significant one.

The final result M is also compiled in this manner, but is delayed by the time required to perform the computation. This latency shall be denoted δ and be defined here in terms of the number of iterations between a multiplier digit entering the computation and a result digit of the same significance being computed. Hence on the jth iteration, only $M_{j-\delta}$ is available, where

$$M_{j-\delta} = M_{j-\delta-1} + m_{j-\delta}r^{-j+\delta} = \sum_{i=1}^{j-\delta} m_i r^{-i} \qquad m_i \in \{\pi_{min}, \ldots, \pi_{max}\} \tag{A.3}$$

Where π_{min} and π_{max} represent the two extreme values of the output digit range.

To compute the result in this manner a residual can be defined as:

$$Z_j = r^{j-\delta}(XY_j + A_j - M_{j-\delta}) \tag{A.4}$$

The residual represents the multiply-add operation performed with all available digits of the input minus the partial result $M_{j-\delta}$. The scaling factor $r^{j-\delta}$ has been introduced for convenience only, and ensures that the residual is maintained within a fixed range.

A.2 Recurrence Equation

A recurrence equation for Z_j can be developed to compute successive residuals from the previous one. That is,

$$Z_j = rZ_{j-1} + r^{-\delta}(Xy_j + a_j) - m_{j-\delta} \tag{A.5}$$

Where $Z_0 = 0$.

As formulated, a digit of the result (i.e. $m_{j-\delta}$) is determined on each iteration of the recurrence. As indicated by equation (A.3) the partial result $M_{j-\delta}$ is obtained by simply appending successive result digits, and no modification of previous digits can occur.

To obtain a correct result, the result digit must be chosen on each iteration such that

$$\sum_{i=j-\delta+1}^{\infty} \pi_{min} r^{-i} < M - M_{j-\delta} < \sum_{i=j-\delta+1}^{\infty} \pi_{max} r^{-i} \tag{A.6}$$

That is, the remaining digits of the result must always be sufficient to represent what remains of the multiply-add computation.

Summing the sequence of digits which represent the two extremes given in equation (A.6) yields

$$\frac{\pi_{min} r^{-(j-\delta)}}{r-1} < M - M_{j-\delta} < \frac{\pi_{max} r^{-(j-\delta)}}{r-1} \tag{A.7}$$

The associated bounds on Z_j can be determined by noting from equation (A.4) that

$$M_{j-\delta} = XY_j + A_j - Z_j r^{-(j-\delta)} \tag{A.8}$$

Substituting for the partial result with the above expression into equation (A.7) gives

$$\frac{\pi_{min} r^{-(j-\delta)}}{r-1} < M + Z_j r^{-(j-\delta)} - XY_j - A_j < \frac{\pi_{max} r^{-(j-\delta)}}{r-1} \tag{A.9}$$

and also substituting for M as given in equation (A.1) gives

$$\frac{\pi_{min} r^{-(j-\delta)}}{r-1} < X(Y - Y_j) + (A - A_j) + Z_j r^{-(j-\delta)} < \frac{\pi_{max} r^{-(j-\delta)}}{r-1} \tag{A.10}$$

The term $Y - Y_j$ represent the value of the remaining digits of Y, and $A - A_j$ the remaining digits of A. The two extremes of $Y - Y_j$ occur when all the remaining digits are either κ_{max} or κ_{min}. Summing these two series provides the bounds:

$$\frac{\kappa_{min} r^{-j}}{r-1} < Y - Y_j < \frac{\kappa_{max} r^{-j}}{r-1} \tag{A.11}$$

and likewise the bounds on what remains of A are

$$\frac{\kappa_{min} r^{-j}}{r-1} < A - A_j < \frac{\kappa_{max} r^{-j}}{r-1} \tag{A.12}$$

By applying these limits to equation (A.10) an expression for the bounds of the residual is obtained

$$\frac{1}{r-1}(\pi_{min} - \kappa_{min} r^{-\delta}(|X| + 1)) < Z_j < \frac{1}{r-1}(\pi_{max} - \kappa_{max} r^{-\delta}(|X| + 1)) \tag{A.13}$$

Therefore, the result digits must be selected to ensure that Z_j remains within these bounds.

The selection of $m_{j-\delta}$ is based on the value of rZ_{j-1} (as Z_j is dependent upon $m_{j-\delta}$). Therefore, it is necessary to determine the regions of rZ_{j-1} over which particular result digits should be selected. i.e.

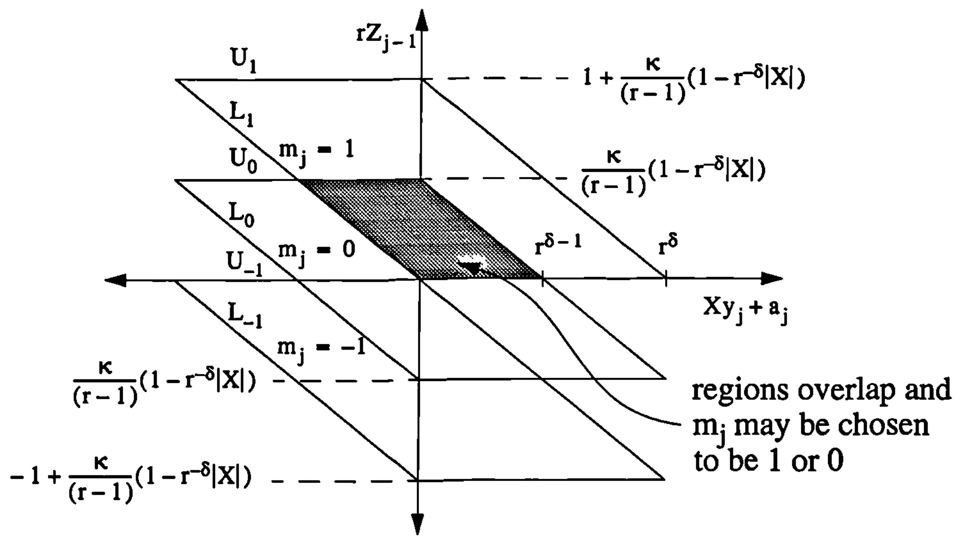
$$\text{if} \quad L_k \leq rZ_{j-1} \leq U_k \quad \text{then} \quad m_{j-\delta} = k \quad \text{for} \quad k \in \{\pi_{min}, \ldots, \pi_{max}\} \tag{A.14}$$

