UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
DEPARTAMENTO DE ELECTRONICA E COMPUTACION

DOCTOR OF PHILOSOPHY DISSERTATION

Division and Square root for Mobile and Scientific computing markets

VIJAYKUMAR HOLIMATH
October 2007

Dr. Javier Díaz Bruguera at the Department of Electronics and Computer Engineering in the University of Santiago De Compostela.

Hereby Certifies:

Research work entitled ”Division and Square Root for Mobile and Scientific Computing Markets” has been developed by Vijaykumar Holimath under my supervision in the Department of Electronics and Computer Engineering of the University of Santiago de Compostela, qualifies him for the PhD degree.

Santiago de Compostela, October 2007

_____

Certifica:

Que el trabajo de investigación titulado ”Division y Raíz Cuadrada para Aplicaciones Móviles y Científicas” ha sido realizado por D. Vijaykumar Holimath bajo mi supervisión en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, y constituye la Tesis que presenta para optar al grado de Doctor.

Santiago de Compostela, October 2007

Prof. Javier Díaz Bruguera,
Director of this PhD dissertation.

Javier Díaz Bruguera,
Chair of Department of Electronics and Computer Engineering.

Vijaykumar Holimath,
PhD Candidate.

Acknowledgment

First of all I would like to thank my supervisor, Prof. Javier Díaz Bruguera. I have learned many techniques working with him and his advice, ideas and guidelines have been invaluable.

I would also like to thank the people I met during my visits, for their assistance in doing the research supported by my supervisor: Prof. Naofumi Takagi (University of Nagoya, Japan), Prof. David Matula (Southern Methodist University, USA) and Prof. Jean-Michel Muller (ENS-Lyon, France).

I would also like to thank all the people in the Department of Electronics and Computer Engineering and the Computer Architecture Group for their help and cordiality. The time spent in Santiago De Compostela has been very productive, both from a professional and personal perspective.

Finally, I would like to thank my parents and my sisters (Beena and Rashmi) whose support enabled me to stay in Spain.

Santiago De Compostela, October 2007

Contents

List of Figures

List of Tables

Glossary

AQA	Accurate Quotient Approximation
CPA	Carry Propagate Addition
CSA	Carry Save Addition
FP	Floating Point
FPU	Floating Point Unit
GLD	Goldschimidt
IEEE	Institute of Electrical, Electronic, Engineering
LUT	Look-Up Table
MAF	Multiply Add Fused
N-R	Newton Raphson
PC	Personal Computer
PDA	Personal Digital Assistant
PP	Partial Product
ROM	Read Only Memory
RM	Round towards $-\infty$
RNE	Rounding to Nearest
RP	Round towards $+\infty$
RZ	Round towards 0

SD	Sign Digit
SRT	Sweeney Robertson Tocher
UPC	Ultra Personal Computer
VLSI	Very Large Scale Integration

Abstract

Division and square root functions are the basic functions in many floating point applications. The uses of these functions are segmented. For example, they are frequent in 3D graphics animation, virtual reality (especially lighting effects), view port transformation, motion synthesis, signal processing, scientific computing applications etc. These types of applications are suitable for the high performance computing market.

Division and square root functions are used less in internet applications, business applications such as audio and video streaming, charting, presentation, snap shot viewing, day to day accounting etc. Most of these applications are suitable for the mobile computing market such as PDA, UPC, tablet PC.

In both of these markets, the silicon area is crucial and in the high performance market speed is the challenge.

In state-of-the-art processors, the silicon area may be regarded as expensive and the speed is to some extent acceptable. Most designers are willing to sacrifice the speed in favor of low cost and complexity.

We therefore aim to develop efficient algorithms both in terms of area and the speed.

We propose a modified Newton-Raphson reciprocation algorithm for single precision computation, without using a look-up table. We regard this algorithm as the best silicon efficient algorithm and the speed is average compared to the state-of-the-art processors. This algorithm could be suitable for the mobile computing market.

It is a variable latency algorithm, requiring a maximum of 22 iterations, which provide a linear convergence. Multiplying with the dividend, results in division. Our method requires a cycle for each iteration, performing multiply add and Booth recoding in one cycle. Initial approximation is a two's complement of the divisor, which can be performed during partial products summation. Multiples of the divisor are constant throughout iteration and are generated only once, in the second iteration.

To evaluate the proposed method, we used a radix-4 multiplier and a radix-8 multiplier. We then compared it with conventional processors. The comparison shows that it offers a good trade-off between performance and area, making it suitable for mobile computing applications such as PDA, UPC, mobile phones, tablet PC, etc.

We also propose a division algorithm using a modified Accurate Quotient Approximation Method for double precision computation and this is extended to a square root algorithm, with a look-up table. It is a better silicon efficient algorithm with a significantly higher performance than the state-of-the-art processors. This algorithm can be suitable for both mobile and scientific computing markets.

The proposed method employs a second degree minimax approximation to obtain an initial 15 bits estimate of reciprocal and inverse square root values. This is then passed through an iteration of Goldschimdt reciprocal and inverse square root algorithms, to enhance the initial approximation to 30 bits and to make the initial approximation more silicon efficient. Following this, we perform an iteration of the Accurate Quotient Approximation method to obtain double precision results. To speed-up these functions, a high seed 30 bits is used and intermediate results are computed in carry-save form.

To evaluate the proposed method, we used a radix-4 multiplier and a radix-8 multiplier and then compared it with conventional processors. This comparison shows that there is a significant improvement in performance, with better silicon efficiency. It offers some advantages over N-R method, GLD algorithm, SRT method and Cyrix algorithm, if an iteration algorithm is considered. It is

expected that, with the proposed method, processor architects will have an option for their next generation processors.

List of Publications

International Conferences

- V. Holimath and J. Bruguera, *A Linear Convergent Functional Iterative Division without a Look-up Table*, Proc 9^{th} EUROMICRO Conference on Digital System Design (DSD), pages 236-239, Dubrovink (Croatia), Sept 2006.

Journals and Transactions

- V. Holimath and J. Bruguera, Division for Mobile Computing Applications, submitted to *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2007.
- V. Holimath and J. Bruguera, Division and Square root using Accurate Quotient Method, submitted to *IEEE Transactions on Computers*, 2007.
- V. Holimath and J. Bruguera, Division and Square root for Mobile and Scientific computing Markets, submitted to *Journal of the ACM*, 2007.

Objectives

Modern applications comprise several floating point operations, among them addition, multiplication, division and square root. The division and square root functions are less frequent than the two basic arithmetic operations (+/-,$\times$) - only somewhat less than 0.8% of the total number of instructions executed [**?**] in general purpose applications. The poor performance of many processors when computing these operations can cause an overall execution time that is comparable to the time spent performing addition and multiplication. While the methodology for designing efficient high-performance adders and multipliers is well understood, the design of division and square root still remains a serious challenge, often viewed as a Black Art among system designers [**?**].

They are frequent in 3D graphics animation, virtual reality [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] (especially lighting effects), view port transformation, motion synthesis, digital signal processing applications and scientific computations. The cost in time of the division and square root function depends on how smartly the compiler can schedule, in order to maximize the distance between the production of the results of other functions and the dependency with respect to division/square root.

These functions can be expressed as the product of the dividend and reciprocal of the divisor, $Q = X \times (1/Y)$ and $Q = Y \times (1/\sqrt{Y})$. There are several techniques described in the literature [**?**] and carried out in state-of-the-art processors such as N-R method, GLD algorithm [**?**, **?**, **?**], SRT algorithm [**?**, **?**]. The performance of these functions depends on the precision; the precision depends on the market targeted. For example, in 3D graphics single precision is used and

in scientific computing applications double precision is used.

To speed-up the division and the square root functions, somewhat more than ten bits of initial approximation of reciprocal and inverse square root are used. The values of the reciprocal and the inverse square root are stored in ROM also called LUT in terms of bits.

Another option to obtain the initial approximation is, storing the coefficient values of linear/quadratic approximation algorithms of reciprocation and inverse square root in the LUT's. Then, the values of the coefficients are added together or the coefficient values are multiplied by the respective input operands of the approximation algorithms and then added. The linear/quadratic approximation algorithms reduce the area for the LUT size.

To obtain the required precision, the LUT's or linear/quadratic approximation algorithms are coupled with the N-R method, GLD algorithm or SRT method. The N-R and GLD methods are quadratic convergence in character and the SRT method linear convergent. On average, the speed of these operations is more than 20 cycles in either single precision or double precision computation in state-of-the art processors.

To enhance performance according to market demands, there is a trade-off between the performance and silicon area. The area for these functions accumulates from LUT's. The more bits of initial approximation that are used, the more silicon area is required to speed-up these functions. This means that the silicon area is directly proportional to the market demands. Most designers are willing to sacrifice the speed in favor of low cost and design complexity.

In the state-of-the-art processors such as Intel, AMD, IBM, the silicon area for the initial approximation may be regarded as expensive and the performance of these functions is to some extent acceptable. These processors are targeted at various markets such as the mobile computing market, the desktop computing market, the scientific computing market etc.

In this thesis, we therefore aim to develop an algorithm that is: a) a best silicon efficiency algorithm and b) a better silicon efficient and best performance

algorithm:

1. A modified N-R reciprocation/division algorithm without using a LUT for single precision computation is proposed. It is a best silicon efficient algorithm and the performance is about the same as the average performance of the state-of-the-art processors.

2. The double precision computation of AQA method for both division and square root, using a LUT is proposed. It is a better silicon efficient algorithm and the performance is significant better compared to the state-of-the-art processors.

We have chosen this approach to provide an option for processor architects that suits their target market.

The modified N-R algorithm is suitable for the mobile computing market. In this market area is crucial. Most architects look for a division algorithm with a low cost, a reduced area with acceptable performance that can run internet applications, business applications (such as audio and video streaming), charting, presentations, snapshot viewing, day to day accounting etc.

The modified AQA method is suitable for the high performance market. In this market the performance is the key. Most architects look for high performance division and square root algorithms with acceptable cost and complexity that can be applied to scientific computing applications, 3D graphics, signal processing applications etc. The modified AQA method is also suitable for the mobile computing market.

In the modified N-R algorithm, we achieve our objective by taking an initial approximation as a two's complement of the divisor and modifying some steps of the N-R algorithm [**?**]. This initial approximation can be performed during partial products summation of the multiplicand (divisor) and it is silicon efficient.

The average performance is maintained by executing a multiply add operation and Booth recoding in the same cycle and keeping the divisor multiples constant in

each iteration. Multiples are generated only once, in the second cycle with respect to the radix-4 multiplier and third cycle with respect to the radix-8 multiplier. This leads to a reduction of a cycle in each of the iterations and each of the iterations is a cycle, which is unlikely to be found in a conventional processors. Furthermore, intermediate results are computed in carry-save form.

The modified N-R method offers some advantages over the N-R method[1] and the SRT method:

- In the modified N-R method multiples of the divisors are constant throughout iteration, it is generated only once, in the second iteration.
- In the modified N-R method, the number of dependent operation is a multiplication, where as in the N-R method it is two dependent multiplications.
- In the modified N-R method, each iteration is a cycle, whereas in the N-R method it is two multiplication cycles, i.e. about 6 cycles.
- The performance gain of the modified N-R method and N-R method are about the same.
- With respect to the SRT method, the modified N-R method does not require a quotient digit selection hardware to select the next quotient digits, which increases the cycle time with respect to the radix.

A similar approach can be extended to perform the square root, but will adversely affect performance because it requires the squaring of the quotient in each of the iterations. Therefore, it requires more than 50 cycles and average performance cannot be maintained. More research on the modified N-R square root algorithm without a LUT is encouraged.

In the modified AQA method, the tactic we used was the taking of a 15 bit initial approximation from the Minimax method [**?**], which we then passed through

[1]The GLD algorithm is similar to the N-R method but the arrangement of the equations in the hardware differs. In the modified N-R method, we modified one of the equations in the N-R method and explained its significance.

an iteration of the GLD algorithm to obtain a 30 bit initial approximation, this method is less complex. Some steps of the Cyrix algorithm [**?**, **?**] were modified. Iteration is required to obtain the double precision results, after a 30 bits initial approximation.

To enhance the performance, we used a high seed 30 bits initial approximation that is silicon efficient. Furthermore, intermediate results are computed in carry save form.

The modified AQA method sidelines the state-of-the-art algorithms:

- With respect to the SRT method, a large number of bits are retired in an iteration and the complexity of the circuit is less than that of the SRT method, if the SRT method uses a high seed 30 bits initial approximation. In the SRT method the complexity of the circuit increases as the radix increases, unlike the proposed AQA method.

- The N-R method, to obtain the double precision results of the division/square root, requires the whole 53 bit accuracy of reciprocal/inverse square root of the divisor and is then multiplied by the respective dividend. The modified AQA method, requires only an initial approximation to obtain the double precision results.

- With respect to GLD algorithm [**?**], the proposed method is similar if the 30 bit initial approximation was used. But the way of computations (equations) in the hardware are different. In the proposed method, intermediate results are computed in CS form and used as a multipliers, moreover, the proposed method offers more parallelism with respect to the GLD algorithm. These features increase performance, which is deficient in the GLD algorithm [**?**].

- The Cyrix processor AQA algorithm requires two iterations to obtain the double precision results. There are four dependent operations if the same 30 bit initial approximation is assumed. In our method, it requires an iteration to obtain the double results with three dependent operations. Unlike

Cyrix algorithm the proposed method can be performed in fixed point computation.

To evaluate the proposed algorithms, we used a radix-4 MAF of IBM $603e^{TM}$ FPU [**?**] targeted at mobile computing applications and a radix-8 multiplier of IBM G5, FPU [**?**, **?**] which is targeted at high performance applications. We then compared them with the state-of-the-art processor algorithms. The comparison shows: the modified N-R offers average performance with the best silicon efficiency. The modified AQA method offers the best performance with better silicon efficiency.

In Chapter 1, we give an introduction of the floating point format and in Chapter 2, we discuss the algorithms in the conventional processors. In Chapter 3, we propose a modified N-R reciprocation algorithm without a LUT for single precision computation. In Chapter 4, we propose a modified AQA method for both division and square root using a LUT for double precision computation. We then summarize and, finally, we propose ideas for future study.

Our research emphasizes on single precision and double precision computation for commercial applications.

Chapter 1

Introduction

In this chapter we describe number formats used in computer arithmetic with special attention to floating point arithmetic. It includes floating formats, rounding, special values and exceptions. We then discuss the role of floating point division and square root in the field of computer arithmetic and floating point computation.

1.1 Number Representation

In a computer a number can be represented in terms of bits, it can be classified into two representations: fixed point representation and floating point representation. A number can be represented by the following equations [**?**, **?**, **?**]:

$$x = X_0 + \sum_{i=0}^{n-1} X_i.r^i \tag{1.1}$$

. . . fixed point representation

where X_0 is a sign bit, X_i is a bit vector having i bits with a base also called *radix* r. The r is usually represented by power of 2, such as $2^0, 2^1, 2^2, 2^3, 2^4 \ldots 2^{n-1}$.

$$M_x = (1)^{s_x} X \beta^{E_x} \tag{1.2}$$

. . . floating point representation

Where M_x is a number represented using a sign bit s_x, X is a normalized significand, and an exponent E_x of base β.

The purpose of using floating point representation is to increase the dynamic range with respect to fixed point representation. The dynamic range is a ratio of the largest and smallest number (non zero and positive) that can be represented.

For a fixed point representation using n radix-r digits for the magnitude, the dynamic range is

$$DR_{fxp} = r^n - 1 \tag{1.3}$$

for the floating point representation,

$$DR_{flp} = \frac{X_{max}\beta^{E_{max}}}{X_{min}\beta^{E_{min}}} \tag{1.4}$$

For instance, if the n bits are portioned so that m bits are used for the significand and $n - m$ bits for the exponent with $\beta = r$ we get

$$DR_{flp} = (r^m - 1)r^{(r^{n-m}-1)} \tag{1.5}$$

For example $n = 32$, $m = 24$, $r = 2$

$$DR_{fxp} = 2^{32} - 1 \approx 4.3 \times 10^9$$

$$DR_{flp} = (2^{24} - 1)2^{2^8-1} \approx 9.7 \times 10^{83}$$

A large dynamic range is required in many applications to avoid overflows and underflows. If the dynamic range of the fixed point representation is not sufficient, complicated scaling operations have to be performed in the program. Therefore, in many applications, the floating point system is preferable.

In IEEE standard-754 [**?**] floating point arithmetic, $\beta = 2$ and four formats are specified. As shown in eqn.1.2, the significand can be represented in sign-and-magnitude notation. There are two basic formats, *single-precision* and *double-precision*, but there is also an *extended* format corresponding to each: single and double-precision. The extended precision is intended to be used in situations where it is desirable to perform computation with a higher precision than the one in which results are displayed (signal processing, 3D graphics require single-precision). The number of bits representing the exponent, significand, and other characteristics of the formats are summarized in Table 1.1. In division and square root, the sign and the exponent can be computed in parallel with the significand. At the end of the iteration, a multiplier cycle for rounding is used to obtain an exactly rounded result.

Parameter	single	single extended	double	double extended
width	32	$\geq$ 43	64	$\geq$ 79
precision	24	$\geq$32	53	$\geq$ 64
exponent width	8	$\geq$11	11	$\geq$15
max.exponent	127	$\geq$1023	1023	$\geq$16383
min.exponent	-126	$\leq$-1022	-1022	$\leq$-16382
exponent bias	127		1023	

Table 1.1: IEEE Standard-754 floating point formats

1.2 Rounding

The result of a floating point operation (such as addition, multiplication, division and square root), to be represented exactly, might require a significand with an infinite number of digits. Since the representation of the siginificand has only f fractional digits, it is necessary to obtain a representation that is close to the exact result.

From the eqn.1.2, M_x a number can be represented as the set of triple (s_x, X, E_x) with X normalized. The X can be infinite precision but it might not be a floating

point number. On the other hand, the X is inside the range of the floating point number, which is $1 \leq X < 2$ for the IEEE standard. We decompose X into two parts X_f and X_d such that

$$X = X_f + X_d \times r^{-f} \tag{1.6}$$

with $0 \leq X_d < 1$. X_f has the precision of the significand in the floating point system and X_d represents the rest.

IEEE standard-754 defines four different rounding modes, which must be supported:

- ▷ *Rounding to Nearest (*RNE*)* (Unbiased, Tie to even) value and to the even number when the result is exactly halfway between two machine numbers.

  In the RNE mode the value represented is the closest possible to the exact value (infinite precision represented by X), it produces the smallest absolute error.

  In terms of the operation on the infinite precision significand, RNE can be described as follows:

$$RNE(X) = \begin{cases} X_f + r^{-f} & \text{if } X_d \geq \frac{1}{2} \\ X_f & \text{if } X_d < \frac{1}{2} \end{cases} \tag{1.7}$$

  The RNE consists of adding $r^{-f}/2$ to the infinite precision significand and keeping the resulting f fractional digits. i.e.

$$RNE(X) = (\lceil (X + \frac{r^{-f}}{2}) r^f \rceil) r^{-f} \tag{1.8}$$

  For $X_d \geq \frac{1}{2}$, the addition of r^{-f} can produce a significand that cannot be represented (significand overflow). In such a case, the resulting significand is multiplied by β^{-1} and the exponent

incremented by 1.

The absolute error is

$$ABRE[RNE] = \begin{cases} -X_d r^{-f} \times \beta^E & \text{if } X_d < \frac{1}{2} \\ (1 - X_d) r^{-f} \times \beta^E & \text{if } X_d \geq \frac{1}{2} \end{cases}$$

The maximum absolute error occurs when $X_d = \frac{1}{2}$, resulting in

$$MABRE[RNE] = \frac{r^{-f}}{2} \times \beta^{E_{max}} \tag{1.9}$$

We now consider the bias. As indicated above, the absolute error for $X_d = a$ and for $X_d = 1 - a$ (for $a < \frac{1}{2}$) have the same magnitude, but different sign. Consequently, with respect to the bias, these errors cancel each other out. The only remaining case is for $a = \frac{1}{2}$, which produces a positive error. To have a bias equal to 0, is a case to be treated in a special manner. The IEEE standard specifies this case, as rounding is done to even[1]. i.e, the *unbiased round to nearest* is

$$RNE(X) = \begin{cases} X_f & \text{if } X_d < \frac{1}{2} \\ X_f + r^{-f} & \text{if } X_d > \frac{1}{2} \\ X_f & \text{if } X_d = \frac{1}{2} \text{ and } X_f = \text{even} \\ X_f + r^{-f} & \text{if } X_d = \frac{1}{2} \text{ and } X_f = \text{odd} \end{cases}$$

Consequently, for this mode

$$RB[RNE] = 0 \tag{1.10}$$

The bias (RB). This is defined as the average absolute error

[1] Rounding to odd in the tie case also has a bias of zero. However, round to even is preferable because it leads to less error when the result is divided by 2–a common computation.

considering an unsigned significand (if the signed significand is used, the bias would be zero for most rounding modes) and measures the tendency toward error of a particular sign. To compute this average, it is necessary to consider a frequency distribution of the values of unsigned significand. The usual assumption is a uniform frequency distribution (this may not occur in typical applications), in which case

$$RB = \lim_{t\to\infty} \frac{\sum_{X\in\{X_{m+t}\}}(Rmode(X) - X)}{\#X} \tag{1.11}$$

where $\{X_{m+t}\}$ is the set of all unsigned significands with $m + t$ bits, and $\#X$ is the number of significands in the set.

The RNE (unbiased) produces the smallest possible absolute error and has zero bias.

▷ *Rounding towards* 0 (RZ) (truncation), in which the result should be the value closest to and no greater in magnitude than the exact value.

In terms of the operation on the infinite precision significand, the rounding significand is obtained by discarding X_d. I.e.

$$RZ(X) = (\lceil X \times r^f \rceil)r^{-f} = X_f$$

The absolute error is

$$ABRE[RZ] = -X_d r^{-f} \times \beta^E$$

Since $X_d < 1$ the maximum absolute error is

$$MABRE[RZ] \approx r^{-f} \times \beta^{E_{max}}$$

This absolute error is larger than the RNE. Furthermore, for an

unsigned significand, the absolute error is always negative and the bias is significant. Its value is

$$RB[RZ] \approx -\frac{1}{2}r^{-f}$$

- ▷ *Rounding towards+*∞ RP or upwards, in which the result should be the value closest to and no less than the exact value.
- ▷ *Rounding towards-*∞ RM or downwards, in which the result should be the value closest to and no greater than the exact value.

  These two directed modes are useful in interval arithmetic, in which the operands and the result of an operation are intervals. This permits the monitoring the accuracy of the result.

  In terms of the infinite precision siginificand and the sign,

$$RP(X) = \begin{cases} X_f + r^{-f} & \text{if } X_d > 0 \text{ and } s = 0 \\ X_f & \text{if } X_d = 0 \text{ or } s = 1 \end{cases}$$

$$RM(X) = \begin{cases} X_f + r^{-f} & \text{if } X_d > 0 \text{ and } s = 1 \\ X_f & \text{if } X_d = 0 \text{ or } s = 0 \end{cases}$$

By default RNE is the active rounding mode for IEEE standard. More information on the rounding can be obtained from [**?**].

1.3 Special values

The exponents in the floating point formats have a biased representation. The bias representation is preferred because it simplifies the comparison of floating point numbers by making it a fixed point comparison and the minimum exponent is represented by 0, so that the representation of the floating point value 0 is all zeros (0 sign, 0 exponent, 0 significand).

In a biased representation with bias B, the signed integer E_x is represented by the positive integer denoted by E_B such that

$$E_B = E_x + B \tag{1.12}$$

To represent the minimum exponent by $E_B = 0$, we obtain

$$B = -E_{min} \tag{1.13}$$

For a symmetric range

$$-B \leq E_x \leq B \tag{1.14}$$

resulting in

$$-0 \leq E_B \leq 2B \tag{1.15}$$

If e is the number of bits of the binary representation of E_B, then

$$2B \leq 2^e - 1 \tag{1.16}$$

Consequently, for B integer we obtain

$$B \leq \frac{1}{2}(2^e - 2) \tag{1.17}$$

for example $e = 8$ (single precision computation) we can make $B = 127$ and

$$E_B = E_x + 127 \tag{1.18}$$

for a symmetric exponent range of $-127 \leq E_x \leq 127$. Note that the maximum value of E_B is 255, so that this value can be used to represent $E_x = 128$

(non-symmetric range) or as a singularity condition.

Two values are not used for real numbers, these special values correspond to the smallest possible exponent, E_{min} which is used for $+0$, -0 and *denormalized numbers*, and the largest possible exponent, E_{max} which is reserved for $-\infty$, $+\infty$ and *NaNs*. The interpretations of a floating point value are the following:

- A maximal exponent and a non-zero significand represents *NaN*. The value in the siginificand may be set according to the machine representation allowing different *NaNs*.
- A maximal exponent and a zero significant represents $-\infty$ or $+\infty$ according to the sign bit.
- A minimal exponent and a non-zero signinficand represent a denormalized number.
- A minimal exponent and a zero significand represents $+0$ or -0 according to the sign bit.
- The remaining exponent values and significands represents normal numbers.

The uses of the special values are: the overflow operation or division of a real number by zero is $-\infty$ or $+\infty$ according to the sign of the intermediate result or the dividend. The IEEE standard 754 calls for two *NaNs*: *signaling* and *quiet*. The signaling *NaN* is intended to provide for uninitialized variables and arithmetic-like enhancements not included in the standard. The quiet *NaN* is intended to provide retrospective diagnostic information for invalid or unavailable operands and results. The result of an under flowing operation is $+0$ or -0, according to the sign of the intermediate result. Denormalized numbers are intended to support *gradual underflow* [**?**, **?**].

1.4 Exceptions

There are five types of exception that must be signaled when detected (i.e. a status flag must be set or a trap routine is called):

- *Invalid operation*, when an operand is not valid for the operation to be performed. The result is a quiet *NaN*.
- *Division by zero*, if the dividend is a finite non-zero number and the divisor is zero. The result is a $+\infty$ or $-\infty$.
- *Overflow*, if the magnitude of a result exceeds the largest finite number representable in the floating point format of the operation. The delivered result may be $+\infty$ or $-\infty$ or plus or minus the largest representable number in the floating point format, depending on the rounding mode.
- *Inexact result*, if the rounded result of an operation is not exact or if overflow occurs but there is no overflow trap.

1.5 Importance of Division and Square root Functions

Use of the division and square root functions are segmented. They are frequent in 3D graphics animation, virtual reality (especially lighting effects), view port transformation, motion synthesis, signal processing, scientific computing applications etc. These types of applications one may call high computing applications.

These functions are less intense in internet applications, business applications (such as audio and video streaming), charting, presentation, snap shot viewing, day to day accounting etc. One may distinguish these types of applications as low computing applications.

In early days, computing division and square root functions were carried out in software emulations. However, most recent processors [**?**, **?**, **?**, **?**] complying

with IEEE-754 standard are equipped with FPU's performing FP addition, subtraction, multiplication, division and square root functions in VLSI. We encourage readers to obtain FP addition, subtraction, and multiplication algorithms from [**?**]. In the next chapter we discuss division and square root algorithms.

A survey performed on FPU's [**?**] reveals that while the majority of the microprocessors surveyed carry out both FP addition and multiplication in 3 cycles, FP division consumes about 60 cycles and usually FP square root slower than FP division. The survey shows that the designers have put their emphasis on the development of faster FP adders and multipliers. This negligence intentionally widens the performance gap by downplaying FP division and square root functions. Software developers take advantage of FP addition and multiplication algorithms to avoid these complex operations.

Another investigation performed by Oberman [**?**] reveals the relationship between the latency of FP division/square root and the system performance. The study shows that FP adders and multipliers are the consumers for 27% and 18% of FP divider/square root results respectively. This means that if an inefficient FP divider and square root is used in the FPU, the processor interlock period generally increases, because the FP divider/square root result consumers have to wait longer for the data.

Now days, there is a trend towards using addition operation in a multiplication. This is also called multiply add fused operation for division and square root functions, such as is found in IBM $603e^{TM}$ microprocessor FPU [**?**]. The multiply-add fused operation can increase the performance of division and square root functions. Dealing with the division and square root more seriously and the better balancing of performance among other FP units, is more reasonable than compromising the overall performance of the whole processor.

Chapter 2

Classification of Hardware Methods

In this chapter we discuss hardware methods for the division and square root functions. They can be classified into two groups: non-iterative methods and iterative methods, we briefly discuss these methods. The non-iterative methods are limited to single precision computation, due to prohibitive area requirements, leading to its use as an initial approximator for the iterative methods. The methods we discuss in this chapter are incorporated in most state-of-the-art processors.

The main types of techniques used for approximating division and square root in hardware can be classified in two groups:

Non-iterative Methods [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]

- ▷ Direct table look-up.
- ▷ Polynomial or rational approximations.
- ▷ Table-based methods.

Iterative Methods [**?**, **?**, **?**, **?**, **?**, **?**]

- ▷ Functional iterative methods.
- ▷ Digit-recurrence and on-line algorithms.

In the first group are direct table look-up, polynomial and rational approxima-

tions, and table based methods. In the second group are hardware implementations of functional iterative methods such as N-R, GLD. The first group is usually suitable for low precision computations up to single-precision floating point or 32-bit fixed point computations, while the iterative methods are used for both low precision and high precision computations, i.e. double precision or double precision-extended floating point operations or fixed-point 64 bit computations.

