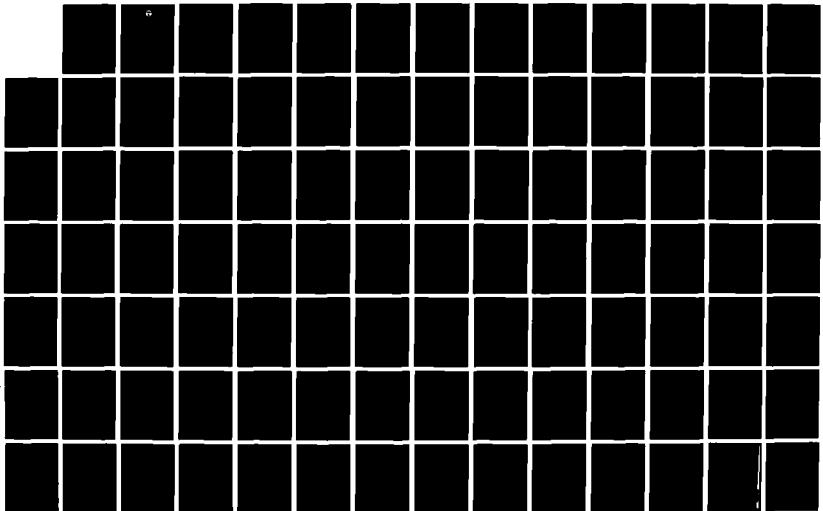


NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
VLSI FLOATING POINT CHIP DESIGN STUDY BY
JG NASH HUGHES RESEARCH LABORATORIES

1 OF 2
CR 312
UNCLAS
NOV 1985

AD-A164198



Contractor Report 312

November 1985

**VLSI FLOATING POINT CHIP
DESIGN STUDY**

J. G. Nash

Hughes Research Laboratories

Naval Ocean Systems Center San Diego, California 92152-5000

Approved for public release,
distribution unlimited

The views and conclusions contained in
this report are those of the authors and
should not be interpreted as representing
the official policies, either expressed or
implied, of the Naval Ocean Systems
Center or the U.S. Government



NAVAL OCEAN SYSTEMS CENTER SAN DIEGO, CA 92152

F. M. PESTORIUS, CAPT, USN
Commander

R.M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This task was performed for the Space and Naval Warfare Systems Command, Washington, DC 20362. Hughes Research Laboratories performed under contract N66001-83-C-0395 with the guidance of K. Bromley, Naval Ocean Systems Center, Code 741, San Diego, CA 92152-5000.

Released by
K. Bromley, Head
Signal Processing Branch

Under authority of
R.L. Petty, Head
Electromagnetic Systems
and Technology Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			Approved for public release; distribution unlimited.		
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S) NOSC CR 312		
6a NAME OF PERFORMING ORGANIZATION Hughes Research Laboratories		6b OFFICE SYMBOL (if applicable)		7a NAME OF MONITORING ORGANIZATION Naval Ocean Systems Center	
6c ADDRESS (City, State and ZIP Code) 3011 Malibu Canyon Road Malibu, CA 90265		7b ADDRESS (City, State and ZIP Code) Code 741 San Diego, CA 92152-5000			
8a NAME OF FUNDING, SPONSORING ORGANIZATION Space and Naval Warfare Systems Command		8b OFFICE SYMBOL (if applicable) S&NW-61R		9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N66001-83-C-0395	
8c ADDRESS (City, State and ZIP Code) Washington, DC 20362		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO 61153N		PROJECT NO XR02102	TASK NO Agency Accession DN305 129
11 TITLE (Include Security Classification) VLSI FLOATING POINT CHIP DESIGN STUDY					
12 PERSONAL AUTHOR(S) J.G. Nash					
13a TYPE OF REPORT Final		13b TIME COVERED FROM Sep 83 TO Nov 84		14 DATE OF REPORT (Year, Month, Day) November 1985	
15 PAGE COUNT					
16 SUPPLEMENTARY NOTATION This study is a continuation of work reported in NOSC CR 232 (March 1984).					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Very large scale integration (VLSI) Floating point chip design		
19 ABSTRACT (Continue on reverse if necessary and identify by block number) This report describes techniques for very large scale integration (VLSI) implementation of arithmetic algorithms. The report describes an algorithm for performing area-time efficient division, on-line techniques for performing bit-serial calculations, and iterative algorithms for performing square root.					
20 DISTRIBUTION AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL K. Bromley			22b TELEPHONE (Include Area Code) (619) 225-7028		22c OFFICE SYMBOL Code 741

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DD FORM 1473, 84 JAN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

1.0	Introduction and Summary	1
2.0	Division	5
2.1	Introduction	5
2.2	Recursive Division Algorithm With Prediction	10
2.2.1	Introduction	10
2.2.2	Range Transformation	12
2.2.3	Division Recursion	18
3.0	Implementing the SVD Computation Using On-Line Arithmetic	32
3.1	Introduction	32
3.2	On-line implementation of the SVD algorithm	33
3.3	Processor organization	39
3.4	Remarks	40
4.0	Algorithms for Square Root & Tangent to Cosine Conversion.....	45
4.1	Introduction	45
4.2	Square root by polynomial approximation and iterative correction	45
4.3	The reciprocal square root	47
4.4	Algorithm "pythag"	48
4.5	Tangent to cosine conversion	48
4.6	Tan ² to cosine conversion	50
4.7	Summary of Previous Algorithms	51
4.8	Estimated run times	52
4.9	"Best" algorithm for computing $(x^2+y^2)^{1/2}$, "HYPOT".....	54

LIST OF APPENDICES

Appendix A:	Division Schemes with Simplified Selection Rules and Prediction of Quotient Digits	A-1
Appendix B:	Radix-4 Division with Range Transformation.....	B-1
Appendix C:	Range Transformation of the Divisor	C-1

VLSI FLOATING POINT CHIP DESIGN STUDY

1.0 Introduction and Summary

This report summarizes the results of a study undertaken by the Hughes Research Laboratories to investigate techniques for VLSI implementations of arithmetic algorithms. While most attention in the VLSI era has gone to activities related to the shrinking of design rules and the introduction of sophisticated architectures, there is still a large contribution that is available from the use of novel algorithms for performing the arithmetic computations that underlie all computations. This is evident from the variety of new arithmetic elements seen in the newest generation of microprocessors, digital signal processing chips, special function arithmetic chips (e.g., multiplier, divider), and co-processor chips.

Within the domain of signal processing there are two arithmetic operations of particular importance, division and square root. These functions are associated with such algorithms as singular value decompositions (SVD), Givens rotations, and various other orthogonal transformations. In this report we investigate techniques for performing division and square root that are suitable for VLSI implementation.

In Section 2 we describe an algorithm for performing area-time efficient division based on a serial/parallel organization. We feel that this circuit considerably advances the state-of-art because our study shows that we can compute at rates as fast as special hardwired parallel circuits that use an order of magnitude more area. These conclusions are based on the comparison of division speed capabilities shown in Table 1 for

Table 1. Comparison of Division Capabilities

CHIP	SPEED (μ sec)	PRECISION (BITS)	COMMENTS
Intel 8087 (Slave Processor)	39	64 (Flt. Pt.)	280 x 280 mil ² (nMOS)
NS 16081 (Slave Processor)	8.9	32 (Fix Pt.)	nMOS
HP (single "÷" chip)	1.2 - 2.4	64 (Flt. Pt.)	CMOS/SOS 210 x 290 mil ²
Weitek (2 chip set)	5 - 10	24 (Flt. Pt.)	300 x 290 Mult 305 x 225 ALU
99000 μ Processor	4.9	16 (Fix Pt.)	
HUGHES	0.88	32 (Fix Pt.)	3 μ 50x150 mil ²

monolithic processor chips. Here, our calculations assumed an NMOS 3-micron technology, which is typical of that used in present day commercial semiconductor products. Unfortunately, there are not a lot of special purpose divider chips on the market with which to make comparisons. Considering that our circuit consumes a minimal amount of area because of its serial-parallel organization (shift-and-subtract), it is clearly much more area-time efficient than any of the other circuits. For example, the Hewlett Packard circuit consumes an entire chip (about 35,000 transistors) yet is half as fast. We have been assisted in this activity by Prof. Milos Ercegovac of UCLA, who performed the various algorithmic investigations.

In Section 3 we describe "on-line" techniques for bit-serial calculations. The advantage of this approach is that parallelism is achieved by overlapping arithmetic operations at the digit level and the associated feature that communication lines are needed for only one digit for each operand mantissa. Our investigation of the on-line approach was based on the numerical requirements imposed by a realistic algorithm, singular value decomposition (SVD). The results of this study, carried out by Paul Tu of UCLA, indicate that modular, efficient on-line approaches to complex algorithms are promising. However, more research work is necessary before detailed implementations are possible. In particular it will be necessary to determine how to deal with variable delays introduced by certain floating point operations, such as addition and subtraction. It was also concluded that fixed point operations would be difficult for complex on-line calculations because of the requirement that all operands fall within a certain range, which would be difficult to monitor during the course of the calculation.

Finally, in Section 4 we describe a variety of iterative algorithms for calculating the square root of a number a^2+b^2 , which is a very important calculation due to the need for obtaining the sines and cosines used in Givens' rotations. This approach, using suitable approximations for the first estimate, can in fact be quite fast. We have looked at several numerical square root algorithms, all of which use some form of polynomial approximation to the first result. It was important to use techniques that avoided loss of precision in the squaring of a and b , yet at the same time did not invoke the use of conditional

branching. We found that by using a 4th order polynomial approximation, only one iteration was needed to provide sufficient accuracy for most calculations. The total time to obtain a sin or cos function this way was equivalent to approximately 13 multiplies.

2.0 Division

2.1 Introduction

As mentioned earlier, division is a very important operation in signal processing because it is found in so many of the various matrix factorization techniques, as for example Gaussian elimination. Even though only a small fraction of the total operation associated with any given algorithm might involve division, it still must be given considerable attention because the division (or square root) operations can "bottleneck" an entire computation when concurrent architectures are in use. An example of this is the systolic array which triangularizes a matrix. In such an array only a single border cell or column of border cells might be computing divisions; however, since all processing elements (PEs) are operating in lock step, the slowest PE will determine the overall cycle time. What is needed are division and square root operations to proceed at the multiplication rate, which is generally the rate limiting factor in all the other cells. If this were the case the system would be maximally efficient.

The basic problem with division is that it is not possible to pipeline it in the same way as can be done with multiplication. This is due to the inability to know what the quotient digits are ahead of time. With multiplication the multiplier bits are known ahead of time so that they can be processed at any convenient time. As a result, our approach will require that we simplify the quotient selection process such that it is sufficiently fast that pipelining would not speed it up even if it were available.

Our divider circuit has been designed in such a way that it fits into the overall framework provided by our Multiplication Oriented Processor (MOP) chip¹ which already contains a fast multiplication circuit. Our chip design approach is characterized by a number of features. First, all arithmetic circuits are serial/parallel (S/P) (e.g., shift-and-add types) that use radix-4 arithmetic and have their own set of dedicated, high speed clocks. The S/P organization saves a large amount of space compared to a fully parallel design, and the high speed clocks and radix-4 operation are intended to prevent loss in speed compared to the pure parallel approach. In addition, all our arithmetic algorithms are intended to be based on some form of carry-save type scheme in order to eliminate carry propagation across the full precision of the word. Each arithmetic circuit has its own set of dedicated control hardware, so that all the programmer is required to do is supply the arithmetic unit with the appropriate operands. The high speed clocks are synchronized with respect to the slower system clocks which are responsible for transferring data on chip and between chips. We expect there will be 4 to 8 high speed clock cycles per slower system cycle. All clocks are of the two-phase, non-overlapping variety.

There are two basic approaches to performing division in a conventional way. The iterative or successive approximation techniques use a fast multiplier to achieve quadratic convergence rates. Often one can use a look-up table to provide a good first

1 J. G. Nash and K. Petrozolin, "VLSI Implementation of a Linear Systolic Array," presented at ICASSP, March 1985, Tampa, Florida.

estimate. For every iteration after that the precision of the result doubles. While this technique is widely used, it does exact a large price in terms of hardware. For this reason it is used primarily in applications where large precision is required and cost or chip area is not the most important criterion. The other popular approach is based on recursive techniques in which precision is proportional to the number of recursion cycles. We feel that this is the best approach for us because it lends itself to very area-time-efficient VLSI implementation and because we feel that we can obtain division rates comparable to our multiply times, which is not the case for the iterative technique. For concurrent architectures the issue of integration is a very important one because large numbers of PEs are required to achieve high throughputs. If each PE were excessively complex, then the overall system could become unwieldy.

In addition, our relatively simple S/P bit-slice arithmetic units provide an important degree of system modularity in that it is a very simple task to configure a PE with a variety of options insofar as arithmetic units and memory are concerned. This is important because we feel that it is unlikely that a single chip type will satisfy a variety of system requirements. In other words each concurrent system implementation might be built from the same basic chip modules (multipliers, dividers, adders, registers, etc.), but the actual PEs would be different

We have looked at three different recursive division algorithms, the radix-2² and radix-4 SRT,³ and the prediction technique of Ercegovic.^{4 5 6} The SRT approaches are the basic non-restoring shift-and-subtract techniques. Table 2 presents a comparison of these approaches in terms of important parameters and figures of merit. Here the S/P multiplier is used as a basis for comparison. As can be seen, the radix-2 SRT approach is the simplest, but is the slowest. On the other hand the prediction

Table 2 Comparison of division algorithms (n = number of bits) in terms of important VLSI parameters. Here CPA refers to full precision 32-bit add times to propagate carry. Relative area estimates are approximate at best.

ALGORITHM	GATE DELAYS, RECURSION	# RECURSIONS	CPA (CONVERSION REQUIRED?)	DESIGN COMPLEXITY 1 = MULT	RELATIVE AREA	RELATIVE DIVISION TIME	32-BIT DIVISION TIME (msec)	RELATIVE AREA-TIME PRODUCT
SRT Radix-2	10	n	No	1.5	1.1	$10n$	1.9	2.1
SRT Radix-4	14	$n/2$	No	2.0	1.4	$7n$	1.3	2.0
PREDICTION	RANGE TRANSFORM	6	Yes	4.0	2.5	$3n+6$	0.88	2.2
	RECORD SIGN	$n/2$	No			+CPA		
MULTIPLICATION	5	$n/2$	Yes	1.0	1.0	$5n/2 + 3$		.54
						+CPA		

- 2 J. E. Robertson, "A New Class of Digital Division Methods," IEEE Trans. Elect. Comp., EC-7, pp 218-222, Sept. 1958.
- 3 J. E. Robertson, "Methods of Selection of Quotient Digits During Digital Division," Dept. Comp. Science, Univ. Illinois, File 665, 1965.
- 4 M. D. Ercegovic, "A Higher Radix Division with Simple Selection of Quotient Digits," Proc. 6th Symposium on Computer Arithmetic, 1983.
- 5 M. D. Ercegovic and T. Lang, "Radix-4 Division with Range Transformation," unpublished manuscript, August 1984.
- 6 M. D. Ercegovic and T. Lang, "Range Transformation of the Division," unpublished manuscript, December 1984.

technique is the most complex, but has the shortest division time. All have approximately the same area-time product. Since a basic requirement is that division and multiplication times be as balanced as possible, we tend to weight the absolute division time more highly than area-time product. We have assumed for purposes of comparison that the carry propagate adder (CPA) time is 125 nSec (8 MHz) and gate delays are 6 nsec.

2.2 Recursive Division Algorithm With Prediction

2.2.1 Introduction

In this section we describe the division algorithm and its implementation in terms of NMOS circuitry. More detailed description is provided in Appendices A - C containing reports by Prof. Milos Ercegovic.

The essential requirement necessary for a fast recursive divider is a fast recursion time. This speed is usually degraded by the time required to perform the selection of the quotient digit for a particular recursion. For radix-2 operation this is not too difficult, but for radix-4 approaches this becomes more complex, resulting in long loop times. Typically, there are three basic steps in each recursion,

$$R[i+1] = r(R[i] - q_i X) \quad (1)$$

where X is the divisor, $R[i]$ is the partial remainder of the i^{th} step, $R[0]=Y$ is the dividend, q_i is a digit of the quotient, and r is the radix. Here the quotient digits must satisfy $\rho \leq q_i \leq \rho$, ρ being digit set maximum. These steps are listed below:

1. Form $q_i X$
2. Subtract and shift to obtain $R[i+1]$
3. Use quotient digit selection process to obtain q_{i+1} from $R[i+1]$

The prediction algorithm uses two techniques to speed up the set of operations described above. First, a new quotient digit selection procedure is introduced that requires only truncation

or rounding of a limited precision partial remainder to obtain q_i . Second, the quotient digit is obtained by avoiding explicit evaluation of (1); rather, it is obtained from the expression

$$q_{i+1} = \text{round} (r(\hat{R}[i] - q_i))$$

where $\hat{R}[i]$ is the low precision, non-redundant representation of the remainder. Here "round" means that q_{i+1} is selected from a rounded and truncated version of the result. Since q_i is simply an integer, the subtraction is simple. Consequently, the maximum step time for the recursion loop is given by the largest delay time associated with either the quotient selection process or the carry-save evaluation of the new partial remainder. In other words there are two separate operations occurring at the same time, so the step time is

$$T = \max(t_a + t_q + t_l, t_s + t_{cs} + t_l)$$

where

t_a = time for assimilation of $R[i]$ (carry propagation, ~6 bits)

t_q = time to subtract and round

t_s = time to select divisor multiple

t_{cs} = time for carry-save subtraction

t_l = time to load registers.

In order to perform the algorithm in this way, it must be divided into two parts, range transformation and recursion. First, for the simple quotient digit selection process to work, it is necessary that the dividend X be transformed into a range

$$(1 - \alpha) \leq X^* \leq (1 + \alpha)$$

where α is a number on the order of 2^{-6} to 2^{-9} and X^* is the transformed divisor. In the remainder of this section we describe the two basic parts of the overall algorithm, with particular emphasis on circuit implementation. More detailed discussion is provided in Appendix A-C.

2.2.2 Range Transformation

We have looked at several approaches to performing range transformation (Appendix A-C). In this section we discuss the one that appears most promising, that based on reciprocal approximation by power series.

The problem here is to compute a transformed division X^* that satisfies the relation $|X^* - 1| \leq \alpha$. The reciprocal approximation approach finds a multiplier M such that $X^* = XM$, so that this relation is satisfied. To do this we divide the divisor X into two parts, X_1 and X_2 , and set

$$d = X_1 + 2^{-k} X_2$$

where

$$X_1 = (1.X_2 \dots X_{k+1})$$

$$X_2 = (0.X_{k+2} \dots X_n).$$

Then a simple series approximation gives

$$R = 1/D = R_1 - 2^{-k} R_1^2 X_2 - e_t \quad (2)$$

where

$\hat{R}_1$ is $1/X_1$ truncated to u bits

$\hat{R}_1^2$ is $(R_1)^2$ truncated to s bits

$\hat{X}_2$ is X_2 truncated to v bits

e_t is the truncation error.

Note that our definition of X_1 assumes that it is normalized so that the most significant bit is a "1." Since we are using fixed point arithmetic, this implies that a shifter network will be required as a front end to the entire divider to perform this operation. This network will not be described here since details are already given in a previous report.⁷

The values of $\hat{R}_1$ and $\hat{R}_1^2$ can be obtained from a PLA with X_1 as an input. In Table 3, values of $\hat{R}_1$ and $\hat{R}_1^2$ are given for the choice of truncation parameters, $k=5$, $t=9$, $u=3$, $s=3$, and $v=3$. Once these have been obtained Equation (2) can be evaluated. Then we can perform the multiplication XR to yield X^* , since R is an appropriately accurate representation of $1/X$. This multiplication will be performed using the same carry-save adder that is required in the second, recursion step.

7. J. G. Nash and G. R. Nudd, "Design Study of Floating Point Systolic VLSI Chip," NOSC Final Report, Contract No. N66001-82-M-4120, September, 1983.

Table 3. Values of R_1 and R_1^2 obtained from truncated input X_1 .

X_1 $x_1x_2x_3x_4x_5x_6$	$\hat{R}_1$ $z_0z_1z_2z_3z_4z_5z_6z_7z_8$	$\hat{R}^2$ $w_0w_1w_2w_3$
1.00000	0.11111111	0.111
1.00001	0.11111000	0.111
1.00010	0.11110000	0.111
1.00011	0.11101010	0.110
1.00100	0.11100011	0.110
1.00101	0.11011101	0.101
1.00110	0.11010111	0.101
1.00111	0.11010010	0.101
1.01000	0.11001100	0.100
1.01001	0.11000111	0.100
1.01010	0.11000011	0.100
1.01011	0.10111110	0.100
1.01100	0.10111010	0.011
1.01101	0.10110110	0.011
1.01110	0.10110010	0.011
1.01111	0.10101110	0.011
1.10000	0.10101010	0.011
1.10001	0.10100111	0.011
1.10010	0.10100011	0.011
1.10011	0.10100000	0.011
1.10100	0.10011101	0.010
1.10101	0.10011010	0.010
1.10110	0.10010111	0.010
1.10111	0.10010100	0.010
1.11000	0.10010010	0.010
1.11001	0.10001111	0.010
1.11010	0.10001101	0.010
1.11011	0.10001010	0.010
1.11100	0.10001000	0.010
1.11101	0.10000110	0.010
1.11110	0.10000100	0.010
1.11111	0.10000010	0.010

Evaluation of equation (2) involves a 3×3 multiplication ($\hat{R}_1^2 \times_2$), the result of which is shifted by four digits and added to R_1 . It appears easier, rather than to complement this result and add, to first complement $\hat{R}_1$ and later complement the final result. With this scheme we can begin to obtain the radix-4 multiplier digits of R immediately since there is no addition being performed in these bit positions, as shown in Figure 1. This is convenient since we are going to perform the

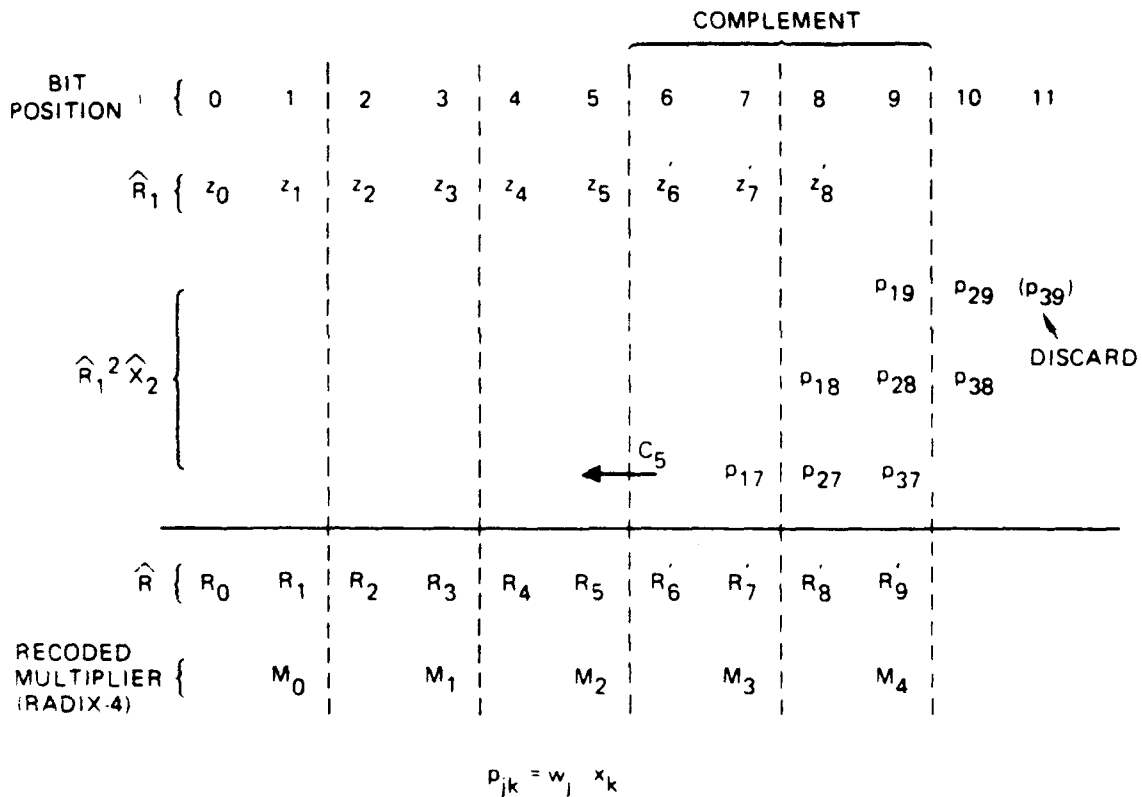


Figure 1. Functional description of the evaluation required in Equation (2).

multiplication beginning with the most significant digit of R . This is very important because we can then begin our range transformation algorithm before completion of the evaluation of (2). Expressions of the recoded multiplier bits in terms of

the variables in Figure 1 are given in Appendix C. These involve simple circuits, typically with two-gate delays.