If recurrence equation (A.5) is substituted into the residual bounds of equation (A.13) then by setting $m_{j-\delta} = k$ the result digit selection bounds are found to be

$$U_k = k + \frac{1}{r-1}(\pi_{max} - \kappa_{max} r^{-\delta}(|X| + 1)) - r^{-\delta}(Xy_j + a_j) \tag{A.15}$$

$$L_k = k + \frac{1}{r-1}(\pi_{min} - \kappa_{min} r^{-\delta}(|X| + 1)) - r^{-\delta}(Xy_j + a_j) \tag{A.16}$$

These bounds are shown for three values of result digit in Figure A.1 for a simplified case where $\kappa_{max} = -\kappa_{min} = \pi_{max} = -\pi_{min} = \kappa$ and $A = 0$. The shaded region indicates where two bounds overlap. If the residual is located in this region it is possible to choose either result digit. To obtain overlap the condition $|X| < r^{\delta-1}$ must be satisfied. The closer X gets to this limit the smaller the overlap becomes, also as δ is decreased, the overlap is also decreased.



rZ_{j-1}
U_1
L_1
U_0
$m_j = 1$
L_0
U_{-1}
$m_j = 0$
L_{-1}
$m_j = -1$
$1 + \frac{\kappa}{(r-1)}(1 - r^{-\delta}|X|)$
$\frac{\kappa}{(r-1)}(1 - r^{-\delta}|X|)$
$r^{\delta-1}$
r^{δ}
$Xy_j + a_j$
$\frac{\kappa}{(r-1)}(1 - r^{-\delta}|X|)$
$-1 + \frac{\kappa}{(r-1)}(1 - r^{-\delta}|X|)$
regions overlap and m_j may be chosen to be 1 or 0

Figure A.1 Result digit selection regions

A.3 Simplified Result Digit Selection

To allow the selection of a valid result digit, the regions must overlap or be continuous. If they overlap, it is necessary in a practical design to decide which of the two possible results will be chosen, and so define a new set of continuous regions. Once done, however, any overlap that did exist, can still be used to allow result digit to be selected based on only an approximate value of the residual. This can be arranged, by ensuring that any digit selected using the new regions is still true to the original bounds, even when there is some uncertainty in the residual due to the use of an approximate value.

The new continuous selection bounds are defined as

$$\text{if} \quad l_k \le \hat{z} \le u_k \quad \text{then} \quad m_{j-\delta} = k \quad \text{for} \quad k \in \{\pi_{min}, \ldots, \pi_{max}\} \tag{A.17}$$

Where $\hat{z}$ is a reduced precision approximation to rZ_{j-1}, which is obtained by truncating the residual to t-bits. In which case,

$$-\Delta_{neg} \le rZ_{j-1} - \hat{z} \le \Delta_{pos} \tag{A.18}$$

where Δ_{pos} and Δ_{neg} are the magnitudes of the positive and negative truncation bounds.

The new selection bounds must lie within the old ones and accommodate the error which arises due to truncation of the residual. Accounting for the worst case truncation errors, the new bounds are related to the old by

$$\begin{gathered} l_k \ge (L_k)_{max} + \Delta_{neg} \\ u_k \le (U_k)_{min} - \Delta_{pos} \\ u_k \ge l_k \end{gathered} \tag{A.19}$$

Where $(L_k)_{max}$ and $(U_k)_{min}$ are worst case values of L_k and U_k respectively. The worst case values are used to make the selection bounds independent of all terms but the truncated residual.

For the regions to be continuous when a t-bit approximation of the residual is used, the following must be true

$$u_{k-1} + 2^{-t} = l_k \tag{A.20}$$

Figure A.2 shows diagrammatically the relationship which must exist between the bounds.