2.1 Non-iterative Methods

Direct table look-up is suitable for very-low precision calculations, but the huge area requirements of such a technique makes it an inefficient method for even single-precision computations: tables of $(2^{24} \times 24)$-bits, i.e. 50 MB would be required.

Another option is approximating elementary functions by polynomial approximation [**?**]. The polynomial approximation uses addition and multiplication in the processor. The degree of the polynomial employed is usually high and a large number of additions and multiplications must be performed, which results in longer execution times.

2.1.1 Table-based methods

Table-based methods are a mixture of direct table look-up and polynomial approximations. Using a table look-up, allows the use of a low-degree polynomial and the low-degree polynomial produces a significant reduction in the size of the look-up tables that, in turn, reduces the number of arithmetic operations. This results in low hardware requirements for polynomial approximations, together with the speed of direct table look-up. There is a trade-off among these parameters, which results in the formulation of three types of table-driven algorithms, divided into: *compute-bound methods* [**?**, **?**], *table-bound methods* [**?**, **?**, **?**, **?**, **?**] and *in-between methods* [**?**, **?**, **?**, **?**].

- ▷ *Compute-bound methods*-these methods are suitable for both software and hardware implementation. They are more suitable for software, because there is a significant amount of addition and multiplication involved. It requires a fused multiply-add, such as that found in Power PC and the Intel IA64 [**?**, **?**, **?**]. The compute-bound methods use table look-up in a very small table to obtain parameters that are used afterwards in intermediate-degree polynomial (or rational) approximations. This method is a slow approach for double precision computations, equivalent to 9 multiplications and 7 additions/subtractions.
- ▷ *Table-bound methods*-use large tables and few additions, such as those found in *Partial Product Arrays* [**?**] and bipartite table methods [**?**]. Symmetric bipartite table methods [**?**] and multipartite methods [**?**] employ more than two look-up tables and a few additions. These generalizations do not suggest using a large number of tables, since the performance of many additions to avoid multiplications is not reasonable. These methods are very fast, but their use is limited to less than 20 bits with current VLSI technology, due to the size of the tables needed to assure the required precision of the result.
- ▷ *In-between methods*-use medium size tables and reduced computation, possibly one or two multiplications or several small / rectangular multiplications. The intermediate size of the look-up tables makes them suitable for performing single-precision computations, achieving fast execution times with reasonable hardware requirements. This type of methods can be further subdivided into linear approximations [**?**, **?**] and second degree interpolation methods [**?**, **?**], depending on the degree of the polynomial employed. Cubic approximations belong to the *Compute-bound methods*. They require the evaluation of a degree-3 polynomial, which results in many multiplications and additions.

Method	Table size (bits)	Multiplier	Adder	Others
Direct	$2^n \times n$	-	-	-
Bipartite tables [**?**]	$(2^{2n/3} \times n) + (2^{2n/3} \times n/3)$	-	n	-
SBTM [**?**]	$(2^{2n/3} \times n) + (2^{2n/3-1} \times n/3)$	-	n	Booth rec
Piecewise linear approx [**?**]	$2^{n/2-2} \times (2(n/2-2)+4) + 2^{n/2-2} \times (n+3)/3$	$(n+5) \times (n+3)/3$	$2n+4$	-
Linear interpolation [**?**]	$2^{n/2} \times (n+2)$	$(n/2+3) \times (n/2+3)$	n	Booth rec
2^{nd}-degree interp. [**?**]	$2^{n/3} \times (n+2n/3+n/3+n/3)$	2 of $(n \times n)$	2 of n	-
2^{nd}-degree interp. [**?**, **?**]	$2^{n/3} \times (n+2n/3+n/3)$	2 of $(n \times n)$	≈ 5 of n	-

Table 2.1: Comparison of different table-driven methods to obtain $n = 24$ bit accuracy

The *In-between methods* can be subdivided, depending upon the type of values stored in the tables. The common practice is to store the polynomial coefficients for each subinterval. In this case, instead, the function values are stored by calculating the polynomial coefficients on-the-fly, resulting in *hybrid* implementations [**?**].

Table 2.1 summarizes the hardware requirements for the type of table-driven methods described for performing single-precision reciprocal and inverse square root functions. Booth recoding (*Booth rec*) [**?**, **?**] is a part of the method.

Bipartite tables

In the bipartite table methods [**?**, **?**], the n bit input operand Y is split into three parts, $Y = Y_1 + Y_2 + Y_3$ corresponding to the *higher*, *medium*, *lower* significant fields:

$$Y = Y_1 + Y_2 2^{-k} + Y_2 2^{-2k} \tag{2.1}$$

Each part is composed of $k = n/3$ bits

The general expression for a bipartite table method is

$$f(Y) \approx f(Y_1 + Y_2 2^k) + Y_3 2^{2k} f'(Y_1) \tag{2.2}$$

which corresponds to approximating $f(Y)$ by the addition of two functions $C_2(Y_1, Y_2)$ and $C_1(Y_1, Y_3)$:

$$C_2(Y_1, Y_2) = f(Y_1 + Y_2 2^{-k}) \tag{2.3}$$

$$C_1(Y_1, Y_3) = Y_3 2^{-2k} f'(Y_1) \tag{2.4}$$

The coefficients C_2 and C_1 are read from look-up tables addressed by Y_1, Y_2 and Y_1, Y_3 respectively and the n bit adder assimilates the two words to obtain the function approximation $f(Y)$. Advantage is taken of some of the symmetry properties of the look-up tables [**?**]. It requires $(2^{n/3} \times n) + (2^{2n/3-1} \times n/3)$ look-up table size, which is about 928K*bits* in single precision floating-point format. It could be considered as expensive for silicon area. Therefore, the bipartite methods are only suitable for very low precision computations of less than 16 bits of accuracy.

The expression of the bipartite method is an addition and the underlying approximation is a linear polynomial, since the term C_1 encloses a multiplication. The result of this multiplication is stored in a look-up table, in order to avoid multiplication operations in the hardware. However, this is done at the expense of significantly increasing the size of the look-up tables for storing the C_1 and it must be addressed by both Y_1 and Y_3. The critical path of these methods is composed of the look-up tables and the n bit addition or booth recoding, the main contribution to the area accumulates from the look-up tables.

Linear approximations

In the piecewise linear approximations, the input operand Y is split into two fields, $Y = Y_1 + Y_2$, the upper and lower parts each having around $n/2$ bits.

An approximation to the function $f(Y)$ in the range $Y_1 \leq Y < Y_1 + 2^{-n/2}$ can be obtained by a first-degree Talyor approximation at the mid-point, $(Y_1 + 2^{-n/2-1})$ [**?**]:

$$f(Y) \approx C_1(Y_1).Y_2 + C_0(Y_1) \tag{2.5}$$

The coefficients C_1 and C_0 depend only on Y_1 and are stored in look-up tables addressed by $n/2$ bit word. After reading the table look-up coefficients, a multiplication and an addition must be performed to evaluate the polynomial.

The input interval is divided into $2^{n/2}$ subintervals and a linear approximation of the function is performed at the centre of each subinterval. The linear approximation is evaluated in terms of the lower part Y_2 by a multiplication and an addition.

The size of the tables to be employed is about $2^{n/2-2} \times (2(n/2-2)+4) + 2^{n/2} \times 5(n+3)/3$ bits for single precision floating point format. This is about 75K*bits*. The size of the multiplication is about $(n+5) \times (n+3)/3$ bits and the final $2n+4$ bit adder.

Second-degree approximations

In the piecewise quadratic approximation, the input operand Y is split into two parts, as in piecewise linear approximations: $Y = Y_1 + Y_2$. In this case Y_1 is about $n/3$ bits wide, while Y_2 has a word length of $2n/3$ bits and the function $f(Y)$ is approximated by a degree-2 polynomial:

$$f(Y) \approx C_2(Y_1).Y_2^2 + C_1(Y_1).Y_2 + C_0(Y_1) \tag{2.6}$$

The coefficients C_2, C_1 and C_0 depend only on Y_1 and are stored in look-up tables addressed by $n/3$ bit word. The size of the tables employed is therefore smaller than in piecewise linear approximations, which is about $2^{n/3} \times (n + 2n/3 +$

$n/3)$ resulting in around 15K*bits* for single precision computations.

The quadratic polynomial is usually evaluated in a two step process [**?**, **?**]:

$$g = C_2 Y_2 + C_1 \tag{2.7}$$

$$f(Y) = g Y_2 + C_0 \tag{2.8}$$

The advantage of using second degree approximations is a smaller table size.

In most functional iterative methods, these approximations are used as an initial seed for the quadratic or linear convergent algorithms. The functional iterative methods are discussed in the following section.

2.2 Iterative methods

Functional iterative methods are based on the multiplication operation and typically have quadratic convergence, which results in low latency algorithms, especially for high precision computations.

Because multiplication is the fundamental operation, these methods can be implemented in software by reusing the existing multiplier with some extra logic. This produces an important reduction in area, but may lead to performance degradation.

2.2.1 Goldschimdt algorithm

Let us assume two n bits[1] inputs Y and X satisfying $1 \leq Y, X < 2$. The GLD algorithm [**?**, **?**, **?**, **?**, **?**] for computing the division operation (Q= X/Y) consists of

[1] $n = 24$ for single precision and $n = 53$ for double precision.

finding a sequence $K_1, K_2, K_3, \ldots$ such that

$$r_i = Y K_1 K_2 \ldots K_i \longrightarrow 1 \tag{2.9}$$

and therefore

$$q_i = X K_1 K_2 \ldots K_i \longrightarrow \frac{X}{Y} \tag{2.10}$$

The reciprocal (Q=1/Y) can be computed as specific case of the division:

$$q_i = K_1 K_2 \ldots K_i \longrightarrow \frac{1}{Y} \tag{2.11}$$

The first factor K_1 is usually obtained as a low precision approximation of the reciprocal. If $r_i = Y \prod_i K_i$ has the form $r_i = 1 - \alpha$, then it is possible to obtain $K_{i+1} = 1 + \alpha$, by a simple two's complementing r_i, leading to $r_{i+1} = 1 - \alpha^2$, which guarantees the convergence of the algorithm.

In summary, the steps to be performed are the following:

- The first factor K_1 is a low-accuracy approximation of the reciprocal $1/Y$, and can be obtained by a table-based method, such as direct table look-up, bipartite table algorithms or polynomial approximation. Let this value have m-bit accuracy, i.e.:

$$1 - 2^{-m} < K_1 Y < 1 + 2^{-m} \tag{2.12}$$

- Define $v = 1 - K_1 Y$. $|v| < 2^{-m}$.

  -Computation of $r_1 = Y K_1 = 1 - v$.
  -Computation of $q_1 = X K_1$ (this multiplication is not necessary for the reciprocal computation).

- By 2's complementing r_1, $K_2 = 1 + v$ is obtained.

-Computation of $r_2 = r_1 K_2 = 1 - v^2$.
-Computation of $q_2 = q_1 K_2$.

- By 2's complementing r_2, $K_3 = 1 + v^2$ is obtained.

  -Computation of $q_3 = q_2 K_3$. At this point, $q_3 < X/Y < q_3(1 + 2^{-8m})$.

The number of iterations required to obtain a result, accurate to a certain precision, is a function of the accuracy of an initial approximation (seed) value. By using a more accurate seed value, the total number of iterations can be reduced.

Square root/inverse square root computation-Let us assume an n-bit input operand Y satisfying $1 \leq Y < 2$. In this case a sequence $K_1, K_2, K_3, \ldots$ should be found such that:

$$K_1, K_2, \ldots K_i \longrightarrow \frac{1}{\sqrt{Y}} \tag{2.13}$$

and therefore

$$q_i = Y K_1, K_2, \ldots K_i \longrightarrow \sqrt{Y} \tag{2.14}$$

The steps to be performed in the square root or inverse square root computation are:

- ▷ Let $Y = 1.d_1, d_2 \ldots d_{n-1}$ and define $\widehat{Y} = 1.d_1, d_2 \ldots d_m$ where $m << n$. Two look-up tables can be employed to obtain both $K_{1r} = 1/\widehat{Y}$ and $K_1 = 1/\sqrt{\widehat{Y}}$ with m-bit accuracy. These tables must be designed so that each table look-up value K_{1r} corresponds at full target accuracy to the square pf the table look-up value K_1. Another way of computing K_{1r} and K_1 is obtaining K_1 with a table driven method (which leads to an important area reduction regarding direct table look-up) and then computing the squaring operation $K_{1r} = (K_1)^2$.
- ▷ The low precision approximations K_{1r} and K_1 are then employed

in some calculations:

-Computation of $r_1 = YK_{1r}$.

-Computation of $q_1 = YK_1$ (only for the square root computation; for inverse square root computation, $q_1 = K_1$).

- ▷ $v_1 = 1 - r_1$ and compute the independent multiplications:

  -Computation of $q_2 = (1 + \frac{v_1}{2})q_1$.

  -Computation of $(1 + \frac{v_1}{2})^2 = (1 + \frac{v_1}{2})(1 + \frac{v1}{2})$.

  -Computation of $r_2 = (1 + \frac{v_1}{2})^2 r_1$.

- ▷ Define $v_2 = 1 - r_2$ and compute the independent multiplications:

  -Computation of $q_3 = (1+\frac{v_2}{2})q_2$. At this point, $q_3 = \sqrt{Y}(1+\alpha)$, when computing the square root function, and $q_3 = (1+\alpha)/\sqrt{Y}$, when computing inverse square root, with $|\alpha| < 2^{8m-2}$.

The total number of iterations to be performed again depends on both the target precision and on the accuracy of the initial approximation. One of the main drawbacks of using the functional iterative method is the difficulty in obtaining a correctly rounded result.

2.2.2 Newton-Raphson Algorithm

In the N-R algorithm [**?**], a primitive function is chosen that has a root at the reciprocal or inverse square root, depending on the function to be computed. The selection of this primitive function is based on the convenience of the resulting iterative form.

Division/Reciprocal computation, the most widely used target root, is the divisor reciprocal 1/Y, which is the root of the primitive function.

$$f(q) = \frac{1}{q} - Y = 0 \tag{2.15}$$

The well known quadratically converging N-R equation is

$$q_{i+1} = q_i - \frac{f(q_i)}{f'(q_i)} \tag{2.16}$$

Applying this equation to the primitive function 2.15, gives the resulting iterative formula used to compute the reciprocal:

$$q_{i+1} = q_i \times (2 - Y \times q_i) \tag{2.17}$$

with quadratically converging error:

$$\epsilon_{i+1} = Y\epsilon_i^2 \tag{2.18}$$

Each iteration involves two multiplications and a subtraction, which is commonly replaced by a two's complement operation. The computation of the division requires a final multiplication by the dividend ($X/Y = X \times (1/Y)$). The multiplications to be performed are dependent operations and therefore cannot be computed in parallel, which leads to performance degradation.

Square root/inverse square root computation- Applying the Newton-Raphson equation to the primitive function:

$$f(q) = \frac{1}{q^2} - Y = 0 \tag{2.19}$$

the following iteration formula is obtained

$$2q_{i+1} = q_i \times (3 - Y \times q_i^2) \tag{2.20}$$

with quadratically converging error:

$$2\epsilon_{i+1} = (3\sqrt{Y}\epsilon_i^2 + Y\epsilon_i^3) \tag{2.21}$$

Each iteration involves three multiplications and the computation of the square root requires a multiplication by the operand Y, since the N-R algorithm converges to an inverse square root.

2.3 Digit-recurrence Algorithm

The algorithm belongs to the same type of initial approximation of the functions in the hardware, usually known as *digit-by-digit* iterative methods due to their linear convergence. This means that a fixed number of bits of the result are obtained in each of the iterations. Implementation of this type of algorithm, typically of low complexity, utilizes a small area and relatively large latencies. The fundamental choices in the design of a digit-by-digit algorithm are the radix, the allowed coefficients or digits and the representation of the partial remainder (residual).

2.3.1 Digit-recurrence division

Digit-recurrence algorithms use subtractive methods to calculate quotients, one digit per iteration. SRT *division* is the name of the most common digit-recurrence division algorithm [**?**, **?**]. The quotient is defined to comprise N radix-*r* digits, with

$$r = 2^b \tag{2.22}$$

and

$$N = \lceil n/b \rceil \tag{2.23}$$

Such a division algorithm requires N iterations to compute the final n-bit result, since b bits of quotient are retired per iteration.

The following recurrence is used in every one of the iterations of the SRT algorithm:

$$W[i+1] = rW[i] - Yq_{i+1}, \tag{2.24}$$

where q_{i+1} is the $(i+1)^{th}$ digit of quotient, $q[i]$ is the quotient in step i, and W[j] is the partial remainder in step i, with an initial value of $W[0] = X$.

In each of the iterations, one digit of the quotient is determined by the selection function

$$q_{i+1} = SEL(W[i], Y) \tag{2.25}$$

The final quotient after N iterations is then

$$Q = \sum_{i=1}^{N} q_i r^{-i} \tag{2.26}$$

Choice of radix-The fundamental method for decreasing the overall latency of the algorithm is to increase the radix *r*, typically chosen as a power of 2 in order to perform the products by *r* as shifts of the operands. To achieve the same precision, the number of iterations required to compute the final result will be reduced. However, the reduction has some side effects, such as increasing the complexity of the selection function, increasing the area and, in most cases, also increasing the cycle time. Therefore, an analysis of trade-offs between area and speed is necessary for determining the values of the radix *r*, which results in efficient implementation.

Choice of digit set-In the digit-by-digit algorithm, some range of digits must be chosen for the allowed values of the quotient in each of the iterations. The simplest case is where, for radix *r*, there are exactly *r* allowed values (non-redundant digit-set). However, to increase the performance of the algorithm, a redundant digit-set is usually employed. Such a digit-set can be composed of symmetric signed-digit consecutive integers, where the maximum digit is *a*:

$$q_i \in \{-a, -a+1, \ldots, -1, 0, 1 \ldots, a-1, a\} \tag{2.27}$$

Thus, to make a digit-set redundant, it must contain more than *r* consecutive

integer values including zero and *a* must satisfy

$$a \geq \lceil r/2 \rceil \tag{2.28}$$

The redundancy of a digit set is determined by the value of the redundancy factor ρ which is defined as

$$\rho = \frac{a}{r-1}, \quad \rho > \frac{1}{2} \tag{2.29}$$

When $a = \lceil r/2 \rceil$ the representation is called *minimally redundant*, while that with $a = r - 1$ is called *maximally redundant*, with $\rho = 1$. A representation is known as *non-redundant* if $a = (r-1)/2$, while a representation $a > r - 1$ is called *over-redundant*.

For the next residual W[i+1] to be bounded when a redundant digit-set is used, the value of the quotient digit must be selected so that:

$$\mid W[i+1] \mid < \rho \times Y \tag{2.30}$$

By using a large number of allowed digits *a* (a large value of ρ), the complexity and latency of the selection function can be reduced. However, choosing the smaller number of allowed digits simplifies the generation of multiples of the divisor Y. Multiples that are a power of two can be generated by simply shifting. If a multiple is required that is not a power of two (e.g. three), an extra operation, such as addition, may be required, adding to the complexity and latency of generating the divisor multiple. The complexity of digit selection and that of generating multiples of the divisor Y must be balanced.

After the redundancy factor ρ is chosen, it is possible to derive the selection function. A *containment condition* determines the selection intervals. A selection interval is the region in which a particular quotient digit can be chosen. These

expressions are given by

$$U_k = (\rho + k)X$$
$$L_k = (-\rho + k)X, \tag{2.31}$$

where $U_k(L_k)$ is the largest (smallest) value of residual $W[i]$, such that it is possible for $q_{i+1} = k$ to be chosen while still keeping the next partial remainder bounded.

However, when a high radix is used, the only practical method is selection by rounding [**?**, **?**, **?**, **?**]. Two alternatives have been used to allow selection by rounding: performing a scaling of recurrence [**?**, **?**] and performing selection by table in the first iteration until the convergence conditions are met [**?**].

Choice of residual representation-The residual can be represented in two different forms, either *redundant* or *non-redundant*. A conventional two's complement is an example of the non-redundant form, while CS two's complement and SD binary representation are examples of the redundant form. Additions and subtractions are involved in digit-recurrence algorithms. If these operations are performed in non-redundant form, they require a full width adder with carry propagation, increasing the cycle time. If the residual is computed in redundant form, a carry free adder, such as a CSA, can be used in recurrence, minimizing the cycle time. However, the complexity of the selection function increases and so, additionally, twice as many registers are required to store the residual between iterations.

It is possible to convert the quotient digits as they are produced, in order to avoid the extra cycle required to perform addition. This scheme is known as *on-the-fly* conversion [**?**]. It can also be extended to perform on-the-fly rounding of the result [**?**].

Combined Division and Square root Computation: for computing the square root, the employed recurrence is

$$W[i+1] = rW[i] - f[i]q_{i+1} \tag{2.32}$$

By defining $f[i] = 2q[i] + q_{i+1}r^{-(i+1)}$ for the square root and $f[i] = Y$ for the division, a single unit for both recurrences can be obtained. A common set of selection constraints must be valid for both operations, and $f[i]$ must be generated without adding extra delay to the critical path of the division recurrence.

2.3.2 Accurate Quotient Approximation method

Digit recurrence algorithms are applicable to low radix division and square root implementation, as radix increases the quotient digit selection hardware becomes complex, increasing the cycle time, area or both. AQA is a variant of a digit recurrence algorithm, for achieving a very high radix division/square root with an acceptable increase in cycle time, area and precise rounding with simpler quotient digit selection hardware. The term "very high radix" is applied when large numbers of bits are retired in the iteration.

The high radix algorithm was independently proposed by: Wong and Flynn [**?**, **?**], and Matula [**?**, **?**]-it is carried out in Cyrix processor. Both these algorithms are identical.

The algorithm is as follows:

1. Initially, set the quotient $Q = 0$, $P = X$ for the division ($Q'_i = X/Y$) and $P = Y$ for the square root ($Q'_i = Y/\sqrt{Y}$). Then, obtain an approximation of $1/Y_h$, $1/\sqrt{Y_h}$ from a look-up table, using the top m bits of Y, returning an h bit approximation. Only $m-1$ bits are actually required to index into the table, as the guaranteed leading bit can be assumed.

2. Scale the truncated divisor by the reciprocal approximation and inverse square root approximation and obtain the partial quotient for the square

root. Similarly, scaling the dividend by the reciprocal approximation partial quotient can be used for the division.

$$\begin{array}{cc} Y' = (1/Y_h) \times Y & q_1 = P \times 1/\sqrt{Y_h} \\ q_i = P \times 1/Y_h \ldots (i \geq 1) & q_i = P/2 \times 1/\sqrt{Y_h} \ldots (i \geq 2) \\ \ldots \textit{division} & \ldots \textit{square root} \end{array} \quad (2.33)$$

The Y' is invariant across the division iterations, therefore, only needs to be performed once. Subsequent iterations use only one multiplication:

$$Y' \times P,$$

The product $P \times 1/Y_h$ and $P \times 1/\sqrt{Y_h}$ can be viewed as the next partial quotient bits.

3. Perform the general recurrence to obtain the next partial remainder:

$$P'_i = P - P \times Y' \ldots \text{for division} \quad (2.34)$$

$$P'_i = P - q_i \times (2Q + q_i) \ldots \text{for square root} \quad (2.35)$$

4. Compute the new quotient as

$$Q'_i = Q + q_i \ldots \text{for division} \quad (2.36)$$

$$Q'_i = Q + q_i \ldots \text{for square root} \quad (2.37)$$

The new quotient is then developed by forming the product $P \times (1/Y_h), P \times (1/\sqrt{Y_h})$ and adding the result to the old quotient Q.

5. The new partial remainder P'_i is normalized by left shifting to remove any leading 0s.

6. Variables are adjusted such that $Q = Q'_i$, and $P = P'_i$.

Initialization	$Y = 1.546949120382167$. $1/\sqrt{Y_{19}} = 0.804010$. $Q_0 = 0, P = Y$, $q_1 = P \times 1/\sqrt{Y_{19}}$ for $i = 1$. $q_i = P/2 \times 1/\sqrt{Y_{19}}$ for $i \geq 2$. $P'_i = P - q_i \times (2Q + q_i)$, $Q'_i = Q + q_i$
$i = 1$	$q_1 = 1.2437625622784660$, $P'_1 = 3.8090566717 \times 10^{-6}$, $Q'_1 = 1.2437625622784660$
	$Q = Q'_1, P = P'_1$
$i = 2$	$q_2 = 1.5312598273 \times 10^{-6}$, $P'_2 = 7.0342 \times 10^{-12}$, $Q'_2 = 1.2437640935382933$
	$Q = Q'_2, P = P'_2$
$i = 3$	$q_3 = 2.827783571 \times 10^{-12}$, $P'_3 = 2.165453955401 \times 10^{-16}$, $Q'_3 = 1.243764093541121083571$
	$Q = Q'_3, P = P'_3$
$i = 4$	$q_4 = 8.705233115169 \times 10^{-17}$, $P'_4 = 2.6280490 \times 10^{-21}$, $Q'_4 = 1.2437640935411212$

Table 2.2: An example of Cyrix processor square root algorithm

7. Repeat steps 2 through 6 of the algorithm for $i \geq 2$ iterations.

In this manner, the quotient bits are generated in series along with intermediate partial quotients and exact remainders. At the end of the process, the exact remainder of a full precision partial quotient is available for the rounding process. This scheme guarantees $h - 2$ bits in each iteration.

An example of Cyrix processor square root algorithm is depicted in Table 2.2 (for $h = 19$ bit initial approximation) in radix-10. It requires four iterations to obtain a double precision result. The Cyrix processor multiplier only produces limited precision result of 19 bits. Because of a specially chosen 19 bit reciprocal, along with the 19 bit quotient digit and 18 bit accumulated partial remainder, it guarantees 17 bits of the quotient in every iteration.

To guarantee an exactly rounded result, most of the discussed algorithms provide an accuracy of ± 0.5*ulp*, as does the proposed method. The details of the rounding operation can be obtained in section 3.3 of the next chapter or in section 4.5 of Chapter 4.

Chapter 3

Modified N-R Algorithm without a LUT

In this chapter we propose a modified Newton-Raphson reciprocation algorithm without using LUT*, a linear convergence algorithm (*N-R *quadratic convergence) for single precision computation. The proposed method requires a maximum of 22 iterations, the number of iterations depends on the input operand leading to a variable latency algorithm. Initial approximation is a two's complement of the divisor, which can be performed during partial products summation. Each iteration is a cycle, performing multiply-add and Booth recoding in the same cycle of partial products summation. Multiples of the divisors are constant throughout iteration. It is only generated once, in the second iteration. To evaluate the proposed method, we used a radix-8 multiplier of G5* FPU *and a radix-4* MAF *of IBM* $603e^{TM}$ FPU*, and we then compared it with state-of-the-art processors. The comparison shows that it requires minimum hardware requirements with average performance. To maintain the average performance, intermediate results are computed in* CS *form. The proposed method offers some advantages over the* N-R *method and* SRT *method. This is detailed in this chapter.*

3.1 N-R Algorithm

In the N-R algorithm [**?**], a primitive function is chosen that has a root at the reciprocal or inverse square root, depending on the function to be computed. The selection of this primitive function is based on the convenience of the resulting iterative form. The N-R algorithm could also be used as a GLD algorithm, both algorithms are similar but the arrangement of the equations in the hardware differs.

Division/Reciprocal computation, the most widely used target root, is the divisor reciprocal 1/Y, which is the root of the primitive function. The following recurrences are performed

$$t = \frac{1}{Y}$$

$$t[0] = \frac{1}{\widehat{y}} \ldots \text{initial approximation} \tag{3.1}$$

$$W[i] = Y \times t[i] \tag{3.2}$$

$$S = 1 - W[i] \tag{3.3}$$

$$t[i+1] = t[i] + S \times t[i] \tag{3.4}$$

with quadratically converging error:

$$\epsilon_{t[i+1]} = Y.\epsilon_i^2 \tag{3.5}$$

Each iteration involves a multiplication, a subtraction and a multiply-add operation. The computation of the division requires a final multiplication by the dividend ($X/Y = X \times (1/Y)$). The multiplications to be performed are dependent operations and therefore cannot be computed in parallel.

By modifying a step in the above eqn.3.4, the multiply-add operation is replaced by just an addition operation, recalling the above equations

$$W[i] = Y \times t[i] \tag{3.6}$$
$$S = 1 - W[i] \tag{3.7}$$
$$t[i+1] = t[i] + S \tag{3.8}$$

with linear converging error (discussed in section 3.3):

$$\epsilon_{t[i+1]} = \overline{Y.\epsilon_i} \tag{3.9}$$

The significance of this modification is discussed in section 3.4.

3.1.1 Algorithm

The proposed method uses the following steps for performing reciprocation:

$$t = \frac{1}{Y}$$
$$t[0] = 2 - Y \ldots \text{initial approximation} \tag{3.10}$$
$$W[i] = Y \times t[i] \tag{3.11}$$
$$S = 1 - W[i] \tag{3.12}$$
$$t[i+1] = t[i] + S \tag{3.13}$$

1. Initialization for approximator ($t[0]$). To initialize we need to subtract the operand from 2 (two's complement in binary)-$t[0]$.

2. Multiply the operand with the approximator-$W[i]$ and subtract the result from 1 (one's complement in binary)-S. This gives the position of the number to be added to the approximator.