Since we will be using the same carry-save adder for division recursion as for the multiplication required in (2), and since we will be performing the multiplication most significant bit first, some of the product will be shifted off the left end of the circuit (divider circuits shift left). However, since the first 7 bits of X^* are either a "0" or a "1," only a couple of bits will be necessary to determine X^* .

A block diagram of the range transformation circuit is shown in Figure 2. The PLA begins its operation after a signal to load the divider circuit, "LD DIV." From this point on in the control section, up to the generation of the "M" values, all logic is combinatorial. There is a multiplexing shifter register that runs off the high speed clocks, ϕ_1 and ϕ_2 , which supplies the recoded multiplier bits to the buffer driver ("BUF") controlling the selection of the appropriate value of X ($-2X$, $-X$, 0 , X , $2X$) to go to the carry-save adder. After the "LD DIV" signal has gone low, this multiplexer starts with $M_{-1} = 0$ as an input. A circuit for determination of $\hat{R}_1^{2\hat{X}}_2$ is given in Appendix C, Figure 2a, which is built from simple full adders. The PLA is a straightforward 5-input, 10-output structure, with an estimated 5 gate delays associated with it. We have performed a first order analysis of it using a standard AND-OR plane approach. Using an estimate of approximately $66 \lambda^2$ microns per cell, where λ is one-half the feature size in microns, we estimate that the entire structure would consume an $11 \times 11 \text{ mil}^2$ area. Since the input to the PLA comes from high capacitance bus lines, there would be no need for large buffer drivers as input to the PLA.

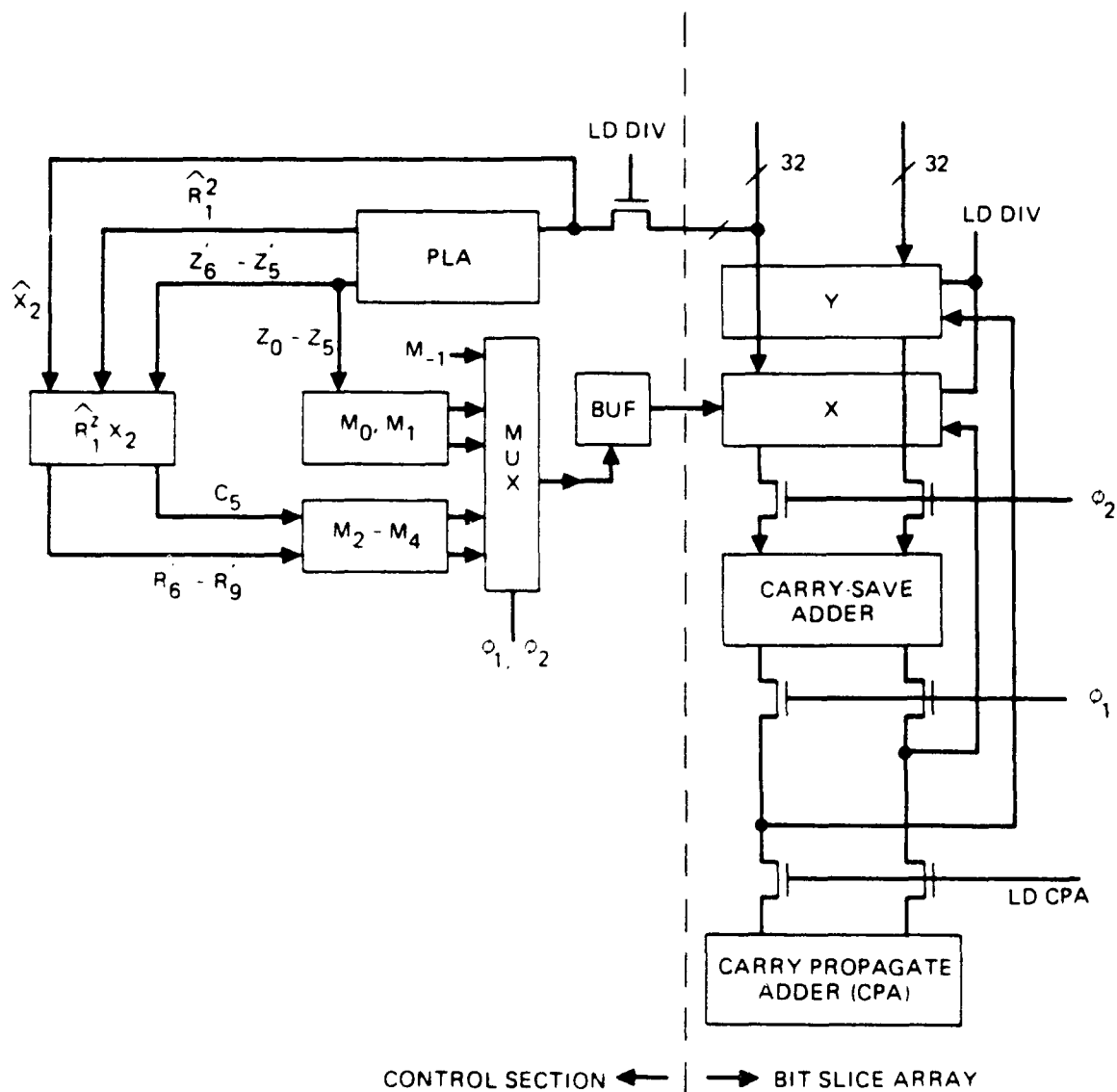


Figure 2. Block diagram of range transformation circuit.

In Table 4 we estimate the number of gate delays associated with the generation of M_i . Assuming a 32-MHz clock, all values of n should be available when required.

Table 4. Estimate of the number of gate delays from transformation initiation until generation of M_i . Here we assume a 32-MHz clock and 6-nsec gate delays (3μ NMOS).

Clock Cycle	Operation	Availability of M_i After Start		Elapsed Time (nsec)	Operation
		In Gate Delays	Time (nsec)		
1	PLA	5	31	30	Mult $M_{-1}=0$
2	Gen M_0	5 1/2	33	63	Mult M_0^{-1}
3	Gen M_1	15 1/2	93	94	Mult M_1^0
4	Gen M_2	15 1/2	93	125	Mult M_2^1
5	Gen M_3	13 1/2	81	156	Mult M_3^2
6	Gen M_4	11 1/2	69	187	Mult M_4^3

After generation of X^* , which is now in carry-save form, it is necessary to send it to the CPA, where it will be transformed into non-redundant form for use in the recursion step next.

2.2.3 Division Recursion

The basic idea in speeding up the recursion operation is to use a redundant number representation for the quotient digits so that a quotient digit can be selected at each step using a limited precision estimate of the partial remainder. This implies that it is not necessary to perform a full precision subtraction at each recursion step, thus avoiding the time consuming carry-propagation across the entire word. Instead a carry-save approach can be used along with a small CPA circuit in the most significant bits (typically 6) to determine the limited precision estimate of the partial remainder. One can use this approach and obtain q_{i+1} from

$$q_{i+1} = \text{round} (r(R[i] - q_i X^*)) \quad (3)$$

where $\hat{R}[i]$ is a limited precision estimate of the partial remainder. As mentioned in the introduction, using this approach is complicated in that it requires the evaluation of $q_i X^*$. One can sacrifice additional precision and compute q_{i+1} from the expression

$$q_{i+1} = \text{round} (r(\hat{R}[i] - q_i)) \quad (4)$$

While (4) is easier to evaluate, it requires more complexity in the range transformation due to a decreased value of α .

The value of α is determined by the degree of precision in these estimates. There are various tradeoffs here associated with the choice of precision in $\hat{R}[i]$, which are explained more fully in Appendix B. These tradeoffs can be represented by the expression

$$\alpha \leq \frac{1}{r(r+1)} (1 - (r-1) \frac{\beta}{\rho})$$

where β is a measure of precision of $R[i]$ and q_i in predicting q_{i+1} or

$$|r(R[i] - q_i) - q_{i+1}| \leq \beta.$$

Thus, reducing the precision of $\hat{R}$ (corresponding to simpler, faster circuitry) implies an increase in β or a decrease in α . The smaller value of α means additional precision required in the range transformation process described in Section 2.2. Typically the value of α would be 2^{-8} to 2^{-9} .

We have looked at a variety of basic circuits for calculations based on (4). Here we describe what we consider the

most efficient implementation of these. The basic scheme is illustrated in Figure 3, which shows the flow of information through the divider and Figure 4, which provides a more circuit oriented representation. In Figure 3 it can be seen that there are two parallel paths of computation producing the two results q_{i+1} and $R[i+1]$. The set of computations to produce q_{i+1} are

1. Propagate (assimilate) carry to produce $\hat{R}[i]$
2. Subtract q_i and shift to produce $r(\hat{R}[i] - q_i)$
3. Round result to generate q_{i+1} .

The operations in the other path are as follows:

1. Select appropriate value of $X^*(q_i X^*)$ as input to CSA
2. Perform carry-save subtraction to yield $R[i+1]$.

A block diagram of the operations required for the q_{i+1} path are illustrated in Figure 5. The CPA addition can be carried out using a "relay-type" adder as shown in Figures 6 and 7. This type of adder is relatively simple, yet carry propagation is fast, since carry information is propagated by transmission gates. (Details on this adder are provided in Ref. [7].) The carry-in input to this CPA requires a small amount of logic as described in Appendix B.

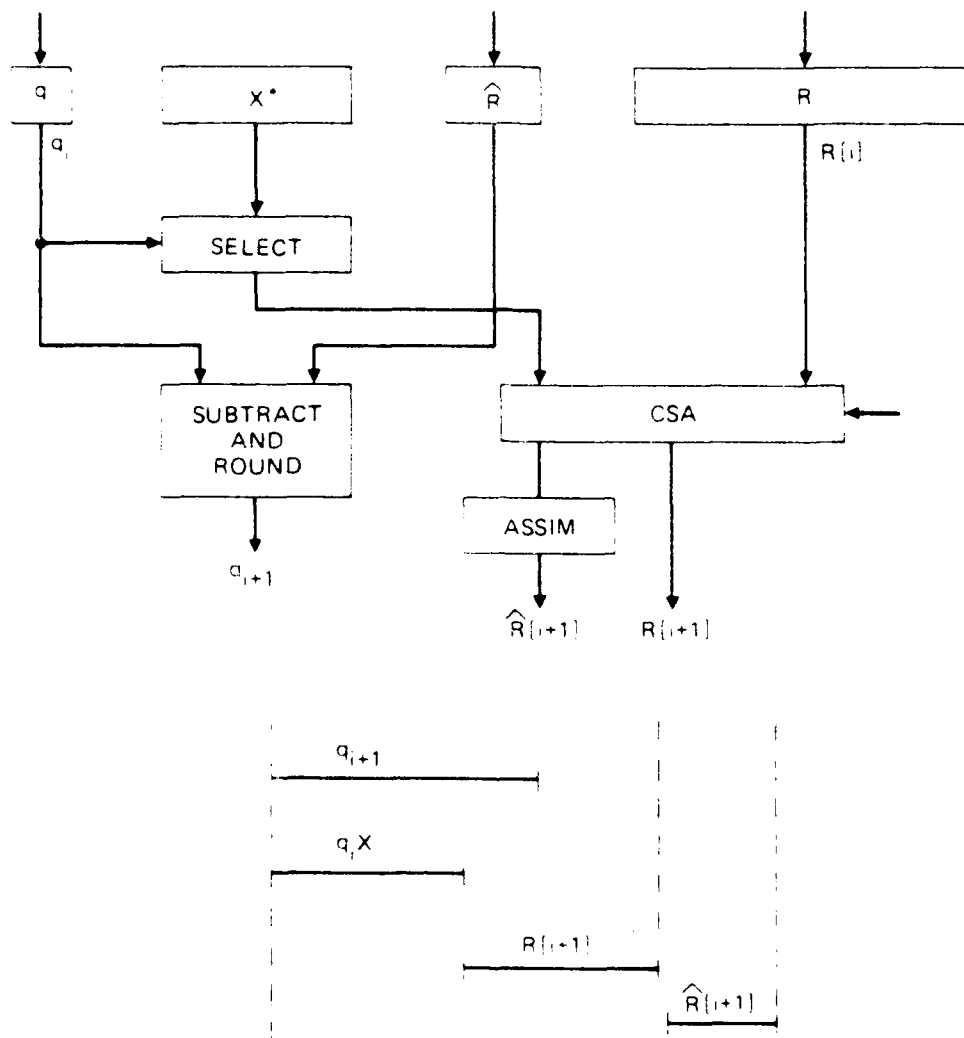


Figure 3. Functional block diagram of recursion operation data flow.

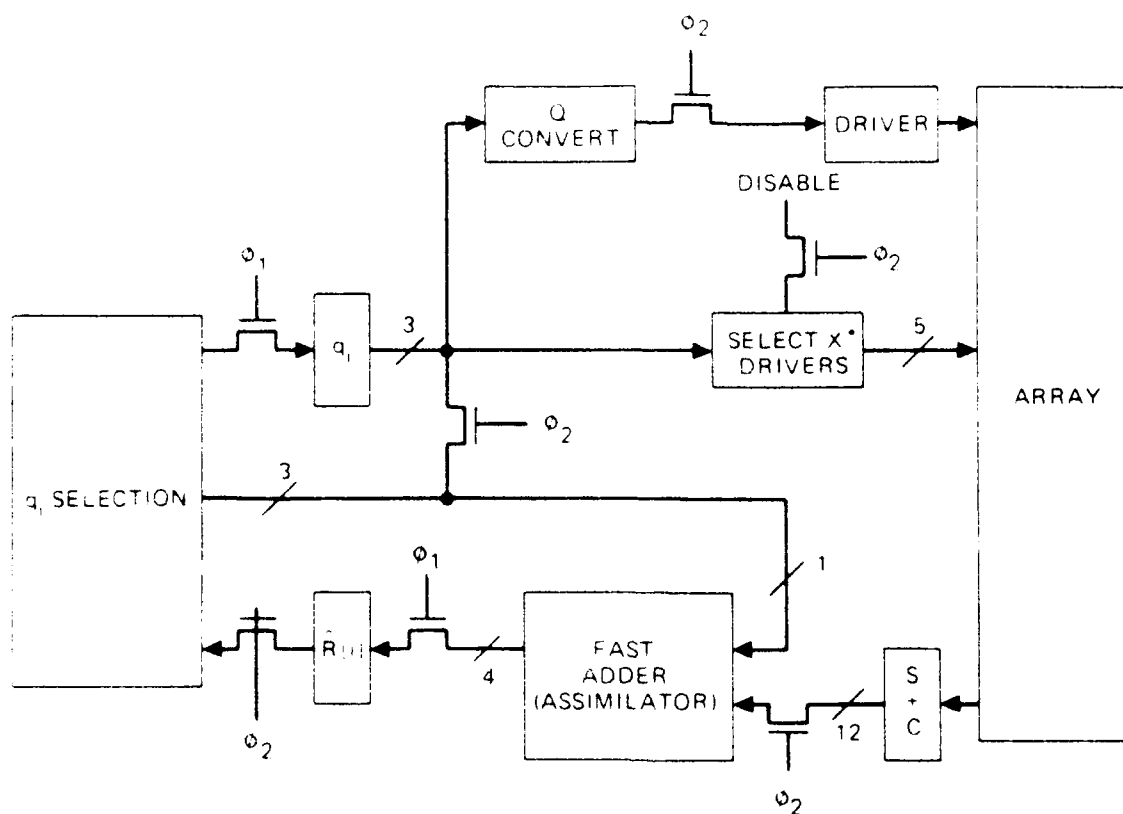


Figure 4. Basic circuit implementation of recursion control hardware.

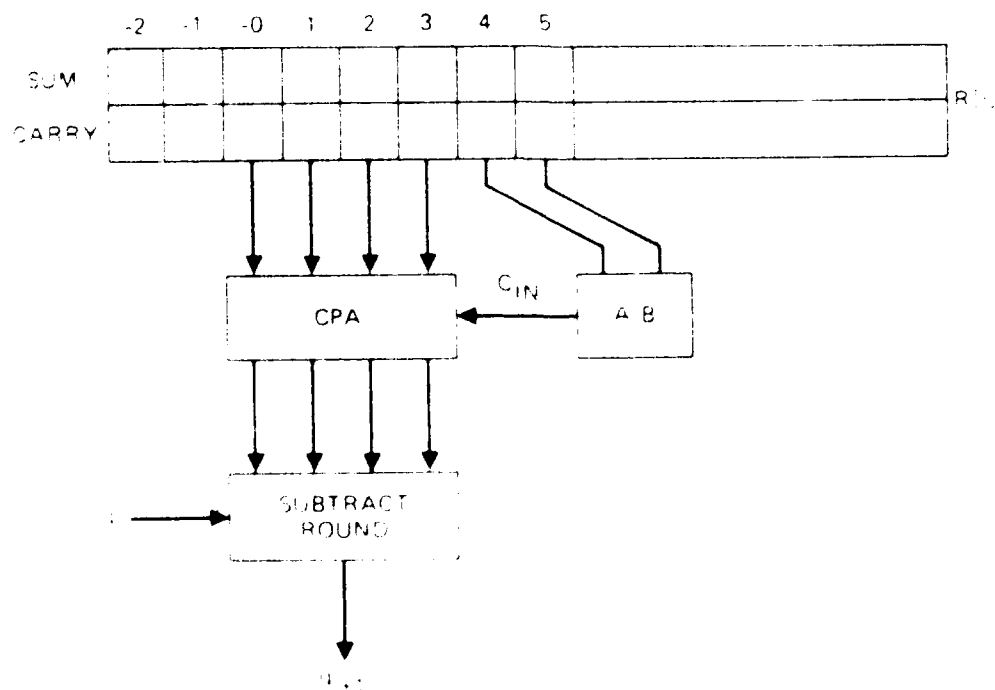


Figure 5. Block diagrams of set of operations leading to generation of q_{i+1}

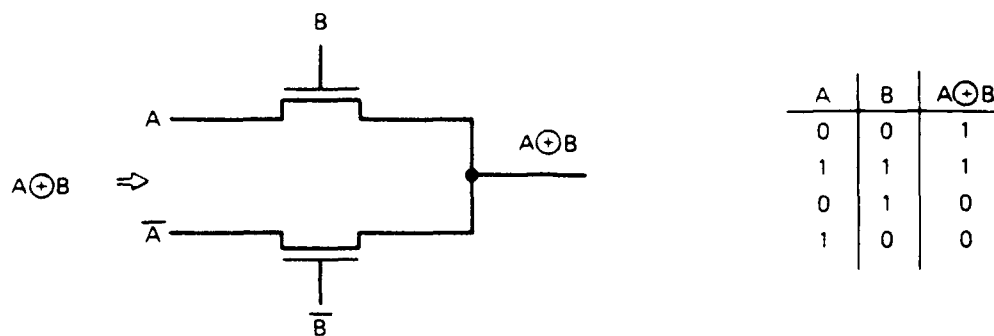
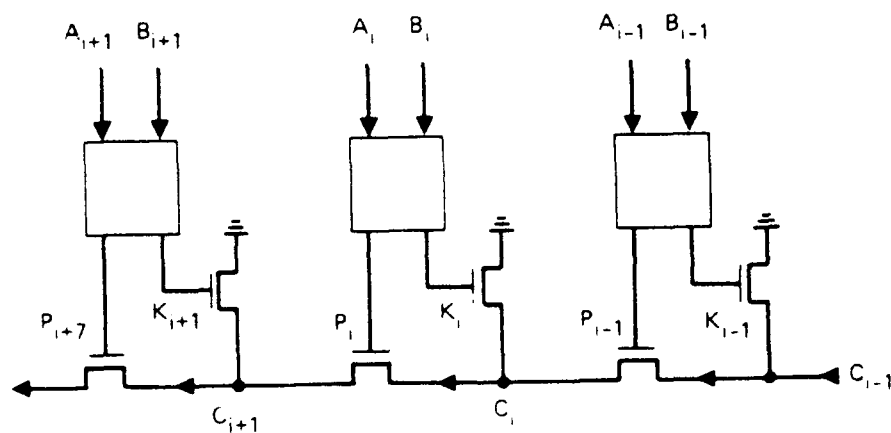


Figure 6. Exclusive OR circuit.



$$K_i = \overline{A_i + B_i}$$

$$P_i = A_i \oplus B_i$$

$$S_i = P_i \oplus C_i$$

Figure 7. Carry propagate adder used to obtain R.