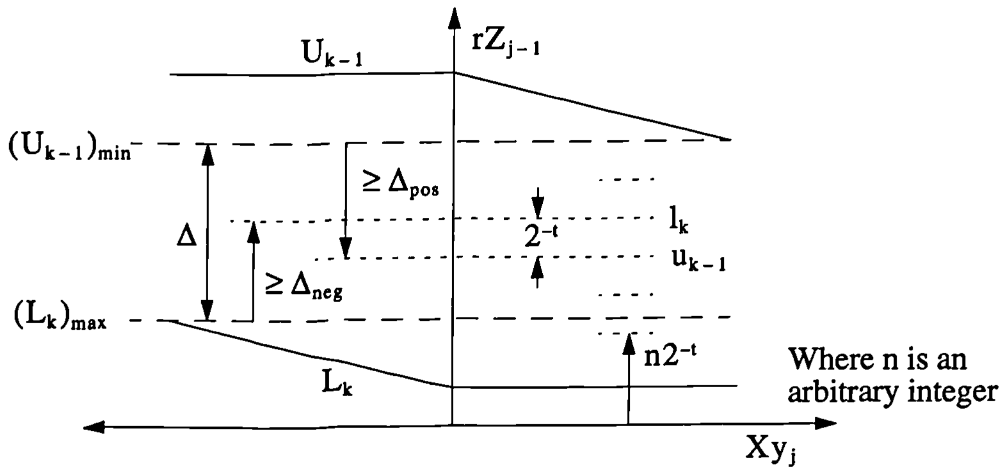


Figure A.2 Non-overlapping selection regions

The overlap of the original bounds is shown on the figure as Δ. The larger the overlap, the fewer bits of the residual which need to be examined to select a result digit. The overlap is given by:

$$\Delta = (U_{k-1})_{min} - (L_k)_{max} \tag{A.21}$$

From the relationship for U_k in equation (A.15) the worst case limit can be determined as

$$(U_{k-1})_{min} = k - 1 + \frac{1}{r-1}(\pi_{max} - \kappa_{max} r^{-\delta+1}(|X| + 1)) \tag{A.22}$$

and similarly from equation (A.16)

$$(L_k)_{max} = k + \frac{1}{r-1}(\pi_{min} - \kappa_{min} r^{-\delta+1}(|X| + 1)) \tag{A.23}$$

Substituting these relationships into equation (A.21) gives the following expression for the overlap

$$\Delta = -1 + \frac{1}{r-1}(\pi_{max} - \pi_{min} - r^{-\delta+1}(|X| + 1)(\kappa_{max} - \kappa_{min})) \tag{A.24}$$

This can be written more succinctly by defining a term which represents the redundancy of a digit set i.e.

$$R_\pi = \frac{\pi_{max} - \pi_{min}}{r - 1} \tag{A.25}$$

This quantity is 1 when the digit-set is non-redundant, and is greater than 1 when the digit-set is redundant. Using this redundancy term in equation (A.24), we obtain for the overlap

$$\Delta = -1 + R_\pi - r^{-\delta+1}(|X| + 1)R_\kappa \tag{A.26}$$

From Figure A.2 it is clear that the overlap must satisfy

$$\Delta \geq \Delta_{pos} + \Delta_{neg} - 2^{-t} \tag{A.27}$$

Substituting the expression for the overlap given in equation (A.26) gives

$$-1 + R_\pi - r^{-\delta+1}(|X| + 1)R_\pi \geq \Delta_{pos} + \Delta_{neg} - 2^{-t} \tag{A.28}$$

From this, an expression for the minimum latency δ can be obtained. i.e.

$$\delta \geq \frac{\log\left(\frac{R_\kappa((|X|)_{max} + 1)}{R_\pi - 1 - \Delta_{pos} - \Delta_{neg} - 2^{-t}}\right)}{\log r} + 1 \tag{A.29}$$

This expression can be used to determine the minimum latency which can be obtained for a particular radix, and input and output digit-sets. The equality occurs when there is no overlap, in which case all bits of the residual must be examined to select the result digit. However, it is unlikely that this expression will be an integer quantity, and the effect of rounding up the latency to the nearest integer will introduce some level of overlap. However, if the result is an integer then overlap must be obtained by increasing the latency by one.

A.4 Approximating the Residual

Assuming that a value of δ is chosen to give some degree of overlap, then only a t-bit truncated value of the residual need be examined to select the result digit. The truncation errors which can be expected for a range of residual representations are listed in Table A.1.

Table A.1 Truncation errors

Representation	Δ_{pos}	Δ_{neg}
Binary	2^{-t}	0
Carry-Save	2^{-t+1}	0
Signed-Binary	2^{-t}	2^{-t}
Minimally Redundant Radix-4	$\frac{2}{3}2^{-t}$	$\frac{2}{3}2^{-t}$
Symmetric Digit Set with Redundancy R_Z	$\frac{R_Z}{2}2^{-t}$	$\frac{R_Z}{2}2^{-t}$

Alternatively, the sum of the positive and negative truncation errors can be expressed in terms of the redundancy of the representation used i.e.

$$\Delta_{pos} + \Delta_{neg} = R_Z 2^{-t} \tag{A.30}$$

Substituting this into equation (A.28) yields,

$$-1 + R_\pi - r^{-\delta+1}(|X| + 1)R_\pi \ge (R_Z - 1)2^{-t} \tag{A.31}$$

Therefore, t must be selected to ensure that

$$t \ge \frac{\log\left(\frac{R_Z - 1}{R_\pi - 1 - r^{-\delta+1}((|X|)_{max} + 1)R_\kappa}\right)}{\log 2} \tag{A.32}$$

A.5 Multiplier-Adder Design Procedure

Based on the preceding analysis, the following procedure can be defined to design a multiplier-adder.

1. For a particular multiplier coefficient range X determine the minimum latency, and the associated overlap Δ.

2. Choose a number representation for the residual and determine the truncation bounds, Δ_{pos} and Δ_{neg}, in terms of t (the bit position to which the residual is truncated).

3. Determine the minimum value of t for which the overlap requirements are met (using equation (A.32)).

4. Solve for the selection bounds u_k and l_k by satisfying the relationships of equation (A.19). These bounds must be multiples of 2^{-t}.