$t = 1/Y$	$X = 0.8123462\, Y = 0.5649120$
$t[0] = 2 - Y$ . . . . . initial approximation	$t[0] = 1.4350880$
$W[i] = Y \times t[i]$	$W[0] = 0.8106984$
$S = 1 - W[i]$	$S = 0.1893016$
$t[i+1] = t[i] + S$	$t[1] = 1.435088 + 0.1893016 = 1.6243896$
	$W[1] = 0.9176371$
	$S = 0.0823628$
	$t[2] = 1.7067524$
	.
	$t[3] = 1.7425875$
	.
	$t[4] = 1.7581789$
	.
	.
	$t[22] = 1.7701872$
	$Q = X \times t[22]$
	$Q = 0.8123462 \times 1.7701872 = 1.4380048$

Table 3.1: Proposed Reciprocation algorithm without a LUT

3. Add this result to that of the approximator ($t[i] + S$) and convert CS to Booth digit representation (discussed in next section).

4. Repeating steps 2 to 3 i.e. eqn.3.11 to eqn.3.13 twenty two times ($t[i+1]$) produces the reciprocal function.

To obtain division, multiply the reciprocal of the divisor by the dividend. As mentioned, the sign and the exponent can be computed in parallel with the mantissa of the quotient. We have focused on the mantissa computation in this thesis.

An example is depicted in Table 3.1. The IEEE standard requires inputs in the range $1 \leq X, Y < 2$. The proposed method operates within the range $0.5 \leq X, Y < 1$, which is a right shift of the operands from the most significant bit to the least significant bit of the input operands. To obtain an IEEE mandated

result in the range $0.5 < t \leq 1$ for reciprocal, a right shift in the final iteration (before normalization) is required. Note that a right shift is not required for the division function to obtain a IEEE mandated result in the range $0.5 < Q < 2$.

3.2 CS Booth digit representation

In our design $t[i]$ is used as the multiplier. In the radix-8 multiplier [**?**], we generate 10 PP's of Y, they are accumulated in four levels of 3:2CSA and in another level of 3:2CSA, $t[i] + S$ operation is performed. It requires five levels of 3:2CSA's out of six levels. Using the $t[i]$ operand as a CS multiplier, the final sixth level of CSA tree [**?**] is unused and we use this final level of CSA tree to convert CS–Booth-3 digit representation for radix-8 multiplier.

In the radix-4 [**?**] multiplier, using $t[i]$ as a multiplier we generate 15 PP's of Y, it requires three levels of 4:2CSA and an additional 4:2CSA is required for adding $t[i] + S$. It consumes all levels of 4:2CSA in the radix-4 multiplier. An extra level of 3:2CSA is required to convert CS–Booth-2 digit representation.

The $t[i]$ multiplier in CS form can maintain the average performance of the division with respect to the state-of-the-art processors.

To avoid the general increase in hardware size due to redundant binary input, attention has been focused on the redundant input to the multiplier recoder input [**?**, **?**]. We used the technique of [**?**] because it requires minimal circuitry for redundant binary output for forwarding and feedback.

Compression from CS–redundant Booth-digit representation

Let $t' = t'_{n-1:0} = (t'_{n-1}, \ldots t'_0) \in \{0,1\}^n$. We denote then by $\langle t' \rangle = \sum_{i=0}^{n-1} t'_i.2^i$ the value represented by t'. Looking at the multiplier input of a radix-4 multiplier each Booth digit t'_{2i} is computed from three consecutive bits $t'_{2i+1}, t'_{2i}, t'_{2i-1}$ by the formula

$$t'_{2i} = -2t'_{2i+1} + t'_{2i} + t'_{2i-1} \tag{3.14}$$

If each tripel $t'_{2i+1}, t'_{2i}, t'_{2i-1}$ substituted by a tripel $(rt'3_i, rt'2_i, t'y1_i)$, the value of the multiplier changes to

$$\langle t \rangle = \sum_{i=0}^{m'-1} (-2.rt'3_i + rt'2_i + rt'1_i).4^i \tag{3.15}$$

Therefore, one can define the set of tripels $(rt'3_i, rt'2_i, rt'1_i)$, with $0 \leq i < m'$ to be a *redundant Booth-2 digit representation* of $\langle t' \rangle$, iff $\langle t \rangle = \langle t' \rangle$. Where m' is a PP's reduction factor for multiples of the multiplicand with b bit multiplier ($b = 29$ bit in our case discussed in the next section) having $t'_{m+1} = t'_m = t'_{-1} = 0$ then $m' = \lceil (b+1)/2 \rceil$. Thus, it does not change the product if fed to the multiplier or the set of tripels $t'_{2i+1}, t'_{2i}, t'_{2i-1}$ based on the non-redundant representation of $\langle t \rangle$ *or* a redundant Booth-digit representation $(rt'3_i, rt'2_i, rt'1_i)$ of $\langle t \rangle$.

Initial approximation is a two's complement of the divisor Y. To obtain $t[1]$, first making a one's complement of the divisor and thereafter performing a one's complemented multiple generation of $t[0]$. Then, performing a PP's summation of $t[0]$ using Y as the multiplier, during the PP's summation, the constant 1 can be added. During 21 iterations (i.e. to generate $t[i+1]$), the divisor Y is used as a multiplicand and $t[i]$ as a multiplier. Truncating the intermediate result, $t[i]$, at the 29^{th} position, deletes the fraction portion $t_{30}, t_{31} \ldots$, keeping a value in the fraction range (-1/2,1/2)*ulp*.

We assume to have a carry-save representation of $\langle t \rangle + const$, that already includes the additive constant $const = \sum_{i=0}^{m'-1} 2.4^i$. The $const$ is required to obtain t in eqn.3.15 form.

Looking at bit windows with a width of 2 in this carry-save representation of the number $\langle t \rangle + const$, each window contains 4 bits (see Figure 3.1); two with a weight of one and two with a weight of two. The binary value w_j of the part of the number within a window i is in the range $w_i \in \{0, \ldots 6\}$. The number $\langle t \rangle + const$ can then be written by:

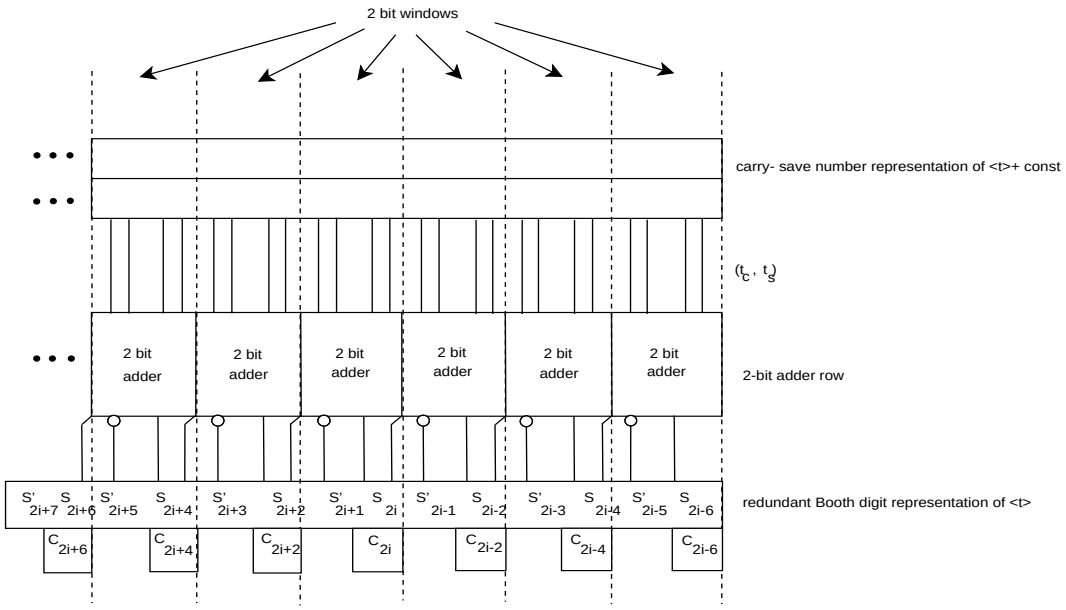
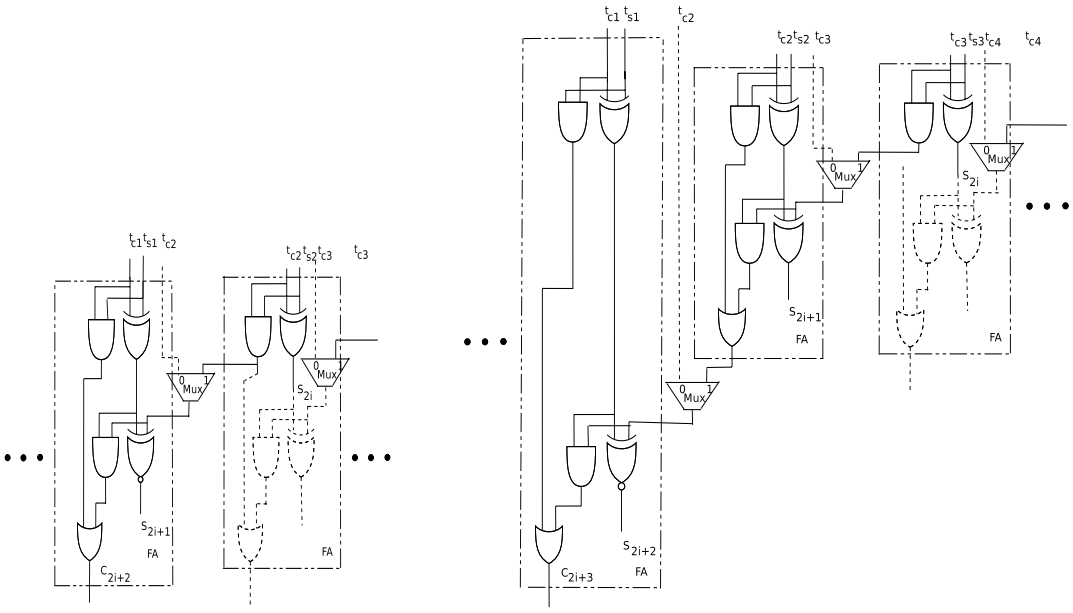


Figure 3.1: Compression from carry-save to redundant Booth-digit representation



(a) 2 bit addition in CSA (b) 3 bit addition in CSA

Figure 3.2: CS-Booth digit representation in a CSA cell

$$\langle t \rangle + const = \sum_{i=0}^{m'-1} w_i.4^i \tag{3.16}$$

If we input the 4 bits of a window i into a 2-bit adder, we get three output bits c_{2i+2}, s_{2i+1} and s_{2i}, that represent the value of the window by

$$w_i = 4.c_{2i+2} + 2.s_{2i+1} + s_{2i} \tag{3.17}$$

$$\langle t \rangle + const = \sum_{i=0}^{m'-1} (4.c_{2i+2} + 2.s_{2i+1} + s_{2i})4^i, \tag{3.18}$$

Subtract the additive constant *const* on both sides, and obtain the value of $\langle t \rangle$

$$\langle t \rangle + const - \sum_{i=0}^{m'-1} 2.4^i = \sum_{i=0}^{m'-1} (4.c_{2i+2} + 2(s_{2i+1} - 1) + s_{2i})4^i, \tag{3.19}$$

As $x - 1 \equiv -\overline{x}$ for $x \in \{0, 1\}$, then one can substitute $s_{2i+1} - 1$ by $-\overline{s_{2i+1}}$ and we can get

$$\langle t \rangle = \sum_{i=0}^{m'-1} (4.c_{2i+2} - 2\overline{s_{2i+1}} + s_{2i})4^i \tag{3.20}$$

In radix-4 recoding [**?**, **?**, **?**, **?**, **?**] the multiplier is recoded with $s_{2m'+1} = s_{2m'} = c_0 = 0$, then we have

$$\langle t\rangle + const = \sum_{i=0}^{m'}(2.s_{2i+1} + s_{2i} + c_{2i})4^i \tag{3.21}$$

Subtracting the additive constant *const* on both sides, and obtain the value of $\langle t\rangle$

$$\langle t\rangle + const - \sum_{i=0}^{m'-1} 2.4^i = \sum_{i=0}^{m'}(2.(s_{2i+1} - 1) + s_{2i} + c_{2i})4^i \tag{3.22}$$

$$\langle t\rangle = \sum_{i=0}^{m'}(-2\overline{s_{2i+1}} + s_{2i} + c_{2i})4^i \tag{3.23}$$

Therefore, the set of triples $(\overline{s_{2i+1}}, s_{2i}, c_{2i})$ is a redundant Booth-2 digit representation of $\langle t'\rangle$ and the sequence of operations described before is a partial compression from a carry-save representation of $\langle t\rangle + const$ to a redundant Booth 2 digit representation of $\langle t'\rangle$. The implementation of this partial compression is depicted in Figure 3.1.

In the radix-4 multiplier [**?**], the $const$ can be introduced by replacing the fourth level of 3:2CSA with 5:2CSA as shown in Figure 3.5 (on the right hand side). In the proposed method, performing CS-Booth-2 digit representation is just wire crossing in adjacent CSA cells. Figure 3.2 (a) depicts the CS–Booth-2 digit representation in a CSA for radix-4 multiplier.

Where t_{si}, t_{ci} are the intermediate sums and carries of t'. We assumed a 3:2CSA consisting of a combination of *X-or*'s, *OR*'s and *AND* gates, however it can be built with *NAND* or *NOR* gates.

In the radix-8 multiplier, the constant is $const = \sum_{i=0}^{m'-1} 4.(8)^i$, the addition of $const$ to t (i.e. $\langle t\rangle + const$) can be performed by replacing the fifth level

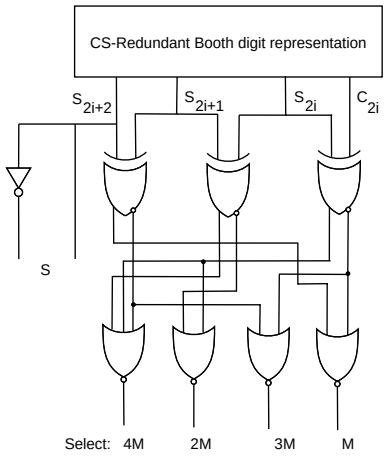


Figure 3.3: CS-Booth recoding

of 3:2CSA with 4:2CSA (the 3:2CSA requires two sums and a carry from the previous level, by replacing 3:2CSA with 4:2CSA we introduce another input which is $const$). A row of 3 bit adders are required for partial compression from CS representation of $\langle t \rangle + const$ to redundant Booth-3 digit representation of $\langle t' \rangle$, which is a wire crossing in an adjacent CSA cell, Figure 3.2(b) depicts the redundant Booth-3 digit representation in the final sixth level of 3:2CSA. An extra radix-8 recoder is required at the end of the CSA tree, in order to recode in the same pipe line stage of PP's summation. In the radix-8 multiplier, the recoder is coupled with a $3M$ multiple generation, which is another pipe line stage [**?**]. Figure 3.3 depicts CS-radix-8 Booth recoder. The results of this recoder can be directly applied to the multiplexer to select the multiplicands.

In the radix-4 multiplier, an extra radix-4 recoder [**?**] is not required, because the radix-4 recoder of the multiplier is placed in the same pipe line stage of the PP's summation. More details on the proposed method of architecture can be obtained in Section 3.4.

Note that, the multiplexers in Figure 3.2 (a)& (b) may not add additional delay in the cycle time, because they can be selected before the start of the PP's summation. In the case of the radix-4 multiplier, it can be selected at the start of the first stage and in the case of the radix-8 multiplier; it can be selected at the start of

the second stage.

This partial compression can easily be extended to the carry-save representation to higher radix Booth recoding multipliers. For compressions to redundant radix-2^k Booth digits, *const* has to change to $const = \sum_{i=0}^{m'-1} 2^{k-1}.(2^k)^i$ and a row of k-bit adders has to be used for partial compression.

3.3 Error Computation

For single precision computation, an estimate accurate to $n+3$ bits before rounding must be obtained. In principle, a result accurate to $n+1$ bits before rounding would suffice to guarantee exact rounding. However, when a multiplier and adder are used in the iterations, performing either truncation or rounding at $(n+1)^{th}$ position of the intermediate result, returns less than n bits of precision. In this case, an accuracy of $n+3$ bits of the estimate is required instead of $n+1$.

In order to guarantee an exactly rounded result, we performed exhaustive simulation of the reciprocation algorithm for all 2^{24} arguments. We obtained an accuracy of ± 0.5*ulp* (for 22 iterations). Under the described conditions, the result before rounding *t* satisfies:

$$-2^{-n-3} < t < 2^{-n-3} \tag{3.24}$$

$$-2^{-n-3} < Q < 2^{-n-3} \tag{3.25}$$

Rounding is performed by adding 2^{-n-3} to *t* and then truncating the resulting value to $n+2$.

Therefore, we must guarantee $\epsilon_{ti+1_{n+2}} \leq 2^{-26}$, since the range of results is $0.5 < t \leq 1$. Error computation for reciprocation can be set as:

$$\epsilon_{t[0]} = t[0] + \epsilon_0 \tag{3.26}$$

$$\epsilon_s = \overline{(t[i] + \epsilon_i).Y} + \epsilon_{sr} \tag{3.27}$$

$$\epsilon_{t[i+1]_{n+2}} = t[i] + \epsilon_i + \overline{t[i].Y} + \overline{\epsilon_i.Y} + \epsilon_{sr} + \epsilon_{tr} \tag{3.28}$$

where ϵ_{sr} and ϵ_{tr} are the error introduced by S and the $t[i+1]$ operation due to finite word length at n=29, $\epsilon_{sr}, \epsilon_{tr} < 2^{-29}$. ϵ_0 is the error introduced by the initial approximation at n=4. $\epsilon_0 < 2^{-4}$, therefore for *i* iterations $\epsilon_i < 2^{-26}$. Therefore, the error bound for S (for *i* iterations), with respect to truncation, is $\epsilon_s \leq 2^{-26}$.

The error in the final result must be bounded by:

$$\begin{aligned} \epsilon_{t[i+1]_{n+2}} &= \epsilon_i + \overline{\epsilon_i.Y} + \epsilon_{sr} + \epsilon_{tr} \\ &\leq 2^{-26} + 2^{-26} + 2^{-29} + 2^{-29} \end{aligned} \tag{3.29}$$

The result before rounding Q must satisfy eqn.3.25. Rounding can be performed by computing a corresponding remainder: $rem = X - Y \times Q$. By observing the sign and magnitude of *rem* and the value of *n+1*, all IEEE rounding modes can be implemented by selecting either $Q, Q + 2^{-n}$ or $Q - 2^{-n}$. The action table for correctly rounding Q is shown in Table 3.2 [**?**].

Guard Bit	Remainder	RN	RP(+/-)	RM(+/-)	RZ
0	= 0	trunc	trunc	trunc	trunc
0	-	trunc	trunc/dec	dec/trunc	dec
0	+	trunc	inc/trunc	trunc/inc	trunc
1	= 0	RNE	inc/trunc	trunc/inc	trunc
1	-	trunc	inc/trunc	trunc/inc	trunc
1	+	inc	inc/trunc	trunc/inc	trunc

Table 3.2: Action table for rounding

For directed rounding modes RP and RM, the action depends on the sign of estimate: those entries that contain two operations such as *pos/neg* correspond to the final result being positive or negative respectively. Most industrial multipliers

incorporate these modes.

3.4 Architecture and Performance

The architecture of the proposed method is, simply to introduce a radix-8 Booth recoder at the end of CSA tree in the radix-8 multiplier and an extra 3:2CSA is required for CS–Booth-2 representation in the radix-4 multiplier. We used a radix-4 multiplier of IBM Power PC $603e^{TM}$ FPU [**?**], which is targeted at the mobile computing market and a radix-8 multiplier of IBM G5 FPU [**?**] which is targeted at high performance applications.

The radix-8 multiplier (64×56 bit) is pipelined in three stages: in the first stage, generating $3X$ multiplicand and Booth recoding, in the second stage, 19 PP's summation in 6 levels of 3:2CSA, and the third and final stage is for carry propagate addition (see Figure 3.4).

The proposed method requires $24(Y) \times 29(t[i])$ bit multiplication and 10 PP's are accumulated out of 19 PP's. It requires 4+1 levels of CSA tree for the multiply add operation:

- ▷ The first 3 levels of 3:2CSA's for 9 PP's summation.
- ▷ In the 4^{th} level, 4:2CSA summation for 10^{th} PP, a carry, a sum and a carry from previous 3:2CSA's.
- ▷ That plus 1 level of 3:2CSA for adding $t[i] + S$ (not shown in Figure 3.4).

As mentioned, in the sixth level, CS–Booth-3 digit representation is performed with an extra radix-8 recoder at this sixth level. The results for this recoder can be directly applied to the 4:1 multiplexer to select the multiplicands. The two's complement for the initial approximation can be done by first making a one's complement of the multiplicand and then adding 1 during the PP's summation.

Note that, in the case of the radix-4 MAF (i.e. $24(B) + 24(A) \times 24(C)$ multiplied result added to another 24 bit operand say, $B\ (B + A \times C)$) [**?**], commonly

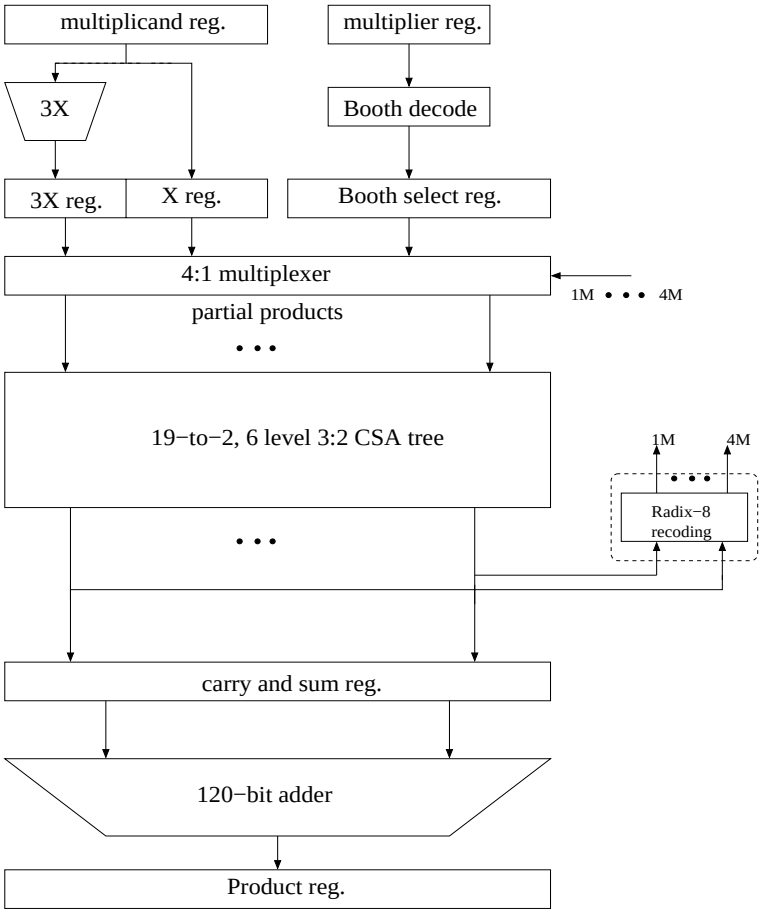


Figure 3.4: Proposed architecture in the radix-8 multiplier

implemented in industry due to its simplicity and ease in creating multiples of the multiplicand, a similar architecture and performance can be obtained to that of the radix-8 multiplier. This is an architecture with a three stage pipeline.

In the first stage, Booth recoding and 14 PP's summation in three levels of 4:2CSA tree. The CS result is added to the aligned B operand in the final fourth level 3:2CSA as shown in the first stage of Figure 3.5. The CS result is passed to the second stage, to obtain the result in non redundant form. The third stage is for normalization and rounding. The C is 28 bits, it can be made by concatenating four 0's for a single precision multiplier. It can also support double precision MAF . More details of this architecture are discussed in the next chapter.

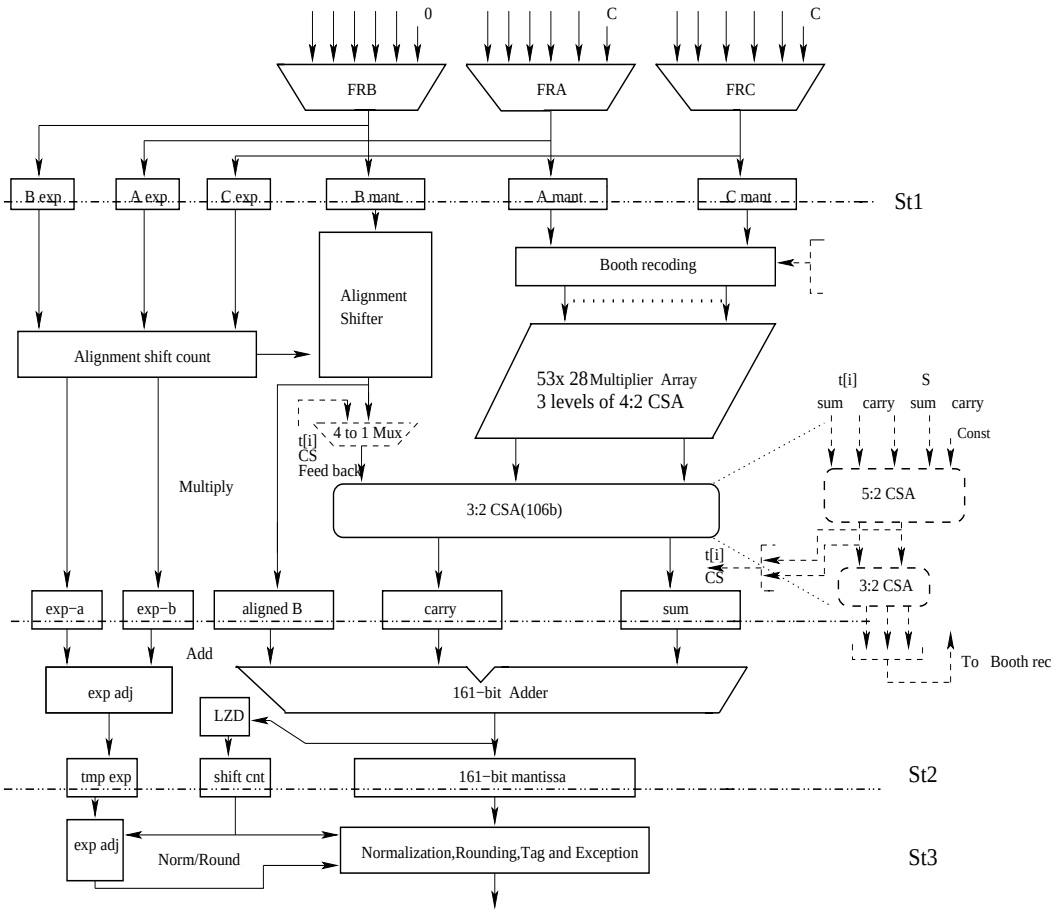


Figure 3.5: Proposed architecture in the MAF

With the $24(Y) \times 29(t[i])$ bit multiplication in the proposed method, the first stage is for Booth recoding and 15 PP's summation in three levels of 4:2CSA tree and in the final fourth level $t[i] + S$ is performed (see Figure 3.5 on the right hand dashed lines) by replacing 3:2CSA with 5:2CSA. To obtain the $t[i]$ in CS–Booth-2 representation, an extra 5^{th} level of 3:2CSA is required at the end of 5:2CSA of the fourth level in the first stage, as shown in Figure 3.5 with dashed lines. An extra 3 bit Booth-2 recoder to 14 bit Booth-2 recoder and an extra row of 5:1 multiplexer is required during multiplexing of the multiplicands to generate 15 PP's.

The advantages of the proposed method with respect to N-R method are that

the multiples of the divisor are constant in each iteration, it is generated only once, in the second cycle with respect to radix-4 multiplier and in the third cycle with respect to radix-8 multiplier, i.e. in the second iteration. The eqn.3.10-eqn.3.13 can be performed in one stage of the multipliers, which is a PP's summation stage. In a standard radix-4 multiplier, the eqn.3.10-eqn.3.13 are performed in the first stage and in a standard radix-8 multiplier they are performed in the second stage. The modification $t[i+1] = t[i] + S$ leads to each iteration of a cycle. The eqn.3.10-eqn.3.13 could be considered as a one equation in hardware terms, we can consider this as a multiplication. In the N-R method, the number of required operations are: a multiplication and multiply add operation-follow the eqn.3.2-eqn.3.4 and each of the iterations requires two cycles. To perform the multiply add operation (eqn.3.4), the multiplicand (S) should be in non-redundant form, it is possible to use the redundant multiplicand, but the data width will double. One could consider that the N-R method without a LUT similar to the proposed method but the performance is about the same. The N-R method requires 4 iterations to compute a single precision result. Each iteration requires two multiplications, which is about 6 cycles for each iteration (we assumed a multiplier latency of three cycles).

The proposed method without LUT is a variable latency algorithm. Variable latency algorithms are useful in speed independent design, they are also called self timed dividers, and self timed dividers execute the power according to the task at hand without sacrificing performance.

The proposed method with respect to the SRT method, does not require quotient digit selection hardware to select the next quotient digits, which increases the cycle time with respect to the radix.