After assimilation, it is necessary to do subtraction of q_i , followed by shifting and rounding. If we set $P = 4(R[i] - q_i) = (P_{-2}, P_{-1}, P_0, P_1)$ and $q = (Q_{-2}, Q_{-1}, Q_0)$ then the subtraction corresponds to obtaining P_{-2} from

$$P_{-2} = \hat{R}_0[i] \odot Q_0[i]$$

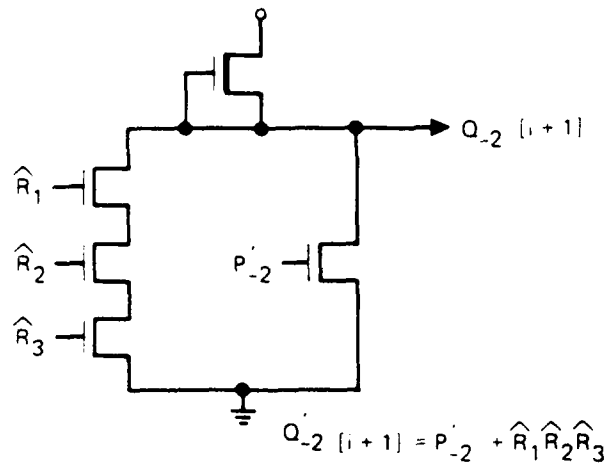
and the remainder of the P values are simply obtained by shifting the R[i] or

$$(P_{-1}, P_0, P_1) = (\hat{R}_1, \hat{R}_2, \hat{R}_3)$$

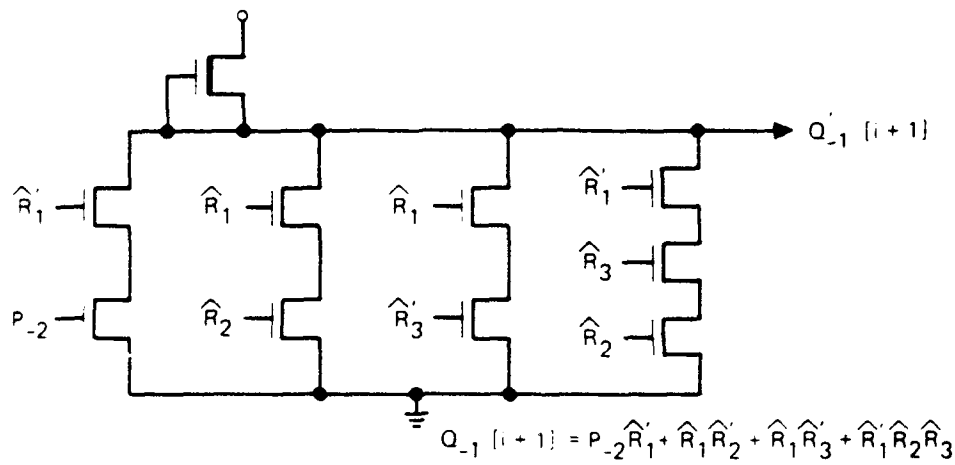
The quotient digit is obtained by rounding P if it is smaller than 2 and by the integer part of P if it is equal to or larger than 2. This corresponds to the following expressions for q_{i+1} as

$$\begin{aligned} Q_{-2}[i+1] &= P_{-2}(\hat{R}_1' + \hat{R}_2' + \hat{R}_3') \\ Q_{-1}[i+1] &= P_{-2}\hat{R}_1' + \hat{R}_1\hat{R}_2' + \hat{R}_1\hat{R}_3' + \hat{R}_1'\hat{R}_2\hat{R}_3 \\ Q_0[i+1] &= (P_{-2} + \hat{R}_1)(P_{-2} + \hat{R}_1')(\hat{R}_2 + \hat{R}_3)(\hat{R}_2' + \hat{R}_3'), \end{aligned}$$

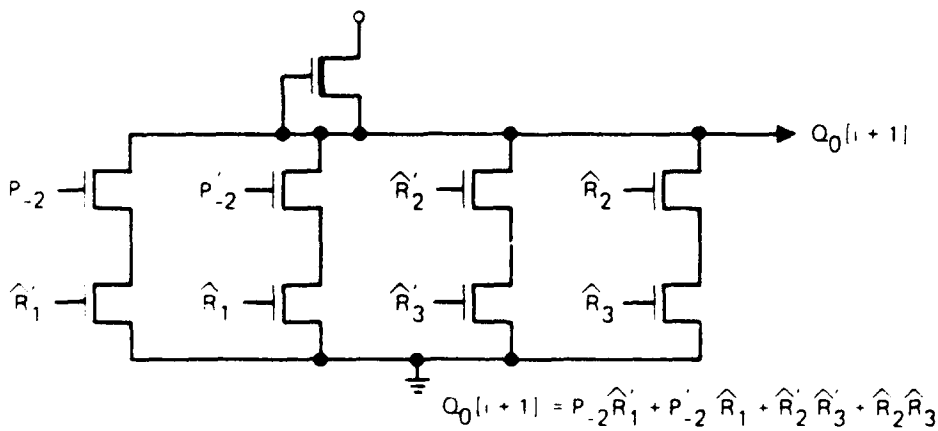
where "'" indicates complement. These expressions can be mapped into the circuits shown in Figure 8a-8c, where we have rewritten some of the switching expressions for more efficient circuit implementation. As can be seen, the subtraction and rounding can be performed in about two gate delays.



(a)



(b)



(c)

Figure 8. Circuits and logic expressions for calculation of quotient digit $q_{i+1} = (Q_{-2}, Q_{-1}, Q_0)$.

The second parallel computation path involves the selection of the appropriate value of X^* (e.g., $q_i X^*$) to add into the carry save adder in the bit slice array. This involves the use of a decoder that takes as input q_i and then drives one of 5 lines, which are running across the bit-slice array. These lines are connected to multiplexers in each bit slice which select $-2X$, $-X$, 0 , X , or $2X$ as shown in Figure 9. The full adder cell shown in Figure 9 can be built as shown in Figure 10, a circuit presently used in our S/P multiplier. We estimate that the total gate delay through this second computational path to be about 6 gate delays, 4 for the $q_i X^*$ operation and 2 for the full adder operation.

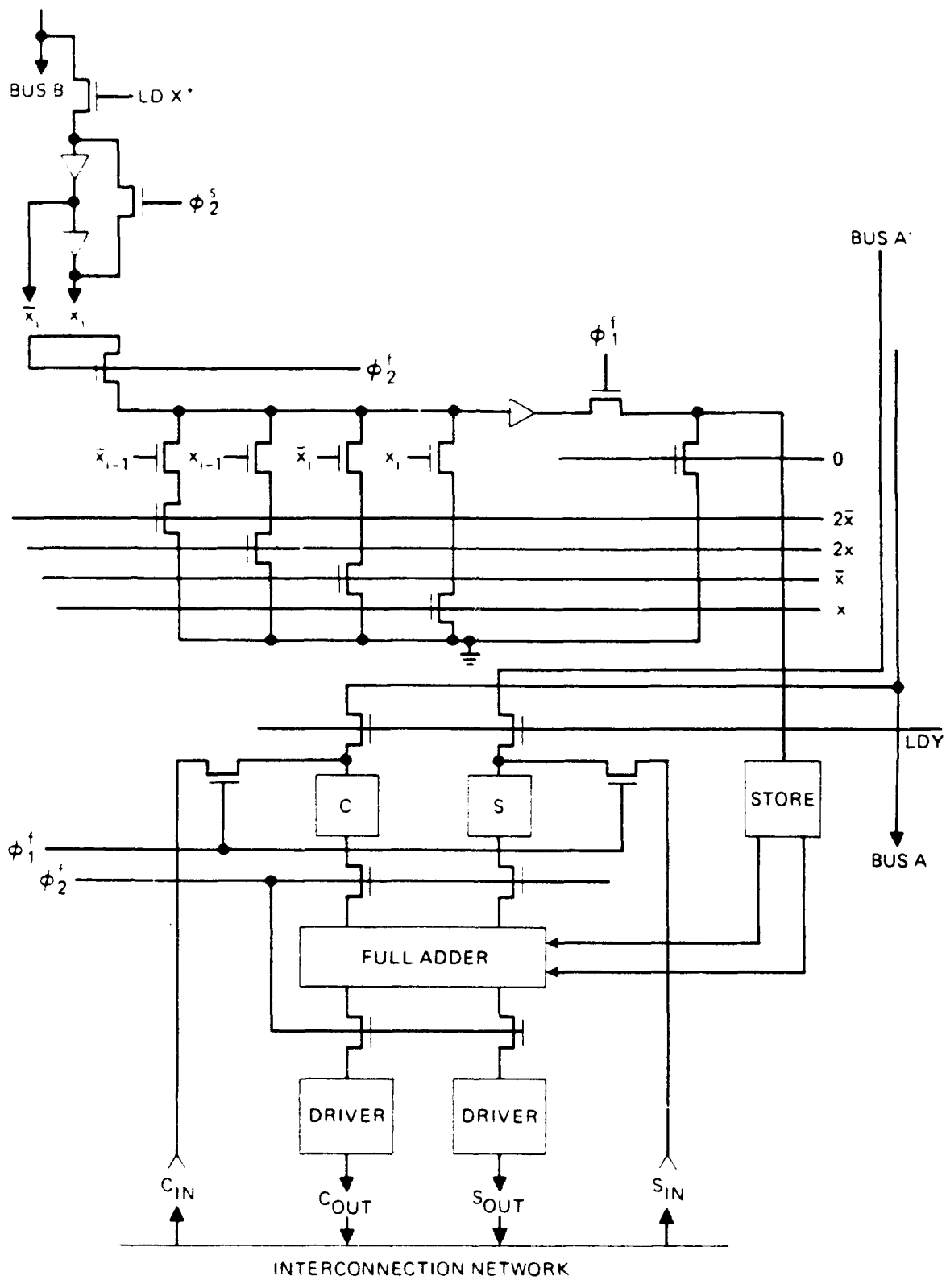


Figure 9. Divider bit-slice cell containing X^* register, multiplexer, full adder and carry, sum register. Here clock superscripts f and s refer to fast and slow clocks.

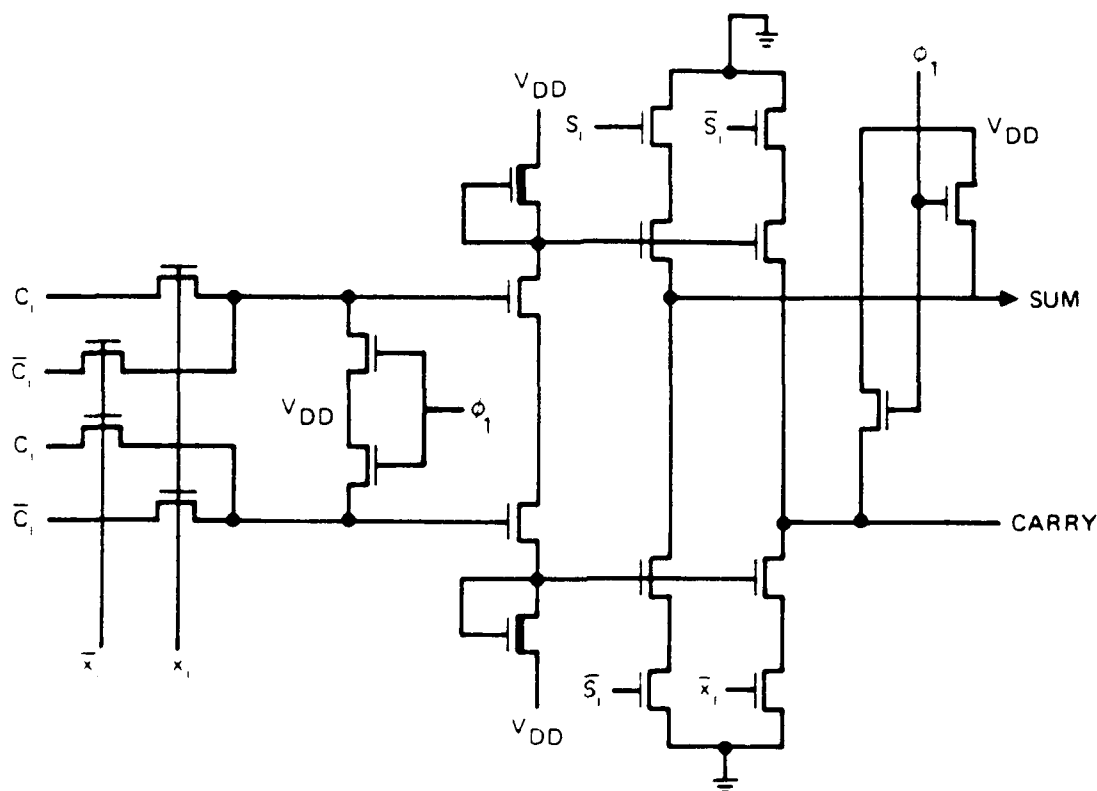


Figure 10. Tentative design of carry-save adder for use in divider bit-slice. Inputs to this cell are sum bit, S_i , carry bit C_i , and divisor bit X_i .

In Figure 9 we see that the value of X^* is taken from Bus B and loaded into a latch. This result comes from the CPA after the range transformation. The necessity of performing a CPA to produce a non-redundant X^* is an important consideration because this is a relatively slow operation. That is, after range transformation, the carry-save value of X^* must be transferred to the CPA by the slower system clocks, followed by another transfer back to the divider after the actual CPA is required. As shown in Table 5, which tabulates the overall division speed in terms of the number of high speed clock cycles, assuming the high speed clocks are four times faster than the slow speed clocks, this

round trip operation to get a non-redundant X^* is considerably time consuming. For this reason it would probably be more reasonable to include a dedicated CPA internal to the divider to avoid this delay. Although we have not achieved our goal of equalizing multiply and division rates, Table 5 indicates that the rates are comparable. Some time is saved in the division operation because the quotient digits are converted back to non-redundant form as they are created, so that a final CPA is not necessary as with the multiplier.

Table 5. Division and multiplication times assuming 32 MHz and 8 MHz arithmetic and system clocks.

Division:	Number 32 MHz Clock Cycles	
	External CPA	Internal CPA
Transformation	6	6
Transfer to CPA	4	-
CPA Operation	4	4
Return from CPA	4	-
Recursion	16	16
Total Cycles	34	26
Total Time	1.1 μ sec	880 nsec
Multiplication:		
Recursion	18	-
CPA Operation	4	-
Total Cycles	22	-
Total Time	730 nsec	-

One issue that has not been discussed is that associated with conversion of quotient digits to non-redundant form. We expect that this can be done in an "on-the-fly" manner, so that the final quotient would be available immediately after finishing the

recursion loops. We also expect that as part of this operation we can perform the necessary corrections associated with the initial normalization of X required before beginning the range transformation operation.

Based on the previous discussions we estimate that the total number of gate delays per recursion will be approximately six. Assuming a 3-micron NMOS technology, corresponding to approximately 5 nsec per gate delay, the clock speed of the arithmetic units would be approximately 32 MHz. The corresponding slow or system clocks would then operate at 8 MHz. Using these values we estimate the divider speed for 32-bit fixed-point operation would be 880 nsec. We compare this speed to data available on other, commercially available circuits in Table 1. For the purposes of comparison we note that the Hewlett Packard circuit is a dedicated divider chip based on a fully parallel, recursive implementation. As can be seen, our tentative design compares very favorably in terms of speed with all the other multiplier circuits. However, from the standpoint of area-time product, which is more appropriate for VLSI implementation, the comparison is even more favorable.

This divider design is very efficient in its usage of area because the range transformation and recursion steps share the same carry-save adder. This adder circuit will be comparable in size to that used in our S/P multiplier because the range transformation and recursion steps share the same carry-save adder. Compared to our S/P multiplier, the divider will have extra circuitry associated with the front-end normalizing circuit and a register to store the quotient digits as they are

generated. In addition, more area will be required outside the bit-slice array for control operations. We estimate that approximately $150 \times 50 \text{ mil}^2$ area would be necessary for a $3\text{-}\mu$ NMOS design of this circuit.

3.0 Implementing the SVD Computation Using On-Line Arithmetic

3.1 Introduction

On-line algorithms perform arithmetic operations in a digit-serial fashion. The operands of a computation come in one digit at a time, with the most significant digit first. After a small number of input digits have arrived, the most significant digit of the result is generated, and thereafter one more digit of the result is generated in each time step. On-line algorithms for addition/subtraction, multiplication, division and square root operations have been developed and simulated by Watanuki,⁸ Raghavendra and Ercegovic,⁹ Trivedi and Ercegovic¹⁰ and Oklobdzija and Ercegovic.¹¹

On-line arithmetic algorithms have two very attractive features. The first feature is that parallelism is achieved by overlapping arithmetic operations at the digit level. A short delay after the first digits of the operands have arrived, the

-
- 8. Osamu Watanuki, "Floating-point on-line arithmetic for highly concurrent digit-serial computation: application to mesh problems," Report No. CSD810529, Computer Science Dept. UCLA, May, 1981.
 - 9. C. S. Raghavendra and M. D. Ercegovic, "A simulator for on-line arithmetic," Proceedings of 5th Symposium on Computer Arithmetic, May 1981, pp 92-98.
 - 10. K. S. Trivedi and M. D. Ercegovic, "On-line algorithms for division and multiplication," IEEE Trans. on Computers, Vol. C-26, No. 7, July 1977, pp 681-687.
 - 11. V. G. Oklobdzija and M. D. Ercegovic, "An on-line square root algorithm," IEEE Trans. on Computers, Vol. C-31, No. 1, Jan. 1982, pp 70-75.

first digit of the result becomes available for the following operations. Thus the computation of the operations which use as input the result of previous operations does not have to wait for the previous ones to finish. The second feature of on-line arithmetic is that since data is transmitted in digit serial fashion, communications lines are needed for only one digit for each operand mantissa and the interconnection requirements are greatly reduced. This feature makes on-line arithmetic very attractive for VLSI implementation where interconnection requirement is of great concern.

In the following we discuss some of the issues involved in applying floating-point on-line algorithms to a processor array implementation of the SVD algorithm. The example we use here is the SVD algorithm by Luk,¹² which is implemented using a triangular processor array. For details of the design and the algorithm, please refer to the above mentioned reference.

3.2 On-line implementation of the SVD algorithm

The algorithm by Luk has two phases. In the first phase, the matrix x is transformed into upper triangular form. In the second phase, the upper triangular matrix is diagonalized using 2-sided Jacob rotations. The algorithm is performed on a triangular array of processors. The required computations for each node is summarized as follows. Suppose each processor is associated with 4 matrix elements $\begin{bmatrix} w & x \\ y & z \end{bmatrix}$

12. F. Luk, "A triangular processor array for computing the singular value decomposition," TR84-625, Dept. of Computer Science, Cornell Univ., July 1984.

Triangularization phase:

diagonal node processor (calculation of rotation angle)

$$\rho = \frac{w}{y}$$

$$\sin\theta = \frac{\text{sign}(\rho)}{\sqrt{1+\rho^2}}$$

$$\cos\theta = \rho \sin\theta$$

$$w' = w \cos\theta + y \sin\theta$$

$$x' = x \cos\theta + z \sin\theta$$

$$z' = -x \sin\theta + z \cos\theta$$

non-diagonal node processor (rotation)

$$w' = w \cos\theta + y \sin\theta$$

$$x' = x \cos\theta + z \sin\theta$$

$$y' = -w \sin\theta + y \cos\theta$$

$$z' = -x \sin\theta + z \cos\theta$$

Diagonalization phase:

diagonal node processor (calculation of rotation angle)

$$\rho = \frac{w + z}{x}$$

$$\sin\psi = \frac{\text{sign}(\rho)}{\sqrt{1+\rho^2}}$$

$$\cos\psi = \rho \sin\psi$$

$$w' + p = w \cos\psi$$

$$z' + r = z \cos\psi + x \sin\psi$$

$$x' + y' + q = w \sin\psi$$

$$\beta = \frac{r - p}{2q}$$

$$t = -\text{sign}(\beta) [|\beta| + \sqrt{1+\beta^2}]$$

$$\cos\phi + \frac{1}{\sqrt{1+t^2}}$$

$$\sin\phi + t \cos\phi$$

$$w'' + d_1 = p \cos^2\phi + r \sin^2\phi - 2q \sin\phi \cos\phi$$

$$z'' + d_2 = p \sin^2\phi + r \cos^2\phi + 2q \sin\phi \cos\phi$$

$$x'' + y'' = 0$$

non-diagonal node processor

$$w' + w \cos\phi - y \sin\phi$$

$$y' + w \sin\phi + y \cos\phi$$

$$x' + x \cos\phi - z \sin\phi$$

$$z' + x \sin\phi + z \cos\phi$$

$$w'' + w' \cos\psi - y' \sin\psi$$

$$y'' + w' \sin\psi + y' \cos\psi$$

$$x'' + x' \cos\psi - z' \sin\psi$$

$$z'' + x' \sin\psi + z' \cos\psi$$

$$w''' + w'' \cos\theta - x'' \sin\theta$$

$$x''' + w'' \sin\theta + x'' \cos\theta$$

$$y''' + y'' \cos\theta - z'' \sin\theta$$

$$z''' + y'' \sin\theta + z'' \cos\theta$$

A computation tree can be developed for the computation performed in each processor, which shows the data dependency between operations and provides an easy way to estimate the total on-line delay of the computation. A computation tree for the computations of the diagonalization phase for a diagonal processor node is shown in Figure 11.