Appendix B Low-Latency, High-Throughput Redundant Squarer

B.1 Algorithm

A multiplier may be used to square a number, however, this is inefficient as it is possible almost to halve the number of partial products by designing a dedicated squarer. This reduction is possible as most product terms appear twice in the multiplication of a number by itself. This can be clearly observed by partitioning the number before it is squared i.e.

$$\begin{aligned} X^2 &= (x_0 r^{-0} + (x_1 r^{-1} + x_2 r^{-2} + \ldots + x_n r^{-n}))^2 \\ &= (x_0 r^{-0})^2 + 2x_0 r^{-0}(x_1 r^{-1} + x_2 r^{-2} + \ldots + x_n r^{-n}) + (x_1 r^{-1} + x_2 r^{-2} + \ldots + x_n r^{-n})^2 \\ &= x_0^2 + 2x_0 r^{-0} \sum_{i=1}^{n} x_i r^{-i} + (x_1 r^{-1} + x_2 r^{-2} + \ldots + x_n r^{-n})^2 \end{aligned} \tag{B.1}$$

The middle term groups the two repeated partial products. The other pairs can be found by repeating the expansion for the last term, and those that follow to yield:

$$X^2 = \sum_{i=0}^{n} \left(x_i^2 r^{-2i} + 2x_i \sum_{j=i+1}^{n} x_j 2^{-i-j} \right) \tag{B.2}$$

This can be implemented using the same bit reduction techniques as used in the redundant multiplier in Chapter 3. A block diagram of a redundant squarer using a MinR4 coding is shown in Figure B.1.

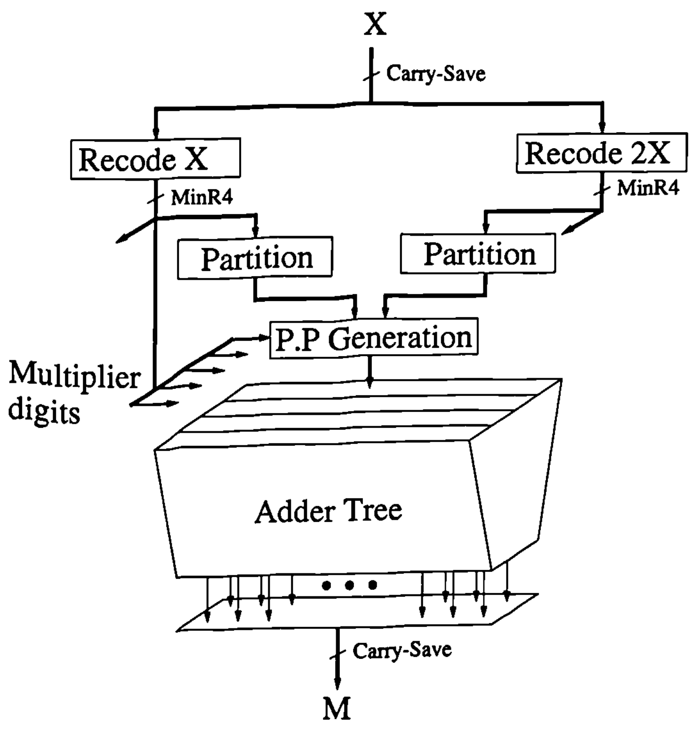


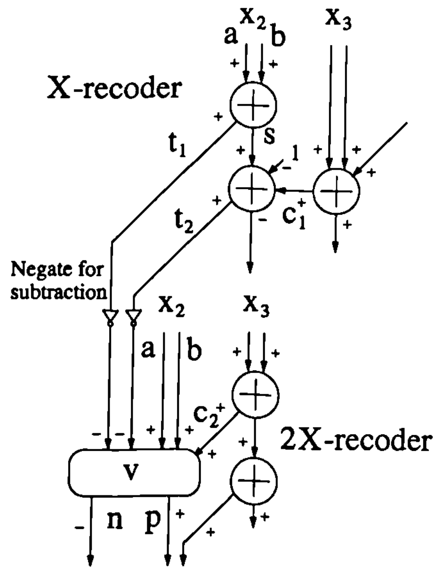
Figure B.1 Block diagram of the redundant number squarer

The partial products are formed between a digit of the recoded input and a partition of the recoded input (as indicated by equation (B.2)). Using a MinR4 digit set, only recodings of input multiples X and $2X$ are required and the same recoders as presented in the MinR4 multiplier design may be used to produce them. Ideally, the input should be recoded once partitioned, but this would require separate recoders for each input. A simplification is to recode the input once and then partition it. To do this correctly it is necessary to modify the 2X-recoder output to ensure that its partitioned output is in fact twice that of the partitioned output of the X-recoder. This modification is considered next.

B.2 Recoding of Partitioned Numbers

The X-recoder defines how the input is recoded into digits, as it is used to supply the digits of the multiplier. When the output of the 2X-recoder is partitioned it must generate a value which is twice that of the partitioned output of the X-recoder. Therefore, it must be designed so that the same number of carries occur between the two partitions as occurs in the recoding of X i.e. the number is split in the same way in both recoders. This can be achieved by subtracting the carries which occur in the X-recoder from the partitioned output of the 2X-recoder. Fortunately, this effects only the most significant digit of the partitioned output of the 2X-recoder.