Performance: The reciprocation/division operation is depicted in Figure 3.6:

- ▷ In the radix-8 multiplier [**?**], the first cycle is for Booth-3 recoding of the divisor (Y) and the generation of the one's complemented $3M$ multiple.
- ▷ In the second cycle, multiplexing of multiplicands and 10 PP's

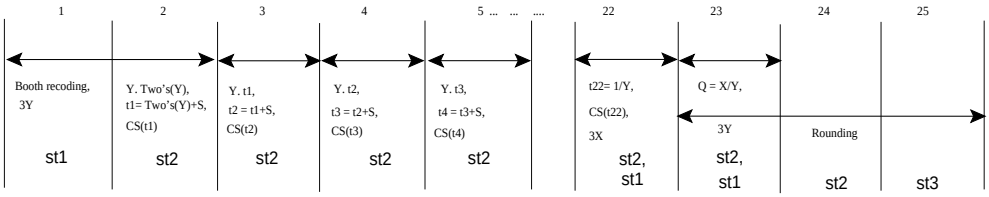


Figure 3.6: Timing Diagram

summation for $t[1]$ is performed i.e. passing through eqn.3.10-eqn.3.13.

- ▷ From the third cycle onwards, iterations through eqn.3.11-eqn.3.13 are performed until the twenty second cycle is reached. In the same twenty second cycle, an independent instruction for generating $\pm 3X$ multiples for the division operation is performed. This independent instruction can be performed because the radix-8 multiplier has a throughput of 1.
- ▷ In the twenty third cycle, a CS of $Q = X/Y$ is obtained and thereafter, another two cycles are used for rounding.

Thus, the result can be obtained in 25 cycles. Note that, during cycle four to twenty-one, we do not need to generate $0, \pm 1Y, \pm 2Y, \pm 3Y, \pm 4Y$ multiples as they are generated in the third cycle and can be stored. Thus, we can say that it is a constant.

In the case of the radix-4 multiplier, 24 cycles were required. It was reduced by one cycle, because the Booth recoding was performed during the PP's summation.

To use $t[i]$ as the multiplier, one has to replace the fourth level of 3:2CSA with 5:2CSA to add the $t + const$ in the radix-4 multiplier (see Figure 3.5)–following eqn.3.15-eqn.3.23. This can introduce delay in the cycle time. Similarly in the radix-8 multiplier, one has to replace the fifth level of 3:2CSA with 4:2CSA (not shown in Figure 3.4), this 4:2CSA can introduces delay in the cycle time. However, as the technology is scaling down, in the future the delay of 5:2CSA and 4:2CSA may be equivalent to that of 3:2CSA.

Note that to generate 15 PP's in the radix-4 multiplier, as mentioned, requires an extra 3 bit Booth recoder and an extra 5:1 multiplexer to the 14 bit Booth-2 recoder and a row of 5:1 multiplexers. This hardware can be introduced without additional delay in the cycle time.

The proposed method requires a maximum of 22 iterations. Note that in Figure 3.6 the CS of quotient is obtained in the 23^{rd} cycle. This is due to the fact that the radix-8 recoding of the multiplier is the first stage of the multiplier (first cycle). In the radix-4 multiplier, the CS of quotient Q is obtained in 22 cycles. The number of iterations depends on the divisor Y. For example, if the divisor is in the range $0.5 \leq Y < 0.5999999$ the iterations required are between 22 and 18. If the divisor is in the range $0.6 \leq Y < 0.6999999$, the iterations required are between 17 and 13. If the divisor is in the range $0.7 \leq Y < 0.7999999$, the iterations required are between 12 and 8, and so on. One can detect the exit of the loop eqn.3.11-eqn.3.13 by looking at the eqn.3.11. If the product of the divisor Y and approximator $t[i]$ is 0.9999999, which is a $W[i]$, then exit the loop. The approximator at this point has an accuracy of ± 0.5*ulp*. In hardware terms, a comparator is required in the PP's summation stage, for comparing $W[i]$ with $\underbrace{1111\ldots1}_{24\ bits}$ (0.9999999) during the iteration.

3.5 Modified N-R square root algorithm without a LUT

This modified N-R reciprocation algorithm can be extended to inverse square root. The $t[i]$ has to be squared in eqn.3.11 and a right shift for S in eqn.3.13. The modified equation for the inverse square root is shown below:

$t = 1/\sqrt{Y}$	$Y = 0.5649120$
$t[0] = 2 - Y \ldots \ldots$ initial approximation	$t[0] = 1.4350880$
$W[i] = Y \times (t[i])^2$	$W[0] = 1.1634236$
$S = 1 - W[i]$	$S = -0.1634236$
t$[i+1] = t[i] + S/2$	$t[1] = 1.435088 + (-0.1634236/2) = 1.3533762$
	$W[1] = 1.0347082$
	$S = -0.0347082$
	$t[2] = 1.3360221$
	.
	$t[3] = 1.3318508$
	.
	$t[4] = 1.3308229$
	.
	.
	$t[22] = 1.3304838$
	$Q = Y \times t[22]$
	$Q = 0.5649120 \times 1.3304838 = 0.7516063$

Table 3.3: Proposed Square root algorithm without a LUT

$$t = \frac{1}{\sqrt{Y}}$$

$$t[0] = 2 - Y \ldots \text{initial approximation} \tag{3.30}$$

$$W[i] = Y \times (t[i])^2 \tag{3.31}$$

$$S = 1 - W[i] \tag{3.32}$$

$$t[i+1] = t[i] + S/2 \tag{3.33}$$

An example is depicted in Table 3.3. The IEEE standard requires input within the range $1 \leq Y < 4$, the proposed method operates within the range $0.5 \leq Y < 1$. For the proposed method, dividing the range $1 \leq Y < 4$ into two intervals: $1 \leq Y < 2$ and $2 \leq Y < 4$. That is, a right shift is required, operands within the

range $1 \leq Y < 2$ and two right shifts are required the operands within the range $2 \leq Y < 4$ for the proposed method to operate within the range $0.5 \leq Y < 1$.

To obtain an IEEE mandated result $0.5 < t \leq 1$ for the inverse square root, multiply the *constant* $1/\sqrt{2}$ for the range $1 \leq Y < 2$ and a right shift for the range $2 \leq Y < 4$ in the final iteration (before normalization). To obtain an IEEE mandated result $1 \leq Q < 2$ for the square root, multiply this scaled inverse square root result with the unscaled dividend Y (the scaling is not shown in Table 3.3).

It requires the same maximum of twenty two iterations. Furthermore, the architecture is similar but the performance is adversely affected. It requires more than 50 cycles, because $t[i]$ has to be squared in each iteration. Each iteration requires at least 3 extra cycles compared to the reciprocal: one of the cycles is to obtain the $t[i+1]$ in non-redundant form and the remaining two cycles for the $(t[i])^2$, to obtain the result in CS form. In these two cycles, one cycle is for the Booth recoding and $3t[i]$ multiple generation, and the other cycle is to obtain $t[i]$ in CS form, with respect to the radix-8 multiplier. In the radix-4 multiplier, a cycle can be reduced due to the Booth recoding in the same cycle of PP's summation. This square root algorithm requires some more research.

3.6 Comparisons

We compare the proposed method with the conventional processors for single precision computation of division in Table 3.4. The rough model we used for area estimates was taken from [**?**]. The unit employed is the size of 1-bit *fa* (full adder), since the area of the main blocks, adders and multipliers can be easily expressed by this unit. A standard 1-bit full-adder has a hardware complexity equivalent to 9 *nand/nor* gates.

Look-up-tables: Estimates for the look-up table can be found in [**?**]. We assumed a pessimistic model. Our model assumes 40*fa*/Kbit rate for tables addressed up to 6 bits, a 35*fa*/Kbit rate for 7-11 input bit tables, 30*fa*/Kbit rate for

State-of-the-art Processors	Area estimates (*fa*)	Latency/Throughput
Intel *Itanium* [**?**, **?**, **?**]	Implemented in software	30/–
IBM G5 FPU[**?**, **?**]	≈ 90	27-30/27-30
AMD K-7 [**?**]	≈ 824	16/13
IBM $603e^{TM}$ [**?**]	–	18
Proposed Method (radix-8 [**?**])	≈ 6	25/23
Proposed Method(radix-4 [**?**])	≈ 31	24/22

Table 3.4: Comparison of the proposed method with Conventional Processors

12-13 input bit tables and a 25*fa*/Kbit rate for 14-15 input bit tables.

The G5 FPU uses about 8 bits in and 10 bits out of look-up (90 *fa*) and AMD 10 bits in and 16+7 out, which is about 824*fa* for reciprocal approximation. The estimated values presented here are reliable approximations, the actual speed-up and the area ratios depend on the technology employed and its implementation.

Note that, in the $603e^{TM}$FPU, a radix-4 SRT divider is used and shared with the multiplier. The proposed method can fit into a radix-4 MAF with extra hardware of about 31*fa*: 29*fa* for CS–Booth-2 digit representation, leaving 2*fa* for an extra Booth-2 recoder and 5:1 multiplexer for the 14 bit Booth recoder and for a row of 5:1 multiplexers, to generate 15PP's. The proposed method can fit into a radix-8 multiplier with extra hardware of about 6*fa*, this is an extra Booth recoder. Which is less than a 1.5% increase in area with respect to industrial multipliers. In the radix-4 MAF the proposed method is 1τ slower in cycle time–where τ is the delay of one full adder, due to another level of 3:2CSA for CS-Booth representation.

Note that, in the radix-4 MAF and in the radix-8 multiplier, we do not include the extra hardware for 5:2CSA and 4:2CSA, which are replaced by 3:2CSA, because as the technology is scaling down, the complexity of 5:2CSA and 4:2CSA may be equivalent to that of 3:2CSA.

There is about 16% improvement in the performance with respect to IBM and *Itanium* processors, but a 4% decrease with respect to IBM $603e^{TM}$ and AMD processors. This is because, 1.09 bits are retired per iteration and it is shared with

the multiplier and, also, a large LUT is not available i.e., the performance is about the same as the average performance of state-of-the-art processors.

3.7 Conclusion

Single precision computation of a modified Newton-Raphson reciprocation algorithm is proposed. It is a variable latency algorithm and only requires a maximum of 22 iterations providing a linear convergence in character. A variable latency algorithm is useful in speed independent design, also called a self-timed divider. The self-timed divider provides processing capability according to the task at hand. In this way, it can reduce power consumption without sacrificing performance.

The advantages of our method are that it does not require a look-up table and each of the iterations is a cycle, unlike the pairs of cycles found in the state-of-the-art algorithms. The performance of the proposed method is about the same as the average performance of state-of-the-art processors. To maintain the average performance, intermediate results are computed in CS form. The multiply-add operation and Booth recoding can be performed in the same cycle. Initial approximation in our case is a two's complement of the divisor, which can be performed during partial products summation. The multiples of the divisor are constant throughout each of the iterations. Therefore we generate them only once, in the second cycle with a radix-4 multiplier and in the third cycle with a radix-8 multiplier, i.e. in the second iteration. This leads to a reduction of a cycle in each iteration, unlike the N-R method. The eqn.3.10-eqn.3.13 could be considered as a multiplication in hardware terms.

In the N-R method the number operation required for reciprocation are: a multiplication and multiply-add operation (see eqn.3.2-eqn.3.4) and each of the iterations requires two cycles. The multiply-add operation requires the multiplicand (S) to be in non-redundant form. It is possible to use the redundant form, but the data width required is twice that of the non-redundant form, unlike the pro-

posed method. Without the use of a LUT for the N-R algorithm, the performance is about the same. This is because each iteration requires two multiplications, which is about 6 cycles (we assumed 3 cycles for a multiplication), it requires 4 iterations to obtain the single precision result.

With respect to the SRT method, the proposed method does not require a quotient digit selection hardware to select the next quotient digits, which increases the cycle time with respect to the radix.

We used both a radix-4 and a radix-8 multiplier. In a radix-8 multiplier, there may be a delay of an *OR* gate in cycle time, due to the extra radix-8 Booth recoder. In a radix-4 multiplier, it is 1τ slower in cycle time, due to another level of 3:2CSA for CS–Booth-2 digit representation.

We set-out a reciprocation/division algorithm with better silicon efficiency than conventional methods as shown in Table 3.4. The proposed method can fit into a radix-4 multiplier with extra hardware of about 31*fa* and about 6*fa* for a radix-8 multiplier, which is less than a 1.5% increase in area with respect to industrial multipliers.

There is about 16% improvement in performance with respect to IBM and *Itanium* processors, but 4% decrease with respect to IBM $603e$ and AMD processors, due to the fact that 1.09 bits retired per iteration, the multipliers are shared and a large LUT is not available i.e., performance is about the same as the average performance of state-of-the-art processors.

The proposed method can be further extended to double precision computation, which requires about 50 cycles. This can be reduced by employing an appropriate look-up table with market targeting.

It offers a good trade-off between performance and area making it suitable for the mobile computing market. In the mobile computing market area is crucial. Most architects look for a division algorithm with a low cost and reduced area, and an acceptable performance that can run internet applications, business applications such as audio and video streaming, charting, presentations, snapshot viewing, day to day accounting etc. It is expected that, with the proposed method, processor

architects will have an option for their next generation processors.

Chapter 4

Modified AQA Method Using a LUT

In this chapter we propose a division and square root using the Accurate Quotient Approximation Method for double precision computation. The proposed method employs a second degree minimax approximation to obtain an initial 15 bits estimate of reciprocal and inverse square root values. This is then passed through an iteration of Goldschimdt reciprocal and inverse square root algorithms, to enhance the initial approximation and to make initial approximation more silicon efficient. Thereafter we perform an iteration of the Accurate Quotient Approximation method to obtain a double precision result. To evaluate the proposed method, we used a radix-4 MAF *of IBM* $603e^{TM}$ *and a radix-8 multiplier of G5* FPU. *We then compared it with conventional processors and it shows that there is a significant improvement in performance with better silicon efficiency. To obtain the best performance, intermediate results are computed in* CS *form and some computations are overlapped. We used the* AQA *method because there are some advantages over the* N-R *method,* GLD *algorithm, Cyrix algorithm and* SRT *algorithm, it is detailed in this chapter.*

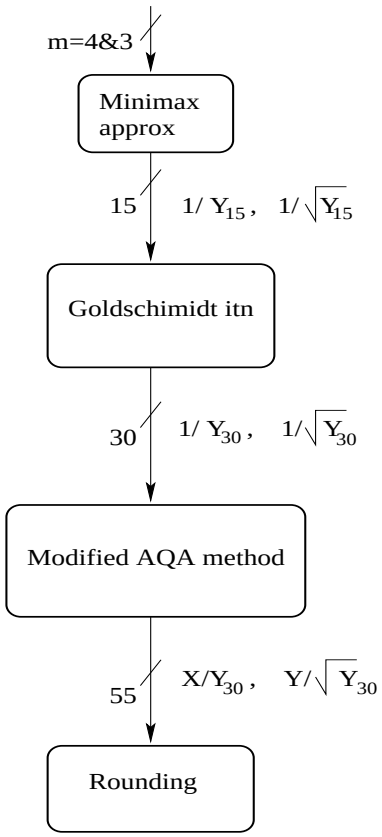


Figure 4.1: Proposed Concept

4.1 Proposed Method

Some steps of the AQA method can be modified for a single iteration algorithm, to obtain the double precision results, using high seed, 30 bits, for the reciprocal and inverse square root. Figure 4.1 exhibits the concept of the proposed method. We interface an initial 15 bit minimax approximation with GLD reciprocation and inverse square root algorithms to make the initial approximation silicon efficient. Interfacing this high seed 30 bits initial approximation with modified AQA method boosts the performance, allows the overlapping of partial remainder computation and, furthermore, offers more parallelism. A brief aspect of these features is depicted in Section 4.6.

Recalling eqn.2.34 - eqn.2.37 and algorithm steps 1-7 of subsection 2.3.2 for $i = 1$ and $2Q = 0$

$$P_1' = P - P \times Y' \qquad P_1' = P - q_1 \times (2Q + q_1)$$
$$P_1' = X - X \times Y \times (1/Y_{30}) \qquad P_1' = Y - Y^2 \times (1/\sqrt{Y_{30}})^2 \tag{4.1}$$
$$Q_1' = Q + q_1 \qquad Q_1' = Q + q_1 \tag{4.2}$$

$$\ldots \textit{division } (Q_i' = X/Y) \qquad \ldots \textit{square root } (Q_i' = Y/\sqrt{Y})$$

Where P is current partial remainder, Q is current partial quotient, P_1' is next partial remainder, Q_1' is next partial quotient and q_1 is next partial quotient bits.

One would substitute Q as X/Y_{30} for the division and $Y/\sqrt{Y}_{30}$ for the square root, assuming that these operations guarantee the first 28 bits of the quotient. Multiplying the next partial remainder P_1' with respective initial approximations i.e. $P_1' \times (1/Y_{30})$ for the division and $P_1'/2 \times (1/\sqrt{Y}_{30})$ for the square root can be viewed as a current partial quotient bits q_1 in our case, adding this partial quotient to the respective Q one can obtain a final Q_1'. Substituting these assumptions results in

$$Q_1' = (X) \times (1/Y_{30}) + (P_1') \times (1/Y_{30}) \ldots \textit{division}$$
$$Q_1' = (Y) \times (1/\sqrt{Y_{30}}) + (P_1'/2) \times (1/\sqrt{Y_{30}}) \ldots \textit{square root} \tag{4.3}$$

We require these assumptions because AQA is a linear convergent algorithm. I.e. $h - 2$ bits are generated in each iteration (28 bits in our case). By performing X/Y_{30} and $Y/\sqrt{Y}_{30}$ we can guarantee the first 28 bits of the quotient. To obtain the double precision result, we need to pass an iteration to obtain the lower 28 bits of the quotient, which is $P_1' \times (1/Y_{30})$ and $P_1'/2 \times (1/\sqrt{Y}_{30})$ for the respective functions. Therefore, the upper 28 bits of Q_1' and lower 28 bits of Q_1' can be performed in parallel.

Simplifying the eqn.4.1 and the eqn.4.3 both the next partial remainder and

the quotient can be viewed as

$$P'' = -Y \times X \times (1/Y_{30}) \qquad P'' = -Y^2 \times (1/\sqrt{Y_{30}})^2 \quad (4.4)$$

$$Q' = (2X + P'') \times (1/Y_{30}) \qquad Q' = (S + P''/2) \times (1/\sqrt{Y_{30}}) \quad (4.5)$$

... division ($Q' = X/Y_{30}$*)* where $S = Y + Y/2$ *... square root (*$Q' = Y/\sqrt{Y_{30}}$*)*

To allow a fair comparison, we assumed a 30 bit initial approximation for the Cyrix processor algorithm. The concept comparisons are explained using Figure 4.2, Table 4.1 and Table 4.2.

In the Cyrix square root algorithm, after obtaining the squared initial approximation $(1/\sqrt{Y_{30}})^2$ and Y^2 (initially $Q = 0$), they are multiplied firstly for iteration $i = 1$. Follow the algorithm steps 1 to 7 of subsection 2.3.2 and Figure 4.2. Then, this result is subtracted from Y ($P = Y$) for $i = 1$ and the result is stored in a register P'_i, this is a next partial remainder. During the multiplication of Y^2 with $(1/\sqrt{Y_{30}})^2$, an independent operation $q_1 = P \times 1/\sqrt{Y_{30}}$ is performed for $i = 1$, thereafter it is added to Q ($Q = 0$ for $i = 1$) and the result is stored in a register Q'_i, this is a next partial quotient. The next partial remainder and quotient are updated as a current partial remainder and quotient i.e. $P = P'_1$, $Q = Q'_1$ in the first iteration to allow iteration for the second iteration $i = 2$.

In the second iteration, the current partial remainder P is right shifted and then multiplied with $1/\sqrt{Y_{30}}$ to obtain q_2. Afterwards, the q_2 is added to a left shifted result of Q i.e. $2Q + q_2$. The result of this addition is then multiplied with q_2 (see eqn.2.35). These operations are shown in a dotted square box of Figure 4.2. The multiplied result is then subtracted from P. The result of this operation is the next partial remainder P'_2. The next partial quotient Q'_2 is obtained by adding q_2 to Q. An example of this algorithm using radix-10 is depicted in Table 4.1.

In the proposed method for the square root, after obtaining the squared initial approximation $(1/\sqrt{Y_{30}})^2$ and Y^2, they are multiplied. The result of this operation we call current partial remainder P''. Follow the eqn.4.4&eqn.4.5 and Figure

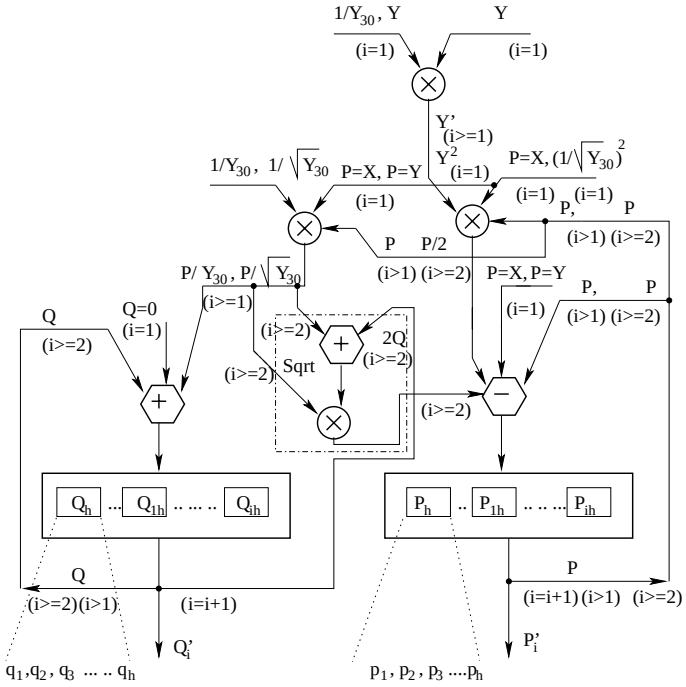
	Proposed Method	Cyrix Processor Square Root Algorithm
	$Y = 1.546949120382167$. $1/\sqrt{Y_{30}} = 0.804010989$. $(1/\sqrt{Y_{30}})^2 = 0.646433670432758121$. $Y^2 = 2.393051581051160208$. $S = Y + Y/2 = 2.3204236805732505$. $P'' = -Y^2 \times (1/\sqrt{Y_{30}})^2$, $Q' = (S + P''/2) \times (1/\sqrt{Y_{30}})$.	$Y = 1.546949120382167$. $1/\sqrt{Y_{30}} = 0.804010989$. $Q = 0, P = Y$, $q_1 = P \times 1/\sqrt{Y_{30}}$, $(1/\sqrt{Y_{30}})^2 = 0.646433670432758121$, $P^2 = 2.393051581051160208$ for $i = 1$. $q_i = P/2 \times 1/\sqrt{Y_{30}}$ for $i \geq 2$. $P'_i = P - q_i \times (2Q + q_i)$, $Q'_i = Q + q_i$. $P = P'_i, Q = Q'_i$
$i = 1$	$P'' = -1.546949117073816457$, $Q' = 1.2437640935411212$	$q_1 = 1.243764092211146147$. $P'_1 = 3.30835050 \times 10^{-9}$, $Q'_1 = 1.243764092211146147$
		$P = P'_1, Q = Q'_1$
$i = 2$		$q_2 = 1.32997500 \times 10^{-9}$. $P'_2 = 3.74683 \times 10^{-17}$, $Q'_2 = 1.2437640935411212$

Table 4.1: Comparison of Proposed method with Cyrix algorithm

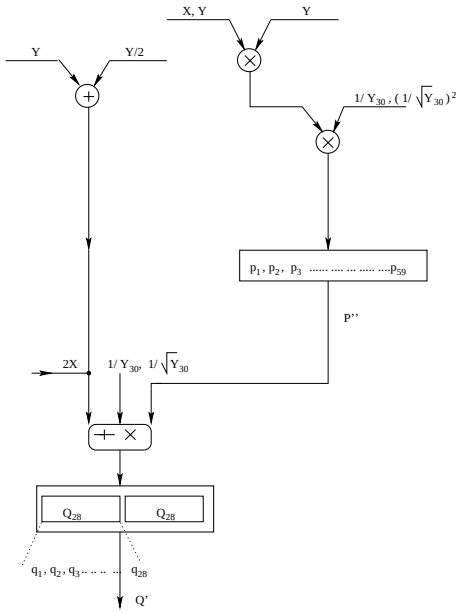
4.2. During the multiplication of Y^2 with $(1/\sqrt{Y_{30}})^2$, an independent operation $Y + Y/2$ (S) is performed. We then add the result of this addition to a right shifted P'' i.e. $S + P''/2$. The result of this addition is then multiplied by $1/\sqrt{Y_{30}}$, which in our case is a quotient. An example of our algorithm is depicted in Table 4.1 (for radix-10).

In the Cyrix division algorithm, after obtaining Y', which is a product of $1/Y_{30} \times Y$ (follow the algorithm steps 1 to 7 of subsection 2.3.2 and Figure 4.2), it is multiplied by P ($P = X$) for the first iteration $i = 1$ and is then subtracted from P. This result is a next partial remainder and is stored in a register P'_i. During the multiplication of P by Y', an independent multiplication P with $1/Y_{30}$ is performed to obtain q_1. This result is added to Q ($Q = 0$ for $i = 1$). The result is a next partial quotient and is stored in a register Q'_i. Both the next partial remainder and quotient are updated as a current partial remainder and quotient i.e. $P = P'_1$, $Q = Q'_1$ in the first iteration to allow to iterate for second iteration $i = 2$.

In the second iteration, current partial remainder P is multiplied by Y' and the



(a) AQA method



(b) Modified AQA method

Figure 4.2: Concept comparison of Cyrix algorithm with the proposed method

result of this product is subtracted from P, the result is a next partial remainder P_2'. Note that the Y' is generated only once, in the first iteration. During the multiplication Y' by P, another independent multiplication P by $1/Y_{30}$ is performed to obtain q_2 and the result of this multiplication is added to Q to obtain the next partial quotient Q_2'. An example of this algorithm using radix-10 is shown in Table 4.2.

In the proposed method (for division), after obtaining $X \times Y$ product and $1/Y_{30}$, they are multiplied. The result of this multiplication we call current partial remainder P'', follow the eqn.4.4&4.5 and Figure 4.2. The result of this multiplication is subtracted from a left shifted X i.e. $2X + P''$ and then this result is multiplied by $1/Y_{30}$ to obtain, in our case, the quotient Q'. An example of the proposed method is depicted in Table 4.2 (for radix-10).

In summary, the number of dependent operations in Cyrix algorithms is (for both division and square root): an addition (in square root), multiplication, subtraction and an addition–follow right to left of the eqn.2.34-eqn.2.37. It requires two iterations to obtain double precision results. The first 28 bits and the last 28 bits of the quotient are generated serially.

In the proposed method, the number of dependent operations for the division and square root are the same: a multiplication, an addition and multiplication–follow the eqn.4.4&4.5. Note that $X \times Y$, Y^2 and $Y + Y/2$ are independent operations. It requires an iteration to obtain double precision results. The first 28 bits and the last 28 bits of Q' are generated in parallel and it is added during multiplication, to obtain final Q' (eqn.4.5). The proposed method can be operated in fixed point mode, unlike Cyrix algorithms.

The 30 bit initial approximation for the reciprocal and the inverse square root can be obtained by indexing the top m bits of Y to the look-up table, which returns more than m bits of precision for these functions. Somewhat more than m bits of precision can be obtained from [**?**, **?**, **?**, **?**]. To obtain a 30 bit result, one would consider $9 \leq m \leq 20$.

	Proposed Method	Cyrix Processor Division Algorithm
	$X = 1.3256711329137561$, $Y = 1.546949120382167$. $1/Y_{30} = 0.646433671$. $X \times Y = 2.050745792976965794$. $P'' = -X \times Y \times (1/Y_{30})$, $Q' = (2X + P'') \times (1/Y_{30})$.	$X = 1.3256711329137561$, $Y = 1.546949120382167$. $1/Y_{30} = 0.646433671$. $Y' = Y \times 1/Y_{30} = 0.999999998738865136$, $Q = 0$, $P = X$, $q_1 = P \times 1/Y_{30}$ for $i \geq 1$. $P'_i = P - P \times Y'$, $Q'_i = Q + q_i$. $P = P'_i, Q = Q'_i$
$i = 1$	$P'' = -1.325671131241906016$, $Q' = 0.8569584580689085$	$q_1 = 0.856958456988168282$. $P'_1 = 1.671850083 \times 10^{-9}$, $Q'_1 = 0.856958456988168282$
		$P = P'_1, Q = Q'_1$
$i = 2$		$q_2 = 1.080740186 \times 10^{-9}$. $P'_2 = 2.10842 \times 10^{-18}$, $Q'_2 = 0.8569584580689085$.

Table 4.2: Comparison of Proposed method with Cyrix algorithm

4.2 Initial Approximation

To make the 30 bit initial approximation silicon efficient, we first obtain a 15 bit initial reciprocal and inverse square root approximation using a Minimax method [**?**] and we then pass it through an iteration of GLD algorithm. Figure 4.3 shows the concept of 30 bit initial approximation.

The Minimax approximation is performed as

$$f(y) = C_0 + C_1 Y_2 + C_2 Y_2^2 \tag{4.6}$$

The n-bit binary input significant Y is split into an upper part Y_1 and a lower part Y_2:

$$Y_1 = [1.y_1, y_2....y_m],\ Y_2 = [...y_{m+1}...y_n] \times 2^{-m}$$

Where C_0, C_1, C_2 are coefficients, the value of these coefficients depends only on the Y_1, the m most significant bits of Y.