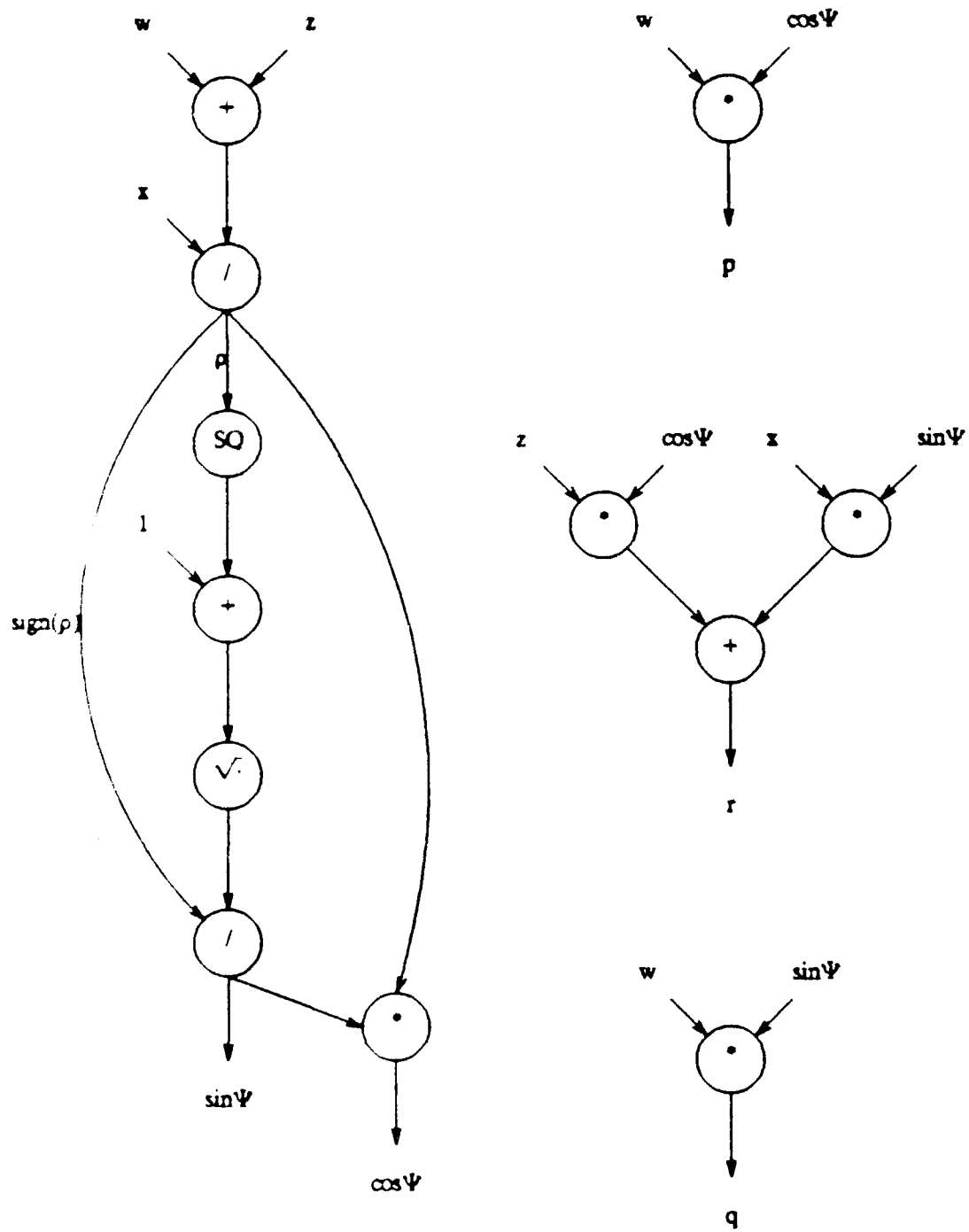


Figure 11 Example of computation tree

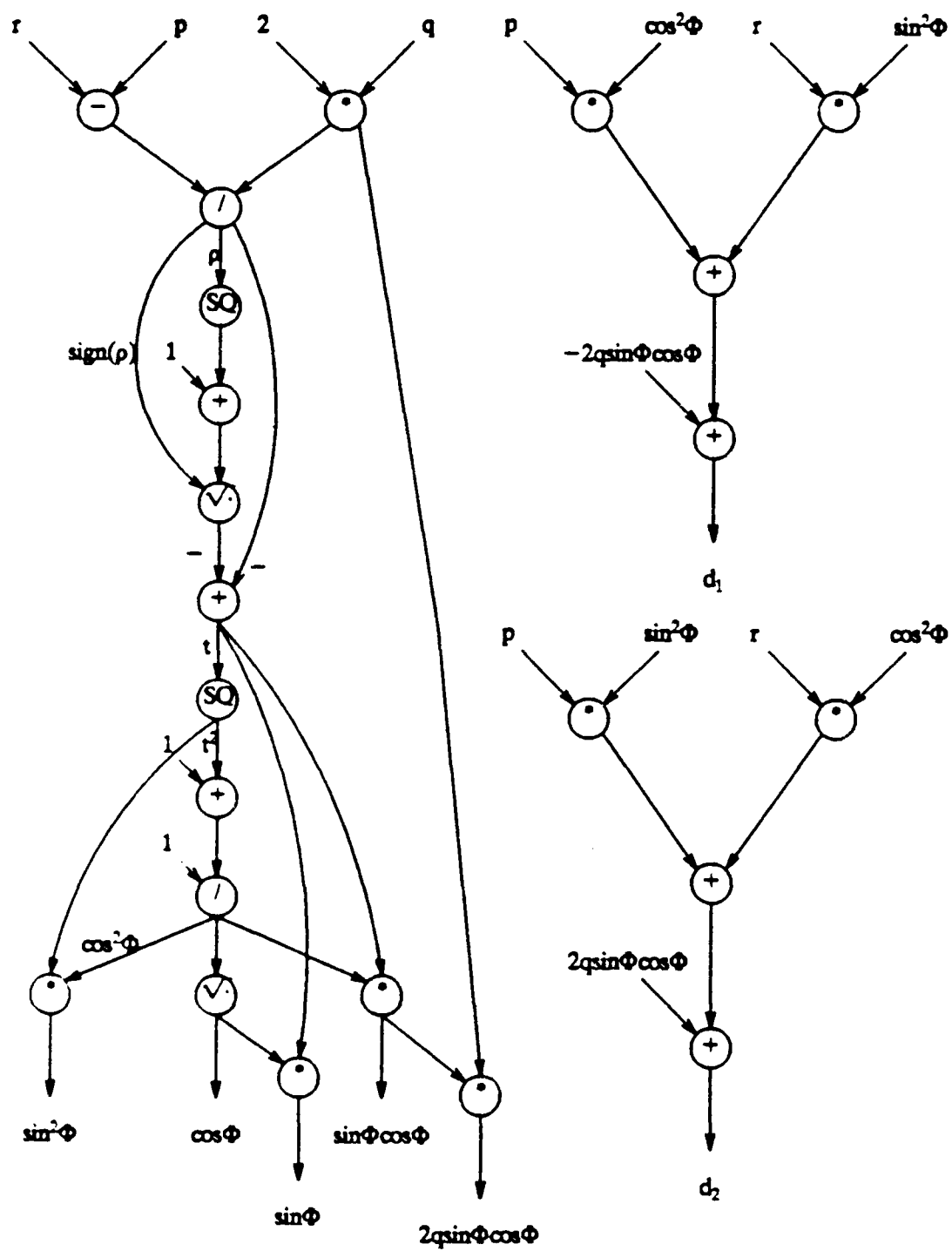


Figure 11. Example of computation tree (continued)

In each processor, one on-line arithmetic unit is needed for each arithmetic operation. This is because when performing on-line arithmetic, the arithmetic operations are highly overlapped and hence the hardware cannot be shared. Counting up the total number of operations performed in each processor for each phase, we have the number of on-line arithmetic units needed in each processor for each phase as shown in Table 6.

phase	diagonal node	non-diagonal node
1	15	12
2	36	36

Table 6. Number of arithmetic units per processor

In floating-point on-line arithmetic operations, there are two kinds of delays. One is the delay introduced by the algorithms themselves which is called the on-line delay and denoted by δ , and is defined as the number of operand digits needed for the algorithm to produce the first output digit. This delay is affected by the degree of redundancy and the radix of the number representation system used, and is usually a small integer between 1 and 4. The second kind of delay is that due to normalization of the operands and results, and is variable. For our SVD algorithm, an estimate for the lower bound of the delay of the computation would be that caused by the on-line delay of each operation. According to [13], this is

$$T_{OL} = \left[n + \sum_{i=1}^L (\delta_i + 1) \right] t_d$$

where L is the number of levels of the computation tree, δ_i is the largest on-line delay of the i^{th} level, n is the number of digits to be calculated for the result, and t_d is the digit-step time which is the time needed to load the input digits and compute one digit of the result.

3.3 Processor organization

The architecture of the on-line arithmetic processor is based on that given by Gorji-Sinaki and Ercegovac.¹³ Here we give only a brief description.

Each processor contains storage for 4 matrix elements, a global control unit (GCU), and a number of on-line arithmetic units (OLU).

Each OLU consists of an exponent unit (EU) and a number (depending upon the precision requirement) of identical processing elements (PEs). Each PE is a digit-slice on-line arithmetic unit. The structure of an on-line division unit is shown in Figure 12, and details of the design are given in [13]. It is mentioned therein that the proposed organization is capable of performing add/subtract and multiply with minor modifications and no increase in hardware. It can readily be shown that an on-line division unit can also perform the on-line square root algorithm. In Figure 12 the number of PEs is equal to the precision required and the X and Y buses are used for

13 A. Gorji-Sinaki and M. D. Ercegovac, "Design of a digit-slice on-line arithmetic unit," Proceedings of 5th Symposium on Computer Arithmetic, May 1981, pp 72-80.

correction factors applied to the square root and division operations. Also, e_x and e_y are the exponents.

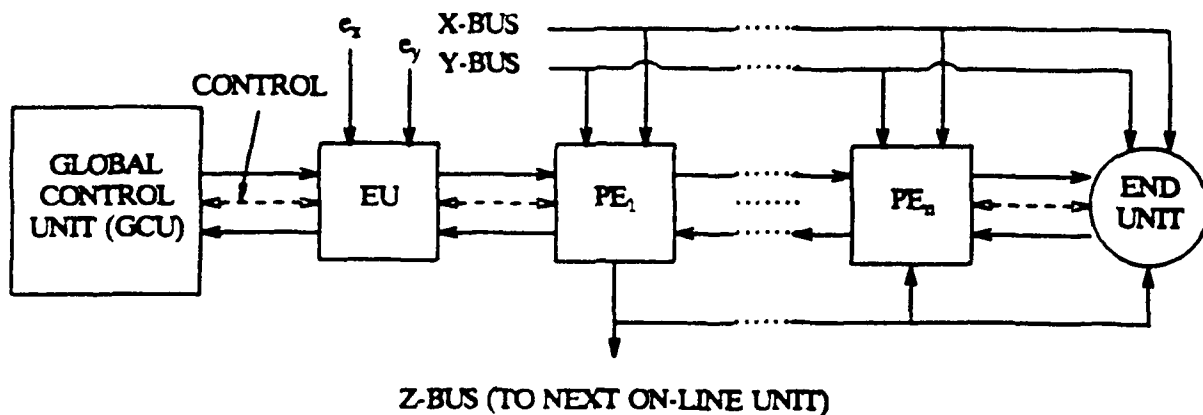


Figure 12 Organization of an on-line division unit.

For each of the data elements transmitted between processors, each processor needs to have two sets of interconnection lines: one for input and the other for output. This is because the connections cannot be shared due to the computation overlap in on-line arithmetic. Each set of connections for a single data element includes lines for the exponent, for one digit of mantissa, and possibly some control line for signaling the arrival of data.

3.4 Remarks

Fixed-point vs. floating-point arithmetic:

In fixed-point on-line arithmetic, there are certain restrictions on the domain of the operands to guarantee that the error of the result is within certain range. Hence if fixed-point arithmetic is used, the operands will have to be converted

to within the required domain before each arithmetic operation starts and later the result will have to be converted back to get the correct result. The range requirement may be different for different arithmetic operations. Thus, fixed-point on-line arithmetic induces substantial overhead and is not suitable for complex algorithms. In floating-point arithmetic, on the other hand, the algorithms always produce normalized results, assuming the operands are normalized. Hence no extra work is needed when a number of on-line operations are cascaded.

Computation of iterative algorithms:

When using on-line arithmetic to implement an iterative algorithm, the usual way to handle the iteration is to replicate the hardware for the number of iterations desired. However, this is not always necessary since usually only finite precision is required for the computation. In the case of the diagonalization phase of our SVD algorithm, for example, the total on-line delay of the operations performed on a diagonal processor node is 34 digit-steps. Each iteration includes operations in the odd-numbered rows of processors and then in the even-numbered rows. Hence the time between the start of consecutive iterations is 68 digit-steps. If the required number of digits of the result is less than, say, 60, then no replication of the processor array is needed. Likewise, doubling the array will allow up to about 130 digits of precision.

The digit-step delay:

The digit-step time, which is the time needed to compute a single digit of the result of an arithmetic operation, depends on the time needed for exponent calculation, for execution of the recursion formula, for loading the input digits, and for performing the output digit selection process. The exponent calculation involves only fixed-point addition/subtraction, and we assume that this requires less time than that required by the mantissa digit computations. So the digit-step time is basically determined by the time needed for the basic recursion formula and the digit selection process. An analysis of the digit-step time is given in [13], and we have the following formula

$$T_{\text{STEP}} = [8k + 7\text{ceilinglog}_2(k + 1) + 24] \delta_g$$

where the radix is $r = 2^k$ and δ_g is the gate delay. For $k = 2$ we note that the time step will be greater than 40 times the gate delay. This value is based on the time required to perform a carry-propagate addition across some limited number of bits, which depends on the selection rules and the precision. Typically, it might involve 8 - 9 bits for division. It is important to minimize the number of delays compared to that associated with full carry propagation in conventional arithmetic. For this reason we feel that less than 10 gate delays would be a minimally allowed value, and therefore more work is required in this regard. We note also that the number of gate delays per time step is a function of the arithmetic operation. Hence, the time step associated with the slowest

operation will determine the system time step, resulting in inefficient usage of hardware.

On-line arithmetic, compared to conventional arithmetic, has the advantages of reduced communication requirements and highly modular and uniform implementation [4]. These advantages make on-line arithmetic highly suitable for LSI/VLSI implementation. The processor organization mentioned in this paper allows replicating the basic PEs to form the OLU, each capable of performing some arithmetic operation, and then putting together a number of OLUs, along with other components such as GCU, storage devices, etc., to form a processor. This design is quite flexible and it is straightforward to design processors for other computations.

More study is needed for problems such as the accumulative error behavior, on-line algorithms for compound operations, and ways to handle the variable delay caused by both normalization and data coming through paths of different length in the computation tree.

Floating Point Operations

For a single floating point on-line arithmetic operation it is not necessary to worry about the problem of normalization of operations (i.e., for subtraction) because the on-line unit can deal with circumstances where cancellation occurs in the most significant bits. However, for multiple on-line units, as for example in the SVD algorithm, this delay, which is variable, introduces a scheduling problem. This must be handled with some appropriate hand-shaking technique, data buffering or data-driven type operation, which incurs extra hardware overhead.

Compound Operations

There are two basic approaches to implementation of an algorithm by on-line techniques. The straightforward approach, on which we will base our analysis, uses a separate on-line unit for each arithmetic operation (add, multiply, etc.), each with its own set of selection rules. An alternative approach which we have not pursued, is to build an on-line unit for an entire compound unit with its own selection rules. The advantages/disadvantages of this approach remain to be determined. However, there are possibilities that properly chosen primitives could provide ways of avoiding scaling problems for fixed point operations and synchronization problems for floating point operations.

4.0 Algorithms for Square Root and for Tangent to Cosine Conversion

4.1 Introduction

A step in the conversion of a coefficient matrix to upper triangular form, by the method of Givens rotations, involves computing the sine and cosine of a vector angle from the values of its x and y coordinates. The obvious approach evaluates the relations

$$\cos \theta = \frac{x}{(x^2 + y^2)^{1/2}} \text{ and } \sin \theta = \frac{y}{(x^2 + y^2)^{1/2}}$$

On a fixed point computer, loss of accuracy can result from the squaring of x or y in cases where these values are small. Also the conventional algorithm for extracting the square root is time consuming and becomes a bottleneck in the speed of processing.

This report discusses a number of algorithms which simplify the square root process, and other algorithms which avoid it altogether in the computation of sine and cosine.

4.2 Square root by polynomial approximation and iterative correction

Approximating the function $\sqrt{N}$ over the entire range of possible arguments is quite impractical. The problem can be simplified by recognizing that the argument need only range between 0.25 and 1.00, since any positive number can be put in this range by repeated shifts of two binary places each. After the square root is found, it can then be restored to its correct range by an equal number of shifts of one binary place each.

Polynomial approximations to the function

$$R = \sqrt{N} \quad 0.25 < N < 1.00$$

can be found by library algorithms which compute the coefficients of the least-squares best fitting polynomial when given a set of points lying on the desired curve. Starting with straight line, parabolic, or cubic approximations, the accuracy increases roughly by a factor of 4 for each increase in the order of the polynomial up to the point where the computation deteriorates due to roundoff error. In general, the square root cannot be calculated to the full accuracy of a given computer by using a high order polynomial approximation. A better approach is to use a lower order polynomial to provide a "first guess" input to a Newton-Raphson iteration formula

$$R_1 = (N/R_0 + R_0)/2$$

where,

N is the given number.

R_0 is the first approximation to the desired root.

R_1 is the second approximation to the desired root

The iteration can be applied several times. Each application will double the number of accurate places in the result until the limit set by the computer register size is reached. A tradeoff must be made between the order of the polynomial and the number of iterations required to bring the result to full accuracy.

Full accuracy on a 32-bit computer can be had from a straight line approximation followed by 3 iterations, or from a parabolic approximation followed by 2 iterations. Considerations involved in making a choice will be discussed in a later section.

4.3 The reciprocal square root

The algorithms of Section 4.2 can be used wherever the square root of a number is required. In the case where the result is to be used as a divisor, such as in the formula

$$\cos \theta = \frac{x}{(x^2 + y^2)^{1/2}}$$

it might be advantageous to use an approximation to the function

$$R = \frac{1}{\sqrt{N}}$$

followed by iterations using the formula

$$R_1 = (3 - N * R_0 * R_0) * R_0 / 2$$

It is found that attaining 32-bit accuracy requires using a 4th order polynomial followed by two iterations. The storage required for two additional coefficients and the two additional multiplications required in each step more than offset the advantage gained by eliminating one division.

4.4 Algorithm "pythag"

Moler and Morrison¹⁴ present an algorithm which computes $(x^2 + y^2)^{1/2}$ directly from x and y without taking a square root.

Initialize:

$$P = \max(|x|, |y|)$$

$$Q = \min(|x|, |y|)$$

Iterate:

$$S = Q^2 / (4P^2 + Q^2)$$

$$P = P + 2 * S * P$$

$$Q = S * Q$$

After 3 iterations, P will contain the desired result to the full accuracy of a computer with less than 60 bits.

The authors state that Pythag is potentially faster than Newton-Raphson iteration for the square root because it is cubically convergent compared with quadratic convergence for Newton-Raphson iteration. This might be true if the result were wanted to hundreds of places, but in real situations the advantage of Pythag is lost because of the greater amount of computation required per step of iteration.

4.5 Tangent to cosine conversion

The use of Givens rotations for matrix triangularization does not require the evaluation of $(x^2 + y^2)^{1/2}$ if some more direct

14 "Replacing Square Roots by Pythagorean Sums," Cleve Moler & Donald Morrison, IBM J. Res. Develop. - Vol. 27, No. 6, Nov. 83, p 577.

way can be found to compute sine and cosine from the value of the tangent.

We have found that this can be done by a variation of the method described in Section 4.2. The method involves the use of a polynomial approximation to the function

$$\cos \theta = f(\tan \theta)$$

followed by one or two steps of iteration using the formulae

$$U = C_0^2 + (C_0 T)^2$$

$$C_1 = \frac{3 - U}{2} C_0$$

where,

T = the given value of $\tan \theta$

C_0 = the first approximation of $\cos \theta$

C_1 = the second approximation of $\cos \theta$

The value of $\sin \theta$ is obtained from

$$\sin \theta = \cos \theta \tan \theta$$

The practicality of this method depends on the observation that the tangent need only range from 0 to 1 in the polynomial approximation of $\cos(\arctan \theta)$. If y is greater than x , these arguments need only be interchanged and the routine will compute $\sin \theta$ from $\cotan \theta$. The cosine is then found from

$$\cos \theta = \sin \theta \cotan \theta$$

The routine is somewhat complicated by the requirement to save the signs of x and y and to find the absolute values of x and y for use in the remainder of the computation. A comparison of these absolute values then determines if an interchange is needed. A flag must be set to indicate that the interchange must be undone at the end of the computation. The original signs of x and y are then applied directly to $\cos \theta$ and $\sin \theta$ respectively.

4.6 $\tan^2$ to cosine conversion

It may be possible to design chip hardware which will determine the greater of x and y in the absolute sense without first computing absolute values of x and y . If this is possible, some time will be saved in the execution of the routine. A consequence is that a sign will now be attached to the computed value of $\tan \theta$. The routine can be made independent of this sign by using a polynomial approximation of

$$\cos \theta = f(\tan^2 \theta)$$

The iteration steps will not require modification since $\tan \theta$ enters as the square in the formula. The signs of x and y must still be saved for restoration to $\cos \theta$ and $\sin \theta$. As before, the range of $\tan^2 \theta$ need only be from 0 to 1.

4.7 Summary of Previous Algorithms

1. Polynomial coefficients

a. Square root

First degree polynomial

$$R_0 = 11/16 N + 11/32$$

With 2 iterations, error = 6×10^{-8} rms

With 3 iterations, error $\approx 10^{-14}$ rms

Second degree polynomial

$$R_0 = -0.316394414 N^2 + 1.052146819 N + 0.259248366$$

With 2 iterations, error = 6×10^{-11} rms

2. Reciprocal square root

Third degree polynomial

$$R_0 = -2.439288044 N^3 + 6.232249739 N^2 - 5.912738903 N + 3.112778043$$

With 2 iterations, error = 6×10^{-9} rms

3. Tangent to cosine conversion

Sixth degree polynomial

$$C_0 = 0.114580644 T^6 - 0.453009177 T^5 + 0.593659040 T^4 - 0.054354432 T^3 - 0.493471808 T^2 - 0.000296313 T + 1.000001911$$

With no iterations, error = $1.597E-6$ rms

With 1 iteration, error = $6.105E-12$ rms

4 $\tan^2$ to cosine conversion

Fifth degree polynomial

$$C_0 = -0.031263642 T^5 + 0.128727929 T^4 - 0.254304918 T^3 + 0.362943041 T^2 - 0.498991485 T + 0.999983578$$

With no iterations, error = 7.678E-6 rms

With 1 iteration, error = 1.353E-10 rms

4.8 Estimated run times

In this section each algorithm is listed as a series of steps, starting in every case with the values of x and y and ending with the output of sine and cosine. The estimated number of high speed (arithmetic) clock cycles for each step is tabulated and the total elapsed time is given. A summary of results is given here:

	ALGORITHM	CLOCK CYCLES
1	a. Square root with 1st degree polynomial	83
	b. Square root with 2nd degree polynomial	77
	c. The above with 2 ALUs	60
2	Reciprocal square root with 4th degree polynomial	101
	Algorithm PYTHAG	167
3	Tan to cos with 6th degree polynomial	103
4	$\tan^2$ to cos with 5th degree polynomial	101

It is evident that 1b and 1c provide the shortest run times. If loss of accuracy due to squaring of x and y is found to be

significant, then algorithms 3 and 4 can be considered. An example of the actual operations performed for 1b is shown in Figure 13.

	load x	1
	square x	5
	store x^2	1
	load y	1
	square y	5
	add x^2	1
	shift to range	2
	store result (R_0)	1
	store shift count	1
	mult by a	5
	add b	1
	mult by N	5
	add C	1
	store result (R_0)	1
	load N	2
	divide by R_0	16
2 times	add R_0	2
	shift 1 place	2
	store result (R_1)	1
	shift to range	2
	store result $(x^2 + y^2)^{1/2} = D$	1
	load x	1
	divide by D	8
	store result (cos)	1
	load y	1
	divide by D	8
	store result (sin)	<u>1</u>
		77

Figure 13. Steps required to perform square root (method 1b).

4.9 "Best" algorithm for computing $(x^2 + y^2)^{1/2}$, "HYPOT"

This can be written as:

$$r = x(1 + \frac{y^2}{x^2})^{1/2} \quad \text{or as} \quad r = x(1 + (\frac{y}{x})^2)^{1/2}$$

On a fixed point computer, the first form results in loss of accuracy if x and y are small numbers. For example, on a computer with 30 bits to the right of the binary point, numbers which occupy only the last 15 bits will become zero when squared. The second form avoids this error by performing the division before the squaring. If only one number is small, accuracy will be improved if this number is made the numerator of the fraction. This is done in the first steps of the algorithm by taking the absolute values of x and y , comparing them, and performing an exchange if required. The algorithm as described thus far can be used with a conventional square-root procedure, and constitutes a valid method for avoiding the squaring error.

The running time of the algorithm can be shortened by eliminating the conventional square root and finding the result by an iterative procedure. Given the expression:

$$r = x(1 + (\frac{y}{x})^2)^{1/2}$$

we will compute $(1 + U)^{1/2}$ where $U = (y/x)^2$. The argument U will range only from 0 to 1 if the comparison and exchange of x and y have been performed as described above. The function $f(U) = (1 + U)^{1/2}$ can be computed by Newton-Raphson iteration:

$$f_1 = \left(\frac{1+U}{f_0} + f_0 \right) \div 2$$

where f_0 = a first guess,

f_1 = the improved value.

The iteration can be repeated any number of times, and at each step will double the number of accurate bits. The most efficient algorithm results if the iteration is performed only once. In our example this requires a first guess accurate to 15 bits to give a final result good to 30 bits.

The function $(1 + U)^{1/2}$ is a smooth function which can be well represented by a polynomial approximation of relatively few terms. The coefficients of the polynomial are computed by the least-squares procedure. This computation is done only once and is not part of the algorithm. The resulting coefficients are stored in permanent memory for use in the algorithm. The 4th order polynomial is adequate to give 17-bit accuracy. The polynomial coefficients are

		rms error:
ORDER 2	-0.070215334983 0.482062943591 1.001323600003	4.701 E-4
ORDER 3	0.024014900953 -0.106237686413 0.496400319759 1.000158637158	5.125 E-5
ORDER 4	-0.010281085549 0.044577072051 -0.19412897544 0.499294445341 1.000020416187	6.240 E-6

		rms error
ORDER 5	0.004933240713	
	-0.022614187331	
	0.055513244505	
	-0.123484054442	
	0.499864663374	
	1.000002729927	8.124 E-7
ORDER 6	-0.002537302797	
	0.012545149104	
	-0.031247590762	
	0.060093537381	
	-0.124615400309	
	0.499974514201	
	1.000000373098	1.106 E-7
ORDER 7	0.001367498511	
	-0.007323547587	
	0.019161948440	
	-0.035823977127	
	0.061743633664	
	-0.124907280763	
	0.499995274613	
	1.000000051632	1.556 E-8
ORDER 8	-0.000762269076	
	0.004416574816	
	-0.012297231345	
	0.023411232648	
	-0.037853839273	
	0.062278473297	
	-0.124978526529	
	0.499999135213	
	1.000000007192	2.242 E-9

and the algorithm steps are (4th degree polynomial)

```
load x
abs value
load y
abs value
compare
exchange (if required)
save abs x (or abs y)
divide
square
save result (U)
mult by a
add b
mult by U
add c
mult by U
add d
mult by U
add e
save result (f0)
load U (shifted)
add 0.5
divide by f0
add f0 (shifted)
mult by abs x (or abs y)
```

Algorithm HYPOT computes $(x^2 + y^2)^{1/2}$ with 30-bit accuracy. An essential feature is the division of y by x before squaring, which avoids the loss of accuracy associated with the squaring of small numbers. The possible interchange of x and y at the beginning of the algorithm does not require the setting of a logical flag, since the operation does not need to be "undone" at the end of the algorithm. There is no need to scale and unscale numbers except for two simple one-place shifts which do not need to be undone.