Figure B.2 shows the relevant slices of the recoders. The two carries from the X-recoder are subtracted from the output of the 2X-recoder using the adder function v. Because the inputs are common to both recoders it is possible to simplify the logic. The truth table shows all possible combinations of the inputs of the relevant adders in the two recoders. The number of combinations has been reduced by noting that if $c_2 = 1$ then $c_1 = 1$.



c_2	a	b	c_1[a]	s	t_1	t_2	t[b]	x_2[c]	Total[d]	n	p
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	-1	0	-1	0	1
0	0	1	0	1	0	1	-1	+1	0	1	0
0	0	1	1	1	0	1	-1	+1	0	1	0
0	1	0	0	1	0	1	-1	+1	0	1	0
0	1	0	1	1	0	1	-1	+1	0	1	0
0	1	1	0	0	1	0	-1	+2	+1	1	1
0	1	1	1	0	1	1	-2	+2	0	1	0
1	0	0	1	0	0	1	-1	0	-1	0	1
1	0	1	1	1	0	1	-1	+1	0	1	0
1	1	0	1	1	0	1	-1	+1	0	1	0
1	1	1	1	0	1	1	-2	+2	0	1	0

a. $c_1=1$ when $c_2=1$.
b. Arithmetic value of the sum of t_1 and t_2.
c. Arithmetic value of x_2.
d. Arithmetic value of the sum of x_2 and t.

Figure B.2 Derivation of function f and its truth-table

From the truth table in Figure B.2 a logic function f can be determined to give the modified msd of the partitioned output of the 2X-recoder. This function is used within the modified recoder shown in Figure B.3 to generate the msd of each partition required (i.e. partitions would take the form $g_{2m}g_3g_4$).

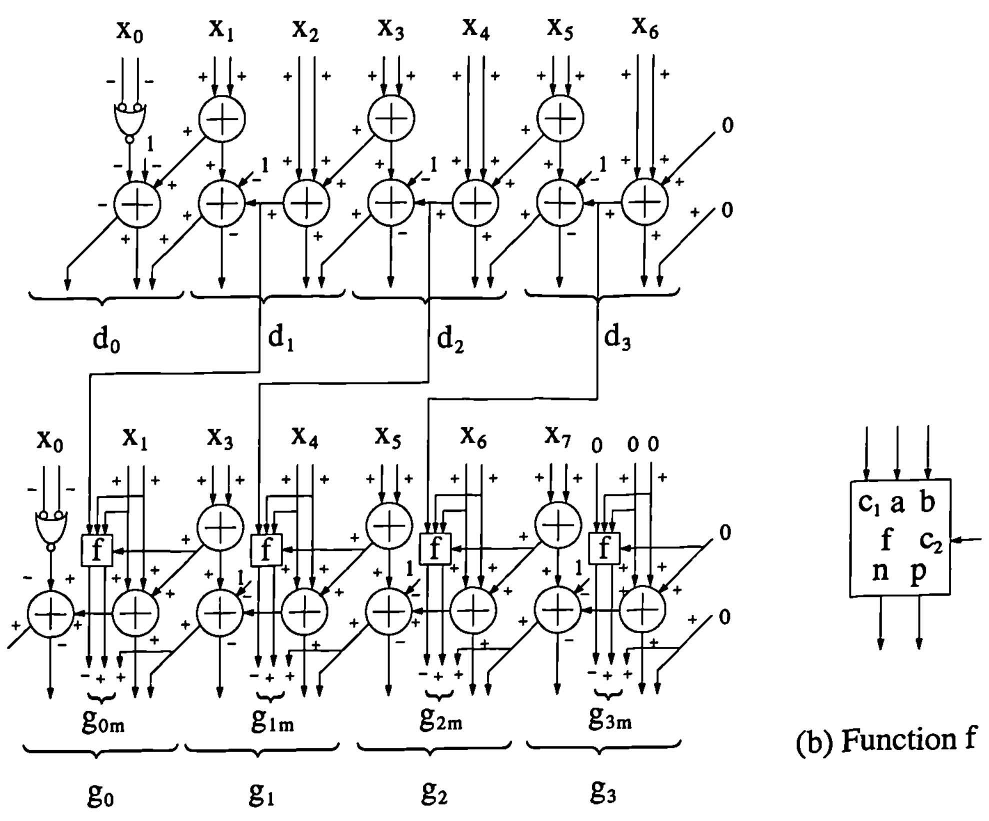


Figure B.3 Modified recoders for correct partitioning of recoded input

B.3 Partial Product Array for Squarer

The adder-tree input for a small squarer is shown in Figure B.4.