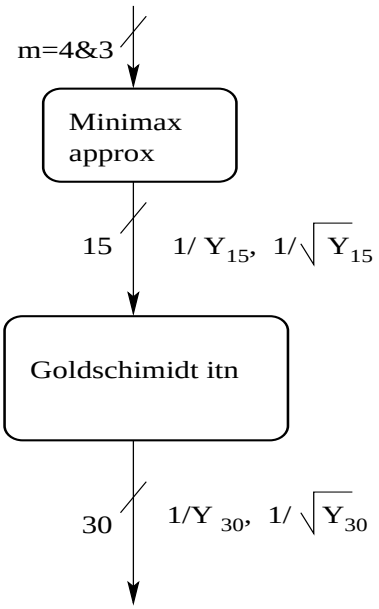


Figure 4.3: The 30 bit Initial Approximation concept

The minimax approximation with polynomial coefficients C_0, C_1, C_2 of unrestricted word length yields very accurate results. The coefficients must be rounded to a finite size to store the values of these coefficients in the look-up tables. In order to compensate for the rounding errors introduced by finite size coefficients, it must be passed through 3-passes. The method can be performed with the computer algebra system using Maple:

- Find the original approximation with non truncated coefficients and round the degree-1 coefficient to p bits to obtain C_1.

  -example

  The computation of the function $1/\sqrt{Y}$ in the interval $(1, 2)$ for $m = 4$. At address i in the table, we will find the coefficients of the approximation for the interval $[1 + i/16, (i + 1)/16)$. Let us say $i = 11$ and $t = 16$, $p = 11$, $q = 9$ bit precisions

  ```
  >pol1:= minimax(1/sqrt(1+11/16+y),y=0..1/16,[2,0],
  1,'err');
  ```

```
> C1:=2.^ (-11)*round(coeff(pol1,y)*2.^ (11));
```

ask for minimax approximation in that domain, the variable y represents Y_2, the lower part of the input operand Y. We obtain the approximation

```
>pol1:= 0.7698000005 +(-0.2279846809 + 0.09685666226
y) y;
> C1:=-0.2280273438;
> err:=0.35841950 10^-6;
```

- Compute $aa2$ using the analytical expression

$$aa2 = a2 + (a1 - C_1) \times 2^m \tag{4.7}$$

where $a1$, $a2$ are the degree-1 and degree-2 coefficients of the original approximation, round $aa2$ to q bits to obtain C_2.

```
>a1:=coeff(pol1,y);
>a2:=coeff(pol1,y^ 2);
> aa2:=a2+(a1-C1)*2.^ 4;
> C2:=2.^ (-9)*round(aa2*2.^ (9));
```

```
>a1:=-0.2279846809;
>a2:=0.09685666226;
> aa2:=0.097539269;
> C2:=0.09765625000;
```

- Based on C_1 and C_2 above, compute the degree-0 coefficient and then round to t bits to obtain C_0.

```
> p0:=minimax(1/sqrt(1+i/16+y)-C1*y-C2*y^ 2,y=0..1/16,
[0,0],1,'err');
> C0:=round(2.^ (16)*(tcoeff(p0)))*2.^ (-16);
> p0:=0.769799943;
> C0:=0.7698059081;
```

```
> err:=0.7580 10^(-6);
```

From the above rounded coefficients, compare the error of the approximation for the target precision (15 bits in our case) with target operation, inverse square root in this case

```
> err:=infnorm(1/sqrt(1+11/16+y)-C0-C1*y-C2*y^ 2,y=0
..1/16);
> err:=0.6722820608 10^(-5);
```

The error (err) of the approximation is much less than 2^{-15}. Figure 4.4 shows a maple program which implements the minimax method for the inverse square root computation, with $m = 3$. To obtain 15 bit initial approximation, the coefficient world lengths we have chosen are $t = 15$, $p = 10$ and $q = 8$ bits, because this combination yields an error of less than 2^{-15}. The number of correct bits are obtained as *goodbits* and the error of the approximation, ϵ'_{approx} (discussed in the next section) is shown as *errmax*. Similar procedure can be followed for the reciprocal. The coefficient word lengths we obtain are: $t = 15$, $p = 8$ and $q = 7$ bits for $m = 4$.

4.2.1 Error Computation

The total error in the final result of the function approximation step can be expressed as the accumulation of the error in the result before rounding, ϵ_{interm}, and the rounding error, ϵ_{round}:

$$\epsilon_{total} = \epsilon_{interm} + \epsilon_{round} < 2^{-r} \tag{4.8}$$

where r depends on the input and output ranges of the function to be approximated, on the target accuracy and defines a specific bound on the final error.

The error in the intermediate result comes from two sources: the error in the

```
> with(numapprox);
>computecoeffts:=proc(t,p,q)
> errmax:=0;
>f:=fopen("table_invsqrt.txt",WRITE);
> fprintf(f,"i\ t  C0 \ t \ t  C1 \ t \ t  C2 \ t \ t  err \  n");
> Digits:=12;
> for i from 0 to 7 do
> pol1:= minimax(1/sqrt(1+i/8+y),y=0..1/8,[2,0],1,'err');
> C1:=2.^ (-p)*round(coeff(pol1,y)*2.^ (p));
> a1:=coeff(pol1,y);
> a2:=coeff(pol1,y^ 2);
> aa2:=a2+(a1-C1)*2.^ 3;
> C2:=2.^ (-q)*round(aa2*2.^ (q));
> p0:=minimax(1/sqrt(1+i/8+y)-C1*y-C2*y^ 2,y=0..1/8,[0,0],1,'err');
> C0:=round(2.^ (t)*(tcoeff(p0)))*2.^ (-t);
> err:=infnorm(1/sqrt(1+i/8+y)-C0-C1*y-C2*y^ 2,y=0..1/8);
> fprintf(f,"%d \ t % e \ t % e\ t % e \ t % e \ n",i,C0,C1,C2,err);
> if errmax< err then errmax:= err fi;
> od;
> fclose(f);
> goodbits:=abs(ln(errmax))/ln(2.0);
> print(goodbits,errmax);
> end;
>
>
>
>computecoeffts(15,10,8);
> 15.0488978369, 0.0000295005660633
```

Figure 4.4: Maple program for obtaining inverse square coefficients $m = 3$

minimax approximation, ϵ_{approx}, and the error due to the use of finite arithmetic in the evaluation of the degree-2 polynomial:

$$\epsilon_{interm} \leq \epsilon_{approx} + \epsilon_{C_0} + \epsilon_{C_1} Y_2 + \epsilon_{C_2} Y_2^2 + |C_2|\epsilon_{Y_2} + |C_2|\epsilon_{Y_2^2} \tag{4.9}$$

The error of the approximation, ϵ_{approx}, depends on the value of m and the function to be interpolated. The maple program we showed in Figure 4.3 contributes the intermediate error of the minimax performed with *rounded* coefficients and therefore we can define:

$$\epsilon'_{approx} = \epsilon_{approx} + \epsilon_{C_0} + \epsilon_{C_1} Y_2 + \epsilon_{C_2} Y_2^2 \tag{4.10}$$

since $\epsilon_{Y_2} = 0$, in this case:

$$\epsilon_{interm} \leq \epsilon'_{approx} + \epsilon_{squaring} \tag{4.11}$$

$\epsilon_{squaring} = |C_2|\epsilon_{Y_2^2}$

The maximum error on the squaring computation bounded to be

$$\sum_{i>2m+12} partial\ product_i < 2^{-2\times(m=3)-10} \ldots \text{inverse square root} \tag{4.12}$$

$$\sum_{i>2m+10} partial\ product_i < 2^{-2\times(m=4)-8} \ldots \text{reciprocal} \tag{4.13}$$

Therefore, the $\epsilon_{squaring} \leq |C_2|.2^{-2m-8}$ for the reciprocal and $\epsilon_{squaring} \leq |C_2|.2^{-2m-10}$ for the inverse square root.

ϵ_{round} depends on how the rounding is carried out. Conventional rounding schemes are *truncation* and *rounding to the nearest*. Using truncation of the intermediate result at position 2^{-r}, the associated error would be bounded by $\epsilon_{round} \leq 2^{-r}$. While performing the rounding to the nearest by adding a one

at position 2^{-r-1} before such truncation, the rounding error would be bounded by 2^{-r-1} instead. In the minimax method round to the nearest is preformed, by adding one to the C_0 in advance at position 2^{-r-1} i.e. $C_0' = C_0 + 2^{-16}$ this is stored in the LUT instead of C_0. We then truncate the intermediate result at position 2^{-16} to guarantee 15 bit results.

Summarizing the total error in the final result can be expressed as:

$$\epsilon_{total} \leq \epsilon'_{approx} + |C_2|\epsilon_{Y_2^2} < 2^{-16} \tag{4.14}$$

4.2.2 Goldschimdt Algorithm

To obtain the double precision result for the division and the square root i.e. 53 bit, we need 30 bit initial approximation, passing this 15 bit approximation through an iteration of GLD reciprocation and inverse square root algorithms [**?**], we can obtain 30 bit initial approximation. Following are the GLD equations

$$1/Y_{30} = 1/Y_{15} \times (2 - (1/Y_{15}) \times Y_{30}) \tag{4.15}$$

$$2(1/\sqrt{Y_{30}}) = 1/\sqrt{Y_{15}} \times (3 - (1/\sqrt{Y_{15}})^2 \times Y_{30}) \tag{4.16}$$

4.2.3 Error computation

The error computation for the reciprocal and inverse square root can be set as

$$|\epsilon_{1/Y_{30}}| = (1/Y_{15} + |\epsilon_{1/Y_{15}}|) \times (2 - [(1/Y_{15} + |\epsilon_{1/Y_{15}}|) \times (Y_{30} + \epsilon_{Y_{30}})] + \epsilon_{vt}) + \epsilon_{t'} \tag{4.17}$$

$$2|\epsilon_{1/\sqrt{Y_{30}}}| = (1/\sqrt{Y_{15}} + |\epsilon_{1/\sqrt{Y_{15}}}|) \times (3 - [(1/\sqrt{Y_{15}} + |\epsilon_{1/\sqrt{Y_{15}}}|)^2 \times (Y_{30} + \epsilon_{Y_{30}})] + \epsilon_{vt}) + \epsilon_{t'} \tag{4.18}$$

The maximum absolute error when computing the reciprocal and the inverse square root, $|\epsilon_{1/Y_{30}}|$, $|\epsilon_{1/\sqrt{Y_{30}}}|$ are proportional to $\epsilon^2_{1/Y_{15}}$ and $\epsilon^2_{1/\sqrt{Y_{15}}}$ respectively. The $|\epsilon_{1/Y_{30}}|$ and $|\epsilon_{1/\sqrt{Y_{30}}}|$ are also depend on the the use of finite arithmetic, this is an intermediate word length truncating at t, introducing error ϵ_{vt} and $\epsilon_{t'}$ at the multiplier and the final result. Furthermore, it also depends on the absolute error of the initial approximations $|\epsilon_{1/Y_{15}}|$, $|\epsilon_{1/\sqrt{Y_{15}}}|$ and the error of the truncated divisors, $\epsilon_{Y_{30}}$.

Therefore, the error in the final result must be bounded by:

$$|\epsilon_{1/Y_{30}}| = \frac{2\epsilon_{1/Y_{15}}.Y_{30}}{Y_{15}} + \frac{\epsilon_{Y_{30}}}{Y_{15}^2} + \frac{\epsilon_{vt}}{Y_{15}} + \epsilon^2_{1/Y_{15}}.Y_{30} + \frac{2\epsilon_{1/Y_{15}}.\epsilon_{Y_{30}}}{Y_{15}} + \epsilon_{vt}. \\ \epsilon_{1/Y_{15}} + \epsilon^2_{1/Y_{15}}.\epsilon_{Y_{30}} + \epsilon_{t'} < 2^{-30} \tag{4.19}$$

$$|\epsilon_{1/\sqrt{Y_{30}}}| = |\epsilon_{1/\sqrt{Y_{15}}}| + \frac{3|\epsilon_{1/\sqrt{Y_{15}}}|.Y_{30}}{2.Y_{15}} + \frac{\epsilon_{Y_{30}}}{2.Y_{15}.\sqrt{Y_{15}}} + \frac{\epsilon_{vt}}{2.\sqrt{Y_{15}}} + \frac{5\epsilon^2_{1/\sqrt{Y_{15}}}.Y_{30}}{2.\sqrt{Y_{15}}} + \\ |\epsilon^3_{1/\sqrt{Y_{15}}}|.Y_{30} + \frac{3|\epsilon_{1/\sqrt{Y_{15}}}|.\epsilon_{Y_{30}}}{2} + \frac{\epsilon_{vt}.|\epsilon_{1/\sqrt{Y_{15}}}|}{2} + \frac{\epsilon^2_{1/\sqrt{Y_{15}}}.\epsilon_{Y_{30}}}{2\sqrt{Y_{15}}} + \epsilon_{t'} < 2^{-30} \tag{4.20}$$

The target 30 bit precision can be met $\epsilon_{Y_{30}}, \epsilon_{vt} < 2^{-32}$ and $\epsilon_{t'} < 2^{-30}$ by keeping $|\epsilon_{1/Y_{15}}|$, $|\epsilon_{1/\sqrt{Y_{15}}}| < 2^{-16}$:

$$|\epsilon_{1/Y_{30}}| < 2^{-15} + 2^{-32} + 2^{-32} + 2^{-32} + 2^{-47} + 2^{-48} + 2^{-64} + 2^{-30} < 2^{-29} \tag{4.21}$$

$$|\epsilon_{1/\sqrt{Y_{30}}}| < 2^{-16} + 2^{-17} + 2^{-33} + 2^{-33} + 2^{-33} + 2^{-48} + 2^{-49} + 2^{-49} + 2^{-65} \\ +2^{-30} < 2^{-29} \tag{4.22}$$

The conservative bounds on $\epsilon_{Y_{30}}, \epsilon_{vt} < 2^{-32}$ and $\epsilon_{t'} < 2^{-31}$ and $|\epsilon_{1/Y_{15}}|$, $|\epsilon_{1/\sqrt{Y_{15}}}| < 2^{-16}$ can be met by employing a truncation on the multipliers.

4.2.4 Architecture of Minimax Approximation

When $m = 3$ for the inverse square root, the lower part Y_2 has 12 bits and Y_2^2 has 6 leading zeros with a word length of 12 bits, if truncation at position 2^{-18} is performed for Y_2^2. Similarly when $m = 4$ for the reciprocal, the lower part Y_2 has 11 bits and Y_2^2 has 8 leading zeros with a word length of 10 bits, if truncation at position 2^{-18} is performed for Y_2^2. This is because Y_2 has m leading zeros, which means that Y_2^2 will have $2m$ leading zeros. We used a specialized squaring unit for squaring operation Y_2^2. More information on the squaring operation can be obtained from [**?**, **?**, **?**].

We use Y_2 and Y_2^2 as multipliers. To use Y_2^2 (Y_2 is in non redundant form) as a multiplier, it is converted from CS to Booth digit representation (discussed in Section 4.4). The Figure 4.6 shows an initial 15 bit approximation in the standard radix-4 and radix-8 multipliers.

The size of the LUT's for the inverse square root are $2 \times 2^3 \times (15 + 10 + 8) = 0.52$Kbits$= 0.064$KB. In the architecture we included the LUT for the reciprocal, which is $2^4 \times (15 + 8 + 7) = 0.47$Kbits$= 0.059$KB. Therefore, the total size of the LUT's to be employed is 0.99Kbits which is 0.123KB.

In Figure 4.6 (a), is a two stage pipeline architecture. In the first stage, the tables are addressed using the operand Y_1 to look-up co-efficient values C_0, C_1, C_2, in parallel computation of Y_2^2. After that, multiplexing for multiplicand and multiplier–to select either for division/square root or general multiplication and then, recoding of Y_2 and Y_2^2. Thereafter, PP's generation and 10 PP's summation (6 PP's of C_1+ 4 PP's of C_2) plus C_0 in three levels of 4:2CSA tree. In the case of the reciprocal, we generate the same 10 PP's. Figure 4.5 shows the PP's accumulation of the initial 15 bit approximation.

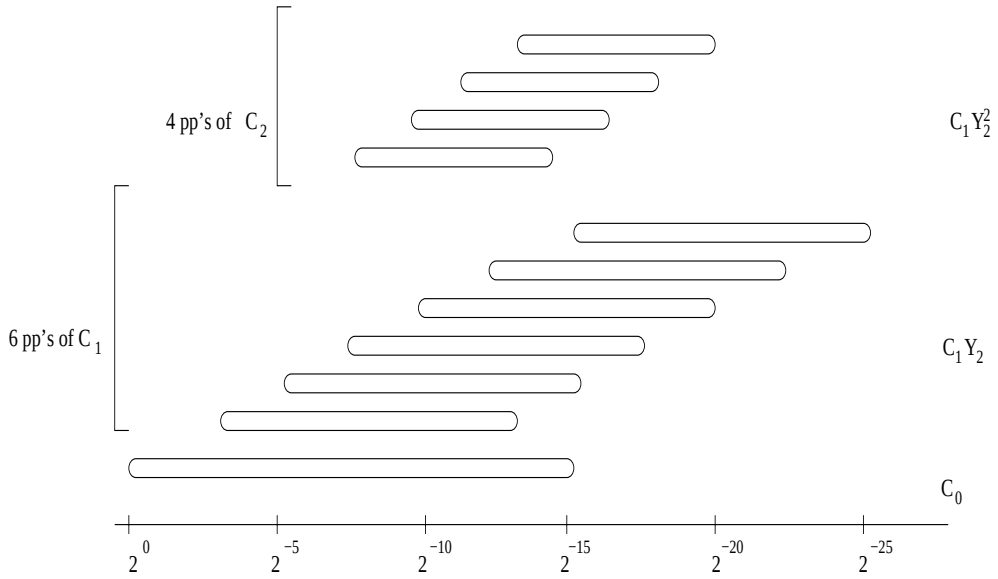
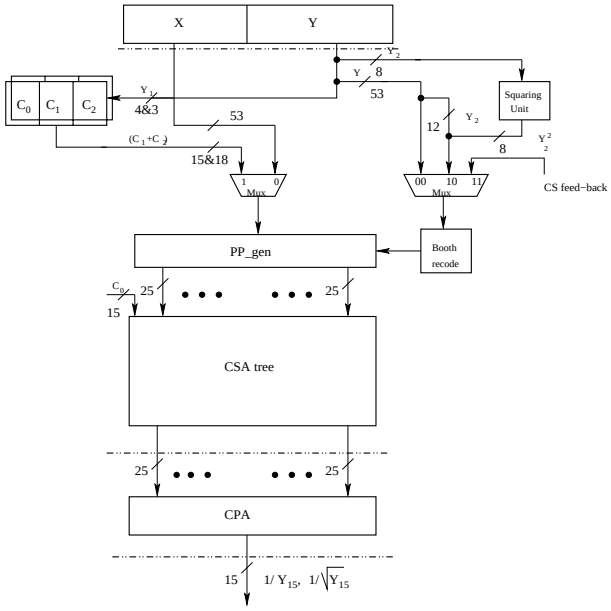


Figure 4.5: Minimax accumulation of PP's

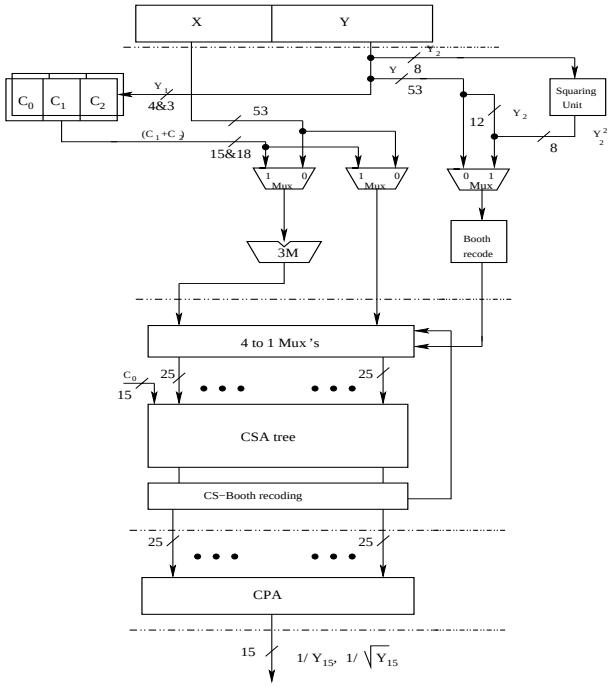
We have chosen a two stage pipeline architecture because it can be easily incorporated in the standard radix-4 multiplier (53×53 bit), the multiplier accumulates 27 PP's in four levels of 4:2CSA tree. The last stage is a CPA of the accumulation tree that performs assimilation from the CS-non redundant form of the result.

A similar architecture for a radix-8 multiplier (53×53 bit) is shown in Figure 4.6 (b). In the case of the radix-8 multiplier, it is a three stage pipeline. In the first stage it looks-up values of C_0, C_1, C_2 from look-up tables, in parallel with squaring of Y_2. Then uses multiplexing for the multiplicand and multiplier–to select, either for division/square root or general multiplication. Thereafter $3M$ multiple generation, which is in parallel with recoding of the multiplier Y_2 and Y_2^2.

In the second stage, multiplexing of multiplicands and 7 PP's summation for inverse square root and reciprocal (4 PP's of C_1 and 3PP's of C_2) plus C_0 in four levels of 3:2CSA tree. The standard radix-8 multiplier requires six levels of 3:2CSA tree for 18 PP's summation. Finally the third stage is a CPA, to obtain



(a) Initial approx in a radix-4 multiplier



(b) Initial approx in a radix-8 multiplier

Figure 4.6: Minimax approximation in the Industrial Multipliers

the result in the non-redundant form.

The access time for LUT's are $1/4^{th}$ of cycle time and $3M$ multiple generation for $(C_1 + C_2)$ is half the cycle time of standard multipliers.

4.2.5 Delay Estimates of Initial Minimax Approximation

In this section we analyze the cycle time for conventional multipliers and the initial approximation. We will show that the cycle time of initial approximation can be matched with the cycle time of the conventional multipliers. The cycle time we analyzed in terms τ-the delay of one full adder [**?**, **?**]. These estimates are reliable approximations. The actual speed-up depends on the technology employed and its implementation. We do not consider inter-connection delay because it is technology dependent. In principle, the cycle time depends upon the scheduling and market targeting. The pipelining of our architecture is flexible and can be performed differently, according to the technology to be employed.

We used a radix-4 multiplier of IBM Power PC $603e^{TM}$ FPU [**?**] and a radix-8 multiplier of IBM G5 FPU [**?**]. The estimated cycle time of the proposed initial approximation is compared with these industrial multipliers.

The Power PC FPU uses a radix-4 multiply-add fused ($53 + 53 \times 53$ bit) operation i.e. $53(A) \times 53(C)$ multiplied result added to another 53 bit operand say, B $(B+A\times C)$. It is a four stage pipeline for double precision MAF (Multiply-Add Fused) and is a three stage pipeline for single precision MAF.

In the case of double precision MAF, $53(A) \times 28(C)$-the lower 28 bits of C multiplied and added to aligned B operand in the final 3:2 CSA as shown in the first stage of Figure 4.13 of Section 4.6, the result (in CS form) is fed back to the second pass. In the second pass, $53(A) \times 25(C)$-the first 25 bits multiplied[1], the PP's of this pass assimilates with the first pass CS result so, 16 PP's were added (14 PP's + carry and sum of the first pass) with three levels of 4:2CSA's. In both

[1]In second pass the first 25 bits of C made to 28 bits with a concatenating three $0^{'}$s to generate 14 PP's.

	Radix-4MAF	Proposed method in radix-4 MAF
first pass & second pass	$(t_{recoding} + t_{5:1mux}) + 3 \times t_{4:2CSA} + 1 \times t_{3:2CSA} + t_{reg}$	$(t_{recodingY_2andY_2^2} + t_{5:1mux}) + 3 \times t_{4:2CSA} + t_{reg}$
	$2\tau+3\times1.5\tau+1\tau+1\tau = 8.5\tau$	$2\tau + 3 \times 1.5\tau + 1\tau = 7.5\tau$
Third stage	$t_{cpa} \parallel t_{exp\ adj} + t_{LZD} + t_{reg}$	$t_{cpa} + t_{reg}$
	$7\tau + 1\tau + 1\tau = 9\tau$	$7\tau + 1\tau = 8\tau$
Fourth stage	$t_{norm} \parallel t_{exp\ adj} + t_{round} + t_{reg}$	–
	$5\tau + 2\tau + 1\tau = 8\tau$	–

Table 4.3: Delay estimates in radix-4 multiplier

passes 14 PP's are generated. In the second stage, CPA which is in parallel with exponent aligning (exp adj)) and leading zero detection (LZD). And in the third stage, normalization and rounding.

The total latency of MAF operation is 4 cycles for double precision: first two cycles for dual pass multiplication, in the third cycle, carry propagate addition, and the final fourth cycle for normalization and rounding.

In the case of single precision it is 3 cycles: in the first cycle single pass multiplication, in the second cycle carry propagate addition and the final third cycle is for normalization and rounding.

The cycle time is set by the delay of the slowest path in the architecture, which may be in the third stage. Table 4.3 shows the delay estimates for MAF and the initial approximation in the MAF. Initial approximation in the MAF requires single pass multiplication. The cycle time of MAF can be estimated as 9τ. Note that the LUT's and squaring unit can be placed either in the first stage or before the first pipeline stage. However, we assumed that it was placed before the first pipeline stage and is coupled with 6:1 multiplexers (see Figure 4.13 on the top right).

The radix-8 multiplier (64×56 bit) [**?**] consists of three stages: in the first stage, $3M$ multiple generation and recoding of the multiplier. In the second stage, multiplexing of the multiplicands and 19 PP's summation in six levels of 3:2CSA's and, finally, the third stage is for CPA. There is a selection of two pos-

	Radix-8 multiplier	Proposed method in radix-8 multiplier
First stage	$t_{3M} \parallel t_{recoding} + t_{reg}$	$t_{lut} \parallel t_{Y_2squaring} + t_{2-to-1\ mux} + t_{3C_1,3C_2,3Y_2} \parallel t_{Y_2andY_2^2recoding} + t_{reg}$
	$7\tau + 1\tau = 8\tau$	$1.5\tau + 0.5\tau + 3\tau + 1\tau = 6\tau$
Second stage	$t_{4-to-1\ mux} + 6 \times t_{3:2CSA} + t_{reg}$	$t_{4-to-1\ mux} + 4 \times t_{3:2CSA} + 1 \times t_{3:2CSA\ for\ cs-booth\ digit\ rep} + t_{radix-8rec} + t_{reg}$
	$1.5\tau + 6 \times 1\tau + 1\tau = 8.5\tau$	$1.5\tau + 4 \times 1\tau + 1\tau + 0.3\tau + 1\tau = 7.8\tau$
Third stage	$t_{cpa} + t_{reg}$	$t_{cpa} + t_{reg}$
	$7\tau + 1\tau = 8\tau$	$7\tau + 1\tau = 8\tau$

Table 4.4: Delay estimates in radix-8 multiplier

sible normalization results in the third cycle. This selection signal is built into a custom designed data flow for speed.

Table 4.4 shows the delay estimates of the radix-8 multiplier and the initial approximation in the radix-8 multiplier. The cycle time of the radix-8 multiplier can be estimated as 8.5τ.

Therefore the cycle time of the initial approximation may not affect the cycle time of the industrial multipliers.

The 15 bit approximation can also be obtained from linear approximation algorithms, bipartite method [**?**, **?**, **?**]. In these cases the table size is more than 0.45KB, the quadratic Minimax approximation requires 0.123KB. Other quadratic methods such as [**?**, **?**] can be used, but they may add delay to the critical path for the industrial multipliers. The Minimax method may not add critical path delay to the industrial multipliers. However, the use of novel efficient algorithms in terms of speed and area is welcome.

4.2.6 Architecture and Performance of initial approximation

To evaluate the proposed 30 bit initial approximation, we used a radix-4 MAF of IBM $603e^{TM}$ FPU [**?**] and a radix-8 multiplier of G5 FPU [**?**, **?**]. As discussed

before, the latency of radix-4 MAF operation is 3 cycles (single pass multiplication) with a throughput of 1 and the latency of the radix-8 multiplier is 3 cycles with a throughput of 1.

In the radix-4 multiplier, after obtaining the 15 bit initial inverse square root approximation in the non redundant form as shown in Figure 4.6 (a), it has to be squared $(1/\sqrt{Y_{15}})^2$ to perform an iteration of GLD inverse square root algorithm–follow the eqn 4.16. For squaring $1/\sqrt{Y_{15}}$, using the $1/\sqrt{Y_{15}}$ as a multiplier and a multiplicand, we generate 8 PP's of $1/\sqrt{Y_{15}}$ and they are accumulated in two levels of 4:2CSA tree. The product is obtained in CS form to convert to Booth-2 digit representation (discussed in Section 4.4) for use as a multiplier and Y_{30} as a multiplicand. Using $(1/\sqrt{Y_{15}})^2$ as a multiplier, we generate 16 PP's of Y_{30} and they are accumulated in three levels of 4:2CSA tree, the *constant* 3 can be introduced in the CSA tree (replacing last 4:2CSA of first level with 5:2CSA[2][**?**]). The result is obtained in CS–Booth-2 digit form to use as a multiplier and $1/\sqrt{Y_{15}}$ as a multiplicand. We generate 16 PP's of $1/\sqrt{Y_{15}}$ and they are accumulated in a three levels of 4:2CSA tree, thereafter CPA addition of the product to obtain $1/\sqrt{Y_{30}}$ in the non-redundant form.