Algorithm HYPOT may be compared with algorithm PYTHAG [14] which is advocated by two authors at IBM. Both accomplish the same result but PYTHAG requires 147 clock cycles compared with 65 cycles for HYPOT. This corresponds to 13 multiply times, only ~six times slower than if one had a hardwired square-root circuit as fast as a multiplier.

APPENDIX A

DIVISION SCHEMES WITH SIMPLIFIED SELECTION RULES AND PREDICTION OF QUOTIENT DIGITS

Milos D. Ercegovac

August 3, 1983

Report No.1

1. Introduction

In a previous report, a paper presented at the 6th IEEE Symposium on Computer Arithmetic [ERCE83], a general division scheme was presented, based on a divisor/dividend transformation technique such that the selection of the quotient digits can be performed by simple rounding.

In this report we elaborate on the implementation and performance aspects of a radix-4 variant. Of particular interest is the fact that the next quotient digit can be obtained in parallel with the next remainder computation.

The discussion and results discussed here are preliminary and require further refinement.

2. Divisor and Dividend Transformation

We follow closely the results from [ERCE83] in this derivation.

Bound on the divisor

$$\alpha \leq \frac{1-3}{4r^2} = \frac{1}{64} > 0.0156$$

so that the transformed divisor X^* is in the interval $[1-1/64, 1+1/64]$

Transformation steps

The scaled remainders for the transformation are defined as

$$D_k = 4^{k-1}(X_k - 1)$$

where $X_0 = X$. We want that $|X_p - 1| \leq 1/64$ or, equivalently, that $D_p 4^{-p+1} \leq 1/64$. Assuming that $|D_p| \leq 1$, $p=4$.

The expressions for the transformation are:

$$D_1 = X_1 - 1 = \begin{cases} 2X_0 - 1 & \text{if } X_0 < 0.75 \\ X_0 - 1 & \text{otherwise} \end{cases}$$

That is, $S_0 \in \{0, 1\}$.

$$D_2 = 4D_1 + S_1 + S_1 D_1$$

Equivalently,

$$D_2 = \begin{cases} 5D_1 + 1 & \text{if } S_1 = 1 \\ 4D_1 & \text{if } S_1 = 0 \\ 3D_1 - 1 & \text{if } S_1 = -1 \end{cases}$$

$$D_3 = 4D_2 + S_2 + S_2 D_2 / 4$$

$$D_4 = 4D_3 + S_3 + S_3 D_3 / 16$$

The transformed divisor is

$$x^* = x_4 = D_4 4^{-3} + 1$$

The initial dividend is transformed using the following recursion:

$$y_{k+1} = y_k (1 + S_k 4^{-k}) \quad k=0,1,2,3$$

Selection of S_1 , S_2 and S_3

The selection intervals are determined by evaluating

$$D_k = (D_{k+1} - S_k) + S_k 4^{-k+1}$$

for $D_{k+1} = d_{\max}/d_{\min}$ and all values of $S_k = -2, -1, 0, 1, 2$. Assuming $-0.99 < D_4 < 0.99$ we obtain the following intervals:

$$d_{\min} = -0.99, \quad d_{\max} = 0.99$$

Selection Intervals for $k= 3$

$s = -2$	$d_{\min} = 0.2606452$	$d_{\max} = 0.7716129$	$\delta = 0.7716129$
$s = -1$	$d_{\min} = 0.0025397$	$d_{\max} = 0.5053968$	$\delta = 0.2447517$
$s = 0$	$d_{\min} = -0.2475000$	$d_{\max} = 0.2475000$	$\delta = 0.2449603$
$s = 1$	$d_{\min} = -0.4898462$	$d_{\max} = -0.0024615$	$\delta = 0.2450385$
$s = 2$	$d_{\min} = -0.7248485$	$d_{\max} = -0.2448485$	$\delta = 0.2449977$

$$d_{\min} = -0.7248484848, \quad d_{\max} = 0.7716129032$$

Selection Intervals for $k= 2$

```

s = -2, dmin = 0.3643290, dmax = 0.7918894, delta = 0.7918894
s = -1, dmin = 0.0733737, dmax = 0.4724301, delta = 0.1081011
s = 0, dmin = -0.1812121, dmax = 0.1929032, delta = 0.1195295
s = 1, dmin = -0.4058467, dmax = -0.0537381, delta = 0.1274740
s = 2, dmin = -0.6055219, dmax = -0.2729749, delta = 0.1328718

```

dmin=-0.6055218855, dmax=0.7918894009

Selection Intervals for k= 1

```

s = -2, dmin = 0.6972391, dmax = 1.3959447, delta = 1.3959447
s = -1, dmin = 0.1314927, dmax = 0.5972965, delta = -0.0999426
s = 0, dmin = -0.1513805, dmax = 0.1979724, delta = 0.0664796
s = 1, dmin = -0.3211044, dmax = -0.0416221, delta = 0.1097584
s = 2, dmin = -0.4342536, dmax = -0.2013518, delta = 0.1197526

```

dmin=-0.4342536476, dmax=1.3959447005

The overlap is indicated by "delta". A set of selection rules is given next. In these rules, d and s denote the corresponding D_k and S_k , respectively.

Select S_1

```

if (d<=-0.1)           s = 1;
else if ((d>-0.1)&(d<=0.165)) s = 0;
else                   s = -1;

```

Select S_2

```

if (d<=-0.33)         s = 2;

```

```

else if ((d>-0.33)&(d<=-0.1)) s = 1;
else if ((d>-0.1)&(d<=0.1))   s = 0;
else if ((d>0.1)&(d<=0.39))   s = -1;
else                           s = -2;

```

Select S_3

```

if (d<=-0.36)                s = 2;
else if ((d>-0.36)&(d<=-0.12)) s = 1;
else if ((d>-0.12)&(d<=0.12)) s = 0;
else if ((d>0.12)&(d<=0.36))  s = -1;
else                          s = -2;

```

3. Main Recursion with Quotient Digit Prediction

Once the divisor and the dividend are transformed into the required range, we apply the following recursion on the partial remainders.

$$q_i = \left\lfloor R_i + \text{sign}R_i * 1/2 \right\rfloor$$

$$R_{i+1} = 4(R_i - q_i x^*)$$

where $R_0 = Y^*$.

A direct implementation of this recursion would require three substeps:

(i) Select q_i ,

(ii) Generate $q_i * x^*$, and

(iii) Compute R_{i+1} .

However, it is possible to overlap the step (i) with steps (ii) and (iii). Assume that q_1 is known. Then, define the recursion as

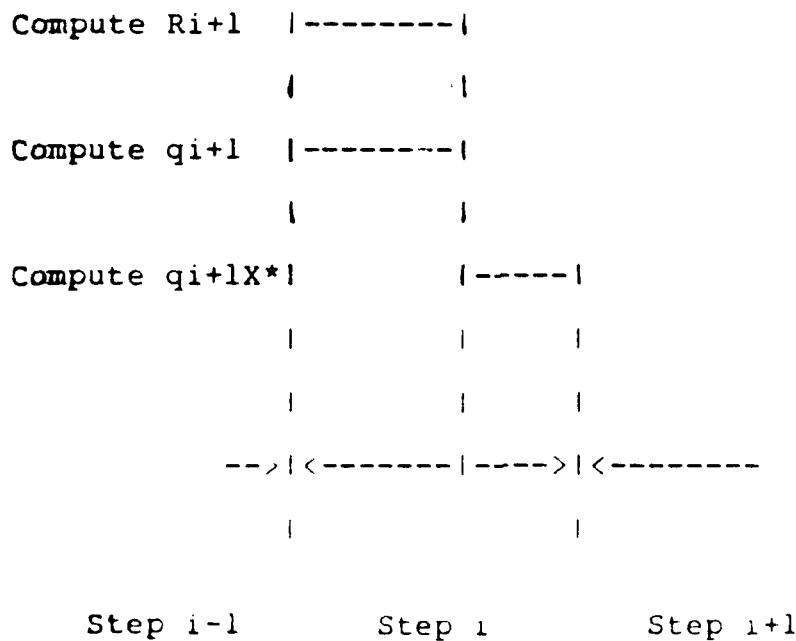
$$R_{i+1} = 4(R_i - q_i X^*)$$

$$q_{i+1} = \left\lfloor 4(\hat{R}_i - q_i + c) \right\rfloor$$

where

$$c = \begin{cases} \frac{1}{2} & \text{if } R_i \geq q_i \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

Therefore, the recursion step contains only two substeps instead of three:



The overall timing of the main recursion would look like

```
      | R1 |      | R2 | R3 | ... | Ri | Ri+1 | ...  
      | q1 | q2 | q3 | ... | qi | qi+1 | ...
```

4. A Complete Radix-4 Algorithm

We give a C version of the complete radix-4 division:

```
#define M 16  
#define X 0.5  
#define Y 0.07401786542  
#define R 4  
#define K 1  
  
main()  
{  
    double x0, y0, d1, y1, d2, y2, d3, y3, d4;  
    double quot, power;  
    float r;  
    double err, xprime, yprime, rem, remnext;  
    int i, q, qnext, s1, s2, s3, m;  
  
    x0 = X; y0 = Y; m = M; r = R;  
  
    /* Step 0 */  
    if (x0 < 0.75 )  
    {  
        d1 = 2.0*x0 - 1.0;  
        y1 = 2.0*y0;  
    }  
    else  
    {  
        d1 = x0 - 1.0;  
        y1 = y0;  
    }  
  
    /* Step 1 */  
    s1 = selone(d1);
```

```

    d2 = r*d1 + s1 + s1*d1;
    y2 = y1*(1 + s1/r );

/* Step 2 */

    s2 = seltwo(d2);

    d3 = r*d2 + s2 + s2*d2/r ;
    y3 = y2*(1 + s2 / (r*r) );

/* Step 3 */

    s3 = seltre(d3);

    d4 = r*d3 + s3 + s3*d3/(r*r);
    yprime = y3*(1 + s3 / ((r*r)*r));
    xprime = d4/((r*r)*r) + 1;
    quot = 0;
    power = 1.0;
    rem = yprime;

    if (rem > 0.0 ) q = rem + 0.5;
        else q = rem - 0.5;

/* Recursion */

    for (i = 1; i < m+1 ; ++i )
    {
        remnext = r*(rem - xprime*q);
        qnext = select(rem, q, xprime);
        quot = quot + q*power;
        err = y0/x0 - quot;
        power = power/r;
        q = qnext; rem = remnext;
    }

/* Select s1 */

selone (d)
double d;
{
    int s;

    if (d <= -0.1 )                s =  1;
    else if (( d > -0.1) & (d <= 0.165 )) s =  0;
    else                          s = -1;

    return(s);
}

/* Select s2 */

```

```

seltwo (d)
double d;
{
    int s;

    if ( d <= -0.33 )           s = 2;
    else if (( d > -0.33 ) & (d <= -0.1 )) s = 1;
    else if (( d > -0.1 ) & (d <= 0.1 )) s = 0;
    else if (( d > 0.1 ) & (d <= 0.39 )) s = -1;
    else                       s = -2;

    return(s);
}

/* Select s3 */

seltre (d)
double d;
{
    int s;

    if ( d <= -0.36 )           s = 2;
    else if (( d > -0.36 ) & (d <= -0.12 )) s = 1;
    else if (( d > -0.12 ) & (d <= 0.12 )) s = 0;
    else if (( d > 0.12 ) & (d <= 0.36 )) s = -1;
    else                       s = -2;

    return(s);
}

/* Select */

select (d, q, div)
double d, div;
int q;
{
    int s, k;
    double rtrunc, dtrunc;
    k = K;

    /* Remainder truncated to 6 bits; divisor replaced by 1 */

    s = d * 64.0; rtrunc = s; rtrunc = rtrunc / 64.0;
    s = div * 64.0; dtrunc = s; dtrunc = dtrunc / 64.0;
    dtrunc = 1.0;

    rtrunc = ( rtrunc - q * dtrunc ) * 4.0;

    if (rtrunc > 0) { s = rtrunc + 0.5;}
    else s = rtrunc - 0.5;

    return(s);
}

```

}

5. Example

$x_0 = 0.5000000000$, $y_0 = 0.0740178654$, $Q = 0.148035/308$
 $d_1 = 0.0000000000$, $y_1 = 0.1480357308$
 $s_1 = 0$

$d_2 = 0.0000000000$, $y_2 = 0.1480357308$
 $s_2 = 0$

$d_3 = 0.0000000000$, $y_3 = 0.1480357308$
 $s_3 = 0$

$d_4 = 0.0000000000$
 $x_{\text{prime}} = 1.0000000000$, $y_{\text{prime}} = 0.1480357308$, $q_1 = 0$

i	Remainder	q	Quotient	Error
predicted next q = 1				
1	0.1480357308	0	0.0000000000	0.1480357308
predicted next q = -2				
2	0.5921429234	1	0.2500000000	-0.1019642692
predicted next q = 2				
3	-1.6314283066	-2	0.1250000000	0.0230357308
predicted next q = -2				
4	1.4742867738	2	0.1562500000	-0.0082142692
predicted next q = 0				
5	-2.1028529050	-2	0.1484375000	-0.0004017692
predicted next q = -2				
6	-0.4114116198	0	0.1484375000	-0.0004017692
predicted next q = 1				
7	-1.6456464794	-2	0.1479492188	0.0000865121
predicted next q = 2				
8	1.4174140826	1	0.1480102539	0.0000254769
predicted next q = -1				
9	1.6696563302	2	0.1480407715	-0.0000050406
predicted next q = -1				
10	-1.3213746790	-1	0.1480369568	-0.0000012259
predicted next q = -1				
11	-1.2854987162	-1	0.1480360031	-0.0000002723
predicted next q = -1				

12	-1.1419948646	-1	0.1480357647	-0.0000000339
	predicted next q = 2			
13	-0.5679794585	-1	0.1480357051	0.0000000258
	predicted next q = -1			
14	1.7280821658	2	0.1480357349	-0.0000000041
	predicted next q = 0			
15	-1.0876713367	-1	0.1480357312	-0.0000000003
	predicted next q = -1			
16	-0.3506853469	0	0.1480357312	-0.0000000003

6. Binary-level Implementation

{to be done }

7. Performance Analysis

{to be done}

8. Alternatives

For transformation part:

- Have a small table of reciprocals of the truncated divisor, perhaps to 4-6 bits; use three stages of CSAs to multiply the divisor (2 bits per stage of the reciprocal); propagate carries to get the transformed divisor; repeat for the dividend but do not propagate carries.
- Use radix-2 in the transformation part; possibly much simpler implementation.

- Use radix-16 in the transformation part - details worked out on the binary level; possibly fewer steps.

For recursion part:

- Implement two steps in one clock period; double the combinational logic (CSAs, selection and multiple generator).

RADIX-4 DIVISION WITH RANGE TRANSFORMATION

M. Ercegovac and T. Lang
August, 1984 (Modified August 22)

The division algorithm described has the following characteristics:

- Is of the typical recurrence type with a redundant quotient representation with digit set between $-p$ and p .
- Simplifies the quotient selection by restricting the range of the divisor to be between $1-\alpha$ and $1+\alpha$.
- Improves the speed of execution by predicting the quotient digit.

The execution consists of two phases (Figure 1):

- (1) Transformation of the divisor X into the range $(1-\alpha) \leq X \leq (1+\alpha)$ and adjustment of the dividend.
- (2) Recurrence to obtain the quotient.

In the next section we describe the algorithm and determine the value of α required to allow the quotient selection to be done by rounding. By reducing α further it is possible to perform the rounding on a limited precision estimate of the partial remainder. Finally we consider the possibility of predicting the quotient digit. For related references see [ERCE83].

The Recurrence Step and the Value of α

This part of the algorithm consists in computing the sequence of partial remainders

$$R[i+1] = r(R[i] - q_i X) \quad (1)$$

where

X is the divisor,

$R[0] = Y$ is the dividend,

q_i is a digit of the quotient $Q = q_0.q_1q_2 \dots q_m$ with $-p \leq q_i \leq p$.

The quotient digit q_i is selected so that

$$|R[i] - q_i| \leq \beta \quad (2)$$

with β a constant to be determined.

Since the quotient digit is in the range $-\rho$ to ρ , this selection implies that

$$|R(i)| \leq \rho + \beta$$

We now determine the restriction in the range of the divisor for which this bound on the partial remainders is satisfied, and show that in this case the computation of the quotient is correct.

Since

$$R[i+1] = r(R[i] - q_i X)$$

we can write

$$R[i+1] = r(R[i] - q_i) + r(1-X)q_i$$

and since $|R[i] - q_i| \leq \beta$, $(1-\alpha) \leq X \leq (1+\alpha)$, and $|q_i| \leq \rho$ we get

$$|R[i+1]| \leq r\beta + r\alpha\rho$$

Consequently, for $|R[i+1]| \leq \rho + \beta$ it is sufficient that

$$r\beta + r\alpha\rho \leq \rho + \beta$$

which results in

$$\alpha \leq (1/r)[1 - \beta(r-1)/\rho] \quad (3)$$

Now we show that this value of α results in a correct quotient. The algorithm computes the correct quotient if

$$|(Y/X) - Q| < r^{-m}$$

with

$$Q = \sum_{i=0}^m q_i r^{-i}$$

By expansion of the recurrence we get

$$Y = X \sum_{i=0}^m q_i r^{-i} + r^{-m-1} R[m+1]$$

Therefore,

$$(Y/X) - Q = r^{-m-1} (R[m+1]/X)$$

and

$$|(Y/X) - Q| \leq r^{-m-1} \frac{|R[m+1]|_{\max}}{|X|_{\min}}$$

Consequently, the quotient is correct if

$$\frac{|R[m+1]|_{\max}}{|X|_{\min}} < r.$$

Introducing the bounds on $R[m+1]$ and on X , we get

$$(\rho + \beta)/(1 - \alpha) \leq r$$

which is satisfied for the bound on α obtained before.

Choice of β and Precision of Remainder Estimate

Since q_i is an integer, it is necessary (from (2)) that $\beta \leq 1/2$. For this value of β a full precision partial remainder is required for the selection of q_i . A larger value of β permits the use of an estimate of reduced precision. Let $\hat{R}[i]$ be an estimate of $R[i]$ to be used for the determination of the quotient digit. Assume that

$$\hat{R}[i] \leq R[i] \leq \hat{R}[i] + \delta$$

Then to assure that $|R[i] - q_i| \leq \beta$ it is sufficient that

$$|\hat{R}[i] - q_i| \leq \beta$$

and

$$|\hat{R}[i] + \delta - q_i| \leq \beta$$

Again, since q_i is an integer the smallest bound on $|\hat{R}[i] - q_i|$ is $1/2$, obtained by using rounding on $\hat{R}$, resulting in

$$|\delta + 1/2| \leq \beta$$

and therefore

$$\delta \leq \beta - 1/2$$

Consequently, to reduce the precision of $\hat{R}$ it is convenient to increase β . On the other hand, increasing β reduces α and therefore requires more preadjusting steps. Eliminating β from the expression (3) for α we get

$$\alpha \leq (1/r)[1 - (\delta + 1/2)(r-1)/\rho] \quad (4)$$

In summary, in order to have $|R[i] - q_i| \leq \beta$ it is necessary to preprocess X into the range $(1-\alpha) \leq X \leq (1+\alpha)$ and to use an estimate $\hat{R}[i]$ with precision δ to compute q_i according to the function

$$q_i = \begin{cases} \text{round}(\hat{R}[i]) & \text{if } \text{integer}(\hat{R}[i]) < \rho \\ \text{integer}(\hat{R}[i]) & \text{if } \text{integer}(\hat{R}[i]) = \rho \end{cases}$$

Quotient Digit Prediction

The division process outlined consists of a sequence of iterations. Each of these iterations is formed of three steps (see Figure 2):

- Determination of the remainder estimate $\hat{R}[i]$ in assimilated form
- Determination of a quotient digit q_i (rounding)
- Selection of a divisor multiple $q_i X$
- Subtraction to obtain new partial remainder $R[i+1]$ in carry-save form

The time of an iteration step is

$$T = t_a + t_q + t_s + t_{cs} + t_l$$

where

t_a = time for assimilation of $\hat{R}[i]$

t_q = time to round

t_s = time to select the divisor multiple

t_{cs} = time of subtraction in carry-save form

t_l = time to load the registers

To reduce the time of an iteration step it is possible to precompute the quotient digit in the previous iteration step. This results in an iteration step consisting of two parallel paths. In one the next partial remainder is obtained while in the other the next quotient digit is computed (Figure 3a).

Using the quotient calculation procedure presented before, the quotient digit q_{i+1} depends on $\hat{R}[i+1]$. In order to predict this digit it is necessary to base the prediction on $\hat{R}[i]$ (and maybe X) since $\hat{R}[i+1]$ has not been computed yet. Since

$$R[i+1] = r(R[i] - q_i X)$$

it is possible to determine q_{i+1} by

$$q_{i+1} = \text{round}(R[i+1]) = \text{round}(r(R[i] - q_i X))$$

which could be approximated by

$$q_{i+1} = \text{round}(\text{assim}(r(R[i] - q_i X)))$$

(where $\text{round}(a)$ is ρ if $a \geq \rho$.)

This prediction does not produce a significant reduction in time since the path requires the same steps as the iterative step without prediction: selection of the multiple, subtraction, assimilation, and rounding (Figure 3b).

A more promising approach is to introduce an additional approximation and compute the digit quotient as

$$q_{i+1} = \text{round}(\hat{r}(\hat{R}[i] - q_i))$$

This eliminates the step of selecting the multiple of the remainder and simplifies the subtraction, since q_i is an integer (Figure 4a). The time of a step is now

$$T = \max(t_a + t_q + t_l, t_s + t_{cs} + t_l)$$

where t_q now includes the subtraction of q_i and the rounding.

If the path for calculating q_{i+1} is longer than that to compute $R[i+1]$, it is possible to balance the two paths by including the assimilation in the second path (Figure 4b) and store $\hat{R}[i]$. In addition, to reduce the critical path, it is possible to use faster circuits in the slice required to compute $\hat{R}$ (even duplicating this slice to reduce the complexity of the interconnection might be convenient). In this case the time is

$$T = \max(t_q + t_l, t_s + t_{cs} + t_a + t_l)$$

Since this procedure of quotient digit prediction introduces an additional approximation (using q_i instead of $q_i X$) it produces the correct quotient if the divisor range is further restricted, that is an additional limitation on α is introduced. We now determine this restriction.