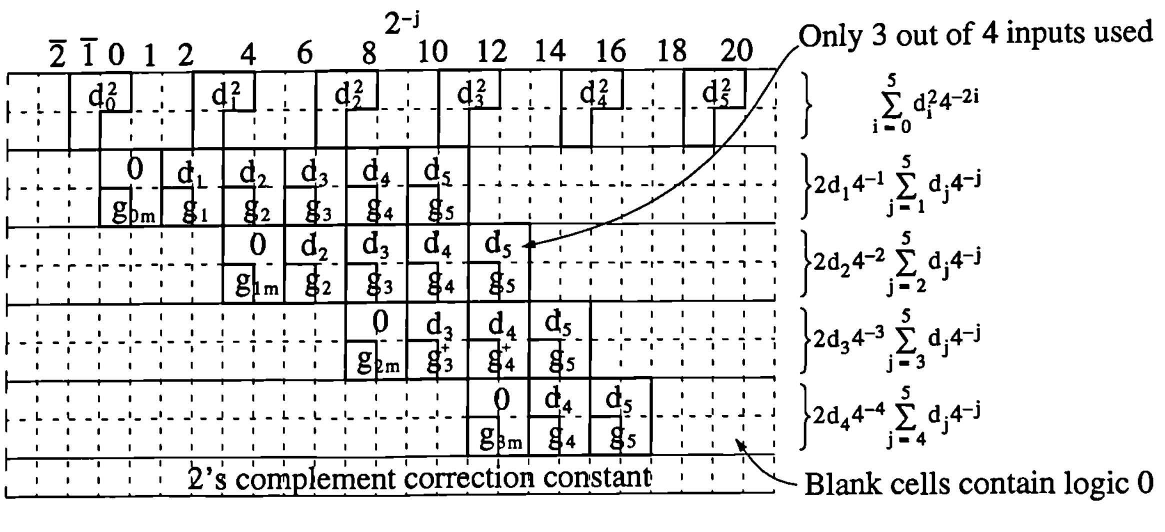

2-j
Only 3 out of 4 inputs used
2's complement correction constant
Blank cells contain logic 0
$\sum_{i=0}^{5} d_i^2 4^{-2i}$
$2d_1 4^{-1} \sum_{j=1}^{5} d_j 4^{-j}$
$2d_2 4^{-2} \sum_{j=2}^{5} d_j 4^{-j}$
$2d_3 4^{-3} \sum_{j=3}^{5} d_j 4^{-j}$
$2d_4 4^{-4} \sum_{j=4}^{5} d_j 4^{-j}$

Figure B.4 Partial-product array for a small squarer

The top row consists of the individual digits of the input X squared. The remaining rows of the array contain the partial products formed from the partitioned outputs of either the X- or the 2X-recoders (or 0). Both sets of recoded outputs are marked on the array to show the bit positions at which partial products formed from them would be entered into the array. For negative multiplier digits, d_i, the appropriate multiple is negated. The same logic as presented for the multiplier in Figure 3.17 may be used for this purpose.

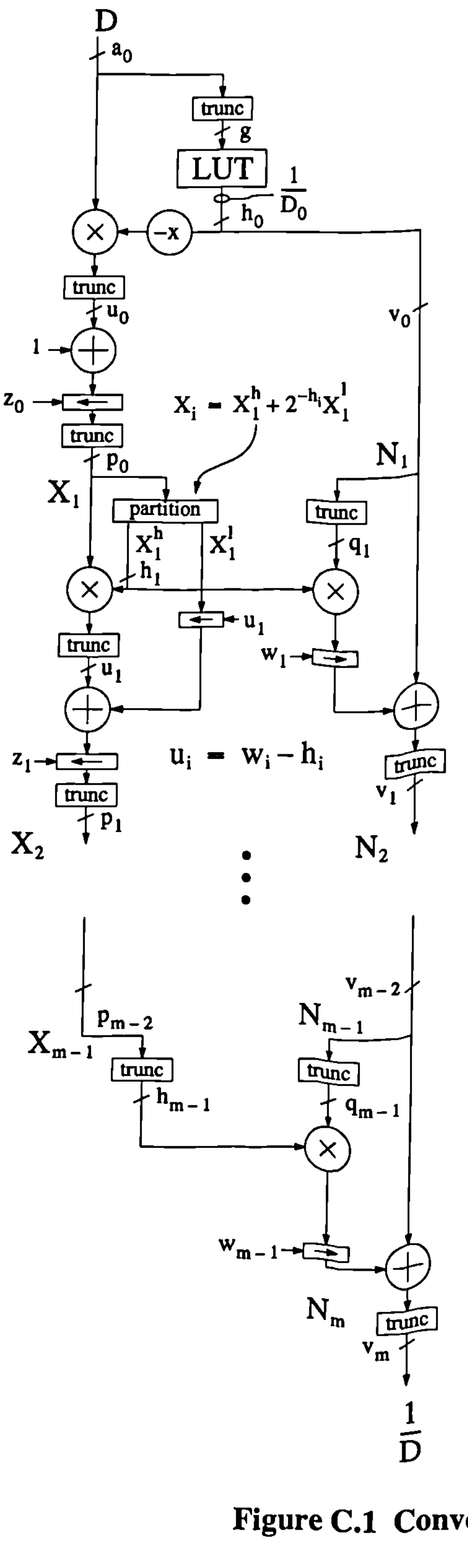
The truth-table describing the digit squaring operation is given in Table B.1. To maintain compatibility with the coding of the partial product array, the coding used to represent the squared digit is different to that used for the input digit itself. Fortunately, this coding is more convenient, and only 3-bits are required to represent the result.

Table B.1 Truth-table of digit squaring logic

MinR4 Input				Output			
bit weighting			Arith. Value	Arith. Value	bit weighting		
-2	1	1			2	2	-1
0	0	0	-2	+4	1	1	1
0	0	1	-1	+1	0	1	0
0	1	0	-1	+1	0	1	0
0	1	1	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	1	+1	+1	0	1	0
1	1	0	+1	+1	0	1	0
1	1	1	+2	+4	1	1	0

Appendix C Summary of Convergence Reciprocal Circuit Parameters

Parameters used to obtain reciprocal circuits are listed in Figure C.1.