Performance: The performance of the initial $1/\sqrt{Y_{30}}$ is followed by above explanation, it is shown in the Figure 4.7.

- ▷ First couple of cycles for obtaining the $1/\sqrt{Y_{15}}$ in the non redundant form.
- ▷ In the third cycle, obtaining $(1/\sqrt{Y_{15}})^2$ CS–Booth-2 digit representation.
- ▷ The fourth cycle is for performing $3 - (1/\sqrt{Y_{15}})^2.Y_{30}$, obtain the result in CS–Booth-2 digit representation.
- ▷ And finally the fifth and sixth cycles are to obtain $1/\sqrt{Y_{30}}$ in the non redundant form.

[2]Replacing 4:2CSA with 5:2CSA in the first level of last 4:2CSA the delay in cycle time may be compensated, as this 5:2CSA is a late arriving signal, first preference could be given to this 5:2CSA for summing in a routing algorithm of PP's summation.

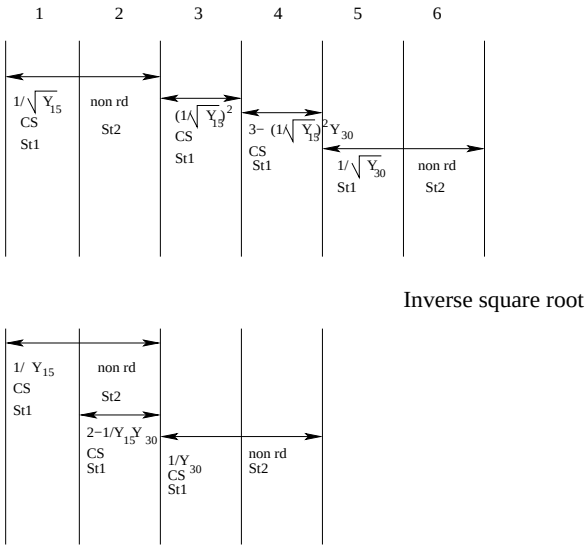


Figure 4.7: Timing diagram of initial 30 bit approximation in the radix-4 multiplier

In the radix-8 multiplier, after obtaining the $1/\sqrt{Y_{15}}$ in the non-redundant form as shown in Figure 4.6 (b), it is squared. Using the $1/\sqrt{Y_{15}}$ as a multiplier and a multiplicand, we generate 6 PP's of $1/\sqrt{Y_{15}}$ and these are accumulated in two levels of a 3:2CSA tree. The result is obtained in CS–Booth-3 digit representation (discussed in section 4.4) and recoded using an extra radix-8 recoder. The result of this recoder can be directly applied to the multiplexer to multiplex the multiplicands for the next cycles. During the PP's summation of $1/\sqrt{Y_{15}}$, an independent instruction $3Y_{30}$ multiple can be performed, because the multiplier has a throughput of 1. Using $(1/\sqrt{Y_{15}})^2$ as a multiplier and Y_{30} as a multiplicand, we generate 11 PP's of Y_{30} and these are accumulated in five levels of a 3:2CSA tree and the *constant* 3 can be introduced in this tree. The result is obtained in CS–Booth-3 digit representation with recoding. During this accumulation, the $3(1/\sqrt{Y_{15}})$ multiple instruction can be performed. Using $3-(1/\sqrt{Y_{15}})^2.Y_{30}$ as a multiplier and $1/\sqrt{Y_{15}}$ as a multiplicand we generate 11PP's of $1/\sqrt{Y_{15}}$ and these are accumulated in five levels of a 3:2CSA tree. The result is allowed to propagate the carry to obtain $1/\sqrt{Y_{30}}$ in the non-redundant form.

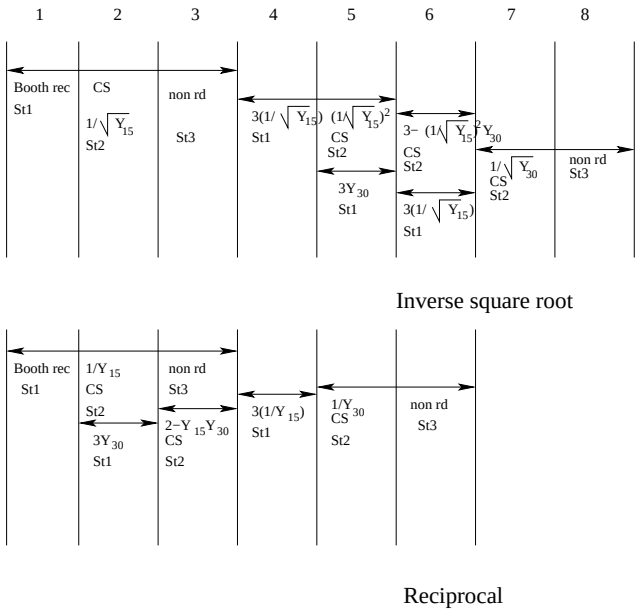


Figure 4.8: Timing diagram of initial 30 bit approximation in the radix-8 multiplier

Performance: is similar to the radix-4 multiplier as shown in Figure 4.8. The timing diagram is followed by the above explanation. Furthermore, it is two cycles slower with respect to the radix-4 multiplier, because the recoding of Y_2 and Y_2^2 (in the first cycle) and $3(1/\sqrt{Y_{15}})$ multiple (in the fourth cycle) is performed in another pipeline stage, unlike the radix-4 multiplier, the recoding and multiple generation is performed in the same pipeline stage of PP's summation.

Similarly for the division, in the radix-4 multiplier, after obtaining the $1/Y_{15}$ in the CS form–as shown in Figure 4.6 (a), it is passed through an iteration of the GLD reciprocation algorithm, follow the eqn.4.15. The CS of $1/Y_{15}$ is converted to CS–Booth-2 digit representation to use as a multiplier and Y_{30} as a multiplicand. We generate 8 PP's of Y_{30} and these are accumulated in two levels of a 4:2CSA tree, the product is two's complemented and obtained in CS–Booth-2 digit representation to use as a multiplier. During the PP's summation of Y_{30}, a carry propagate addition instruction for CS of $1/Y_{15}$ can be performed. Using $2 - 1/Y_{15}.Y_{30}$ as a multiplier and $1/Y_{15}$ as a multiplicand, we generate 16 PP's of $1/Y_{15}$ and these are accumulated in three levels of a 4:2CSA tree. The result of

$1/Y_{30}$ is obtained in both CS and non-redundant form to use as a multiplier and multiplicand for modified AQA method (discussed in next section).

Performance: The performance of the initial $1/Y_{30}$ followed by the above explanation, is shown in Figure 4.7.

- ▷ In the first cycle, obtain the $1/Y_{15}$ in the CS form and in the second cycle the CS result is allowed to propagate the carry. In the second cycle an independent instruction $2 - 1/Y_{15}Y_{30}$ is performed.
- ▷ In the third cycle, multiplying $2 - 1/Y_{15}.Y_{30}$ with $1/Y_{15}$ and obtain the result in CS form.
- ▷ The fourth and final cycle is to obtain $1/Y_{30}$ in the non-redundant form.

In the radix-8 multiplier, during the PP's summation of coefficient bits for the $1/Y_{15}$, another instruction $3Y_{30}$ can be performed. After obtaining the $1/Y_{15}$ in the CS form, it is converted to CS–Booth-3 digit representation with recoding as a multiplier and Y_{30} as a multiplicand in the next cycles. We generate 6 PP's of Y_{30} and these are accumulated in two levels of a 3:2CSA tree, the result is two's complemented. Thereafter it is converted to a CS–Booth-3 digit and recoded. During the PP's summation of Y_{30}, a carry propagate addition instruction of $1/Y_{15}$ can be performed. Thereafter, a $3(1/Y_{15})$ multiple instruction is performed. Using $2 - 1/Y_{15}.Y_{30}$ as a multiplier and $1/Y_{15}$ as a multiplicand we generate 11 PP's of Y_{15} and these are accumulated in five levels of a 3:2CSA tree. This result is allowed to propagate the carry, to obtain the result in non-redundant form $1/Y_{30}$.

Performance: The performance of initial $1/Y_{30}$, followed by the above explanation, is shown in Figure 4.8.

- ▷ First two cycles for obtaining $1/Y_{15}$ in the CS form and in the second cycle $3Y_{30}$ multiple instruction can be performed, because the radix-8 multiplier has a throughput of 1.

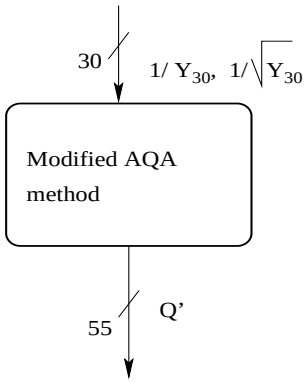


Figure 4.9: Proposed Method

- ▷ In the third cycle, obtaining $2 - 1/Y_{15}.Y_{30}$ CS-Booth 3 digit with recoding and a CPA for $1/Y_{15}$.
- ▷ The fourth cycle is for performing $3(1/Y_{15})$.
- ▷ Finally the fifth and sixth cycle are for obtaining $1/Y_{30}$ in the non-redundant form.

It is two cycles slower, with respect to the radix-4 multiplier, because the recoding of Y_2 and Y_2^2 (in the first cycle) and $3(1/Y_{15})$ multiple (in the fourth cycle) is performed in another pipeline stage. Unlike the radix-4 multiplier, the recoding and multiple generation is performed in the same pipeline stage of PP's summation.

4.3 Algorithm

The algorithm presented here is for both the division and square root. In order to obtain double precision results, an iteration is required after 30 bit initial approximation. The concept shown in Figure 4.9, recalling the eqn.4.4&eqn.4.5

$$P'' = -Y \times X \times (1/Y_{30}) \qquad P'' = -Y^2 \times (1/\sqrt{Y_{30}})^2 \quad (4.23)$$

$$Q' = (2X + P'') \times (1/Y_{30}) \qquad Q' = (Y + Y/2 + P''/2) \times (1/\sqrt{Y_{30}}) \quad (4.24)$$

$$\ldots \textit{division } Q' = X/Y_{30} \qquad \ldots \textit{square root } Q' = Y/\sqrt{Y_{30}}$$

Following are some steps to compute modified AQA method:

1. Obtain the 30 bit initial approximation, in parallel perform $X \times Y$ (Y^2 in the case of square root) and then take first 59 bits (detailed in error computation section) of the multiplied result.

2. In the case of square root, square the initial approximation and in parallel add $Y + Y/2$. The addition of this result is say, S.

3. Multiply $1/Y_{30}$ with the truncated $X \times Y$ product. Multiply $(1/\sqrt{Y_{30}})^2$ with the truncated Y^2 product in the case of square root. The result of this multiplication is P''.

4. Subtract the above result P'' to $2X$ for the division. Subtract the above result $P''/2$ to S for the square root. It can be introduced in the CSA tree. Obtain the result in CS to Booth digit representation (discussed in next section).

5. And then, multiply $1/Y_{30}$ ($1/\sqrt{Y_{30}}$ for the square root) with the above result to get Q'.

In the division and square root, the exponent can be computed in parallel with the mantissa of the quotient, therefore we focused on the mantissa in this thesis.

4.4 CS to Redundant Booth digit representation

In our design, the operands $1/Y_{15}$ and $(1/\sqrt{Y_{15}})^2$, $1/Y_{30}$ and $(1/\sqrt{Y_{30}})^2$, $2 - 1/Y_{15}Y_{30}$ and $3 - (1/\sqrt{Y_{15}})^2 Y_{30}$, $2X + P''$ and $S + P''/2$, and Q' (for division) are used as multipliers, and they are obtained in CS form. These intermediate results we call B and for the non-redundant representation of these intermediate results we call B'. Using these operands as a CS multiplier, the final level of CSA tree [**?**, **?**] are unused and we use this final level of CSA tree to convert CS to Booth 2&3 digit representation for the radix-4 and radix-8 multipliers respectively. The multipliers in CS form will enhance the performance of the division and the square root. Recalling the equations eqn.3.15 to eqn.3.23, and Figure 3.1, Figure 3.2 and Figure 3.3 of Chapter 3, by changing the notation t to B and t' to B' we obtain

$$\langle B \rangle = \sum_{i=0}^{m'-1} (-2.rB'3_i + rB'2_i + rB'1_i).4^i \tag{4.25}$$

The set of tripels $(rB'3_i, rB'2_i, rB'1_i)$, with $0 \leq i < m'$ to be a *redundant Booth-2 digit representation* of $\langle B' \rangle$, iff $\langle B \rangle = \langle B' \rangle$. Where m' is a PP's reduction factor for multiples of the multiplicand with b bit multiplier.

As discussed in Subsection 4.2.1 and 4.2.3 of Section 4.2, the $1/Y_{15}$ has $b = 16$ bits and $(1/\sqrt{Y_{15}})^2$ has $b = 32$ bits, the $1/Y_{30}$ has $b = 30$ bits and $(1/\sqrt{Y_{30}})^2$ has $b = 60$ bits , the $2 - 1/Y_{15}Y_{30}$ has $b = 32$ bits and $3 - (1/\sqrt{Y_{15}})^2 Y_{30}$ has $b = 32$ bits , the $2X + P''$ and $S + P''/2$, and Q' have a maximum $b = 59$ bits (discussed in next section); having $B'_{m+1} = B'_m = B'_{-1} = 0$ then $m' = \lceil (b+1)/2 \rceil$. Thus, the product not changed if fed to the multiplier in either the set of tripels $B'_{2i+1}, B'_{2i}, B'_{2i-1}$ based on the non-redundant representation of $\langle B' \rangle$ *or* a redundant Booth-digit representation $(rB'3_i, rB'2_i, rB'1_i)$ of $\langle B' \rangle$.

We assume to have a carry-save representation of $\langle B \rangle + const$, that already includes the additive constant $const = \sum_{i=0}^{m'-1} 2.4^i$. The $const$ is required to

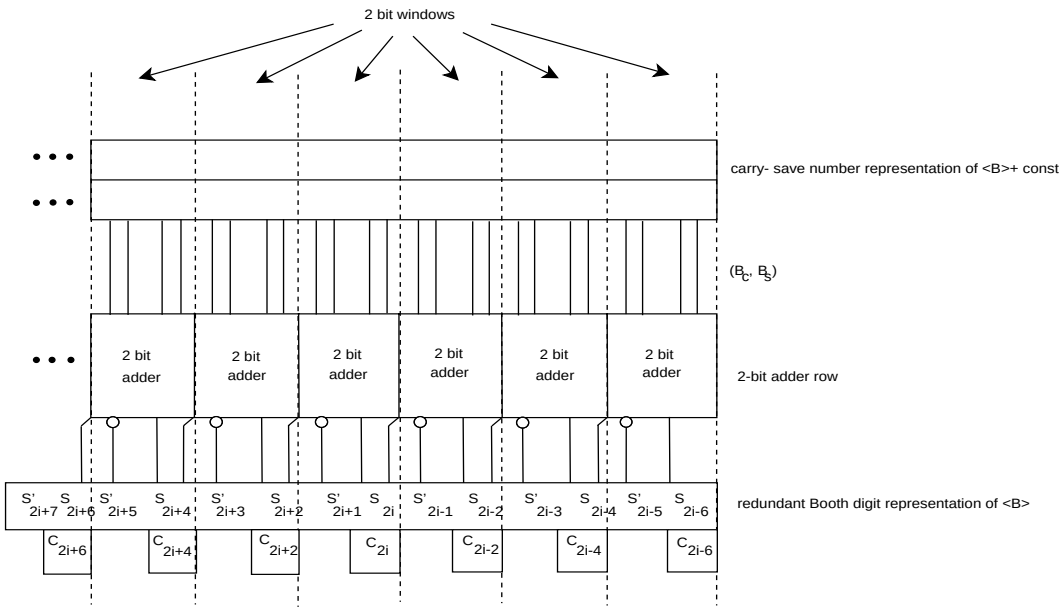


Figure 4.10: Compression from carry-save to redundant Booth-digit representation

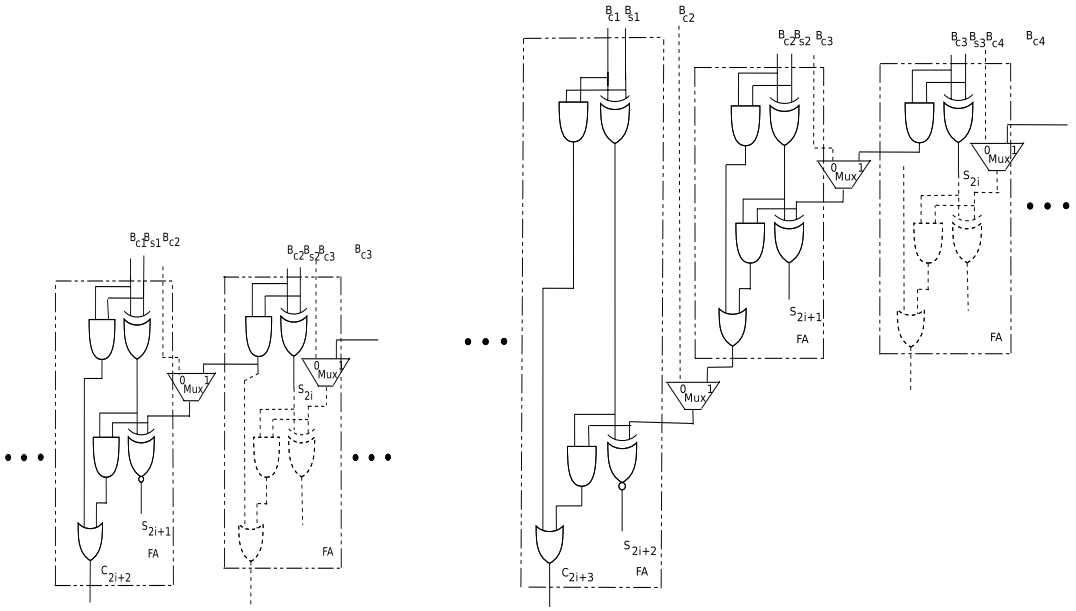
obtain B in eqn.4.25 form.

Looking at a bit windows of width 2 in this carry-save representation of the number $\langle B \rangle + const$, each window contains 4 bits (see Figure 4.10); two with a weight of one and two with a weight of two. The binary value w_j of the part of the number within a window i is in the range $w_i \in \{0, \ldots 6\}$. The number $\langle B \rangle + const$ can then be written by:

$$\langle B \rangle + const = \sum_{i=0}^{m'-1} w_i.4^i \tag{4.26}$$

If we input the 4 bits of a window i into a 2-bit adder, we get three output bits c_{2i+2}, s_{2i+1} and s_{2i}, that represent the value of the window by

$$w_i = 4.c_{2i+2} + 2.s_{2i+1} + s_{2i} \tag{4.27}$$



(a) 2 bit addition in CSA

(b) 3 bit addition in CSA

Figure 4.11: CS-Booth digit representation in a CSA cell

$$\langle B\rangle + const = \sum_{i=0}^{m'-1} (4.c_{2i+2} + 2.s_{2i+1} + s_{2i})4^i, \tag{4.28}$$

Subtract the additive constant *const* on both sides, and obtain the value of $\langle B\rangle$

$$\langle B\rangle + const - \sum_{i=0}^{m'-1} 2.4^i = \sum_{i=0}^{m'-1} (4.c_{2i+2} + 2(s_{2i+1} - 1) + s_{2i})4^i, \tag{4.29}$$

As $x - 1 \equiv -\overline{x}$ for $x \in \{0, 1\}$, then one can substitute $s_{2i+1} - 1$ by $-\overline{s_{2i+1}}$ and we can get

$$\langle B\rangle = \sum_{i=0}^{m'-1} (4.c_{2i+2} - 2\overline{s_{2i+1}} + s_{2i})4^i \tag{4.30}$$

In radix-4 recoding [**?**, **?**, **?**, **?**, **?**] the multiplier is recoded with $s_{2m'+1} = s_{2m'} = c_0 = 0$, then we have

$$\langle B \rangle + const = \sum_{i=0}^{m'} (2.s_{2i+1} + s_{2i} + c_{2i}) 4^i \tag{4.31}$$

Subtracting the additive constant *const* on both sides, and obtain the value of $\langle B \rangle$

$$\langle B \rangle + const - \sum_{i=0}^{m'-1} 2.4^i = \sum_{i=0}^{m'} (2.(s_{2i+1} - 1) + s_{2i} + c_{2i}) 4^i \tag{4.32}$$

$$\langle B \rangle = \sum_{i=0}^{m'} (-2\overline{s_{2i+1}} + s_{2i} + c_{2i}) 4^i \tag{4.33}$$

The set of triples $(\overline{s_{2i+1}}, s_{2i}, c_{2i})$ is a redundant Booth 2 digit representation of $\langle B \rangle$. The implementation of this partial compression is depicted in Figure 4.10.

In the radix-4 MAF [**?**], the $const$ is introduced by replacing the third level of 4:2CSA with 5:2CSA (the 4:2CSA requires two carries and sums of previous level, by replacing this 4:2CSA with 5:2CSA we introduce another input which is $const$). Performing CS–Booth-2 digit representation is a just wire crossing in a adjacent CSA cells, it can be performed in the final 3:2CSA level. Figure 4.11 (a) depicts the CS–Booth-2 digit representation in a CSA for radix-4 multiplier.

Where B_{si}, B_{ci} are the intermediate sums and carries of B'. We assumed a 3:2CSA consists of a combination of *X-or*'s, *OR*'s and *AND* gates, however it can be built with *NAND* or *NOR* gates.

In the radix-8 multiplier [**?**], the constant is $const = \sum_{i=0}^{m'-1} 4.(8)^i$, it can be introduced by replacing the fifth level of 3:2CSA with 4:2CSA (the 3:2CSA re-

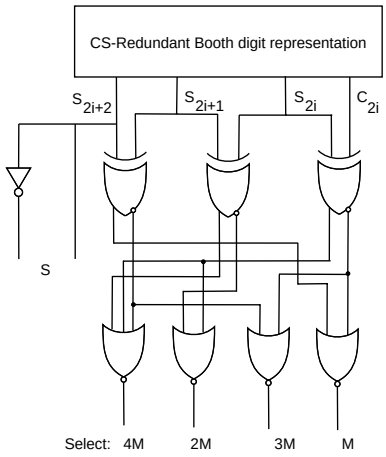


Figure 4.12: CS-Booth recoding

quires two sums and a carry of previous level, by replacing this 3:2CSA with 4:2CSA we introduce another input which is $const$). Performing CS–Booth-3 digit representation is a just wire crossing in an adjacent CSA cells, it can be performed in the final sixth level of 3:2CSA. An extra radix-8 recoder is required at the end of the CSA tree. Because of the need to recode in the pipe line stage of PP's summation. In the radix-8 multiplier, the recoder is coupled with $3M$ multiple generation which is another pipe line stage [**?**]. Figure 4.11 (b) and Figure 4.12 depicts the 3 bit addition and CS–radix-8 Booth recoding. The results of this recoding can be directly applied to the multiplexer to select the multiplicands.

In the radix-4 multiplier, an extra radix-4 recoder [**?**] is not required because the radix-4 recoder of the multiplier is placed in the same pipeline stage of PP's summation.

The multiplexers in Figures 4.11 (a)& (b) may not add additional delay in cycle time because they can be selected before start time of the PP's summation. In the case of the radix-4 multiplier, it can be selected at the start of the first stage and in the radix-8 multiplier it can be selected at the start of the second stage.

The work presented in [**?**] requires an additional hardware for 2 bit adder[3], in

[3]The author proposed a 30 bit unfolded architecture for redundant reciprocation approximation.

our design, as mentioned, this can be performed by just wire crossing in the CSA. We suggest that you obtain more information on CS-Booth k digits from [**?**].

In [**?**, **?**] other partial compressions of redundant number representation were described. Most of these recoders require at least four logic levels. Each of the two logic levels is a delay of mux & xor gate. From this point of view, the described technique is simpler, faster and the hardware can be shared with the CSA tree.

4.5 Error Computation

An accuracy of $n+2$ bits of the final estimate is required, in order to guarantee exact rounding. The result before rounding Q'_1 must satisfy:

$$-2^{-n-2} < [Q'_1] < 2^{-n-2} \tag{4.34}$$

Rounding is performed by adding 2^{-n-2} to Q'_1 and then truncating the result to a value to $n+1$ bits to form Q''_1, which has an error of strictly ± 0.5*ulp*. Therefore, we must guarantee $|\epsilon_{q'}| < 2^{-55}$, since the range of output result $0.5 \leq Q'_1 < 2$ and $1 \leq Q'_1 < 2$ for division and square root. Error computation for the division and square root operation using eqn.4.1& eqn.4.3 can be set as :

$$|\epsilon_{p'_1}| = X - (X.Y + \epsilon_{X.Y}) \times (1/Y_{30} + |\epsilon_{1/Y_{30}}|) + \epsilon_{pt'} \tag{4.35}$$

$$|\epsilon_{q'_1}| = (X + |\epsilon_{p'_1}|) \times (1/Y_{30} + |\epsilon_{1/Y_{30}}|) + \epsilon_{qt'} \tag{4.36}$$

$$|\epsilon_{p'_1}| = Y - (Y^2 + \epsilon_{Y^2}) \times (1/\sqrt{Y_{30}} + |\epsilon_{1/\sqrt{Y_{30}}}|)^2 + \epsilon_{pt'} \tag{4.37}$$

$$|\epsilon_{q'_1}| = (Y + |\epsilon_{p'_1}|/2) \times (1/\sqrt{Y_{30}} + |\epsilon_{1/\sqrt{Y_{30}}}|) + \epsilon_{qt'} \tag{4.38}$$

error accumulates from the initial approximation (Minimax method and GLD algorithm) and error due to finite word length of the intermediate result at $b =$

59, $\epsilon_{pt'}, \epsilon_{X.Y}, \epsilon_{Y^2}$ and $\epsilon_{qt'} \leq 2^{-59}$. The target precision can be met keeping $|\epsilon_{1/Y_{30}}|$, $|\epsilon_{1/\sqrt{Y_{30}}}| < 2^{-29}$ (truncating at position 2^{-30}), solving the above equations results

$$|\epsilon_{q_1'}| < -\frac{X.Y\epsilon_{1/Y_{30}}}{Y_{30}} - X.Y\epsilon_{1/Y_{30}}^2 + \frac{\epsilon_{pt'}}{Y_{30}} - \frac{\epsilon_{X.Y}}{Y_{30}^2} - \frac{2\epsilon_{X.Y}\epsilon_{1/Y_{30}}}{Y_{30}} + \epsilon_{pt'}|\epsilon_{1/Y_{30}}| - \epsilon_{X.Y}\epsilon_{1/Y_{30}}^2 + \epsilon_{qt'} < 2^{-55} \tag{4.39}$$

$$|\epsilon_{q_1'}| < 2^{-29} + 2^{-58} + 2^{-59} + 2^{-59} + 2^{-87} + 2^{-88} + 2^{-117} + 2^{-59} < 2^{-55} \tag{4.40}$$

$$|\epsilon_{q_1'}| < \sqrt{Y_{30}}Y_{30}|\epsilon_{1/\sqrt{Y_{30}}}|/2 - \sqrt{Y_{30}}Y\epsilon_{1/\sqrt{Y_{30}}}^2 + \epsilon_{pt'}/2\sqrt{Y_{30}} - Y^2|\epsilon_{1/\sqrt{Y_{30}}}^3|/2 + \epsilon_{pt'}|\epsilon_{1/\sqrt{Y_{30}}}|/2 + \epsilon_{qt'} < 2^{-55} \tag{4.41}$$

$$|\epsilon_{q_1'}| < 2^{-30} + 2^{-58} + 2^{-60} + 2^{-88} + 2^{-89} + 2^{-59} < 2^{-55} \tag{4.42}$$

These conservative bounds on $\epsilon_{pt'}, \epsilon_{X.Y}, \epsilon_{Y^2}$ and $\epsilon_{qt'}$ can be met by employing truncation in the multiplier, which carry out P_1' and Q_1' at $b = 59$.

The result before rounding Q_1' must satisfy eqn.4.34. Rounding can be performed by computing a corresponding remainder. By observing the sign and magnitude of remainder and the value of *n+1*, all IEEE rounding modes can be implemented by selecting either $Q_1', Q_1' + 2^{-n}$ or $Q_1' - 2^{-n}$. The action table for correctly rounding Q_1' is shown in Table 4.5 [**?**].

For directed rounding modes RP and RM, the action depends on the sign of estimate: those entries that contain two operations such as *pos/neg* correspond to the final result being positive or negative respectively. Most industrial multipliers incorporate these modes.

Guard Bit	Remainder	RN	RP(+/-)	RM(+/-)	RZ
0	= 0	trunc	trunc	trunc	trunc
0	-	trunc	trunc/dec	dec/trunc	dec
0	+	trunc	inc/trunc	trunc/inc	trunc
1	= 0	RNE	inc/trunc	trunc/inc	trunc
1	-	trunc	inc/trunc	trunc/inc	trunc
1	+	inc	inc/trunc	trunc/inc	trunc

Table 4.5: Action table for rounding

4.6 Architecture and Performance

The architecture of the proposed method is, simply to introduce LUT's and a squaring unit during input operand multiplexing in a multiplier. We used the IBM Power PC $603e^{TM}$ microprocessor FPU [**?**, **?**], which is targeted at portable computers and an IBM G5 FPU [**?**, **?**] that is targeted at scientific computing applications. The proposed method requires minimum hardware requirements in these multipliers.