The basic recurrence for $i+1$ can be written

$$R[i+2] = r(R[i+1] - q_{i+1}X)$$

Replacing $R[i+1]$ in terms of $R[i]$ we get

$$R[i+2] = r(r(R[i] - q_i X) - q_{i+1}X)$$

This can be transformed into

$$R[i+2] = r(r(R[i] - q_i) - q_{i+1} + r(1-X)q_i + (1-X)q_{i+1})$$

Since the prediction is done so that

$$|r(R[i] - q_i) - q_{i+1}| \leq \beta$$

and $|1-X| \leq \alpha$, $|q_i| \leq \rho$, we obtain

$$|R[i+2]| \leq r(\beta + r\alpha\rho + \alpha\rho)$$

Consequently, for $|R[i+2]| \leq \rho + \beta$, we need

$$\rho + \beta \geq r(\beta + (r+1)\rho\alpha)$$

which results in

$$\alpha \leq \frac{1}{r(r+1)} \left(1 - (r-1)\frac{\beta}{\rho}\right) \quad (5)$$

which is $1/(r+1)$ times the value without prediction.

Prediction of the Quotient Digit and Approximation of the Remainder Estimate

To reduce the critical path in Figure 4b it is possible to compute an approximation $S[i+1]$ of $\hat{R}[i+1]$ instead of the exact value (of course this would require a further reduction of the range of X to get a correct quotient). A suitable expression for $S[i+1]$ is

$$S[i+1] = r(R[i] - q_i)$$

The calculation of $S[i+1]$ is simpler than that of $\hat{R}[i+1]$ because it does not require selection of the multiple and because the subtraction of q_i is simpler than the subtraction of $q_i X$ since q_i is an integer. The resulting time is (Figure 5)

$$T = \max(t_a + t_l, t_q + t_l, t_s + t_{cs} + t_l)$$

The restriction on the range of X is now

$$\alpha \leq \frac{1}{r(r^2 + r + 1)} \left[1 - (r-1) \frac{\beta}{\rho} \right]$$

This value of α is small and would require many transformation steps for the divisor.

Range Transformation of the Divisor

The previous algorithm requires the divisor to be transformed into the range $(1-\alpha) \leq X^* \leq (1+\alpha)$.

This transformation can be done by the following recurrence:

$$X[i+1] = X[i] + s_{i+1}X[0]2^{-(i+1)}$$

with $1/2 \leq X[0]=X < 1$ and $1-2^{-p} < X[p]=X^* < 1+2^{-p}$.

The selection of s_{i+1} is done by

$$s_{i+1} = \begin{cases} 1 & \text{if } X_0[i]=0 \text{ and } X_{i+1}[i]=0 \\ -1 & \text{if } X_0[i]=1 \text{ and } X_{i+1}[i]=1 \\ 0 & \text{otherwise} \end{cases}$$

Since $X_1[i]=X_2[i]=\dots=X_i[i]=X_0[i]$, it is more convenient to define

$$z[i] = 2^i(X[i]-1)$$

and perform the equivalent recurrence

$$z[i+1] = 2z[i] + s_{i+1}X[0]$$

with $z[0]=X-1$ and $X^* = X[p] = 2^{-p}z[p] + 1$.

Now the selection is

$$s_{i+1} = \begin{cases} 1 & \text{if } z_1[i]=1 \text{ and } z_2[i]=0 \\ -1 & \text{if } z_1[i]=0 \text{ and } z_2[i]=1 \\ 0 & \text{otherwise} \end{cases}$$

We now determine the selection intervals and show that, since the intervals overlap, it is possible to use a z with limited precision for the selection.

Since $1-2^{-k} < X[k] < 1+2^{-k}$, the range of $z[k]$ is

$$-1 < z[k] < 1$$

The selection intervals are determined by solving

$$z[i] = 1/2(z[i+1] - s_{i+1}X[0])$$

for $z[i+1] = \pm 1$ and $s_{i+1} = -1, 0, 1$.

Consequently

$$s_{i+1} = 1 \text{ if } z[i] \in (-(1+X[0])/2, (1-X[0])/2)$$

$$s_{i+1} = 0 \text{ if } z[i] \in (-1/2, 1/2)$$

$$s_{i+1} = -1 \text{ if } z[i] \in ((-1+X[0])/2, (1+X[0])/2)$$

Since $1/2 \leq X[0] < 1$, we obtain the intervals of Figure 6a. Consequently, the following selection rule results

$$s_{i+1} = \begin{cases} 1 & \text{if } z[i] \leq -1/4 \\ 0 & \text{if } -1/4 < z[i] < 1/4 \\ -1 & \text{if } z[i] \geq 1/4 \end{cases}$$

This results in an overlap of $\delta = 1/4$ so that it is necessary to assimilate over positions 0, 1, 2, and 3.

The dividend has to be adjusted in accordance to the transformation of X . That is,

$$Y^* = Y(1 + \sum s_i 2^{-i})$$

Instead of adjusting the dividend, it is possible to adjust the quotient. That is, compute $Q^* = Y/X^*$ and obtain the true quotient as

$$Q = Q^*(1 + \sum s_i 2^{-i})$$

The transformation process consists of the following steps (Figure 6b):

(1) Compute $z[0] = X[0] - 1$.

For $i=0, \dots, p$ do

(2) Determine s_{i+1} from an estimate of $z[i]$.

(3) Select the corresponding multiples of X and Y .

(4) Compute $z[i+1]$ and $Y[i+1]$.

(5) Compute $X^* = z[p]2^{-p} + 1$

Radix-4 Implementations (with TTL Timings)

We now discuss the time and complexity for several radix-4 implementations. To compare them with the TTL design presented by Taylor [TAYL81] we give timings using Taylor's delay estimates.

We choose the following parameters:

$$r=4, \quad \rho=2, \quad \delta=1/8.$$

This results in

$$\beta=5/8 \text{ and } \alpha=1/64 \text{ (without prediction) and } \alpha=1/320 \text{ (with prediction).}$$

The choice of ρ limits the divisor multiples required to -2,-1,0,1,2.

The choice of δ requires a (assimilated) remainder estimate of 7 bits (3 for the integer part and 4 for the fraction since a truncation of the carry-save partial remainder after the k -th bit produces an error of 2×2^{-k}). A better possibility is to truncate after the 6th bit and add a carry to this 6th bit. This produces an error of $\pm 2^{-k}$.

If the resulting α is too small, especially with prediction, it is possible to increase δ to 1/16 resulting in $\beta=9/16$ and $\alpha=1/128$ (with prediction). The increase in δ results in a remainder estimate of 8 bits (or 7 if the addition of the carry is done).

A) No prediction: carry-save remainder and carry-propagate estimate

In this design we implement the partial remainder in carry-save form and compute the estimate using a carry-propagate adder (CPA) of four bits and a two-level network to compute the carry into the 4-bit slice (only 4 bits of the estimate are required for the rounding). The selection of the quotient digit is performed by rounding the estimate (Figure 7). The time estimate for TTL is:

- determination of estimate (4-bit CPA and two-level network) 30 ns.
- rounding (two gate levels) 12 ns.
- select multiple 19 ns.
- carry-save subtraction 12 ns.
- set register 5 ns.

TOTAL 78 ns.

This time can be reduced to 70 ns. if the selection of the multiple is done by vector AND gates and a decoded quotient (11 ns. instead of 19 ns.)

B) Slice-save partial remainder and estimate

In this design the partial remainder is computed using slices of 2 bits with the carries between slices saved (Figure 8)(this 2-bit slice is compatible with the radix-4 design, a 4-bit slice does not seem possible).

The estimate is computed from the 9 bits corresponding to the three most significant slices (only 6 bits of the remainder have to be assimilated in this case since there is just one bit in the 7th position (Figure 8b)). The estimate can be computed using a 4-bit CPA and a 3-input AND gate (Figure 8b).

The TTL time is now:

- determination of estimate (4-bit CPA and one gate) 24 ns.
- rounding (two-level network) 12 ns.
- select multiple 19 ns.
- subtract (2-bit slice) 12 ns.
- set register 5 nsec.

TOTAL 72 ns.

Again the time can be reduced to 66 ns. by reducing the time for selection.

C) PLA for quotient generation

In this design the remainder is computed in carry-save (1-bit or 2-bit slices) form and the 6 most significant bits (12 bits or 9 bits) are used directly for the quotient generation (with a PLA)(Figure 9). The PLA probably has many AND terms since addition

is involved in the function.

The TTL time is:

- computing q_i (rounding) (PLA) 25 ns.
- select multiple 19 ns.
- subtract (carry-save) 12 ns.
- set register 5 ns.

TOTAL 61 ns.

Reducing the time of selection we get in this case 53 ns.

D) Second-level prediction

As mentioned before this prediction uses

$$q_{i+1} = \text{round}(\text{assim}(r(R[i] - q_i)))$$

For the values of these designs the reduction of α is from 1/64 to 1/320.

In this case there are two concurrent paths: the computation of q_{i+1} and that of $R[i+1]$. As mentioned in section x, the computation of the estimate can be included in any of the two; the choice being made in such a way that the critical path is reduced. We consider both possibilities.

Scheme I: Estimate calculation in q_{i+1} path (Figure 10).

The two paths are as follows:

i) Estimate and calculation of q_{i+1} .

This path includes the computation of the estimate, the subtraction of q_i and the rounding. The computation of the estimate can be done as in cases A) or B) (Figures 7 and 8).

The subtraction of q_i and the rounding is done as follows: Let us call $P = 4(\hat{R}[i] - q_i) = (P_{-2}, P_{-1}, P_0, P_1)$. Since we subtract and then multiply by 4 (and q_i is an integer), the sign of P is obtained by subtracting Q_0 from $\hat{R}_0$, that is,

$$P_{-2} = \hat{R}_0[i] \oplus Q_0[i]$$

The value of P is obtained by shifting $\hat{R}[i]$, that is,

$$(P_{-1}, P_0, P_1) = (\hat{R}_1, \hat{R}_2, \hat{R}_3)$$

The quotient digit is obtained by rounding P if it is smaller than 2 and by the integer part of P if it is equal or larger than 2. This results in the following table:

	P_{-2}	$\hat{R}_1$	$\hat{R}_2$	$\hat{R}_3=1$			$\hat{R}_3=0$		
				Q_{-2}	Q_{-1}	Q_0	Q_{-2}	Q_{-1}	Q_0
0 to 1	0	0	0	0	0	1	0	0	0
1 to 2	0	0	1	0	1	0	0	0	1
2 to 3	0	1	0	0	1	0	0	1	0
3 to 4	0	1	1	-	-	-	0	1	0
-4 to -3	1	0	0	1	1	0	-	-	-
-3 to -2	1	0	1	1	1	0	1	1	0
-2 to -1	1	1	0	1	1	1	1	1	0
-1 to 0	1	1	1	0	0	0	1	1	1

From the table the following expressions result:

$$Q_{-2}[i+1] = P_{-2}(\hat{R}_1' + \hat{R}_2' + \hat{R}_3')$$

$$Q_{-1}[i+1] = P_{-2}\hat{R}_1' + \hat{R}_1\hat{R}_2' + \hat{R}_1\hat{R}_3' + \hat{R}_1'\hat{R}_2\hat{R}_3$$

$$Q_0[i+1] = (P_{-2}' + \hat{R}_1)(P_{-2} + \hat{R}_1')(\hat{R}_2 + \hat{R}_3)(\hat{R}_2' + \hat{R}_3')$$

Substituting P_{-2} in these expressions it is clear that the subtraction and rounding can be performed in two gate levels.

The timing for this scheme is

- determination of the estimate (like in A or B) 30 ns. or 24 ns.
- subtraction of q_i and rounding (2 levels) 12 ns.
- set register 5 ns.

TOTAL 47 ns. or 41 ns.

As in C) the complete calculation of the quotient could be done using a PLA with 12 inputs (or 9 inputs if 2-bit slices are used) resulting in a total time of $25+5=30$ ns. As commented there, since addition is involved the PLA might have many AND terms.

ii) Calculation of the partial remainder

- selection of the multiple 19 ns.
- subtraction (1-bit or 2-bit slice) 12 ns.
- set register 5 ns.

TOTAL 36 ns.

Again the selection can be reduced to 11 ns resulting in a total of 28 ns.

The longest path is therefore 30 ns if the PLA is used and 41 ns if it is not used.

If the PLA cannot be used, the critical path can be reduced by using faster circuits in the calculation of the estimate and in the rounding. For example using FAST circuits would produce a delay of...

Scheme II: Estimate calculation as part of partial remainder path

From scheme I it can be seen that moving the calculation of the estimate to the partial remainder path would lengthen it excessively (at least for the TTL timings being considered).

What could be done is to move part of the estimate calculation. An attractive possibility would be to include here the calculation of the carry into the 4-bit slice (one

gate) and to compute also the p's and g's of the 4-bit slice (Figure 12). This would add one level to the partial remainder path and reduce one level from the quotient path. The result would be that both paths would be approximately of 35 ns. Of course, different balances can be achieved if some of the circuits are of a faster technology.

References

[ERCE83] M.D. Ercegovic, "A Higher Radix Division with Simple Selection of Quotient Digits", Proc. 6th Symposium on Computer Arithmetic, 1983.

[TAYL81] G. S. Taylor, "Compatible Hardware for Division and Square Root", Proc. 5th Symposium on Computer Arithmetic, 1981, pp.127-134.

DIVISOR $X(0)$ DIVIDEND $I(0)$

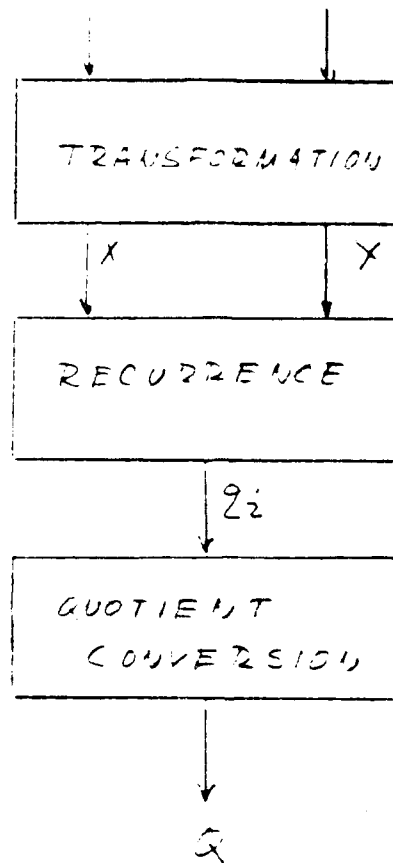
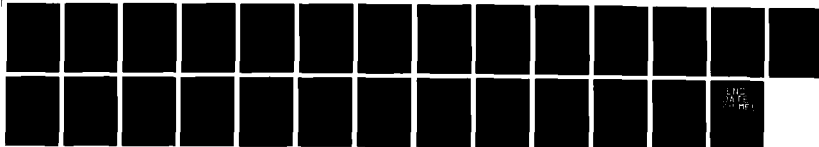


FIGURE 1. DIVISION SCHEME WITH
PRELUDE TRANSFORMATION

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA
VLSI FLOATING POINT CHIP DESIGN STUDY BY
JG NASH HUGHES RESEARCH LABORATORIES

2 OF 2
CR 312
UNCLAS
NOV 1985

A0-A164/98



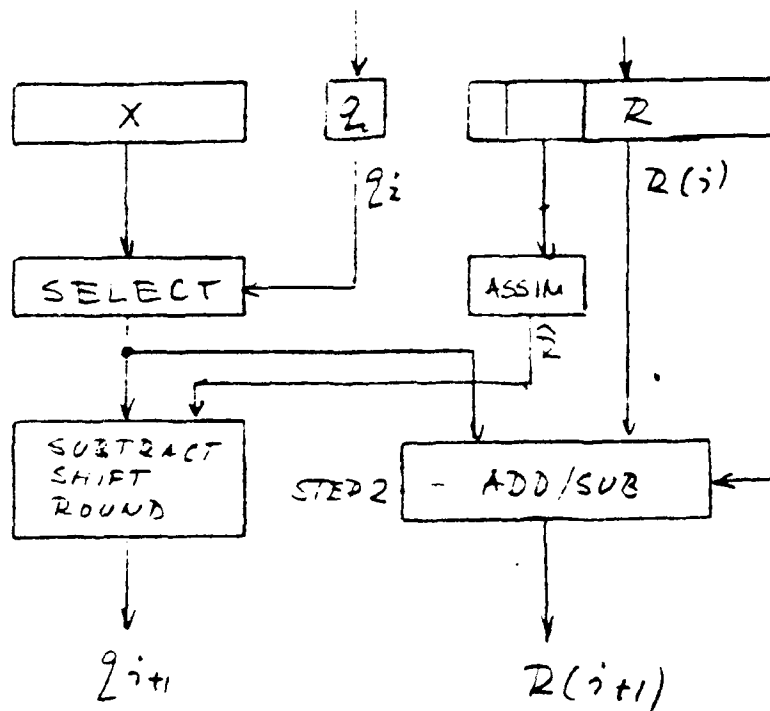


FIGURE 3b: SCHEME WITH PREDICTION BASED ON $\hat{Z}(i)$ AND $q_i X$

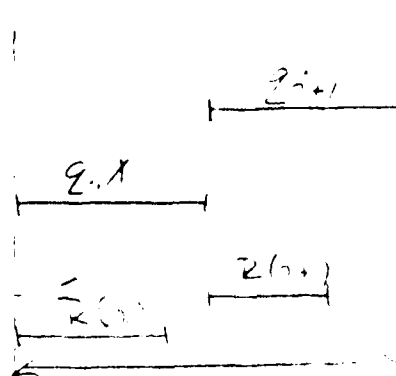


FIGURE 3a: ITERATION WITH PREDICTION

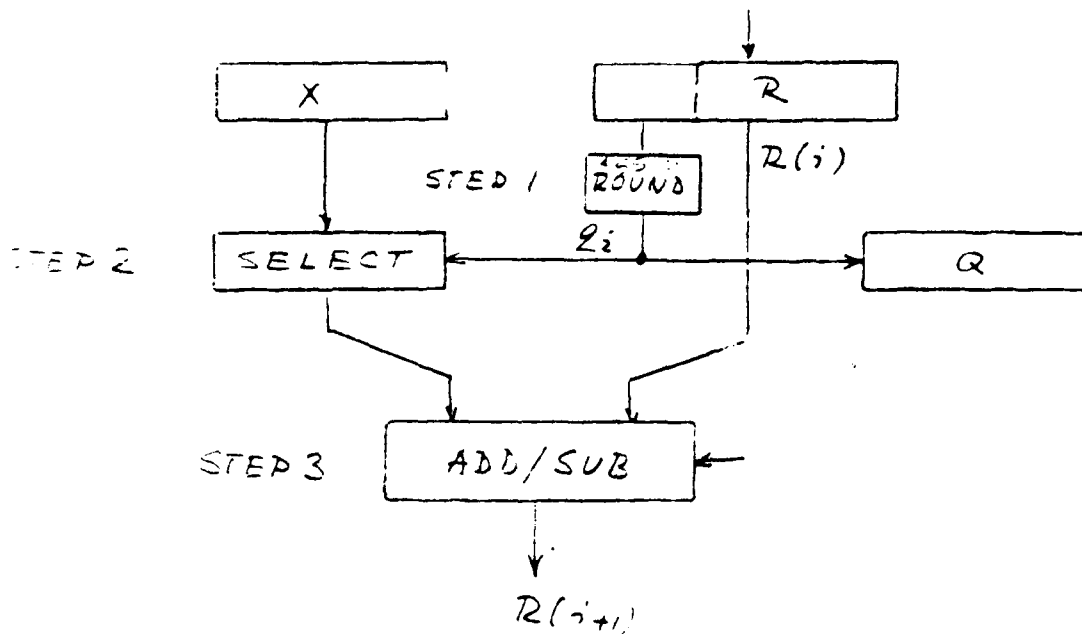


FIGURE 26: BASIC ITERATION SCHEME

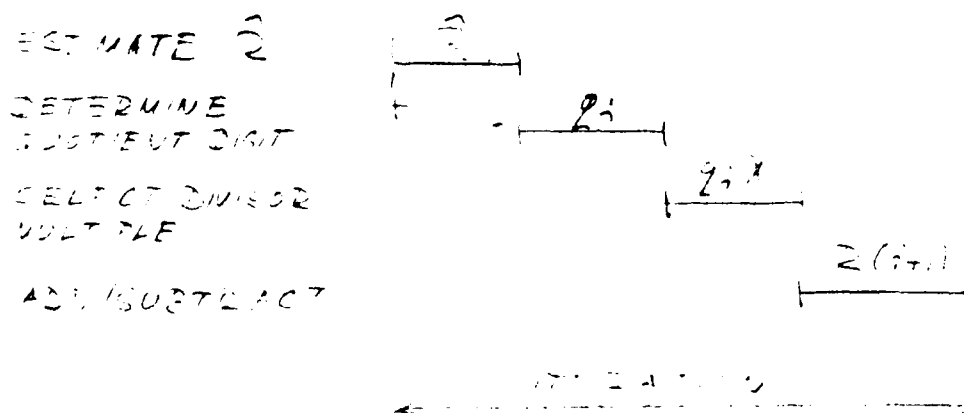


FIGURE 20: BASIC ITERATION

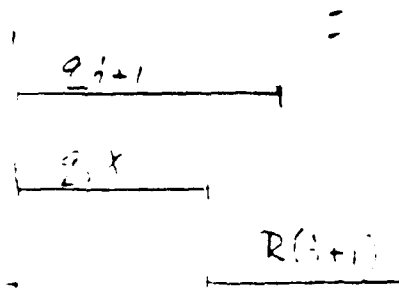
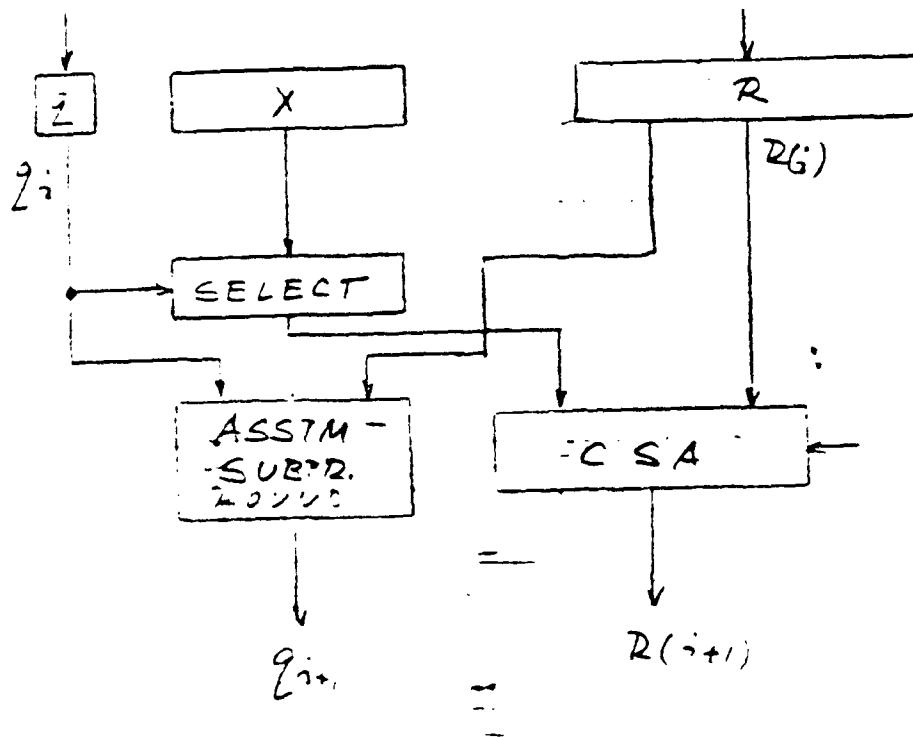