Parameter	Wordlength			
	12	16	20	20[a]
a_0	12	16	20	20
g	10	8	9	9
h_0/v_0	9	7	8	8
p_0	12	16	20	20
z_0	9	7	8	8
h_1	-	8	9	9
z_1	-	0	0	0
p_1	11	14	17	17
q_1	9	7	8	8
w_1	9	7	8	8
v_1	20	22	25	25
h_2	-	-	17	9
z_2	-	-	-	8
p_2	-	-	17	9
q_2	-	14	17	17
w_2	-	14	16	16
v_2	-	26	32	32
p_3	-	-	-	9
q_3	-	-	-	9
w_3	-	-	-	24
v_3	-	-	-	32

a. Multiplier restricted to 9-bits (i.e. $h_0, h_1, h_2, p_3 = 9$)

Figure C.1 Convergence Reciprocal DG

Appendix D Numerical Effect in Normalised Algorithm

The increased numerical error in the residual when using the cell normalised algorithm is introduced by the term Δ_r which is calculated to adjust the normalisation of the cell output from one row to the next. i.e.

$$x_{out} = \frac{x_{in} - sr}{c\Delta_r} \tag{D.1}$$

where $\Delta_r = \sqrt{1 - r^2}$.

As r approaches 1, the value of Δ_r becomes more sensitive to the error in r. When Δ_r is applied using equation (D.1), this error is transferred to the cell output.

The effect can be quantified by considering the impact which a small change in r has on Δ_r in the vicinity of $r = 1$. Defining $\Delta_r = f(r)$ and ε_r as the small perturbation in r, the perturbed result $f(r + \varepsilon_r)$ can be obtained by performing a Taylor's series expansion. Two terms are sufficient for this purpose, and

$$f(r + \varepsilon_r) = f(r) + \frac{d}{dr}f(r)\varepsilon_r \tag{D.2}$$

Performing the differentiation and factorising yields

$$f(r + \varepsilon_r) = f(r)\left(1 + \frac{r\varepsilon_r}{1 - r^2}\right) \tag{D.3}$$

Therefore, as $r \rightarrow 1$ the quantity $\frac{r}{1 - r^2} \rightarrow \infty$ and the effect of ε_r on the output is magnified.

These results have been confirmed by simulations. This was achieved by choosing a range of channel spread parameters which cause one of the r quantities to approach one. As this happened, so the quiescent value of the residual grew very rapidly, as predicted.

Appendix E Overview of Channel Equaliser Application

A block diagram of the channel equaliser is presented in Figure E.1. It shows the triangular array being used as an adaptive linear combiner, and the channel modelled by a 3-tap FIR filter and a noise source.

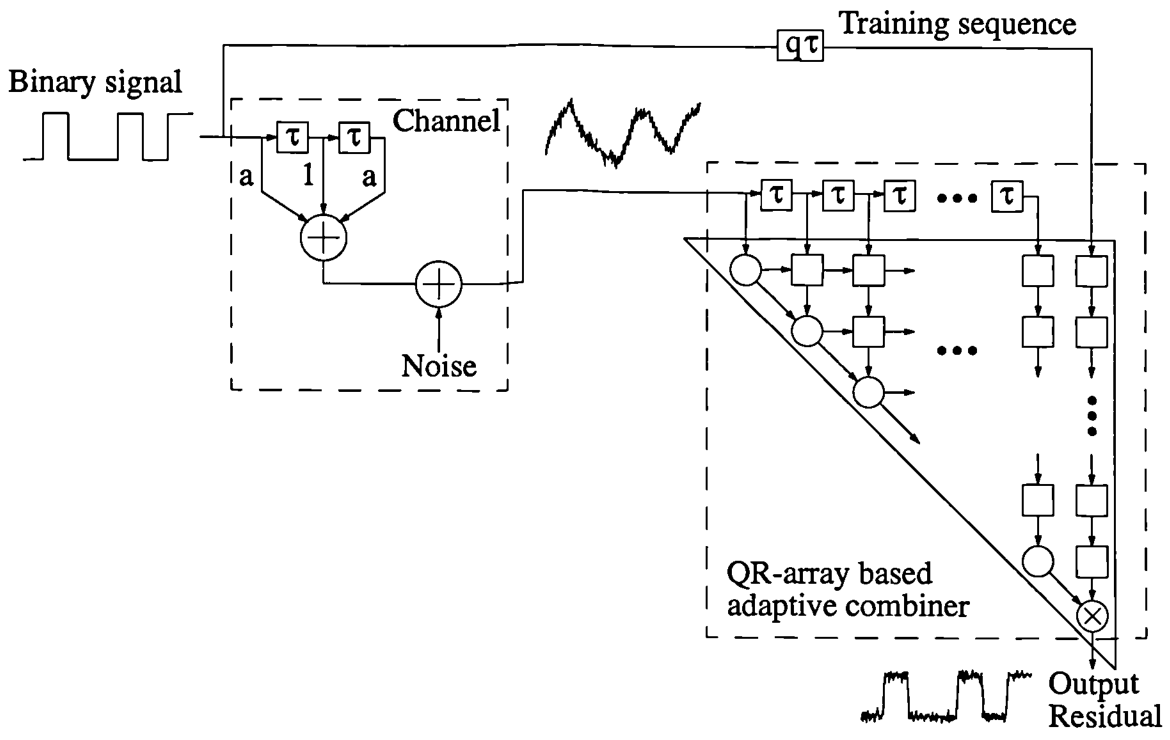


Figure E.1 Block diagram of channel equaliser for algorithm evaluation

The amount of inter-symbol interference introduced by the channel is controlled by the parameter a. The adaptive filter aims to undo this by approximating an inverse to the channel filter. In this simple case, the ideal inverse channel filter can be calculated to see what form it takes. The z-transform of the ideal inverse filter is given by