The Power PC FPU uses radix-4 MAF operation. As discussed previously, the latency of MAF operation is 4 cycles: the first two cycles for dual pass multiplication, in the third cycle, carry propagate addition, and the final fourth cycle for normalization and rounding.

In a radix-4 multiplier [**?**], after obtaining the squared 30 bit initial approximation $(1/\sqrt{Y_{30}})^2$, Y^2 and S in the square root, we need to multiply i.e. an iteration of $-Y^2 \times (1/\sqrt{Y_{30}})^2$ is required to obtain double precision result (eqn.4.23). By using Y^2 as a multiplicand and $(1/\sqrt{Y_{30}})^2$ as a multiplier, we generate 15 PP's (P'') in each pass and compressed in three levels of 4:2CSA, the S (algorithm step 2 of section 4.3) can be subtracted during the PP's summation of the second pass (using the space of addition feed back of CS result i.e. replacing the last 4:2CSA of the first level with 5:2CSA see [**?**]) as shown in Figure 4.13 on the right dashed arrow. And then, with this result (acting as a multiplier) multiplied by $1/\sqrt{Y_{30}}$ (59×30 bit), we generate 30 PP's of $1/\sqrt{Y_{30}}$-a dual pass multiplica-

tion, thereafter the result in CS form is allowed to propagate the carry to obtain Q'.

Similarly, for the division, after obtaining the 30 bit initial approximation $1/Y_{30}$ and $X \times Y$ product, an iteration of $-X \times Y \times (1/Y_{30})$ is required to obtain the double precision result. By using the result of $X \times Y$ as a multiplicand and $1/Y_{30}$ as a multiplier, we generate 15 PP's and this is summed in three levels of a 4:2CSA. The operand $2X$ can be introduced during this PP's summation (using the space of addition feed back of CS result i.e. replacing one of 2:1 multiplexer for CS feed back with 3:1 multiplexer, which is in parallel with the 5:1 multiplexer for multiplicand selection see [**?**]) as shown in Figure 4.12 on the right dashed arrow. And then, with this result multiplied by $1/Y_{30}$ (59×30 bit), we generate 30 PP's of $1/Y_{30}$-a dual pass multiplication and the result Q' is obtained in CS form.

Note that, 15 PP's can be generated, by adding an extra 3 bit Booth recoder to the 14 bit Booth recoder and adding an extra 5:1 multiplexer to the row of 5:1 multiplexers (for selecting the multiplicands) without additional delay. By replacing 4:2CSA with 5:2CSA in the first level of last 4:2CSA the delay in cycle time may be compensated. As this 5:2CSA is a late arriving signal, first preference could be given to this 5:2CSA for summing in routing algorithm of PP's summation.

As mentioned, the results of $(1/\sqrt{Y_{15}})^2$ and $1/Y_{15}$, $(1/\sqrt{Y_{30}})^2$ and $1/Y_{30}$, $3 - (1/\sqrt{Y_{15}})^2 Y_{30}$ and $2 - 1/Y_{15}Y_{30}$, $S + P''/2$, $2X + P''$ and Q' (for division) are obtained in CS form to enhance the performance of the division and square root, to use as a multipliers. The results of these operations are represented in Booth digit representation by just wire crossing in the final level of 3:2CSA (see Figure 4.10& 4.11 (a) followed by eqn's. 4.25-4.33). To use these operands as a CS multipliers, one has to replace third level of 4:2CSA with 5:2CSA to introduce the $const = \sum_{i=0}^{m'-1} 2.4^i$. The third level of 4:2CSA requires two sums and carries from the previous level, by replacing 4:2CSA with 5:2CSA we introduce another input which is $const$ (not shown in the Figure 4.13).

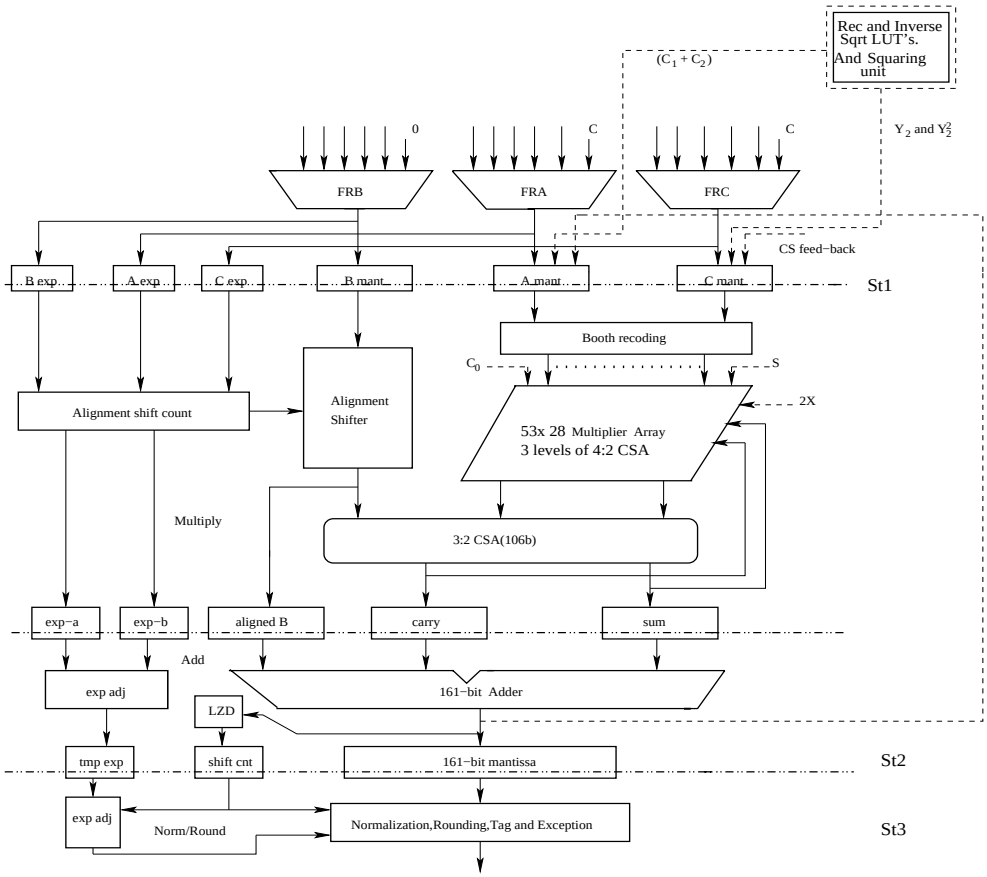


Figure 4.13: Proposed architecture

Performance: The square root performance shown in Figure 4.14.

- ▷ In the first two cycles, generating a 15 bit minimax approximation, a single pass multiplication with carry propagate addition is obtained.
- ▷ During two to five cycles Y^2, a dual pass multiplication is performed.
- ▷ Four to seven cycles for the generation of $1/\sqrt{Y_{30}}$ approximation through GLD algorithm (eqn.4.16). During four and five cycles some instructions of $1/\sqrt{Y_{30}}$ can be performed because MAF [**?**] has a throughput of 1.

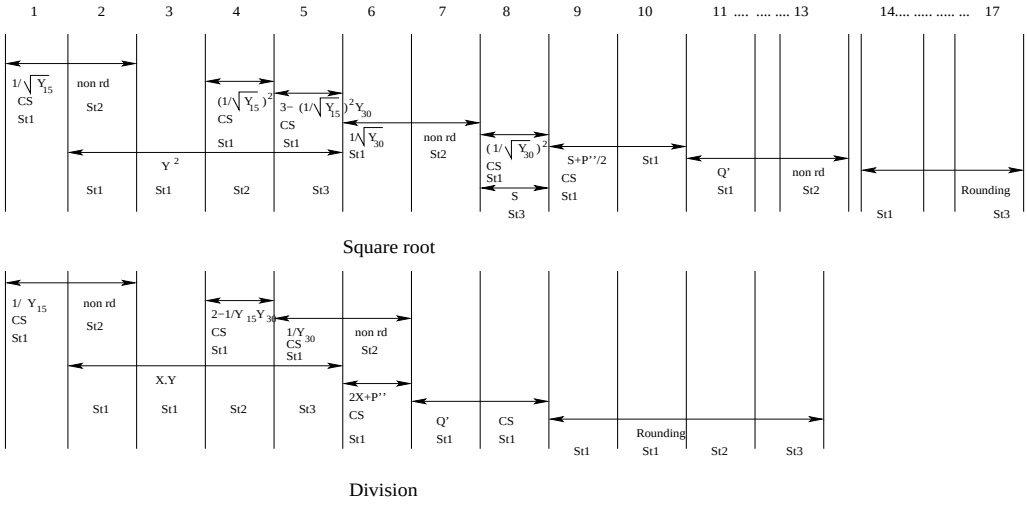


Figure 4.14: Performance in the MAF multiplier

- ▷ Eight to ten cycles to obtain partial part of Q' (eqn.4.24). Note the nine and the ten cycles for dual pass multiplication to obtain $S + P''/2$ in CS–Booth digit form. The PP's summation of these cycles assimilates with added $Y + Y/2\ (S)$, which was performed in the eighth cycle (as mentioned, the MAF has a throughput of 1). The $S + P''/2$ operation acts as a multiplier in the next cycles.
- ▷ Thereafter, eleven to thirteen cycles for Q' acting $1/\sqrt{Y_{30}}$ as a multiplicand and the final, fourteen to seventeen cycles for rounding. This can be performed by computing a remainder: $rem = Y - Q'^2$.
- ▷ Thus the square root is performed in 17 cycles.

Similarly, for the division, use the timing diagram shown in Figure 4.14. It can be expected to perform in 12 cycles.

In the radix-8 multiplier [**?**, **?**], similar architecture and performance can be obtained with a three stage pipeline: in the first stage booth recoding and 3M multiple generation, in the second stage PP's summation and in the third stage carry propagate addition. There is a selection signal built into the custom designed

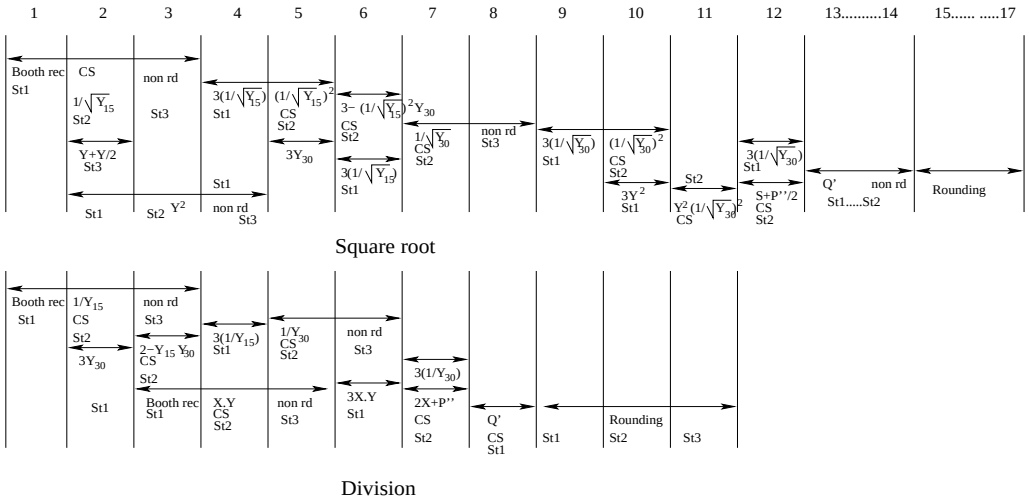


Figure 4.15: Performance in the radix-8 multiplier

data flow for the speed. This selection signal selects two possible normalization results in the third cycle.

The square root operation requires $59(Y^2) \times 60(1/\sqrt{Y_{30}})^2$ bit multiplication and 20 PP's are accumulated out of 19 PP's (64×56 bit), all six levels of 3:2 CSA's are required to compress the result into sum and carry. An extra 3:2CSA addition i.e. another cycle is required to subtract a compressed result $P''/2$ to S, this is because the $(1/\sqrt{Y_{30}})^2$ operation accumulates all levels of 3:2CSA's. In the final sixth level, CS–Booth-3 representation is performed as shown in Figure 4.11 (b). Thereafter it is recoded using an extra radix-8 recoder (see Figure 4.12). The performance of the square root is similar to that of the radix-4 multiplier as shown in Figure 4.15.

Similarly, for the division, with the $59(X.Y) \times 30(1/Y_{30})$ bit multiplication, 10 PP's were accumulated out of 19 PP's, the operand $2X$ can be introduced in a 3:2CSA tree. It requires five levels of 3:2 CSA's out of six and plus 1 level for CS-Booth digit representation along with radix-8 recoder. Multiplying this result with the $1/Y_{30}$ results in Q'. The division can be expected in 11 cycles, a cycle is reduced because it is a three stage pipeline. Figure 4.15 depicts the division performance in the radix-8 multiplier.

To enhance the performance of the division and square root in the radix-8 multiplier, we use the CS result of the intermediate results[4] as a multipliers. To use these operands as a multipliers, one has to replace fifth level of 3:2CSA with 4:2CSA. The 3:2CSA requires two sums and a carry from the previous level, by replacing 3:2CSA with 4:2CSA we introduce another input which is $const = \sum_{i=0}^{m'-1} 4.(8)^i$.

Note that, replacing 4:2CSA with 5:2CSA in the MAF and replacing 3:2CSA with 4:2CSA in the radix-8 multiplier, can introduce delay in the cycle time. However, as the technology is scaling down, in the future, the delay of 5:2CSA and 4:2CSA may be equivalent to that of 3:2CSA.

[4] $1/Y_{15}$ and $(1/\sqrt{Y_{15}})^2$, $1/Y_{30}$ and $(1/\sqrt{Y_{30}})^2$, $2-1/Y_{15}Y_{30}$ and $3-(1/\sqrt{Y_{15}})^2 Y_{30}$, $2X+P''$ and $S+P''/2$, and Q' (for division).

Conventional algorithms	Division	Square root
Newton-Raphson Method [**?**]	$1/Y_{60} = (1/Y_{30}) \times (2 - Y \times 1/Y_{30})$	$2Q = 1/Y_{30} \times (3 - Y \times (1/\sqrt{Y_{30}})^2)$
	$Q = X \times 1/Y_{60}$	$Q = Y \times 2(1/\sqrt{Y_{60}})$
GLD algorithm [**?**]	$G_d = X \times 1/Y_{30}$	$G_s = Y \times 1/\sqrt{Y_{30}}$
	$V_d = 2 - (1/Y_{30}) \times Y$	$V_s = 3 - (1/\sqrt{Y_{30}})^2 \times Y$
	$Q = G_d \times V_d$	$2Q = G_s \times V_s$
Cyrix algorithm [**?**]	$P'_i = P - P \times Y'$	$P'_i = P - q_i \times (2Q + q_i)$
	$Q'_i = Q + q_i$	$Q'_i = Q + q_i$
SRT method [**?**, **?**]	$W[i+1] = rW[i] - Yq_{i+1}$, where $W[0] = X$, $r = 2^{30}$	$W[i+1] = rW[i] - 2q[i]q_{i+1} + q_{i+1}^2 r^{-(i+1)}$, where $W[0] = Y$, $r = 2^{30}$
	$q_{i+1} = SEL(W[i], Y)$	$q_{i+1} = SEL(W[i], Y)$
	$Q = \sum_{i=1}^{N} q_i r^{-i}$, where $N = \lceil n/b \rceil$	$Q = \sum_{i=1}^{N} q_i r^{-i}$, where $N = \lceil n/b \rceil$
Proposed Method	$P'' = -Y \times X \times (1/Y_{30})$	$P'' = -Y^2 \times (1/\sqrt{Y_{30}})^2$
	$Q' = (2X + P'') \times (1/Y_{30})$	$Q' = (Y + Y/2 + P''/2) \times (1/\sqrt{Y_{30}})$

Table 4.6: Comparison of Proposed method with Conventional algorithms

The advantage of using the modified AQA method is explained using Table 4.6. In the N-R method, it requires the whole 53 bits accuracy of the reciprocal/inverse square root of the divisor and this is then multiplied by the dividend to obtain double precision quotients, even though the 30 bits initial approximation was used. In the proposed method it requires only 30 bits of initial approximation to obtain double precision quotients and, as mentioned, the first 28 bits and the last 28 bits of the quotient are generated in parallel and the result is added during the multiplication. In the quadratic convergence algorithm, the partial remainder is not available, unlike the AQA method.

The modified AQA method and GLD algorithm [**?**] are similar, if the same 30 bit initial approximation were used. But the method of computation in the hardware differs. In the proposed method, intermediate results are computed in CS form and used as a multipliers in the next computation, furthermore, overlapping some computations (see Figure 4.14&4.15)– offers more parallelism. We use CS multiplier and overlapping to amplify the performance for these functions. These features are unlike in the GLD algorithm [**?**]. The timing diagram of GLD algorithm is shown in Figure 4.16 for the radix-8 multiplier for comparison with the modified AQA method timing diagram in Figure 4.15. In the radix-4 multiplier the latency of the GLD algorithm is three cycles less, this is because the Booth recoding of the multiplier is in the same pipeline stage of the PP's summation. In the radix-8 multiplier, the Booth recoding is in another pipeline stage and is usually coupled with $3M$ multiple generation. On average, there is more than 25% improvement in performance for both the division and the square root, with respect to the GLD algorithm [**?**].

In the case of the Cyrix algorithm, it would require two iterations to obtain the double precision results, if the same 30 bit initial approximation was used. In the proposed method, it requires an iteration after the 30 bit initial approximation. The number of dependent operations in the Cyrix algorithm are: an addition (for square root), multiplication, subtraction and an addition. The first 28 bits and the last 28 bits of the quotient are generated sequentially. In the proposed method, a multiplication, an addition and a multiplication dependent operation are required.

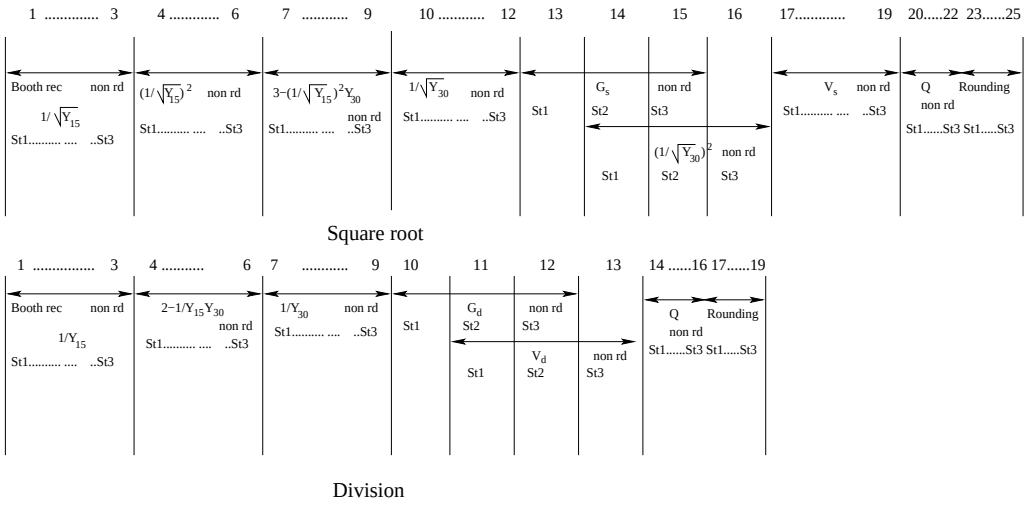


Figure 4.16: Performance of GLD algorithm [**?**] using 30 bit initial approximation

Furthermore, the proposed method can be operated in fixed point mode unlike the Cyrix algorithm. As mentioned, the first 28 bits and the last 28 bits of the quotient are generated in parallel and the result is added during the multiplication (eqn.4.1-4.5).

The modified AQA method with respect to the SRT method, although large numbers of bits are retired in the iteration, the complexity of the circuit is less than that of the SRT method. If the SRT method is shared with a multiplier and a 30 bit initial approximation is used. The performance gain of the SRT method for double precision computation may not be substantial even though the high accuracy of the initial approximations (30 bit) was used.

4.7 Comparisons

Most conventional processors share the hardware for the division and square root with the multiplier. The extra hardware required for these functions are initial approximation LUT's of the reciprocal and inverse square root. Therefore, we compare the proposed method with the state-of-the-art processor LUT's in Table 4.7 for double precision computation.

State-of-the-art Processors	Extra hardware for division and square root (*fa*)	Latency/Throughput Division and Square root
Intel *Itanium*[**?**]	Implemented in software	30/–, 45/–
IBM G5 FPU[**?**, **?**]	≈ 180	27-30/27-30, 37-46/37-46
AMD K-7 [**?**]	≈ 2473	16/13, 27/24
SunUltraSPARC IV [**?**, **?**]	≈ 127	20, 29/–
IBM Power PC $603e^{TM}$[**?**]	—	33/–, Not Supported
Proposed Method ($603e^{TM}$[**?**])	≈ 60	12/10, 17/15
Proposed Method (G5 FPU [**?**, **?**])	≈ 72	11/9, 17/15

Table 4.7: Comparison of Proposed method with Conventional Processors

The hardware estimates are shown in terms of full adders (*fa*). A standard 1-bit full-adder has a hardware complexity equivalent to 9 *nand/nor* gates.

Look-up-tables: Estimates for the look-up table can be found in [**?**]. We assumed a pessimistic model. Our model assumes 40*fa*/Kbit rate for tables addressed up to 6 bits, a 35*fa*/Kbit rate for 7-11 input bit tables, 30*fa*/Kbit rate for 12-13 input bit tables and a 25*fa*/Kbit rate for 14-15 input bit tables.

The estimated values presented here are reliable approximations, the actual speed-up and the area ratios depend on the technology employed and its implementation.

The G5 FPU uses about 8 bits in and 10 bits out of look-up (90 *fa*), AMD 11 bits in and 16+7 out, which is about 1649*fa* and SunUltra 8 bits in and 9 bits out of look-up which is about 81*fa* for inverse square root approximation. For the reciprocal, LUT's sizes are: 90 *fa* in the G5FPU, AMD uses 10 bits in and 16+7 out which is about 824*fa* and in SunUltra it is 7 bits in and 10 bits out which is about 46*fa*.

The proposed method requires: 0.99Kbits for both reciprocal ($2^4 \times (15+8+7)$)

and inverse square root ($2 \times 2^3 \times (15 + 10 + 8)$) LUT's, which is about 40*fa* and another 20*fa* for 8 bit squaring unit. This means that the proposed method can fit into a radix-4 MAF [**?**] with extra hardware of about 60*fa*.

In the case of a radix-8 multiplier [**?**], another extra 60 bit Booth recoder is required in the second stage, which is about 12*fa*. This is because it is necessary to perform redundant recoding in the PP's summation stage to increase the performance of these operations.

Note that we do not included the area estimates of extra 3 bit Booth recoder and 5:1 multiplexer in the MAF . This is because in order to obtain exact IEEE rounded result for both division and square root, multipliers could support 60×60 bit multiplication as found in the Sun Ultra SPARC IV processor [**?**]. Supporting 60×60 bit multiplication (both in a radix-4 and in a radix-8 multiplier) does not increase the cycle time nor extra CSA's are required.

Comparison shows that, on average, approximately more than 50% improvement in the performance and more than 70% of the hardware can be saved with respect to the conventional processor LUT's. Note that, in a Power PC, by utilizing most of the existing hardware, no extra hardware for division is required, because it uses SRT algorithm with two bits retired per iteration. In a radix-8 multiplier of the G5 architecture, there is an increase in performance, due to the enhanced initial approximation with half the table size of the G5 initial approximation.

In the initial 15 bit approximation, one could eliminate the squaring unit and perform two dependent multiplications $(C_2.Y_2).Y_2$ in CSA tree, but there is about 2τ increase in the cycle time due to these dependent multiplications. As technology scales down and when the market targeting of the processor architect is acceptable, these two dependent multiplications could perform in state-of-the-art multipliers.

4.8 Conclusions

We discussed the computation of the division and square root function using an accurate quotient approximation method. The advantages of the proposed method are (see Table 4.6): there may not be an increase in the cycle time, though large numbers of bits are retired in the iteration. This is unlike the digit recurrence method, which increases the cycle time with respect to the radix. Unlike the Newton-Raphson method, it requires only the 30 bit initial approximation to obtain the final quotient. The Cyrix algorithm requires two iterations to obtain the double precision results and the first 28 bits and the last 28 bits of the quotient are generated sequentially. Three dependent operations are required for the division and four dependent operations for the square root. The proposed method requires an iteration to obtain the double results and three dependent operations for both division and square root. The first 28 bits and the last 28 bits of the quotient generated in parallel and added during multiplication. It is similar to the GLD algorithm, if the 30 bit initial approximation is used, but the method of computation in the hardware is different. We perform intermediate results in CS form and used as a multipliers and, overlapping of some computations to enhance the performance. In the GLD algorithm these features are imperfect (see Figure 4.15&4.16)[5]. These features lead to a significant boost in the performance with respect to the GLD algorithm.

The look-up table size for the initial approximation is less than half the table size of conventional processors. Booth recoding can be performed in the same cycle of PP's summation and overlapping some computation of the partial remainder (see Figure 4.14&4.15) which increases performance.

We set-out a division and square root algorithm with better silicon efficiency than the conventional methods as shown in Table. 4.7. The proposed method can fit into a radix-4 multiplier with extra hardware of about 60*fa* and about 72*fa* for a radix-8 multiplier, which is about more than a 70% decrease in the silicon area,

[5]In the radix-4 multiplier the latency of the GLD algorithm is three cycle less. this is due to the fact that the Booth recoding is in the same pipeline stage of PP's summation.

with respect to state-of-the-art processor LUT's.

There is a significant improvement in performance with respect to the state-of-the-art processors. This is because we used high accuracy of initial approximation (30 bit) and intermediate results are computed in carry save form. Firstly we obtain a 15 bit initial approximation from the Minimax method, which is silicon efficient and may not add critical path delay to industrial multipliers. We then interface this approximation with a GLD algorithm to obtain a 30 bit initial approximation. However, the use of state-of-the-art efficient initial approximation algorithms in terms of speed and area is encouraged. An iteration is required after the 30 bit initial approximation to obtain double precision results.

It offers the best trade-off between performance and area, making it suitable for both the mobile and scientific computing market. In both the mobile computing and high performance computing market area is crucial. Moreover, the speed is the key in the high computing market. Most architects look for division and square root algorithms that can run graphics applications, scientific computing applications, signal processing applications, audio and video streaming applications, charting etc. It is expected that, with the proposed method, processor architects will have an option for their next generation processors.

Conclusions

Summary

We discussed the division and square root computation for the single precision and double precision computation. Basics of various floating point precisions, rounding modes, special values and exceptions are explained in Chapter 1.

The use of division and square root functions depends on the target market. For example, in mobile computing markets such as PDA, UPC, tablet PC , where it can run internet applications, day to day accounting, snapshot viewing, presentations etc. In these markets, the use of division and square root is less intense and so single precision can suffice. Also, in these markets the processor architects look for low cost and low complexity, but with acceptable performance. The modified N-R reciprocation/division algorithm could be suitable for these markets.

The square root function without a LUT, by modifying the N-R algorithm, is proposed. However, this adversely affects performance, because it has to be squared in each iteration and each iteration requires at least three cycles. Somewhat more than 50 cycles are required to compute the square root. More research on the "without a LUT for modified N-R square root algorithm" would be very welcome.

These functions are frequent in 3D graphics, scientific computing applications, mathematical computing applications etc. Double precision computation is required in many applications such as the previously mentioned scientific and mathematical computing applications. In these markets, the processor architects

look for higher performance with acceptable levels of cost and complexity. The modified AQA method can be applied for these markets. Furthermore, the modified AQA method can be used in the mobile computing market for future processors.

To obtain a higher performance for these functions, the initial approximation in the form of LUT or LUT with linear/quadratic approximation algorithms can be used to reduce the area required for these functions. The initial approximations are usually coupled with the N-R method, the GLD algorithm and the Digit recurrence algorithm, also called the SRT method. A discussion of the initial approximation algorithms and the N-R method, the GLD algorithm and the Digit recurrence algorithm are detailed in Chapter 2.

These functions are slower because they require sequences of addition or subtraction, multiplication and multiply-add operations, blocking the FPU throughout the computation process. Some processors implemented these functions in the software, such as in *Itanium* processors and others implemented them in the hardware, such as in IBM and AMD processors. Implementing these functions in the software provides a high degree of parallelism and does not block the FPU during the computation, but performance is lower. By implementing these functions in the hardware, performance can be enhanced, but this is complex and the cost is higher. Most designers are willing to sacrifice the speed in favor of low cost and low complexity. Performance and complexity depends on the precision and the precision is dependent on the target market or on market demands.

We proposed the modified AQA method, which offers significant improvements in performance, with better silicon efficiency than the conventional processors for double precision computation. We also proposed the average performance modified N-R reciprocation algorithm, which offers the best silicon efficiency when compared to the conventional processors used for single precision computation.