FIGURE 4: PREDICTION BASED ON
 $R(i)$ AND Z_i

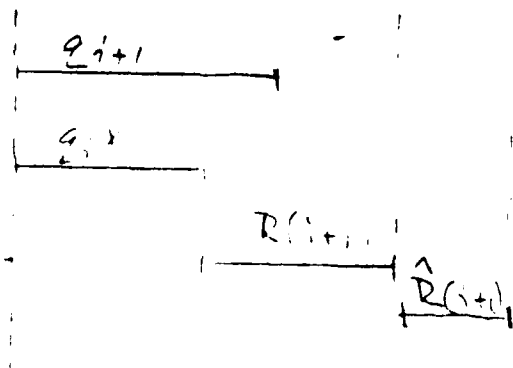
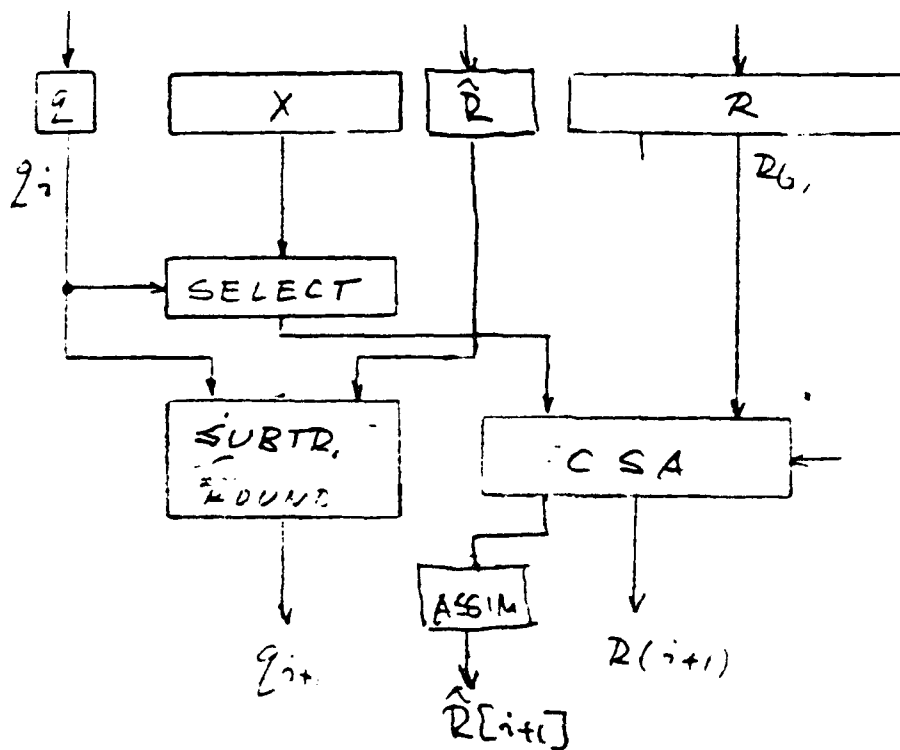


FIGURE 6: RECURSION BASED ON $R(i)$ AND 2^i

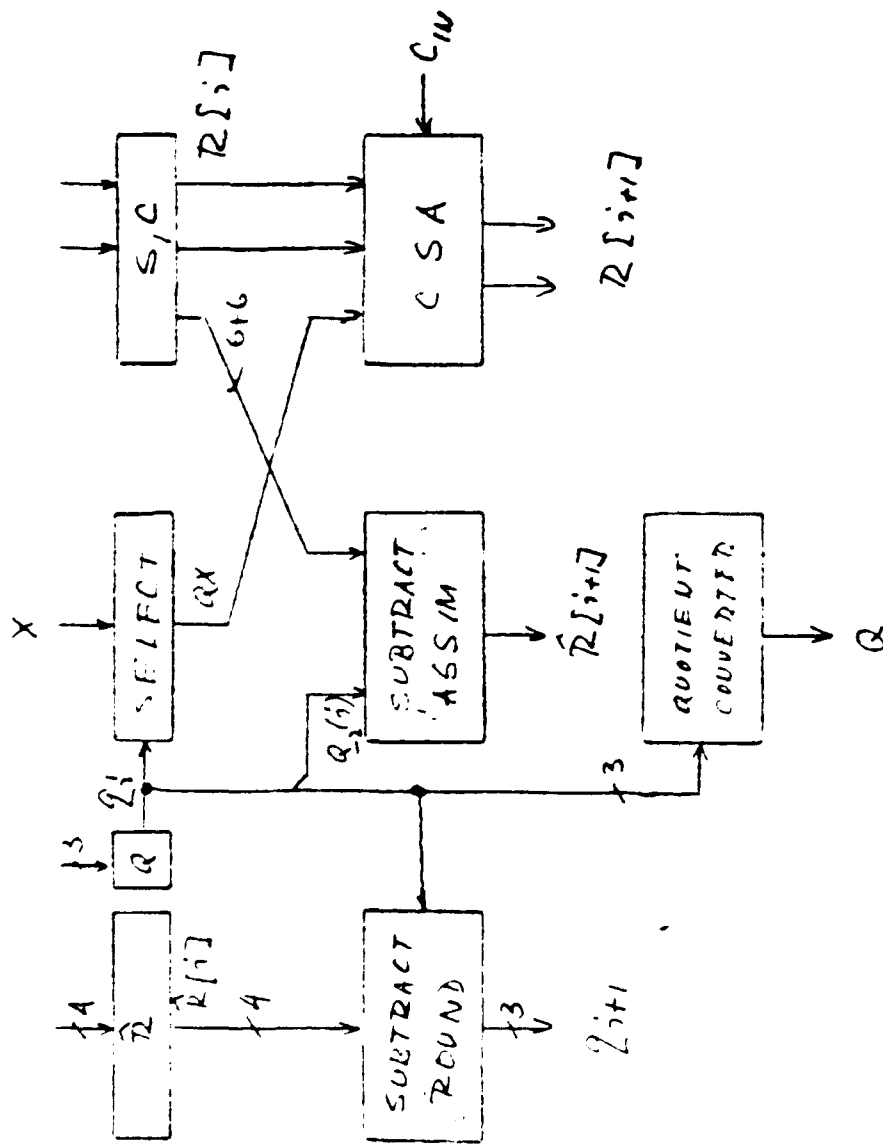


FIGURE 5: RADIX-4 DIVISION SCHEME
WITH PREDICTION

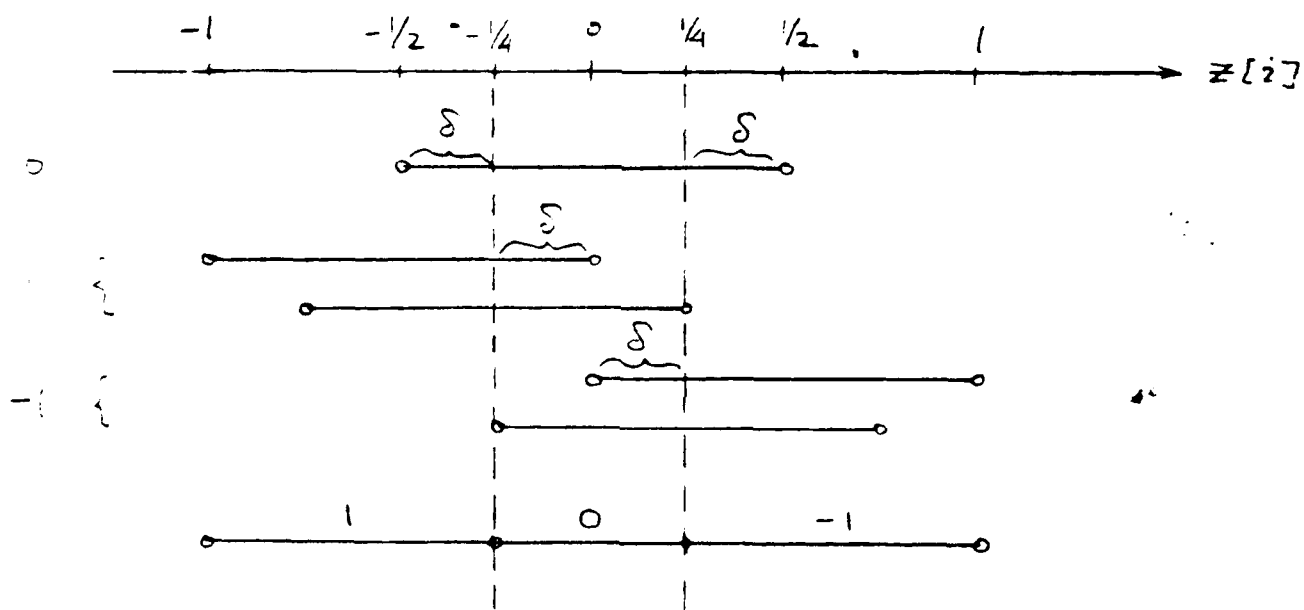


FIGURE 62: SELECTION INTERVALS

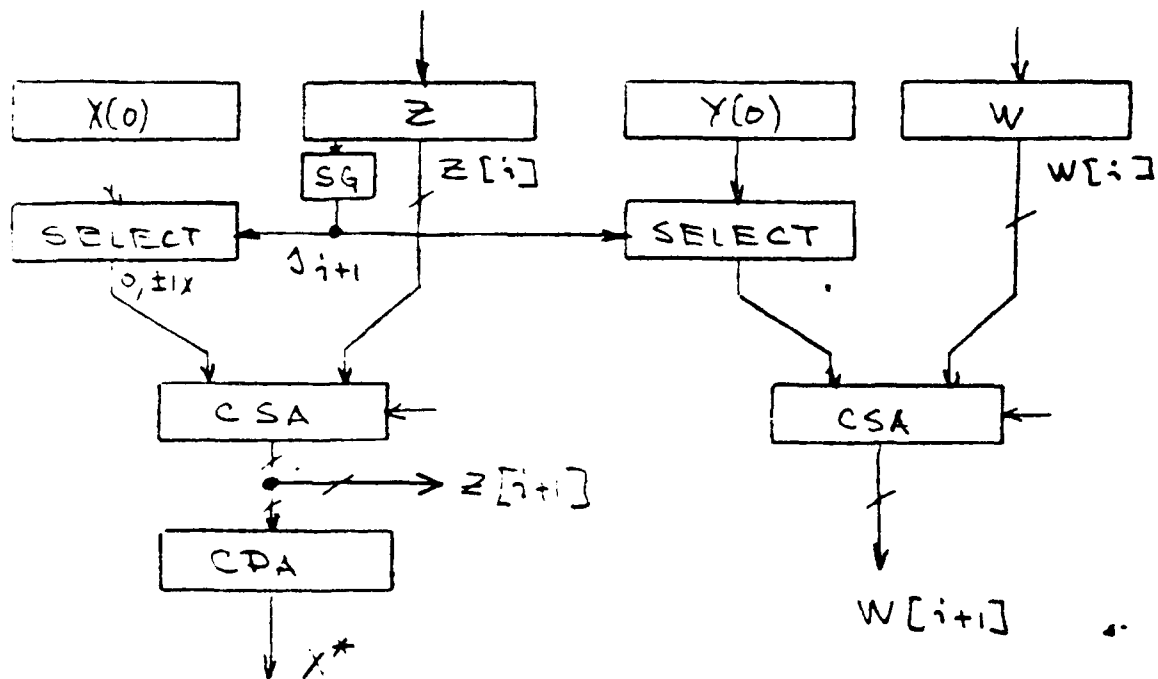
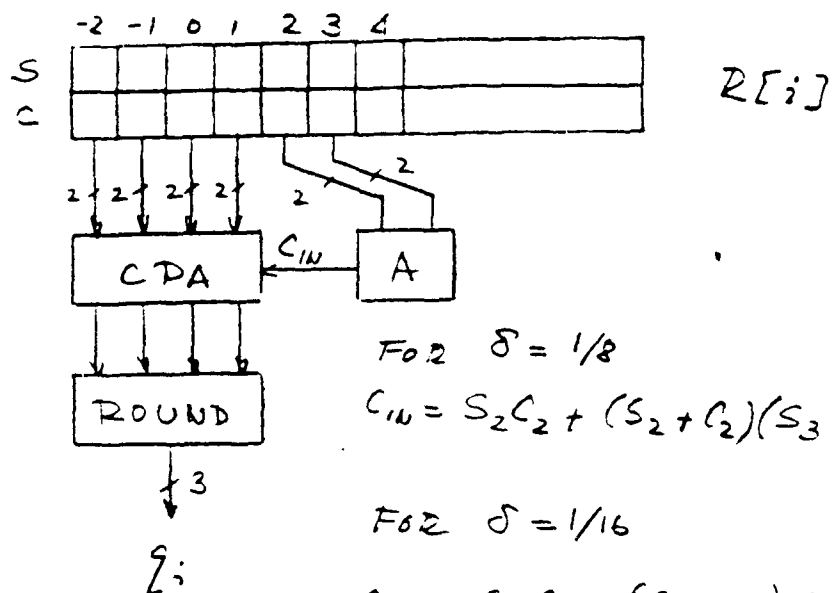


FIGURE 6L TRANSFORMATION SCHEME



For $\delta = 1/8$

$$C_{1N} = S_2 C_2 + (S_2 + C_2)(S_3 + C_3)$$

For $\delta = 1/16$

$$C_{1N} = S_2 C_2 + (S_2 + C_2) S_3 C_3 + (S_2 + C_2)(S_3 + C_3)(S_4 + C_4)$$

FIG. 7: NO PREDICTION, 1-BIT SLICE CSA

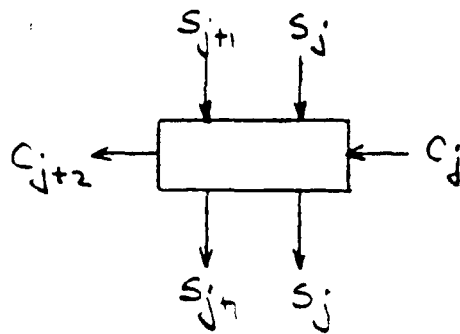
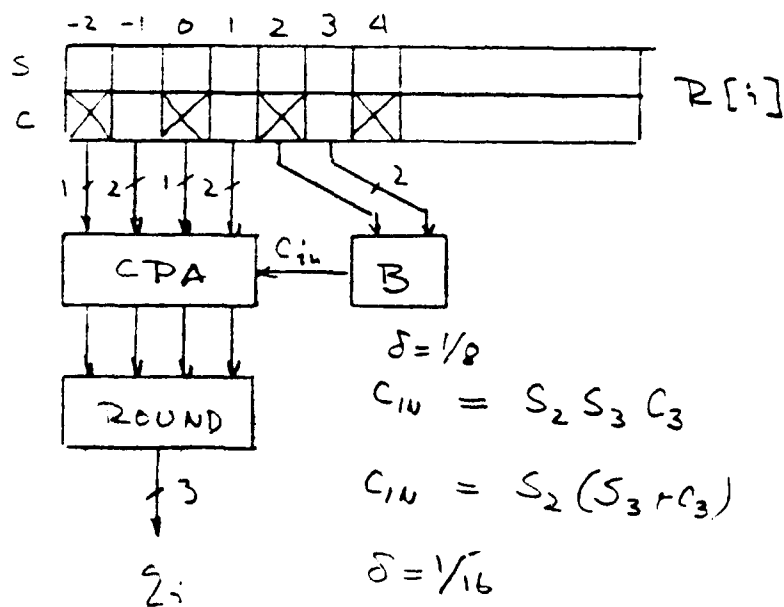


FIG. 8a: 2-BIT SLICE CSA



$$\delta = 1/8$$

$$C_{in} = S_2 S_3 C_3 \quad (\text{NO 1 ADDED})$$

$$C_{in} = S_2 (S_3 + C_3) \quad (\text{1 ADDED})$$

$$\delta = 1/16$$

$$C_{in} = S_2 S_3 C_3 + S_2 (S_3 + C_3) S_4$$

FIG. 8b: NO PREDICTION - 2-BIT SLICE CSA

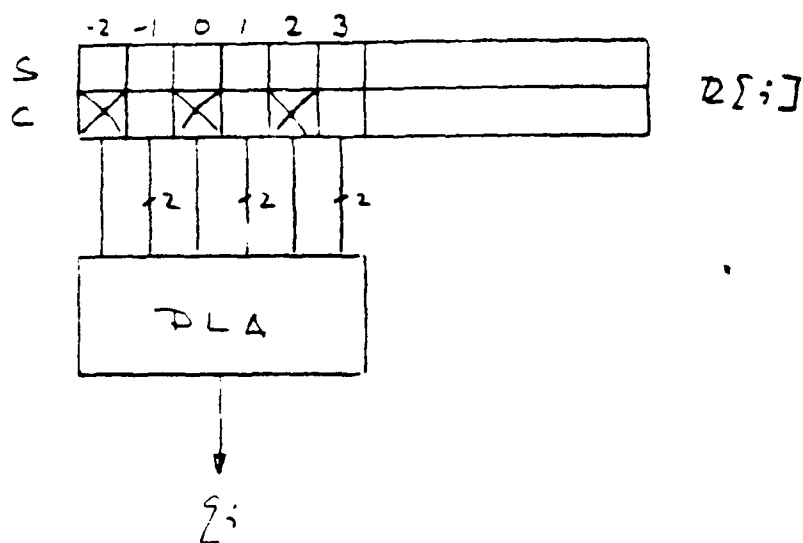


FIG. 9 : NO PREDICTION, WITH DLA
(2-BIT SLICE)

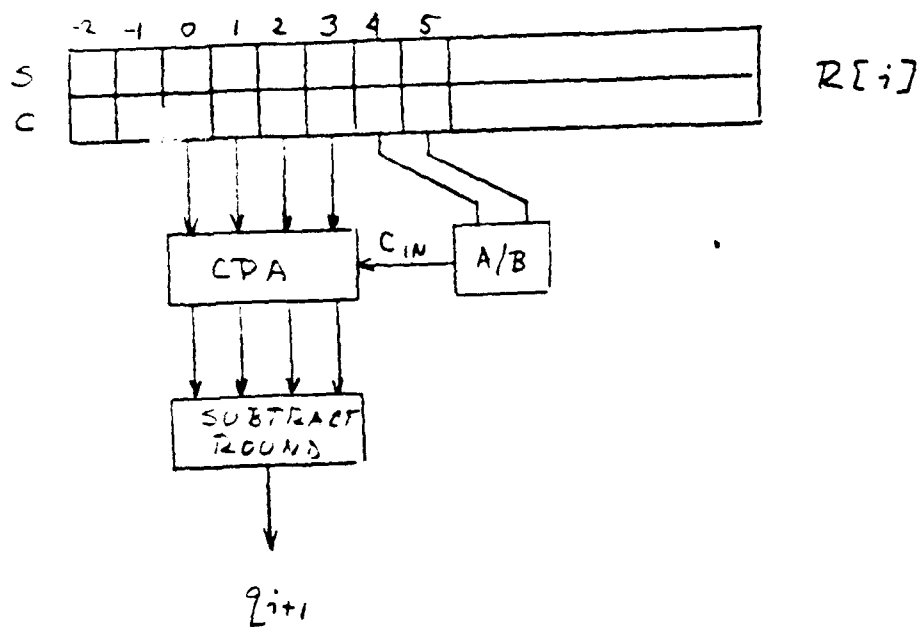


FIG. 10 PREDICTION, ESTIMATE AND q_{i+1}

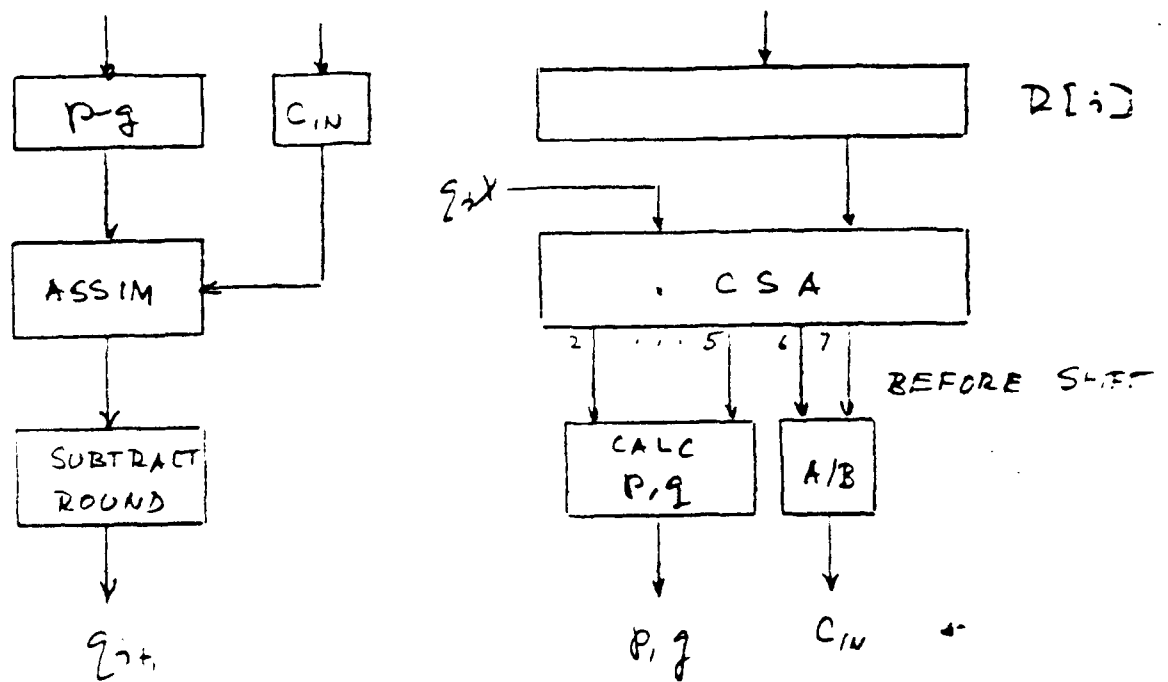


FIG. 11 : PART OF ASSIMILATION IN REMAINDER PATH

Range Transformation of the Divisor

M. Ercegovac and T. Lang

December 20, 1984

The division algorithm discussed in the previous reports [ERLA84] requires that the divisor be transformed into the range $(1-\alpha) \leq X^* \leq (1+\alpha)$. In [ERCE83] a transformation based on the continued product normalization algorithm [ERCE72, ERCE73] is given. The implementation of this algorithm corresponds to a recurrence which is difficult to speed-up sufficiently with the available hardware complexity. We also considered several radix-2 iterative algorithms for the divisor transformation. The one-bit-per-step alternative has an undesirably long step time while that with the prediction, on the other hand, requires very complex selection rules. All of the above mentioned approaches are characterized by a sequential generation of digits used in the divisor transformation thus precluding any overlap between the steps. As an alternative that has more potential for a faster transformation, we consider here an approach based on a reciprocal approximation by power series. The method consists of two parts:

- (i) Compute M , an approximation to the reciprocal of the divisor X , such that

$$|M - 1/X| \leq \alpha/X$$

- (ii) Multiply the divisor by M to obtain X^* , such that

$$X^* = XM \quad \text{and} \quad |X^* - 1| \leq \alpha$$

1. Power Series Approximation

Let $R = 1/D$ where $1 \leq D < 2$ is $2X$. That is,

$$D = (1.x_2x_3\dots x_kx_{k+1}\dots x_n)$$

and

$$R = (0.1r_2r_3\dots r_n)$$

To compute R , we decompose D such that $D = X_1 + 2^{-k}X_2$ where

$$X_1 = (1.x_2\dots x_{k+1}) \quad 1 \leq X_1 < 2 - 2^{-k}$$

$$X_2 = (0.x_{k+2}\dots x_n) \quad 0 \leq D_2 < 1 - 2^{-(n-k)}$$

By McLaurin's series expansion we have

$$\begin{aligned}
 R &= \frac{1}{D} = \frac{1}{X_1 + 2^{-k}X_2} \\
 &= \frac{1}{X_1} \frac{1}{1 + 2^{-k}X_2 \frac{1}{X_1}} \\
 &= R_1 \frac{1}{1 + 2^{-k}X_2 R_1} \\
 &= R_1 [1 - 2^{-k}X_2 R_1 + 2^{-2k}(X_2 R_1)^2 - \dots]
 \end{aligned}$$

For the approximation of R we use the first two terms of the expansion and truncate the result to t bits. We get

$$\hat{R} = \hat{R}_1 - 2^{-k}\hat{R}_1^2\hat{X}_2 - e,$$

where

$\hat{R}_1$ is R_1 truncated to u bits,

$\hat{R}_1^2$ is $(\hat{R}_1)^2$ truncated to s bits,

$\hat{X}_2$ is X_2 truncated to v bits, and

$0 \leq e, < 2^{-t}$ is the truncation error.