$$I(z) = \frac{1}{a + z^{-1} + az^{-2}} \tag{5.1}$$

This results in an infinite impulse response filter. However, it can be approximated by an a-causal FIR filter, which has the z-transform

$$I(z) \cong \frac{bz^{-1}}{a(1-b^2)}(\ldots -b^3z^3 + b^2z^2 - bz + 1 - bz^{-1} + b^2z^{-2} - b^3z^{-3} + \ldots) \quad \text{(E.1)}$$

where

$$b = \sqrt{\frac{1}{4a^2} - 1} - \frac{1}{2a} \quad \text{(E.2)}$$

If $a \le 0.5$ then b is real and $|b| < 1$. Therefore, in these circumstances a causal FIR filter can be obtained for practical purposes by truncating the sequence and delaying the training data. (See box labelled $q\tau$ in Figure E.1.)

The adaptive FIR filter will strive to implement this inverse channel filter, and it is clear that the greater the number of taps the better should be the approximation (under ideal conditions), but the larger the triangular array required to determine it. The channel inverse is determined during an adaption phase, when a predefined sequence of data is transmitted so that the adaptive filter can be provided with the undistorted data upon which to base the adaption.

The output of the filter represents the least squares error, which will be high at the start of the adaptation phase, but rapidly reduces as the filter converges to the channel inverse. The level to which the residual will ultimately descend is limited by three factors

- the ability of the FIR adaptive filter to form the inverse of the channel (depends upon the order of the filter)
- the noise in the channel
- numerical errors introduced by limited precision arithmetic in the adaptive filter

The last of these factors is the main interest of this study. So that it could be directly measured, the channel noise was set to zero and the order of the adaptive filter was made high enough so as not to be the limiting factor. In the simulations presented in this chapter, an 11-th order filter was used (i.e. the number array inputs $p = 12$).

The forget-factor, β, is an important parameter of the adaptive filter, as this determines the length of the window of input data over which the filter parameters r and u are calculated. The window size is given by $\frac{1}{1-\beta}$, so as $\beta \rightarrow 1$ it increases. The larger the window, the better the estimate of the filter parameters in the presence of noise, but the slower the response to changes in the channel. Typical values for β range from 0.9 to 0.9999.

The filter output residual is dependent upon the particular sequence of data transmitted over the channel. Therefore, 100 random sequences of the input are used and a single output sequence is obtained by taking the rms value over the ensemble for each time instance in the sequence. Figure E.2 shows a typical set of results obtained in this way.

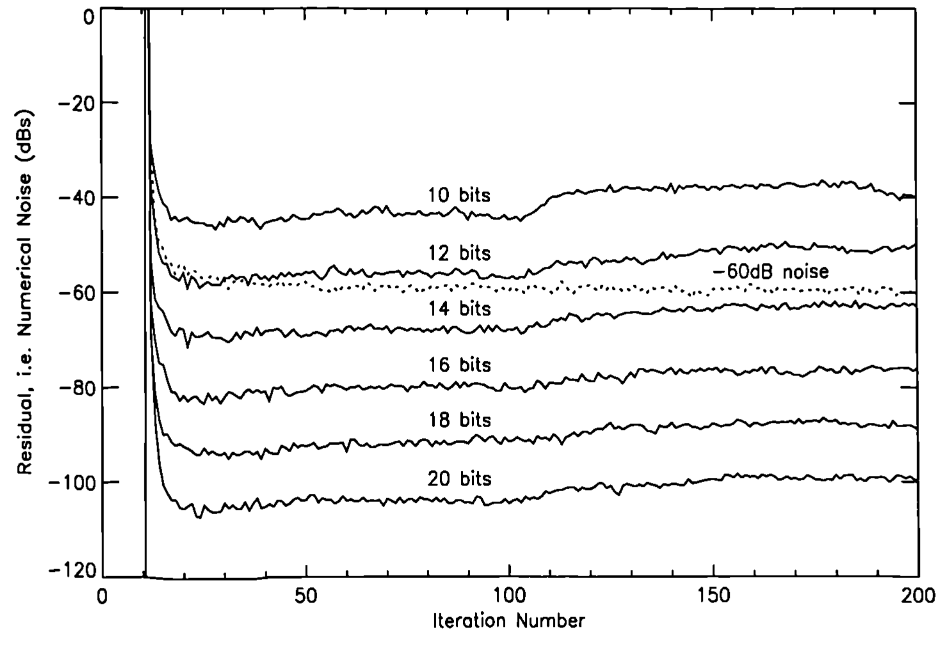


Figure E.2 Channel equaliser output during adaption phase

The solid lines represent the residual calculated using floating-point arithmetic with the shown mantissa wordlengths. The residual is initially high but rapidly decays to a steady value (which will be referred to as the quiescent value) as the filter adapts to the channel inverse. As expected, increasing the mantissa wordlength reduced the numerical error and causes a reduction in the quiescent residual level. For small wordlengths (i.e. 10- to 14-bits) the residual initially falls, but then increases before settling to a quiescent value. This is due to the accumulation of arithmetic errors in the recursive computations to update the r and u parameters. Note that these results were obtained without channel noise. If noise were introduced by the channel at a level of -60dB, then the residual shown using a dotted line on the graph would be obtained when using high precision arithmetic. In this example, it is clear from the graph that the channel noise will mask any benefit of using a mantissa wordlength greater than 16-bits.