In Chapter 3, a single precision computation of a modified N-R reciprocation algorithm without a LUT is proposed, which is the best silicon efficient algorithm.

It is a variable latency algorithm and only requires a maximum of 22 iterations with a linear convergence. A variable latency algorithm is useful in a self-timed divider, because the processing capability depends on the task at hand. In this way, it can reduce power consumption without sacrificing performance.

To evaluate the proposed method, we used a radix-4 and radix-8 multiplier. In a radix-8 multiplier, there may be a delay of an *OR* gate in cycle time, due to the extra radix-8 Booth recoder. In a radix-4 multiplier, it is 1τ slower in cycle time, due to another level of 3:2CSA for CS–Booth-2 digit representation.

We compared the proposed method with the state-of-the-art processors in Table 3.4. The proposed method can fit into a radix-4 multiplier with extra hardware of about 31*fa* and of about 6*fa* for a radix-8 multiplier, which is less than a 1.5% increase in area with respect to industrial multipliers.

There is about 16% improvement in performance with respect to IBM and *Itanium* processors, but a decrease of 4% with respect to IBM $603e^{TM}$ and AMD processors. This is due to the fact that 1.09 bits are retired per iteration; it is shared with the multiplier and a large LUT is not available. Therefore, the performance is about the same as the average performance of the state-of-the-art processors. The significance of the proposed N-R algorithm is discussed in the next section.

In Chapter 4, we proposed division and square root functions by modifying the AQA method, using a LUT for high performance computing applications. This requires iteration after the 30 bit initial approximation. The proposed modified AQA method offers some attractive strategies additional to the conventional N-R, GLD, SRT and Cyrix algorithms, which are discussed in the following section.

To evaluate the proposed modified AQA method, we used a radix-4 multiplier of IBM $603e^{TM}$ FPU , which is targeted at the mobile computing market and a radix-8 multiplier of IBM G5 FPU it targeted at high performance applications. We then compared it with the conventional processors for double precision computation in Table 4.7. Comparison shows that better silicon efficiency with significant improvement in performance can be expected.

It can be fitted into a radix-4 multiplier with extra hardware of about 60*fa* and

with about 72*fa* in a radix-8 multiplier. This, on average, is somewhat more than a 70% decrease in the silicon area, with respect to the state-of-the-art processor LUT's. Performance is significantly better, because we used high seed, 30 bits, and the intermediate results are computed in CS form, i.e. the intermediate results are converted to CS to Booth digit representation for use as multipliers. This is wire crossing in our design.

Contributions

The advantages/contributions of the proposed modified N-R reciprocation algorithm are that it does not require a look-up-table and that performance is about the same as the average performance of the state-of-the-art processors. The average performance can be maintained by computing intermediate results in CS form. The initial approximation is a two's complement of the divisor which can be performed during partial products summation. Each of the iterations is a cycle, unlike the pairs of cycles found in the state-of-the-art processors. The multiples of the divisor are constant throughout each of the iterations. Therefore we generate them only once, in the third cycle in the case of the radix-8 multiplier and in the second cycle in the case of the radix-4 multiplier, i.e. in the second iteration. This leads to a reduction of a cycle for each of the iterations. Moreover, the multiply-add operation and Booth digit representation can be performed in the same cycle of partial products summation. It is a variable latency algorithm and requires only 22 iterations. Variable latency algorithms are useful in speed independent designs that provide power according to the task at hand. The eqn.3.10-eqn.3.13 could be considered to be a multiplication in hardware terms.

In the N-R method, to obtain the reciprocation, the required operations are: a multiplication and multiply add operation (see eqn.3.2-eqn.3.4) and each of the iterations require two cycles. In the N-R method, the multiply-add operation requires (eqn.3.4) the multiplicand S to be in the non-redundant form. It is possible to use the multiplicand in the redundant form, but this doubles the data width.

These features are different in the proposed method (eqn.3.10-eqn.3.13–could be considered to be a multiplication in hardware terms).

It is possible to use the N-R method as a without LUT algorithm, as in the proposed method, but performance is about the same. This is because each iteration requires two multiplications i.e. about 6 cycles (we assumed 3 cycles for a multiplication) and requires 4 iterations to obtain the single precision result.

With respect to the SRT method, the proposed method does not require a quotient digit selection hardware to select the quotient digits which increases the cycle time with respect to the radix.

The proposed modified N-R algorithm can also be extended to the square root computation and it only requires a maximum of 22 iterations. But it does not offer the average performance of the conventional processors. This is due to the squaring of the quotient in each of the iterations. Each of the iterations requires at least three cycles. Therefore, more than 50 cycles are needed to obtain the result. Breakthrough research on this algorithm is substantial.

The advantages/contributions of the proposed AQA method are shown in Table 4.6.

- Unlike the N-R method, only the 30 bit initial approximation is required to obtain the final quotient.
- With respect to the GLD algorithm the proposed method is similar if the 30 bit initial approximation was used, but the method of computation in the hardware is different. We perform intermediate results in CS form used them as multipliers and, furthermore, we use the overlapping of some computations. These features increase performance and the comparisons made can be seen in Figures 4.15 and 4.16.
- The cycle time might not be increased, although large numbers of bits are retired in the iteration, unlike the digit recurrence method which increases the cycle time with respect to the radix.

- With respect to the Cyrix algorithm, the modified AQA requires an iteration to obtain double precision results. It requires a multiplication, an addition and a multiplication dependent operation. The first 28 bits and the last 28 bits of the quotient generated in parallel, is added during multiplication. To obtain double precision results, the Cyrix algorithm requires two iterations and an addition (in square root) and multiplication, subtraction and addition-dependent operations are required. The first 28 bits and the last 28 bits of the quotient are generated sequentially, if the same 30 bits initial approximation is assumed. Furthermore, the proposed method can be operated in fixed point mode, unlike the Cyrix algorithm.

- The look-up table size for the initial approximation is less than half the table size of the conventional processors. Booth recoding can be performed in the same cycle of partial products summation and overlapping some of the computations of the partial remainder (see Figures 4.14 & 4.15) which increases performance.

- To make the proposed method silicon efficient, we first obtain a 15 bit initial approximation using the Minimax method, which is silicon efficient and may not add critical path delay to industrial multipliers. We then interface this approximation with the GLD algorithm to obtain a 30 bit initial approximation. However, the use of novel initial approximation algorithms that are efficient in terms of speed and area, is encouraged.

- An iteration is required after a 30 bit initial approximation to achieve double precision results.

The modified AQA method can also be extended to the mobile computing market, as it can support high performance applications for future processors. The proposed modified AQA method can fit into industrial multipliers with extra hardware of less than 70*fa*. The choice is up to the architects.

Future Work

Performing manual calculations is a part of human nature. Computers readily support binary arithmetic. A recent processor, the IBM z900 series, is the only one capable of performing decimal instructions in the hardware [**?**, **?**]. However, its decimal computation capability is limited to integer operands. Recently, decimal arithmetic has become more attractive in the financial and commercial world such as banking, tax calculation, currency conversion, insurance and accounting, etc. The following facts may explain this interest:

- A survey of commercial databases [**?**] shows that more than 90% of numbers are stored in decimal or integer form, while more than half of them are represented in a purely decimal format.
- It is well understood that, when converting decimal and binary formats, most fractional decimal numbers are approximately represented in binary floating point representation and, therefore, may loose precision [**?**, **?**]. This means that using binary floating point numbers in financial applications, where errors cannot be tolerated, does not necessarily mean that correct results will be obtained.
- Regulations, such as the European Directorate General II [**?**], specify decimal digits for currency calculations.

It is likely that in the near future, most high-end processors will perform decimal operations in decimal floating point format using dedicated decimal floating point units.

In decimal arithmetic there are some complex functions such as sequential multiplication, division and square root. The fundamental operations of division and square root are based on multiplication. Computing partial products of the multiplication and adding each newly computed product to the previous partial product, cannot be accomplished in a reasonable time [**?**] without a fast circuit for decimal addition. This can adversely affect speed if division and square root were also taken into consideration. One method for computing partial products is by taking advantage of carry free addition. The building of fast circuits for decimal carry free addition may be a possibility. Designing the computation of division and square root, based on fast addition and multiplication, is challenging.

It may be interesting to focus the Accurate Quotient Approximation method on decimal arithmetic. The decimal divider and square root can demand a larger area and, thus, may obtain a reasonable speed-up.

Achieving the highest speed in the smallest chip area is one of the most important goals. However, there may be some other areas for designers to consider. Among these is the design of methods using low power consumption [**?**], motivated by battery operated devices demanding intensive computation in portable environments, and for the proposed use of divider and square root computations, it might be interesting to study whether they can be redesigned for low power use, with or without performance loss in small calculations.

Resumen en Español

En esta Tesis se analizan los algoritmos para el cálculo de la división y la raíz cuadrada para una representación punto flotante con simple y doble precisión y se proponen varias implementaciones alternativas. Los aspectos básicos relacionados con la representación punto flotante, modos de redondeo, valores especiales y excepciones, se resumen en el capítulo 1.

El uso que se hace de las funciones de división y raíz cuadrada depende, en gran medida, del mercado al que se dirige la implementación. Por ejemplo, en mercados de computación móvil, tales como PDAs, tablet PC, dónde se requiere la ejecución de aplicaciones de internet, visión de fotogramas, presentaciones, etc., no se realiza un uso masivo de estas operaciones; por este motivo, una representación de punto flotante con precisión simple es suficiente. Por otra parte, en estos segmentos de mercado, se imponen procesadores con una arquitectura de bajo coste y baja complejidad, pero con rendimiento aceptable. El algoritmo Newton–Raphson (N–R) modificado para cálculo del recíproco y división, que se propone en esta memoria, puede ser adecuado para este mercado.

Otra de las implementaciones propuestas, raíz cuadrada sin tabla de aproximación inicial *(Look–Up Table, LUT)* está basado una modificación del algoritmo N–R. Sin embargo, su rendimiento es algo deficiente, porque cada iteración requiere tres ciclos y son necesarias varias iteraciones. El número total de ciclos que se necesitan para el cálculo de la raíz cuadrada con este algoritmo es, aproximadamente, 50. Creemos que es necesario profundizar todavía más en este algoritmo para logar una implementación eficiente.

Estas funciones son frecuentes en aplicaciones de computación gráfica 3D, aplicaciones de cálculo científico, aplicaciones matemáticas, etc. La mayor parte de estas aplicaciones necesitan utilizar una representación punto flotante de doble precisión. En estos campos de aplicación, el procesador tiene una arquitectura de alto rendimiento con niveles de complejidad y coste aceptables. Otro de los algoritmos propuestos, el método AQA *(Accurate Quotient Aproximation)* puede ser de utilidad. Además, este método podría ser utilizado para procesadores futuros orientados a aplicaciones móviles.

Para aumentar el rendimiento de estas implementaciones, la aproximación inicial se obtiene mediante la LUT o la LUT combinada con algoritmos de aproximación lineal o cuadrática para reducir el área necesaria. Esta aproximación inicial está acoplada a las iteraciones del algoritmo N–R, el algoritmo de Goldschmidt *(GLD)* y los algoritmos de dígito recurrencia *(DR)*. En el capítulo 2, se discute de forma detallada la obtención de la aproximación inicial y los algoritmos N–R, GLD y DR.

Estas funciones son lentas porque efectúan secuencias de sumas y restas, multiplicaciones y suma-multiplicación, bloqueando la unidad de punto flotante del procesador *(FPU)* durante el cálculo. Algunos procesadores implementan estas funciones en software, por ejemplo el Itanium, y otros en hardware, tales como los procesadores de IBM y AMD. La implementación software proporciona un alto grado de paralelismo y no bloquea la FPU, sin embargo su rendimiento es bastante pobre. Las implementaciones hardware obtienen mejores rendimientos, pero a costa de una complejidad y coste mayores. Muchos diseñadores están dispuestos a sacrificar la velocidad para obtener costes y complejidad menores. El rendimiento y la complejidad dependen de la precisión, y la precisión depende a su vez del mercado al que está dirigido el procesador y de los requerimientos de este mercado.

En esta memoria se propone un método, el método AQA modificado, que proporciona mejoras significativas en el rendimiento, con una mayor eficiencia en términos de área de silicio que los procesadores convencionales utilizados para

cálculos en precisión simple.

En el capítulo 3 se describe una modificación del algoritmo N–R sin LUT para el cálculo del recíproco, que es la implementación de mayor eficiencia en términos de área. Es un algoritmo de latencia variable, con convergencia lineal que requiere un máximo de 22 iteraciones. Un algoritmo de latencia variable es útil en divisores auto-controlados, dónde la capacidad de procesamiento depende de la aplicación. De esta forma, se puede reducir el consumo de potencia sin sacrificar el rendimiento.

Para evaluar el método propuesto utilizamos multiplicadores de radix 4 y radix 8. El resultado de la evaluación muestra que puede haber un ligero incremento de tiempo debido a los componentes extra que se necesitan, recodificador de Booth radix 8 y recodificación entre representaciones redundantes.

Hemos comparado el método propuesto con implementaciones en procesadores actuales (Tabla 3.4). El método propuesto puede ser implementado en un multplicador radix 4 y un multiplicador radix 8 con incremento de área de un 1.5 % con respecto al área del multiplicador integrado en el procesador.

Por otra parte, se observa un incremento del 16 % en el rendimiento con respecto al procesador Itanium y a los procesadores de IBM, pero una pérdida del 4% con respecto al IBM $603e^{TM}$ y los procesadores de AMD. Esta pérdida es debida a que se retiran 1,09 bits por iteracción y no hay disponible una LUT de tamaño elevado. Por lo tanto, el rendimiento es parecido al rendimiento medio de los procesadores actuales.

En el capítulo 4, se propone una modificación del método AQA para división y raíz cuadrada, utilizando una LUT, para aplicaciones de cálculo de alto rendimiento. Esta implementación ofrece algunas estrategias atractivas adicionales a los algoritmos N–R convencional, GLD, DR y Cyrix, que se discuten más adelante.

Para evaluar el algoritmo AQA modificado hemos utilizado el multiplicador radix 4 de la FPU del IBM $603e^{TM}$, orientado a aplicaciones móviles, y el multiplicador radix 8 de la FPU del IBM $G5$, orientado a aplicaciones de alto rendimiento;

en ambos casos se ha utilizado una representació de doble precisión (ver Tabla 4.7). La comparación muestra que se obtiene una mayor eficiencia en términos de área y un mayor rendimiento. Se obtiene una reducción de área de un 70% respecto a implementaciones tradicionales basadas en LUT.

Las principales contribuciones de las propuestas presentadas en la Tesis se resumen a continuación.

Con respecto al algoritmo N–R modificado para el cálculo del recíproco, podemos destacar que no necesita tablas para la aproximación inicial y que el rendimiento se mantiene similar al obtenido con los procesadores actuales, debido a que se están calculando los resultados intermedios en representación redudante. La aproximación inicial es el complemento a dos del divisor, que puede obtenerse en paralelo con la suma de los productos parciales. Cada iteración es un ciclo, a diferencia de los dos ciclos por iteración de los procesadores actuales. Los múltiplos del divisor se mantienen constantes durante la iteración por lo tanto, es necesario generarlos solamente una vez. Además, la operación multiplicación–suma y la recodificación de Booth pueden ser realizadas en el mismo ciclo que la suma de los productos parciales. El resultado es un algoritmo de latencia variable que require sólo 22 iteraciones.

Este método puede ser extendido al cálculo de la raíz cuadrada y sólo necesita 22 iteraciones. Pero no mejora el rendimiento medio de los procesadores convencionales debido a que en cada iteración hay que cálcular el cuadrado del cociente. Cada iteración precisa tres ciclos por lo tanto, son necesarios más de 50 ciclos para obtener el resultado final.

Las principales contribuciones y ventajas del método AQA modificado son las siguientes. En comparación con el método N–R, solamente se necesita la aproximación inicial de 30 bits para obtener el cociente. Con respecto al algoritmo GLD, la metodología es parecida, pero la organización del hardware es diferente. Los resultados intermedios en representación redundante se utilizan como multiplicadores y se solapan diversas operaciones. Estas características incrementan el rendimiento. La duración del ciclo no aumenta aunque el número de bits que se

retiran en cada iteración es elevado. Esto no ocurre en los algoritmo DR, en los que la duración del ciclo aumenta con el radix.

Con respecto al algoritmo Cyrix, el algoritmo AQA modificado necesita una iteración para obtener el resultado en doble precisión. Los 28 bits más significativos y los 28 bits menos significativos del cociente se generan en paralelo y se suman durante la multiplicación. Para obtener doble precisión con el algoritmo Cyrix se necesitan dos iteraciones. Los 28 bits más significativos y los 28 bits menos significativos del cociente se generan de forma secuencial. Por otra parte, el algoritmo propuesto puede adaptarse a operar con operandos en punto fijo, mientras que el algoritmo Cyrix no es posible adaptarlo.

El tamaño de la LUT para la aproximación inicial es, aproximadamente, la mitad que el tamaño de la LUT en los procesadores convencionales. La recodificación de Booth puede llevarse a cabo en el mismo ciclo que la suma de los productos parciales y solapandose con alguna de las operaciones necesarias para la obtención del resto, lo cual aumenta el rendimiento.

El algoritmo AQA modificado puede ser extendido también al mercado de la computación móvil y al de aplicaciones de alto rendimiento de procesadores futuros. Puede ser incorporado a multiplicadores industriales con poco hardware extra.

Bibliography

[1] Ansi/ieee standard 754-1985 for binary floating-point arithmetic. 1985.

[2] *European Commission Directorate General II. The Introduction of Euro and the Rounding Amounts. Note II/28/99-EN Euro Papers No.22., 32pp, DGII/C-4-SP (99) European Commission, Directorate General, Economic and Financial Affairs, Brussels, Belgium*. February 1999.

[3] *Divide, Square root and Remainder Algorithms for Itanium Architecture*. Application Note, November 2000.

[4] Intel itanium architecture software developer's manual. 1:191–203, October 2002.

[5] Itanium 2 processor reference manual for software development and optimization. May 2004.

[6] *UltraSPARC IV Processor Architecture Overview*. Technical white paper, February, 2004.

[7] J. C. Bajard, S. Kla, and J. M. Muller. Bkm: a new hardware algorithm for complex elementary functions. *IEEE Trans. Computers*, pages 955–964, August 1994.

[8] G. W. Bewick. *Fast multiplication: algorithms and implementation*. Stanford University PhD dissertation, 1994.

[9] A. D. Booth. A signed binary multiplication technique. *Quart. J. Mech. Appl Math*, 4(2):236–240, 1951.

[10] W. S. Briggs, T. B. Brightman, and D. W. Matula. *Method and Apparatus For Performing Division Function Using A Rectangular Aspect Ratio Multiplier*. US Patent 5 046 038, US Patent Office, September 1991.

[11] W. S. Briggs, T. B. Brightman, and D. W. Matula. *Method and Apparatus For Performing The Square Root Function Using A Rectangular Aspect Ratio Multiplier*. US Patent 5 060 182, US Patent Office, October 1991.

[12] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough. The ibm z900 decimal arithmetic unit. In *Proc* 35^{th} *Asilomar Conference on Signals, Systems and Computers*, pages 1335–1339, November 2001.

[13] J. Cao and B. Wei. High-performance hardware for function generation. In *Proc* 13^{th} *Int'l symp.on Computer arithmetic*, pages 184–188, 1997.

[14] J. Cao, B. Wei, and J. Cheng. High-performance architectures for elementary function generation. In *Proc* 15^{th} *Int'l symp.on Computer arithmetic*, pages 136–144, 2001.

[15] T. C. Chen. A binary multiplication scheme based on squaring. *IEEE Trans. Computers*, 20:678–680, 1971.

[16] W. D. Clinger. How to read floating point numbers accurately. In *Proc of the Conference on Programming Language Design and Implementation*, pages 92–101, June 1990.

[17] J. N. Coleman, E. I. Chester, C. I. Softely, and J. Kaldec. Arithmetic on the european logarithmic microprocessor. *IEEE Trans. Computers*, 49(7):702–715, July 2000.

[18] M. Daumas and D. Matula. *Recoders for Partial Compression and Rounding*. Laboratoire de I'Informatique du Parallelisme, report no.97-01, January 1997.

[19] K. Diefendorff, P. K. Dubey, R. Hochprung, and H. Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, pages 85–95, March/April 2000.

[20] E.Antelo, T.Lang, and J.D.Bruguera. Computation of $\sqrt{x/d}$ in a very-high radix combined division/square-root unit with scaling and selection by rounding. *IEEE Trans. Computers*, 47(2):15–161, February 1998.

[21] E.Antelo, T.Lang, and J.D.Bruguera. Very-high radix cordic vectoring with scaling and selection by rounding. In *Proc* 14^{th} *IEEE symp.on Computer arithmetic*, pages 204–213, April 1999.

[22] E.Antelo, T.Lang, and J.D.Bruguera. High-radix cordic rotation based on selection by rounding. *Journal of VLSI Signal Processing*, 25(2):141–153, 2000.

[23] M. D. Ercegovac, L. Imbert, D. Matula, J. M. Muller, and G. Wei. Improving goldschimdt division, square root and square root reciprocal. *IEEE Trans. Computers*, 49(7):759–763, 2000.

[24] M. D. Ercegovac and T. Lang. On-the-fly conversion of redundant into conventional representations. *IEEE Trans. Computers*, 36(7):895–897, July 1987.

[25] M. D. Ercegovac and T. Lang. On-the-fly rounding. *IEEE Trans. Computers*, 41(12):1497–1503, December 1992.

[26] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[27] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Kluwer Academic Publishers, 2004.

[28] M. D. Ercegovac, T. Lang, and P. Montuschi. Very-high radix division with selection by rounding and prescaling. *IEEE Trans. Computers*, 43(8):909–918, 1994.

[29] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root and some elementary functions using small multipliers. *IEEE Trans. Computers*, 49(7):628–637, 2000.

[30] M. J. Flynn. On division by function iteration. *IEEE Trans. Computers*, 19:702–706, 1970.

[31] D. M. Gay. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*. Numerical Analysis Manuscript 90-10. AT & T Bell Laboratories, 1990.

[32] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[33] R. E. Goldschimdt. *Applications of Division by Convergence*. Massachussets Institute of Technology Master dissertation, June 1964.

[34] D. Harris. A powering unit for an opengl lighting engine. In *Proc* 35^{th} *Asilomar Conference on Signals, Systems and Computers*, pages 1641–1645, 2001.

[35] V. K. Jain, S. A. Wadecar, and L. Lin. A universal nonlinear component and its application to wsi. *IEEE Trans. Components, Hybrids and Manufacturing Technology*, 16(7):656–664, 1993.

[36] T. Jayarshee and D. Basu. On binary multiplication using the quarter square algorithm. In *Proc. Spring Joint Computer Conf.*, pages 957–960, 1974.

[37] R. Jessani and C. Olson. The floating-point unit of the powerpc 603e. *IBM J.Res.Develop*, 40(5):559–566, September 1996.

[38] R. Jessani and M. Putrino. Comparison of single-and dual-pass multiply-add fused floating-point units. *IEEE Trans. Computers*, 47(9):927–937, September 1998.

[39] I. Koren. Evaluating elementary functions in a numerical co-processor based on rational approximations. *IEEE Trans. Computers*, 40:1030–1037, 1994.

[40] T. Lang and P. Montuschi. Very-high radix square root with prescaling and rounding and a combined division/square root unit. *IEEE Trans. Computers*, pages 827–841, August 1999.

[41] D. M. Lewis. 114 mflops logarithmic number system arithmetic unit for dsp applications. *IEEE Journal of Solid-State Circuits*, 30(12):1547–1553, 1995.

[42] P. E. Madrid, B. Millar, and E. E. Swartzlander. Modified booth algorithm for high radix multiplication. In *IEEE Computer design conference*, pages 118–121, 1992.

[43] P. Markstein. *IA-64 and Elementary Functions*. Hewlett-Packard Professional Books, 2000.

[44] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture: a specification for a New Family of RISC Processors*. Morgan Kaufman Publishers Inc., San Francisco CA, 1994.

[45] J. M. Muller. *Elementary Functions. Algorithms and Implementation*. Birkhauser, 1997.

[46] J. M. Muller. A few results on table based methods. *Reliable Computing*, 5(3), 1999.

[47] A. Naini, A. Dhablania, W. James, and D. Sarma. 1-ghz hal sparc64 dual floating point unit with ras features. In *Proc* 15^{th} *IEEE symp.on Computer arithmetic*, pages 173–183, June 2001.

[48] S. Oberman, G. Favor, and F. Weber. Amd-3d now! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[49] S. F. Oberman. Floating point division and square root algorithms and implementation in the amd-k7 microprocessor. In *Proc* 14^{th} *IEEE symp.on Computer arithmetic*, pages 106–115, April 1999.

[50] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Trans. Computers*, 46(2):154–161, February 1997.

[51] S. F. Oberman and M.J.Flynn. Division algorithms and implementations. *IEEE Trans. Computers*, 46(8):833–854, August 1997.

[52] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[53] W. J. Paul and P. M. Seidel. On the complexity of booth recoding. In 3^{rd} *Conference on Real Numbers and Computers, RNC3*, pages 199–218, April 1998.

[54] J. A. Piñeiro and J. D. Bruguera. High-speed double precision computation of reciprocal, division, square root and inverse square root. *IEEE Trans. Computers*, 51(12):1377–1388, December 2002.

[55] J. A. Piñeiro, S. F. Oberman, J. M. Muller, and J. D. Bruguera. High-speed function approximation using a minimax quadratic interpolator. *IEEE Trans. Computers*, 54(3):304–318, March 2005.

[56] S. Playstation2. 2.44-gflops 300-mhz floating-point vector-processing unit for high performance 3d graphics computing. *IEEE Journal of Solid-State Circuits*, 35(7):1025–1033, July 2000.

[57] S. Playstation2. Vector unit architecture for emotion synthesis. *IEEE Micro*, 20(2):40–47, 2000.

[58] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice Hall Publishers, 2002.

[59] D. Sarma and D. Matula. Faithful interpolation in reciprocal tables. In 13^{th} *Symp.on Computer arithmetic*, volume 13, pages 82–91, 1997.

[60] D. Sarma and D. W. Matula. Faithful bipartite rom reciprocal tables. *IEEE Trans. Computers*, 47(11):1216–1222, November 1998.

[61] M. J. Schulte and J. E. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Trans.Computers*, 48(8):842–847, 1999.

[62] M. J. Schulte and J. E. Stine. The symmetric addition method for accurate function approximation. *Journal of VLSI signal processing*, 21(2):167–177, 1999.

[63] M. J. Schulte and K. E. Wires. High-speed inverse square roots. In *Proc* 14^{th} *Int'l symp.on Computer arithmetic*, pages 124–131, April 1999.

[64] E. Schwarz, L. Sigal, and T. McPherson. Cmos floating-point unit for the s/390 parallel enterprise server g4. *IBM J. Res. Develop*, 41(4/5):475–488, July/September 1997.

[65] E. M. Schwarz, R. M. Averill*III*, and L. J. Sigal. A radix-8 cmos s/390 multiplier. In 13^{th} *Symp.on Computer arithmetic, Asilomar, CA*, pages 2–9, July 1997.

[66] E. M. Schwarz, M. A. Check, C. L. K. Shum, T. Koehler, S. B. Swaney, J. D. MacDougall, and C. A. Krygowyski. The micro architecture of the ibm eserver z900 processor. *IBM Journal Research and Development*, 46(4/5):381–394, July/September 2002.

[67] E. M. Schwarz and M. J. Flynn. Hardware starting approximation for the square root operation. In *Proc* 11^{th} *Symp.on Computer arithmetic*, pages 103–111, 1993.

[68] E. M. Schwarz and C. A. Krygowski. The s/390 g5 floating-point unit. *IBM J. Res. Develop*, 43(5/6):707–721, September/ November 1999.

[69] P.-M. Seidel. High-speed redundant reciprocal approximation. *Integration the VLSI journal*, 28:1–12, 1999.

[70] H. C. Shin, J. A. Lee, and L. S. Kim. A minimized hardware architecture of fast phong shader using taylor series approximation in 3d graphics. In *Proc*

Int'l Conference on Computer Design, VLSI in Computers and Processors, pages 286–291, 1998.

[71] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating point divide and square root implementations. *ACM Computing Surveys*, 28(3):518–564, 1996.

[72] N. Takagi. Powering by a table look-up and a multiplication with operand modification. *IEEE Trans.Computers*, 46(11):1216–1222, 1998.

[73] P. T. P. Tang. Table-driven implementation of the logarithm function in ieee floating-point arithmetic. *ACM Trans. Mathematical Software*, 4(16):378–400, December 1990.

[74] P. T. P. Tang. *Table Look-up Algorithms for Elementary Functions and their Error analysis*. Argonne National Lab. Report, MCS-P194-1190, January 1991.

[75] A. T. Tsang and M. Olschanwosky. *A Study of Database 2 Customer Queries. Technical Report TR-03.413*. IBM Santa Teresa Laboratory, April 1991.

[76] D. Wong and M. Flynn. Fast division using accurate quotient approximations to reduce the number of iterations. *IEEE Trans. Computers*, 41(8):981–995, August 1992.

[77] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations. *IEEE Trans.Computers*, 43(3):278–294, March 1994.

[78] K. C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[79] M. Zhang, S. Vassiliadis, and J. Delgado-Frias. Sigmoid generators for neural computing using piecewise approximations. *IEEE Trans.Computers*, 45:1045–1050, 1996.