We now compute a bound for the approximation error e such that

$$R = \hat{R} + e$$

This error can be written as

$$e = e_T + e_R + e_S$$

where

- e_T is the error due to the use of only two terms of the series,
- e_R is the error due to the truncation of R_1 , and
- e_S is the error due to the truncation of R_1^2 and of X_2 .

We have (*)

$$R = (\hat{R}_1 + 2^{-u}) - 2^{-k}(\hat{R}_1^2 + 2^{-s})(\hat{X}_2 + 2^{-v}) + e_T$$

which results in

$$R = \hat{R} + 2^{-t} + 2^{-u} - 2^{-(k+v)}\hat{R}_1^2 - 2^{-(k+s)}\hat{X}_2 - 2^{-(k+s+v)} + e_T$$

Consequently, since $\hat{R}_1^2 \leq 1 - 2^{-s}$ and $\hat{X}_2 \leq 1 - 2^{-v}$,

$$-(2^{-(k+v)} + 2^{-(k+s)}) \leq e \leq 2^{-t} + 2^{-u} + 2^{-2k}$$

The choice of k , t , s , and v should be made so that

$$|e| \leq \alpha/2 \quad \text{since} \quad M = 2R$$

Let

$$\alpha = 2^{-p}$$

then

$$2^{-t} + 2^{-u} + 2^{-2k} \leq 2^{-p-1}$$

and

$$2^{-(k+v)} + 2^{-(k+s)} \leq 2^{-p-1}$$

Several choices for k , t , u , and v are possible to satisfy these conditions. The selection should simplify the implementation. For example for $\alpha = 2^{-6}$, the following are possible choices:

$$k=4, t=9, u=9, s=4, \text{ and } v=4$$

$$k=5, t=9, u=8, s=3, \text{ and } v=3$$

$$k=6, t=8, u=9, s=2, \text{ and } v=2$$

For $\alpha = 2^{-7}$:

$$k=5, t=9, u=10, s=4, v=4$$

$$k=6, t=9, u=10, s=3, v=3$$

(*) To simplify the notation some of the errors are denoted by their maximum values.

2. Reciprocal Generation for $\alpha = 2^{-6}$ ($p = 6$)

We now consider a detailed design for $p = 6$. We choose $k=5$, $r=9$, $u=8$, $s=3$, and $v=3$.

Table 1 displays the 8-bit truncated reciprocal and its 3-bit truncated square.

Table 1

X_1 $x_1x_2x_3x_4x_5x_6$	$\hat{R}_1$ $z_0z_1z_2z_3z_4z_5z_6z_7z_8$	$\hat{R}^2$ $w_0w_1w_2w_3$
1.00000	0.11111111	0.111
1.00001	0.11111000	0.111
1.00010	0.11110000	0.111
1.00011	0.11101010	0.110
1.00100	0.11100011	0.110
1.00101	0.11011101	0.101
1.00110	0.11010111	0.101
1.00111	0.11010010	0.101
1.01000	0.11001100	0.100
1.01001	0.11000111	0.100
1.01010	0.11000011	0.100
1.01011	0.10111110	0.100
1.01100	0.10111010	0.011
1.01101	0.10110110	0.011
1.01110	0.10110010	0.011
1.01111	0.10101110	0.011
1.10000	0.10101010	0.011
1.10001	0.10100111	0.011
1.10010	0.10100011	0.011
1.10011	0.10100000	0.011
1.10100	0.10011101	0.010
1.10101	0.10011010	0.010
1.10110	0.10010111	0.010
1.10111	0.10010100	0.010
1.11000	0.10010010	0.010
1.11001	0.10001111	0.010
1.11010	0.10001101	0.010
1.11011	0.10001010	0.010
1.11100	0.10001000	0.010
1.11101	0.10000110	0.010
1.11110	0.10000100	0.010
1.11111	0.10000010	0.010

These functions can be implemented using a 5-input, 10-output PLA. If this is not feasible, the table can be decomposed into subtables.

As indicated, the reciprocal R is approximated by

$$\hat{R} = \hat{R}_1 - 2^{-4}\hat{R}_1^2\hat{X}_2$$

To perform the subtraction we could complement the subtrahend and add. However, this would require a range extension of this subtrahend, which complicates the implementation. It seems better to complement the first term (it can be obtained in this form from the PLA), add, and then complement the result. The use of ones' complement seems better since it avoids the addition of 1 in the complementation of the result (note that no end-around-carry is produced during the addition since $z_1' = 0$). The trailing 1's produced by the complementation of $\hat{R}_1$ can be avoided by incrementing a unit in the last significant position of the complement of $\hat{R}_1$.

The configuration of the addition is shown in Figure 1. The multiplication of $\hat{R}_1^2$ by $\hat{X}_2$ is implemented by the addition of three partial products.

Since the reciprocal approximation will be used to multiply the divisor in order to obtain X^* , and a radix-4 multiplication is to be used, it is necessary to recode the reciprocal approximation to a radix-4 representation with digit set $\{-2, -1, 0, 1, 2\}$. Due to the fact that the most significant bits of the adder have only one operand different from zero, it is possible to perform the recoding on the two most significant radix-4 digits without waiting for the carry propagation from the other digits. This is convenient since we are going to perform the multiplication beginning with the most significant digit of the reciprocal approximation; that is, the multiplication can begin before finishing the addition.

To simplify the recoding the six most significant bits of $\hat{R}_1$ are computed in normal (not complemented) form. Table 2 shows the resulting Z , obtained by complementing $\hat{R}_1$, adding 1 in the least significant position and complementing again positions 0 to 5.

Table 2

X_1 $x_1x_2x_3x_4x_5x_6$	$\hat{R}_1$ $z_0z_1z_2z_3z_4z_5z_6'z_7'z_8'$	$\hat{R}^2$ $w_0w_1w_2w_3$
1.00000	0.11111001	0.111
1.00001	0.11110000	0.111
1.00010	0.11100000	0.111
1.00011	0.11101110	0.110
1.00100	0.11100101	0.110
1.00101	0.11011011	0.101
1.00110	0.11010001	0.101
1.00111	0.11010110	0.101
1.01000	0.11001100	0.100
1.01001	0.11000001	0.100
1.01010	0.11000101	0.100
1.01011	0.10111010	0.100
1.01100	0.10111110	0.011
1.01101	0.10110001	0.011
1.01110	0.10110110	0.011
1.01111	0.10101010	0.011
1.10000	0.10101110	0.011
1.10001	0.10100001	0.011
1.10010	0.10100101	0.011
1.10011	0.10011000	0.011
1.10100	0.10011011	0.010
1.10101	0.10011110	0.010
1.10110	0.10010001	0.010
1.10111	0.10010100	0.010
1.11000	0.10010110	0.010
1.11001	0.10001001	0.010
1.11010	0.10001011	0.010
1.11011	0.10001110	0.010
1.11100	0.10000000	0.010
1.11101	0.10000010	0.010
1.11110	0.10000100	0.010
1.11111	0.10000110	0.010

In order to do the recoding of the first two digits without waiting for the carry, the third digit has to absorb the carry c_3 into position 5, as indicated in the figure. Since this carry corresponds to the complement of the result, when this carry is 1, a unit has to be subtracted from the third digit. To absorb this subtraction, we recode the digit to the set $\{-1, 0, 1, 2\}$. The recoding is

$c_5z_4z_5$	M_2
000	0
001	1
010	2
011	(1)(-1)
100	-1
101	0
110	1
111	-2

Consequently, the recoding of M_1 is:

z_2z_3t	M_1
000	0
010	1
100	(1)(-2)
110	(1)(-1)
001	1
011	2
101	(1)(-1)
111	(1)0

where $t = z_4z_5$

Finally, the recoding for M_0 is:

z_2	M_0
0	1
1	2

For the recoding of M_3 and M_4 we use the corresponding bits of the result of the addition, after complementation. This results in:

$R_6 R_7 R_8$	M_3
000	0
001	1
010	1
011	2
100	-2
101	-1
110	-1
111	0

Using a sign-and-magnitude representation of the radix-4 digits we get the following switching expressions:

$$M_{0s} = 0 \quad M_{01} = z_2 \quad M_{00} = z_2'$$

$$M_{1s} = z_2 \quad M_{11} = z_2 z_3' t' + z_2' z_3 t \quad M_{10} = z_3' t + z_2 t + z_2' z_3 t'$$

$$M_{2s} = z_4 z_5 + z_4' z_5' \quad M_{21} = z_4 z_5' c_5' + z_4 z_5 c_5 \quad M_{20} = z_5' c_5 + z_5 c_5'$$

$$M_{3s} = R_6 \quad M_{31} = R_6 R_7' R_8' + R_6' R_7 R_8 \quad M_{30} = R_7' R_8 + R_7 R_8'$$

$$M_{4s} = R_8 \quad M_{41} = R_8 R_9' \quad M_{40} = R_9$$

A design of the adder and the recoding circuit is shown in Figure 2.

3. Divisor Transformation

The divisor is transformed by multiplying it by M . Since the most significant digits of M are ready first, and to use the same carry-save-adder used for the division, we perform the multiplication beginning with the most significant digit of M . This type of multiplication usually requires an adder of increasing precision. However, in this case the most significant bits of X^* are 1.000000 or 0.111111, so that keeping a few extra bits is sufficient to determine X^* .

To reduce the number of multiplication steps, it is possible to load the registers with $M_0 X$ and produce $4M_0 X + M_1 X$ in one multiplication step. The configuration of the multiplication steps is shown in Figure 3.

To put the divisor in the form required by the division, it is necessary to convert it from the carry-save representation to a conventional representation. This step corresponds to a carry-propagate addition.

4. Timing of the Transformation

The time of the transformation is determined by the following components:

a) Determination of $\hat{R}_1$ and $\hat{R}_1^2$ by the PLA. We assume 2 gate delays for this step.

b) Recoding to obtain M_0 and M_1 . This requires two gate delays.

c) Multiplication step to obtain $4M_0X + M_1X$. This corresponds to 5 gate delays.

d) Three more multiplication steps to multiply by M_2 , M_3 , and M_4 . We assume that the addition and recoding of these digits is overlapped with steps b) and c). This requires $5 \times 3 = 15$ gate delays.

e) The carry-propagate addition. We assume 15-20 gate delays.

The total is of 39-44 gate delays. This corresponds to 8-9 cycles.

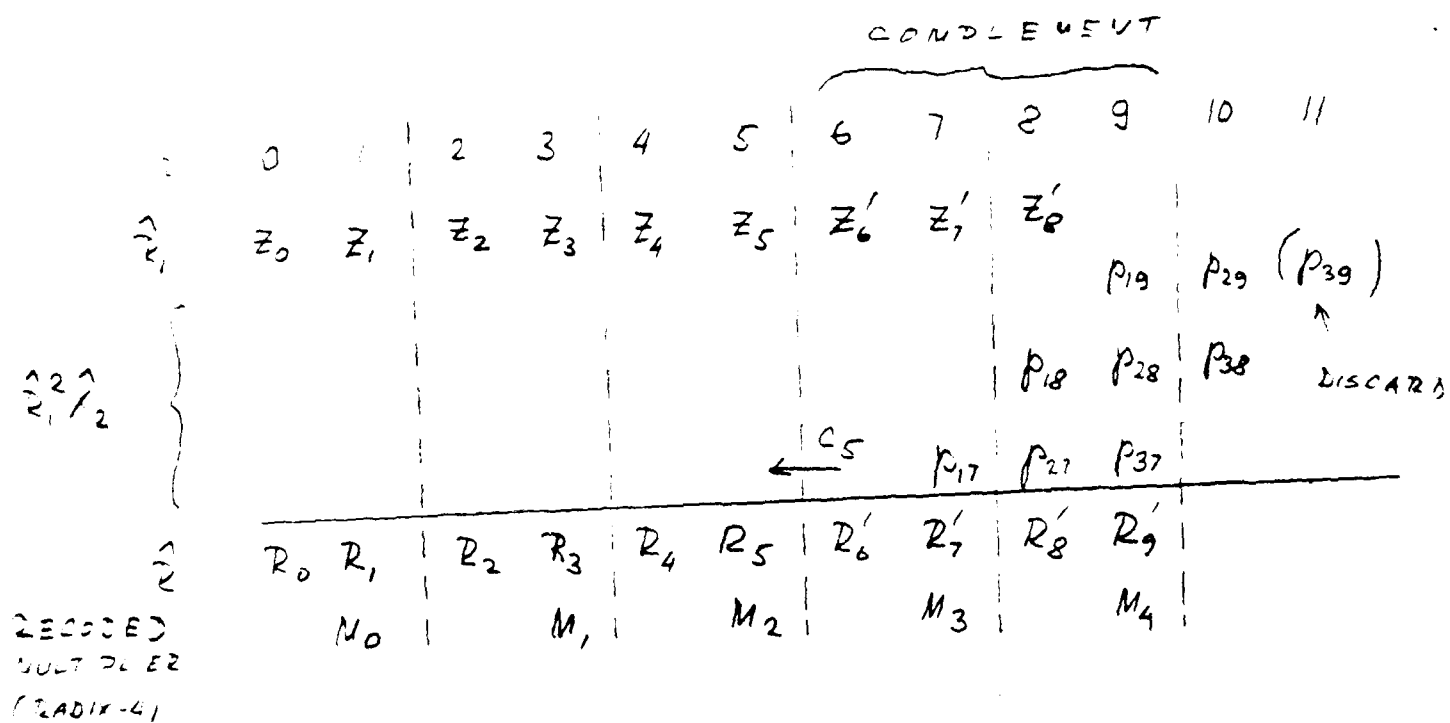
References

[ERLA84] M.D. Ercegovac and T. Lang, "Radix-4 Division with Range Transformation", unpublished manuscript, 1984.

[ERCE83] M.D. Ercegovac, "A Higher Radix Division with Simple Selection of Quotient Digits", Proc. 6th Symposium on Computer Arithmetic, 1983.

[ERCE72] M.D. Ercegovac, "Radix 16 Division, Multiplication, Logarithmic and Exponential Algorithms Based on Continued Product Representations", MS Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1972.

[ERCE73] M. D. Ercegovac, "Radix-16 Evaluation of Certain Elementary Functions", IEEE Trans. on Computers, June 1973, pp.561-566.



$$P_{jk} = w_j \cdot X_k$$

FIGURE 1: COMPUTATION OF $\hat{Z}$ AND RECODED MULTIPLIER M

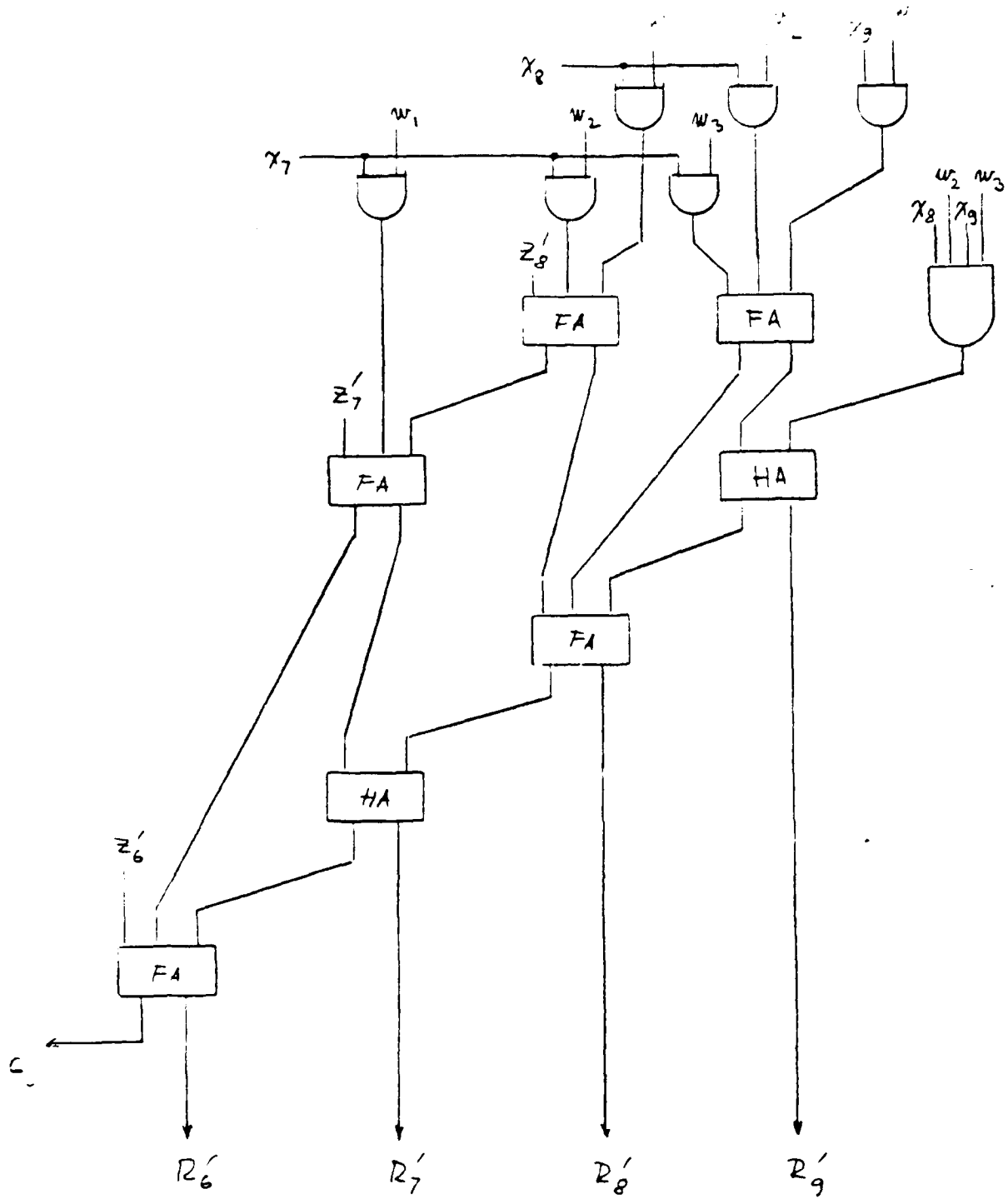


FIGURE 2a: 2's ADDER

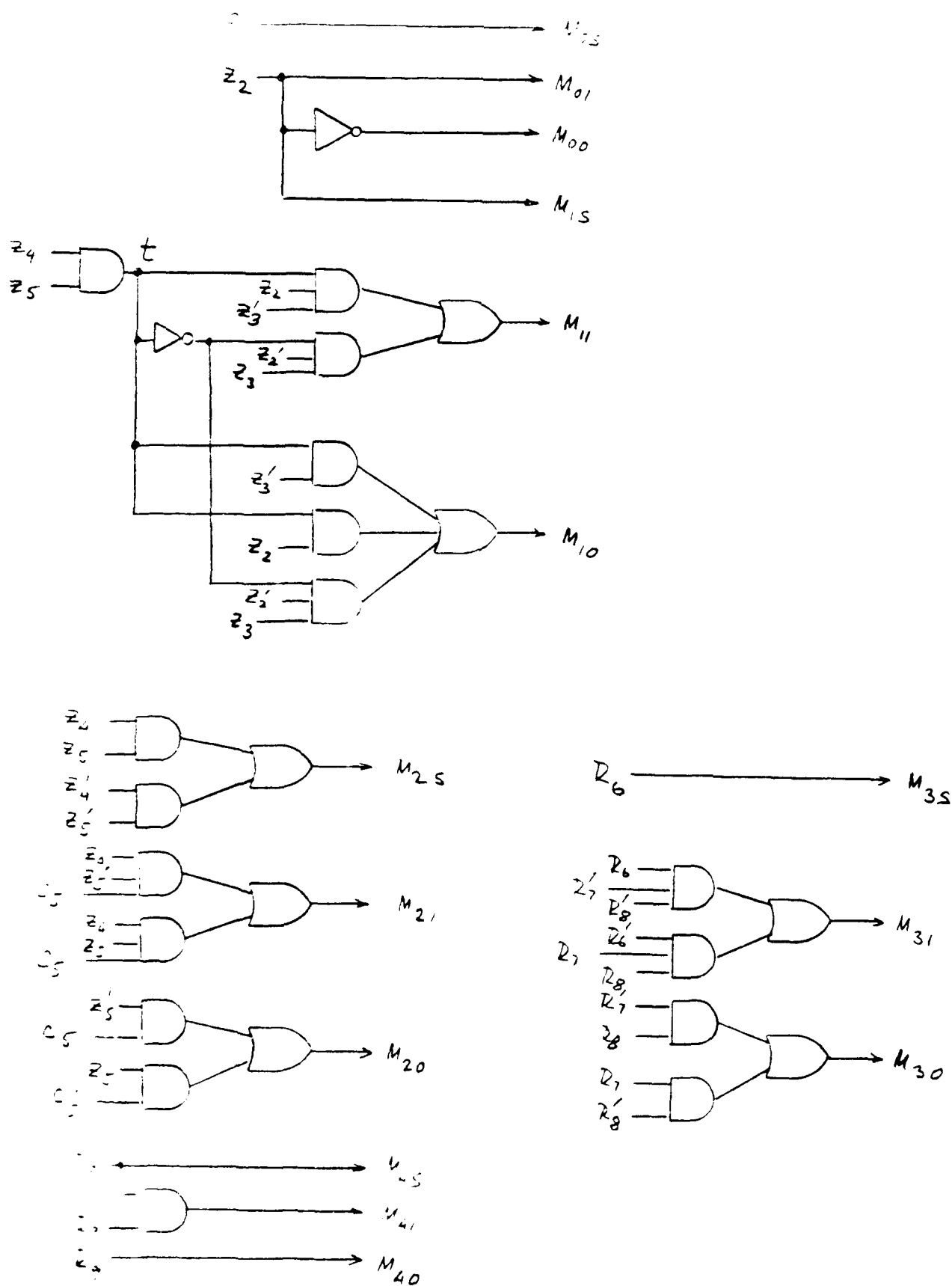


FIGURE 26: DECODER

Approved for public release.
distribution unlimited

The views and conclusions contained in
this report are those of the authors and
should not be interpreted as representing
the official policies, either expressed or
implied, of the Naval Ocean Systems
Center or the U.S. Government

**END
DATE
FILMED**

1-6-86
R.H.